

POWERSHELL FOR NEWBIES

Getting started with PowerShell 4.0

Jeffery Hicks

AVAILABILITY™
for the Modern Data Center

Abstract

This eBook is intended as a quick start guide for IT Professionals who want to learn about Windows PowerShell. The eBook assumes the reader has little to no experience with PowerShell and needs some guidance in getting started. The eBook will assume PowerShell v4, although for a beginner that doesn't matter much since the fundamental topics are version independent.

Contents

Introduction and expectations	4
What is PowerShell?	4
What PowerShell is not	5
Why it matters to you	5
Assumptions and requirements	6
PowerShell help	6
Updating help	6
Saving help	7
Using help	7
Online	11
About topics	12
Terminology	13
Cmdlet	13
Parameter	13
Alias	14
Function	14
Variable	15
Module	15
Snapin	17
Provider	19

The PowerShell paradigm	21
Objects	22
Get-Member	22
The PowerShell pipeline	23
Bigger is not always better	25
A Veeam example	26
Next steps	31
Further reading and resources	31
About the Author	32
About Veeam Software	33

Introduction and expectations

Windows PowerShell has been around for several years and even though at the time of writing this eBook we are at v4.0, and v5.0 is in preview, there are still many IT Pros who are only now coming around and realizing they need to learn PowerShell. They have realized that PowerShell isn't a passing fad and that if they want any sort of job security or career future related to Windows and Microsoft-based technologies, PowerShell is going to play an important role.

The goal of this eBook is to give you a crash course on PowerShell essential terms, concepts and commands. I am assuming you have little to no experience with PowerShell and are what Microsoft considers an IT Pro; that is, someone whose daily work involves managing Windows-based servers, applications and platforms. Naturally, you can't learn PowerShell from one eBook, but I hope to leave you with enough information to get started and I'll wrap up with a list of resources and next steps.

What is PowerShell?

First off, what exactly is PowerShell? My answer is that PowerShell is an object-based management engine based on the .NET Framework. This engine is exposed through a series of commands called cmdlets and providers (note that PowerShell terms are explained later in the Terminology section). PowerShell can be exposed through an interactive console as well as a batch-oriented scripting language.

The engine is hosted in an application. Microsoft ships two out of the box, at least on the client. The PowerShell console you have probably seen is hosted by the legacy cmd.exe command shell. In other words, PowerShell is sitting on top of a CMD window, with all of the benefits and drawbacks that entails. PowerShell can also be hosted in a GUI, like the PowerShell ISE, which I'll touch on later. Vendors, developers and the PowerShell community can create their own hosting applications, and usually PowerShell behaves the same regardless of how you are interacting with it.

In fact, that is one of the great things about PowerShell. Once you learn the fundamentals, it doesn't matter if you are working with files, processes, Active Directory user accounts, Exchange mailboxes or Hyper-V virtual machines. You will use the same skills and often the same cmdlets. Yes, you may struggle at first learning PowerShell, but once you get over the initial hurdle, I think you'll find your learning curve becomes much shallower.

What PowerShell is not

There are also some misconceptions about PowerShell I need to clear up. First, PowerShell is not simply another scripting language like VBScript. Yes, you can create some very powerful scripts using PowerShell's simple scripting language, but you don't have to use a script to use PowerShell. Many of you will simply open a PowerShell prompt and start running commands. All that the scripts do is save time typing.

PowerShell is not a programming language. Although it is built on the .NET Framework and it allows you to create some pretty amazing tools, this is not the intent. There is a PowerShell SDK, but that is intended for developers using Visual Studio to create PowerShell-based solutions. This is not something a typical IT Pro needs to worry about.

Finally, using PowerShell does not mean you are destined to spend your day typing commands at a command prompt. There will still be graphical tools, but they will run on top of PowerShell. More than likely you may be using PowerShell now through a GUI and not even know it. The limitations of the GUI are related to the very nature of its design. You can only do what the GUI is designed to do. Eventually you will want to get out of the GUI and take matters into your own hands at a PowerShell prompt. Or eventually build your own GUI!

Why it matters to you

All of this matters because PowerShell is an integral part of Windows and will only continue to be widespread in its use. Even today we are seeing where PowerShell is the glue that holds a variety of management tools and technologies together. From PowerShell Workflow to Desired State Configuration and new features in v5, PowerShell is the common denominator. If you want to have a career as an IT Pro, I've long held that it isn't a matter of **if** you'll learn PowerShell, but **when**.

In addition to Microsoft, many companies are offering PowerShell solutions for their products. Companies like VMware, Citrix, StarWind and Veeam® now offer PowerShell tools to manage their respective products—often for free! Yes, you can continue to use traditional tools, but I think once you realize what you can accomplish from a PowerShell prompt, you'll be a convert. And once you learn the fundamentals, it doesn't matter where the PowerShell commands come from, they should all behave the same.

Assumptions and requirements

Everything I am going to demonstrate in this eBook will be based on PowerShell 4.0, which at the time of this writing, is the most current production-approved version. I will be using a Windows 8.1 client. For beginners, much of what I'll talk about will also apply to PowerShell 3.0. If you are still running PowerShell 2.0, I would encourage you to update when possible. And if for some reason you are still running PowerShell 1.0, you have my sympathies.

As I stated earlier, this guide is for absolute beginners and will cover fundamental concepts. If you have been using PowerShell for a little while, this eBook might make a nice refresher. I will not be covering advanced topics like workflow and Desired State Configuration, or even intermediate topics like PowerShell remoting. I want you to feel comfortable typing commands at a PowerShell prompt and to let go of some of your anxiety or trepidation.

PowerShell help

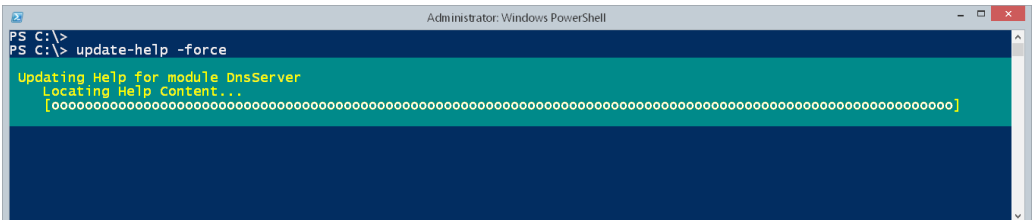
The first thing you must understand in PowerShell is to how to use the help system. The PowerShell team has written thorough documentation. Many other add-ins from Microsoft and third parties will also have help documentation. If you don't know how to use help, you will constantly struggle in learning and using PowerShell.

Updating help

The first thing you need to do on a new install is update the help. By default PowerShell only ships with minimal help. For servers this usually isn't that big of an issue since you are unlikely to be using a server for your day-to-day administration. To update the help, your computer needs to have an Internet connection. From an elevated PowerShell session run this command:

PS C:\>update-help

PowerShell will enumerate all of the installed modules, which are packages of commands, find the link to updated help, download and install the help files.



```

Administrator: Windows PowerShell
PS C:\>
PS C:\> update-help -force
Updating Help for module DnsServer
Locating Help Content...
[*****]
  
```

Figure 1

For out-of-the-box commands, this means your computer will connect to a Microsoft website to get the most up-to-date help. By design the command will only connect once every 24 hours, unless you use the `-Force` parameter as in Figure 1.

Be aware that not every module will have updated help. So don't be surprised if you get some error messages. There's nothing you can do except ignore them and hope that eventually updated help will be provided.

Saving help

One downside to using the `Update-Help` command is that every desktop might need to connect to Microsoft to download the most current help. However, you can save help to a local folder or network share and then update help from that location. The command works essentially the same as `Update-Help` except you need to specify the location, which must already exist.

PS C:\> Save-Help -DestinationPath \\file01\help\v4help -force

If you will have a mix of PowerShell 3.0 and later clients, keep help for each version in a separate folder. Once saved, anyone can run `Update-Help` and specify the alternate location.

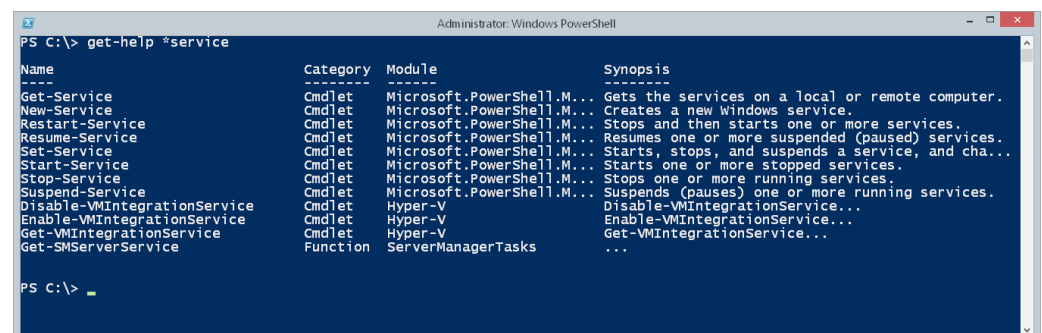
PS C:\> Update-Help -SourcePath \\file01\help\v4help -force

Be aware that saving help will only retrieve help for the commands installed on the computer running the command. If the computer running `Update-Help` has different commands, it may not get updated help.

Using help

Once help has been updated you can begin to use it. All you need to do is ask for help and you can use wildcards. You can use either the `Get-Help` command or the `Help` function, which works essentially the same as `Get-Help` with the addition of paging.

PS C:\> get-help *service



```

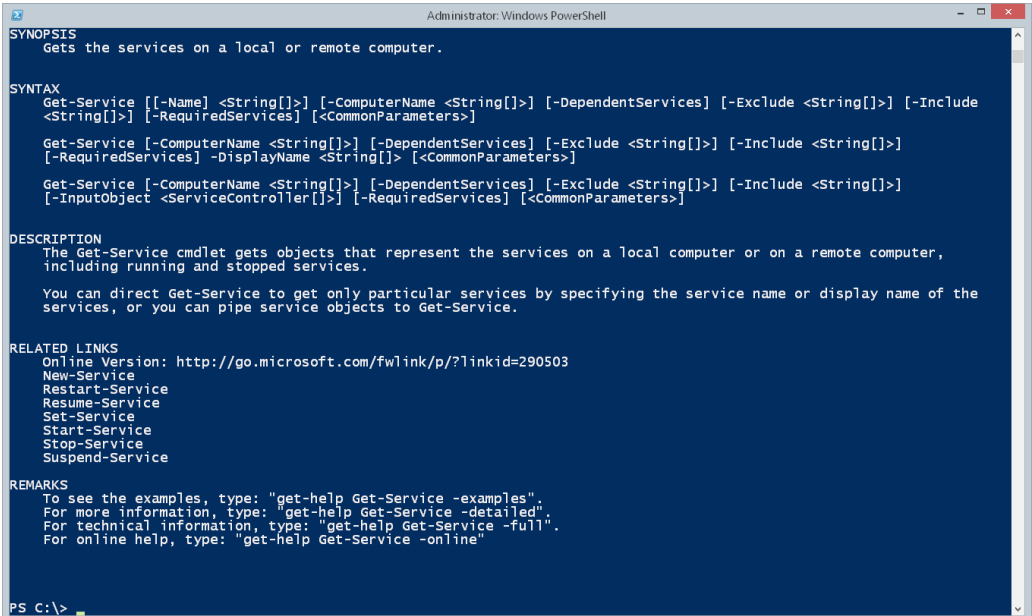
PS C:\> get-help *service
Name                Category  Module                               Synopsis
-----
Get-Service         Cmdlet   Microsoft.PowerShell.M...           Gets the services on a local or remote computer.
New-Service         Cmdlet   Microsoft.PowerShell.M...           Creates a new Windows service.
Restart-Service    Cmdlet   Microsoft.PowerShell.M...           Stops and then starts one or more services.
Resume-Service     Cmdlet   Microsoft.PowerShell.M...           Resumes one or more suspended (paused) services.
Set-Service        Cmdlet   Microsoft.PowerShell.M...           Starts, stops, and suspends a service, and cha...
Start-Service      Cmdlet   Microsoft.PowerShell.M...           Starts one or more stopped services.
Stop-Service       Cmdlet   Microsoft.PowerShell.M...           Stops one or more running services.
Suspend-Service   Cmdlet   Microsoft.PowerShell.M...           Suspends (pauses) one or more running services.
Disable-VMIntegrat Cmdlet   Hyper-V                               Disable-VMIntegratService...
Enable-VMIntegrat Cmdlet   Hyper-V                               Enable-VMIntegratService...
Get-VMIntegratService Cmdlet   Hyper-V                               Get-VMIntegratService...
Get-SMServerService Function  ServerManagerTasks                  ...
  
```

Figure 2

Figure 2 shows all of the commands on my computer that PowerShell can find that end in 'Service.' To know more about a specific command, I simply ask for help.

PS C:\> help get-service

Depending on the size of your console window the display might end with a More command. Press **Enter** to advance a line at a time, press the **space bar** to advance to the next page or press **Q** to quit. For the most part, PowerShell is not case-sensitive.



```

SYNOPSIS
    Gets the services on a local or remote computer.

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

DESCRIPTION
    The Get-Service cmdlet gets objects that represent the services on a local computer or on a remote computer,
    including running and stopped services.

    You can direct Get-Service to get only particular services by specifying the service name or display name of the
    services, or you can pipe service objects to Get-Service.

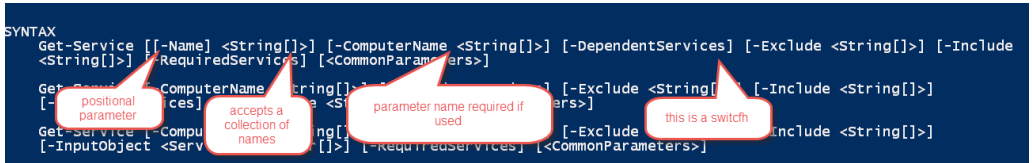
RELATED LINKS
    Online Version: http://go.microsoft.com/fwlink/p/?linkid=290503
    New-Service
    Restart-Service
    Resume-Service
    Set-Service
    Start-Service
    Stop-Service
    Suspend-Service

REMARKS
    To see the examples, type: "get-help Get-Service -examples".
    For more information, type: "get-help Get-Service -detailed".
    For technical information, type: "get-help Get-Service -full".
    For online help, type: "get-help Get-Service -online"

PS C:\>
  
```

Figure 3

The main sections should be self-explanatory. But let's look at the Syntax section a bit more closely (see Figure 3). In this command there are three different ways you can run it. The parameters always begin with a dash (-). Anything that you see in brackets [] indicates it is optional. If you run a command that has a required or mandatory parameter and you do not specify it, PowerShell will prompt you. Figure 4 illustrates some additional help concepts.



```

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

Figure 4

When the entire parameter is in [], as in the case of -Name, this indicates a positional parameter. PowerShell will assume that the first thing you type after Get-Service is a service name. Thus, you could type either of these commands.


```
PS C:\> get-service bits
```

```
PS C:\> get-service -name bits
```

The parameter value must be a string. If the string has spaces you will need to enclose it in quotes. When you see [] as part of the value type, this is your clue that PowerShell will accept an array, or collection of objects—in this case, service names. Generally you can separate the values with a comma.

```
PS C:\> get-service bits,wuauerv
```

Status	Name	DisplayName
Running	bits	Background Intelligent Transfer Ser...
Stopped	wuauerv	Windows Update

The -Computername parameter is completely optional, as indicated by the [] that enclose the entire parameter definition. But if you want to use it, you must specify the parameter name because the name itself is not in [].

```
PS C:\> get-service bits -ComputerName chi-dc04
```

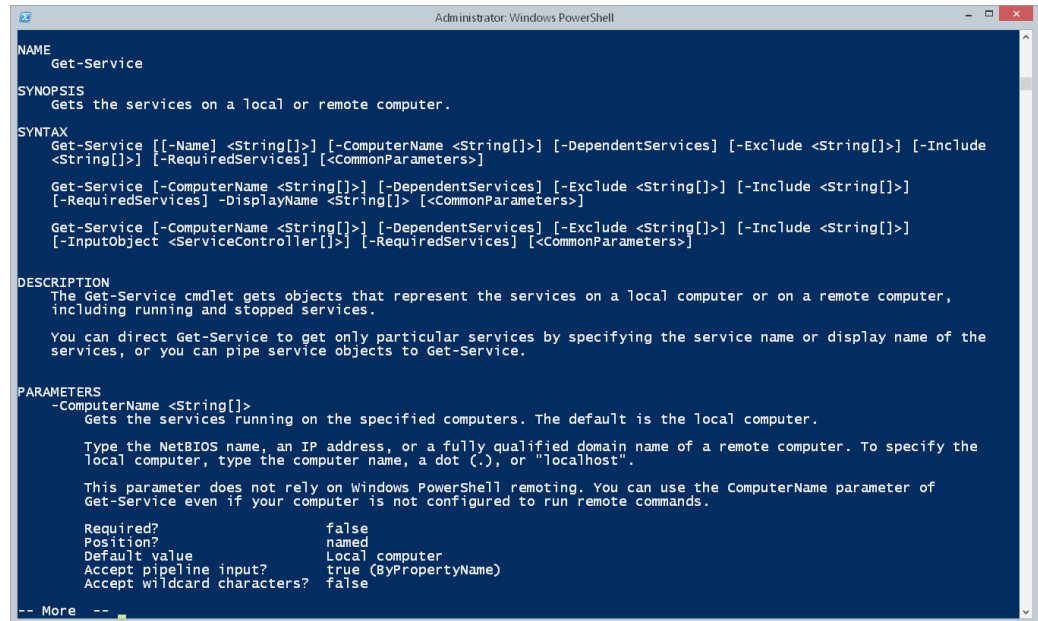
Status	Name	DisplayName
Running	bits	Background Intelligent Transfer Ser...

Finally, you will see some parameters, like -DependentServices, that have no object type for a value. These are referred to as Switch parameters, which are very similar to True/False. If you specify the parameter, PowerShell will treat the parameter as True and do whatever it is designed to do.

```
PS C:\> get-service winmgmt -ComputerName chi-dc04
-DependentServices
```

Status	Name	DisplayName
Running	UALSVC	User Access Logging Service
Stopped	SharedAccess	Internet Connection Sharing (ICS)
Stopped	NcaSvc	Network Connectivity Assistant
Running	iphlpvc	IP Helper

At the end of the help you will see remarks about other help views. I recommend using -Full.



```

NAME
    Get-Service

SYNOPSIS
    Gets the services on a local or remote computer.

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

DESCRIPTION
    The Get-Service cmdlet gets objects that represent the services on a local computer or on a remote computer,
    including running and stopped services.

    You can direct Get-Service to get only particular services by specifying the service name or display name of the
    services, or you can pipe service objects to Get-Service.

PARAMETERS
    -ComputerName <String[]>
        Gets the services running on the specified computers. The default is the local computer.

        Type the NetBIOS name, an IP address, or a fully qualified domain name of a remote computer. To specify the
        local computer, type the computer name, a dot (.), or "localhost".

        This parameter does not rely on Windows PowerShell remoting. You can use the ComputerName parameter of
        Get-Service even if your computer is not configured to run remote commands.

        Required?                false
        Position?                named
        Default value            Local computer
        Accept pipeline input?    true (ByPropertyName)
        Accept wildcard characters? false

-- More --

```

Figure 5

This will display information about every parameter, notes and examples (see Figures 5 and 6).



```

You can pipe a service object or a service name to Get-Service.

OUTPUTS
    System.ServiceProcess.ServiceController

    Get-Service returns objects that represent the services on the computer.

NOTES
    You can also refer to Get-Service by its built-in alias, "gsv". For more information, see about_Aliases.

    Get-Service can display services only when the current user has permission to see them. If Get-Service does
    not display services, you might not have permission to see them.

    To find the service name and display name of each service on your system, type "get-service". The service
    names appear in the Name column, and the display names appear in the DisplayName column.

    When you sort in ascending order by status value, "Stopped" services appear before "Running" services. The
    Status property of a service is an enumerated value in which the names of the statuses represent integer
    values. The sort is based on the integer value, not the name. "Running" appears before "Stopped" because
    "Stopped" has a value of "1", and "Running" has a value of "4".

----- EXAMPLE 1 -----
PS C:\>get-service

This command retrieves all of the services on the system. It behaves as though you typed "get-service *". The
default display shows the status, service name, and display name of each service.

----- EXAMPLE 2 -----
-- More --

```

Figure 6

If you only want to see examples, you can run a help command like this:

```
PS C:\> help get-service -Examples
```

If you only want details on a specific parameter, you can use a command like the following, which also supports wildcards:

```
PS C:\> help get-service -Parameter req*
```

-RequiredServices [<SwitchParameter>]

Gets only the services that this service requires.

This parameter gets the value of the ServicesDependedOn property of the service.

By default, Get-Service gets all services.

Required?	false
Position?	named
Default value	False
Accept pipeline input?	false
Accept wildcard characters?	False

This is a great way to see if the parameter accepts wildcards, if it is positional or if you have to specify the parameter name.

Online

If you are in doubt as to whether you have the most up-to-date help, you can go online for many commands.

PS C:\> help get-service –online

If the command has an online link, your browser will open to the online version of command documentation (see Figure 7). In the case of Microsoft, help is often updated online before it is packaged for download.

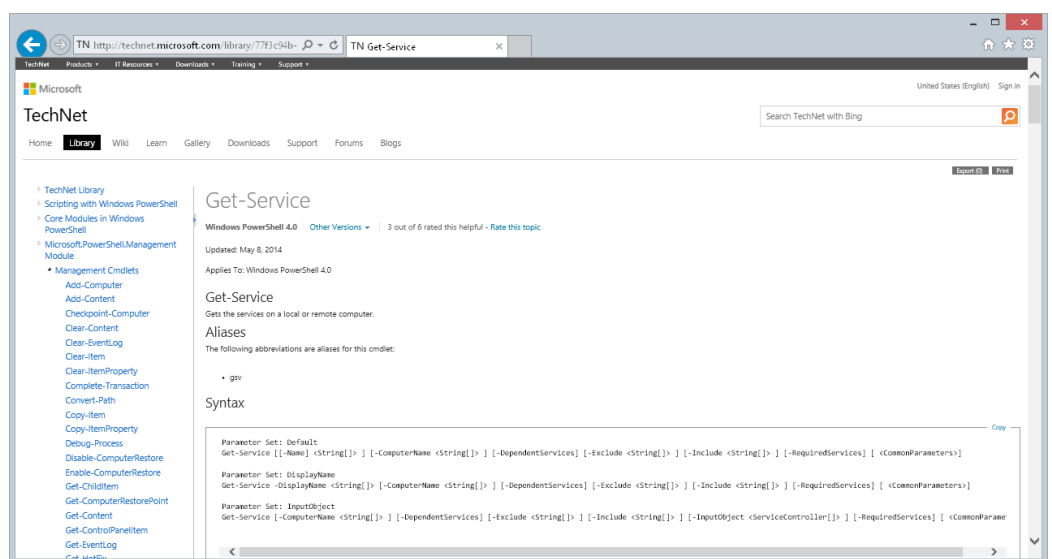


Figure 7

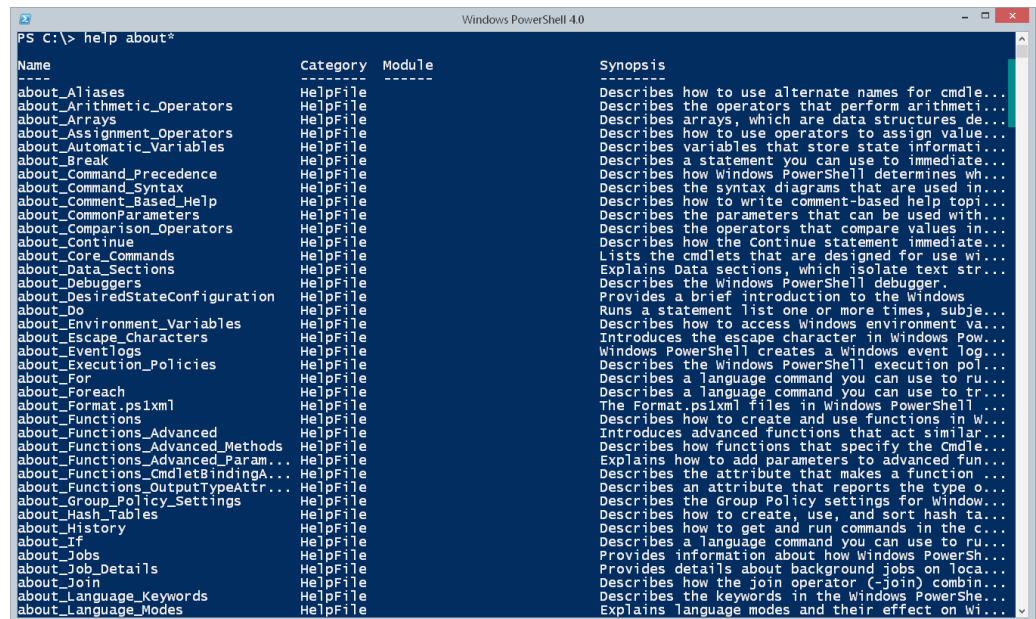
The page will display complete help and examples. There might even be community contribution at the bottom of the page.

About topics

In addition to command help, PowerShell includes a number of concept help topics, referred to as the “about” topics.

PS C:\> help about*

As you can see in Figure 8 there, are many topics.



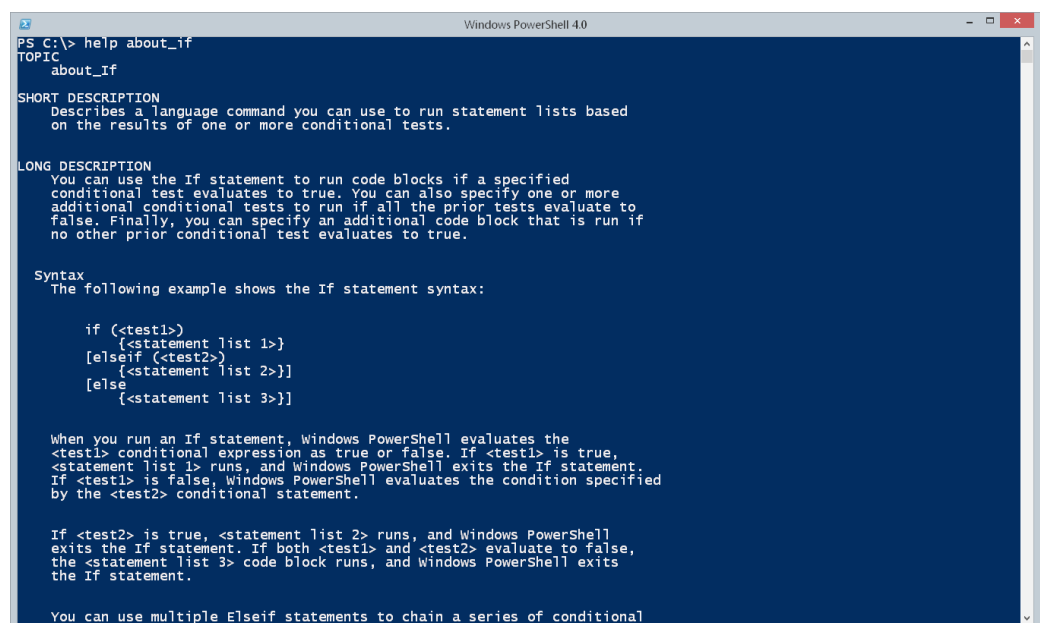
```

PS C:\> help about*
Name                Category  Module  Synopsis
-----
about_Aliases        HelpFile
about_Arithmetic_Operators HelpFile
about_Arrays         HelpFile
about_Assignment_Operators HelpFile
about_Automatic_Variables HelpFile
about_Break          HelpFile
about_Command_Precedence HelpFile
about_Command_Syntax HelpFile
about_Comment_Based_Help HelpFile
about_CommonParameters HelpFile
about_Comparison_Operators HelpFile
about_Continue       HelpFile
about_Core_Commands HelpFile
about_Data_Sections HelpFile
about_Debuggers      HelpFile
about_DesiredStateConfiguration HelpFile
about_Do              HelpFile
about_Environment_Variables HelpFile
about_Escape_Characters HelpFile
about_EventLogs      HelpFile
about_Execution_Policies HelpFile
about_For             HelpFile
about_Foreach        HelpFile
about_Format_psxml   HelpFile
about_Functions      HelpFile
about_Functions_Advanced HelpFile
about_Functions_Advanced_Methods HelpFile
about_Functions_Advanced_Parameters HelpFile
about_Functions_CmdletBindingAttributes HelpFile
about_Functions_OutputTypeAttributes HelpFile
about_Group_Policy_Settings HelpFile
about_Hash_Tables    HelpFile
about_History        HelpFile
about_If             HelpFile
about_Jobs           HelpFile
about_Job_Details    HelpFile
about_Join           HelpFile
about_Language_Keywords HelpFile
about_Language_Modes HelpFile
  
```

Figure 8

Once you know what you want, ask PowerShell to display it (see Figure 9).

PS C:\> help about_if



```

PS C:\> help about_if
TOPIC
    about_if

SHORT DESCRIPTION
    Describes a language command you can use to run statement lists based
    on the results of one or more conditional tests.

LONG DESCRIPTION
    You can use the If statement to run code blocks if a specified
    conditional test evaluates to true. You can also specify one or more
    additional conditional tests to run if all the prior tests evaluate to
    false. Finally, you can specify an additional code block that is run if
    no other prior conditional test evaluates to true.

Syntax
    The following example shows the If statement syntax:

    if (<test1>)
    {
        <statement list 1>
    }
    [elseif (<test2>)]
    {
        <statement list 2>
    }
    [else]
    {
        <statement list 3>
    }

    When you run an If statement, windows PowerShell evaluates the
    <test1> conditional expression as true or false. If <test1> is true,
    <statement list 1> runs, and windows PowerShell exits the If statement.
    If <test1> is false, windows PowerShell evaluates the condition specified
    by the <test2> conditional statement.

    If <test2> is true, <statement list 2> runs, and windows PowerShell
    exits the If statement. If both <test1> and <test2> evaluate to false,
    the <statement list 3> code block runs, and windows PowerShell exits
    the If statement.

    You can use multiple Elseif statements to chain a series of conditional
  
```

Figure 9

The only limitation is that you can't use -Online.

Terminology

Next, let's make sure you understand the different PowerShell terms you are likely to come across.

Cmdlet

This term, pronounced "command-let," is PowerShell's core unit of execution. These are compiled commands written in a .NET Framework language. Cmdlets are small and single purpose. The cmdlet names follow a standard Verb-Noun naming convention. The verb is from a list of standard .NET verbs, like Set, Remove or Get. The Noun is a singular version of the "thing" you want to work with like Service, Process or EventLog. Thus, you can probably make a pretty good educated guess about what command you would use to stop a process.

Parameter

Cmdlet behavior can be customized by the use of Parameters. The syntax when using a parameter is a dash, immediately followed by the parameter name, a space and finally the value. If the value can contain multiple entries, you generally separate them with commas. As I explained in the section on reading help, some parameters are positional, so all you need to do is to specify the value. For other parameters, you only need to type enough of the parameter name so that PowerShell knows which parameter you are referring to.

You could type a command like this:

```
PS C:\> get-service -Name adws -ComputerName chi-dc04
```

Status	Name	DisplayName
-----	----	-----
Running	adws	Active Directory Web Services

Alternatively, you can use the parameter shortcuts and save some typing.

```
PS C:\> get-service adws -c chi-dc04
```

Because Get-Service has no other parameter that starts with 'C,' it knows you mean -Computername. This practice is perfectly acceptable when running interactive commands. But when you begin scripting, use full parameter names. You only have to type it once and it will make your script easier to understand.

Alias

An alias is an alternative command name. An alias might be provided as a transition aid. For example, someone coming from a Linux background would know the *ps* command for listing processes. This same command is an alias for the `Get-Process` cmdlet. Thus a new user could use either of these commands:

```
PS C:\> get-process
```

```
PS C:\> ps
```

However, be aware that the *ps* command is NOT the same command in Linux. It is merely a shortcut to `Get-Process`. To customize the command you would need to use the parameters from `Get-Process`. Fortunately, you can ask for help via the alias. Try these commands in your PowerShell session:

```
PS :\> help ps
```

```
PS :\> help dir
```

Many aliases are built in to PowerShell. What command do you think you can use to *get* an *alias*? You can create your own aliases for any command, even non-PowerShell commands.

```
PS C:\> Set-Alias -Name np -Value Notepad.exe
```

This command created a shortcut called *np*. When typed, Notepad will start. Aliases will only exist for as long as your PowerShell session is open. The next time you open PowerShell you will have to redefine it. Or you can use a PowerShell profile script—I'll leave that for you to read.

```
PS C:\> help about_profiles
```

Function

A function is a modular, reusable block of PowerShell code. It can be written to act like a cmdlet but it is created in a PowerShell script. Functions can accept parameters, interact with the pipeline and have aliases. You will most likely be using built-in PowerShell functions without even realizing it. Eventually you will be able to create your own functions.

The fact that something is a function and not a cmdlet is really irrelevant. But if you are curious about what functions are currently loaded in your PowerShell session, run this command:

```
PS C:\> get-command -CommandType function -ListImported
```

There is a lot of information about functions in help:

```
PS C:\> help about_functions*
```

Variable

A variable is a special type of object that is a placeholder for something else. The variable itself really has no meaning until you assign it a value.

```
PS C:\> $a = 10
```

The variable name is 'a' but it is prefixed with a \$ symbol when you want to access it.

```
PS C:\> $a * 2
```

```
20
```

Variables can also hold the output of PowerShell commands.

```
PS C:\> $services = get-service
```

This variable can now be used in place of Get-Service:

```
PS C:\> $services | where-object {$_.status -eq 'running'}
```

But be aware that the variable is basically a point in time. The values in \$services are those that existed when I ran the command. There are some exceptions, especially when the variable contains a single object. However, as a PowerShell beginner you should assume that variables don't update. The help system has more details:

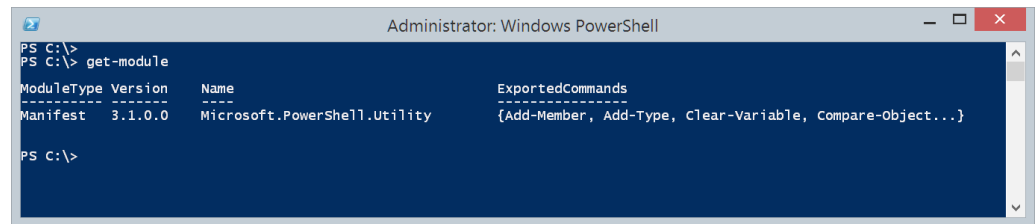
```
PS C:\> help about_variables
```

Module

A module is a package of PowerShell commands. Many modules are tied to Windows products or features. Modules can be defined in binary files but are often nothing more than a collection of PowerShell scripts and functions. You will eventually learn to develop your own modules. To see what modules are installed on your computer, use the Get-Module cmdlet.

```
PS C:\> get-module
```

Figure 10 shows a new PowerShell session with no profile script. This is the only module loaded automatically.



```

Administrator: Windows PowerShell
PS C:\>
PS C:\> get-module
ModuleType Version      Name                               ExportedCommands
-----
Manifest 3.1.0.0    Microsoft.PowerShell.Utility      {Add-Member, Add-Type, Clear-Variable, Compare-Object...}
PS C:\>

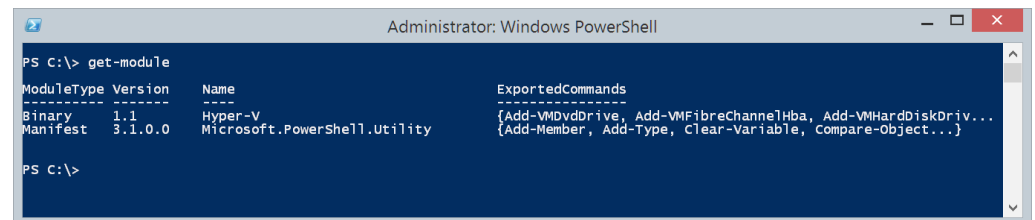
```

Figure 10

As you run commands from modules, the module itself will automatically load. For example, if you have the Hyper-V module installed, all you need to do is run a command from the module.

PS C:\> get-vm -computername chi-hvr2

Now the Get-Module command will show the new module (see Figure 11).



```

Administrator: Windows PowerShell
PS C:\> get-module
ModuleType Version      Name                               ExportedCommands
-----
Binary 1.1        Hyper-V                               {Add-VMCdvdDrive, Add-VMFibreChannelHba, Add-VMHardDiskDrive...}
Manifest 3.1.0.0    Microsoft.PowerShell.Utility      {Add-Member, Add-Type, Clear-Variable, Compare-Object...}
PS C:\>

```

Figure 11

To get a list of all of the available modules, run this command:

PS C:\> get-module -listavailable

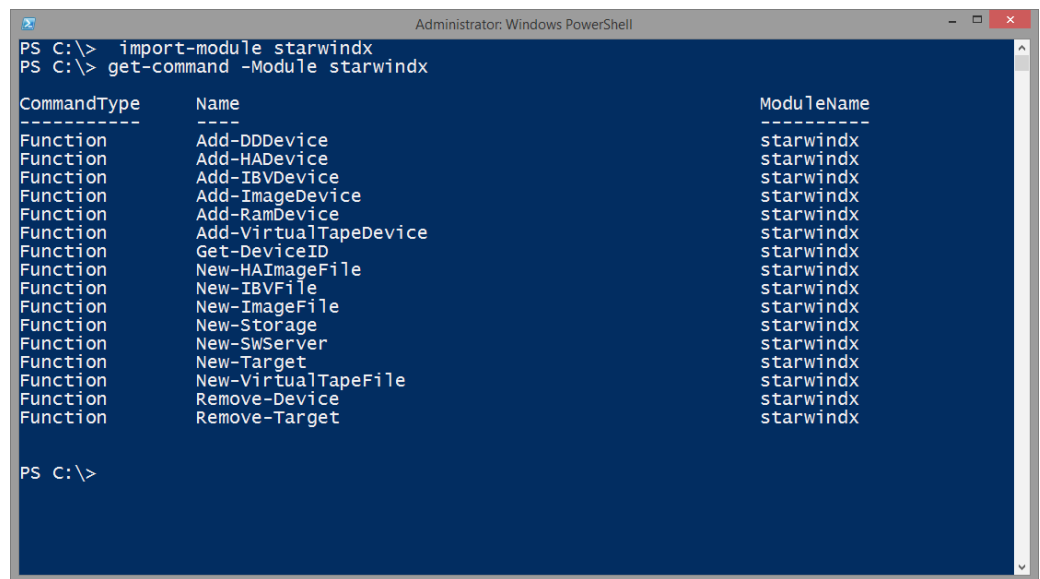
If you want, you can import the module without having to run any of its commands.

PS C:\> import-module StarWindX

Once loaded, you can use the Get-Command cmdlet to list the available commands within the module (see Figure 12).

PS C:\> get-command -module StarWindX

You can also use a wild card with the module name if you don't know the full name or don't feel like typing it.



```

Administrator: Windows PowerShell
PS C:\> import-module starwindx
PS C:\> get-command -Module starwindx

CommandType      Name                                     ModuleName
-----
Function         Add-DDDevice                           starwindx
Function         Add-HADevice                           starwindx
Function         Add-IBVDevice                          starwindx
Function         Add-ImageDevice                        starwindx
Function         Add-RamDevice                          starwindx
Function         Add-VirtualTapeDevice                  starwindx
Function         Get-DeviceID                           starwindx
Function         New-HAImageFile                        starwindx
Function         New-IBVFile                            starwindx
Function         New-ImageFile                          starwindx
Function         New-Storage                            starwindx
Function         New-SWServer                           starwindx
Function         New-Target                             starwindx
Function         New-VirtualTapeFile                   starwindx
Function         Remove-Device                          starwindx
Function         Remove-Target                          starwindx

PS C:\>

```

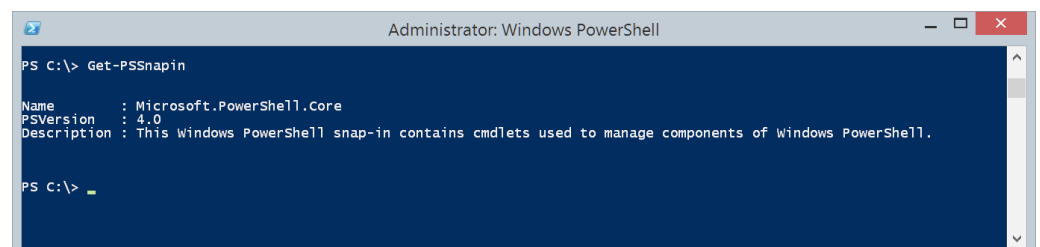
Figure 12

I can now use any of these commands.

Snapin

A snapin, often referred to as a PSSnapin, is another way of adding new commands to your PowerShell session. Snapins are compiled and must be installed on a computer before they can be used. In the early years of PowerShell this was the primary way of packaging commands, but now most things ship as modules. Unlike a module, you must add the snapin to your session before you can use any of the commands. PowerShell will auto-load the snapin like it does a module (see Figure 13).

PS C:\> get-pssnapin



```

Administrator: Windows PowerShell
PS C:\> Get-PSSnapin

Name       : Microsoft.PowerShell.Core
PSVersion : 4.0
Description: This Windows PowerShell snap-in contains cmdlets used to manage components of Windows PowerShell.

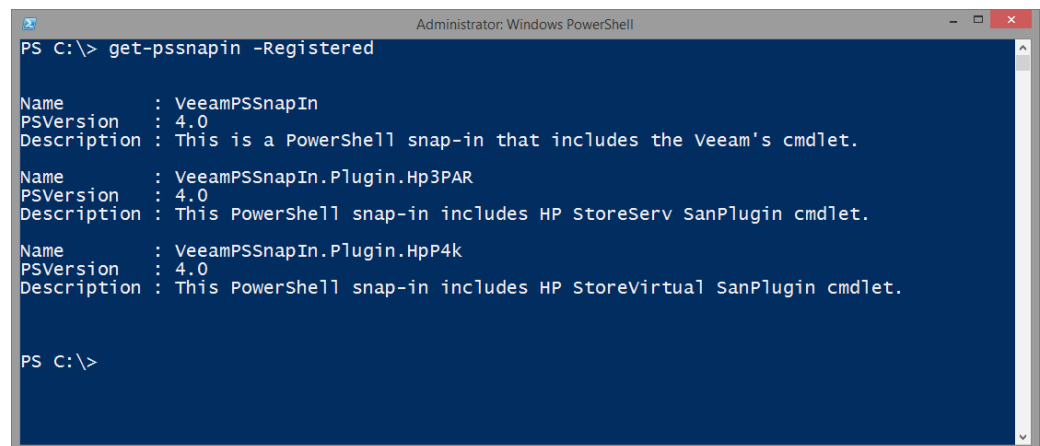
PS C:\>

```

Figure 13

Get-PSSnapin only shows currently loaded snapins. You may have other snapins registered and waiting to be loaded (see Figure 14).

PS C:\> Get-PSSnapin -Registered



```

Administrator: Windows PowerShell
PS C:\> get-pssnapin -Registered

Name       : VeeamPSSnapIn
PSVersion  : 4.0
Description : This is a PowerShell snap-in that includes the Veeam's cmdlet.

Name       : VeeamPSSnapIn.Plugin.Hp3PAR
PSVersion  : 4.0
Description : This PowerShell snap-in includes HP StoreServ SanPlugin cmdlet.

Name       : VeeamPSSnapIn.Plugin.HpP4k
PSVersion  : 4.0
Description : This PowerShell snap-in includes HP StoreVirtual SanPlugin cmdlet.

PS C:\>

```

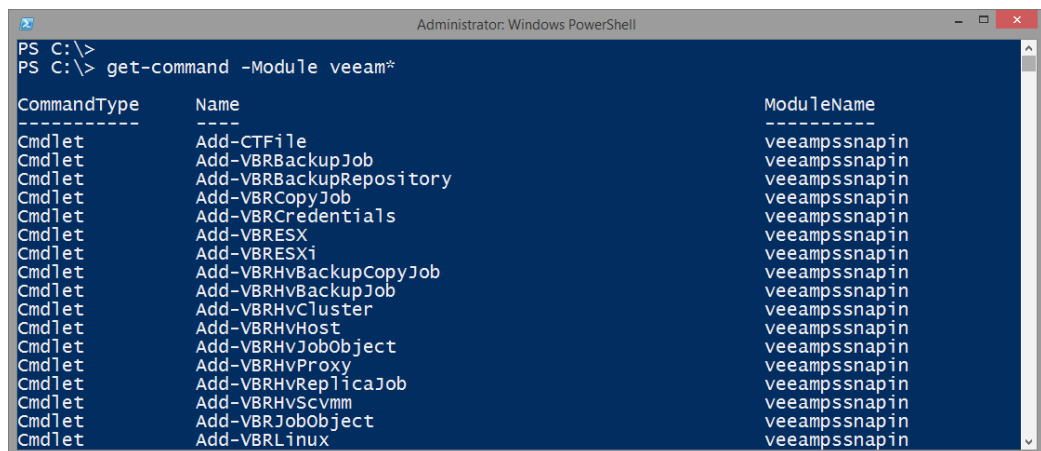
Figure 14

Once you know the snapin name, you can add it to your session.

PS C:\> add-pssnapin veeamPSSnapin

You can now use Get-Command to view the snapin commands (see Figure 15).

PS C:\> get-command -module Veeam*



```

Administrator: Windows PowerShell
PS C:\>
PS C:\> get-command -Module veeam*

CommandType      Name                                     ModuleName
-----
Cmdlet           Add-CTFile                             veeampssnapin
Cmdlet           Add-VBRBackupJob                       veeampssnapin
Cmdlet           Add-VBRBackupRepository                veeampssnapin
Cmdlet           Add-VBRCopyJob                         veeampssnapin
Cmdlet           Add-VBRCredentials                     veeampssnapin
Cmdlet           Add-VBRESX                             veeampssnapin
Cmdlet           Add-VBRESXi                            veeampssnapin
Cmdlet           Add-VBRHvBackupCopyJob                 veeampssnapin
Cmdlet           Add-VBRHvBackupJob                     veeampssnapin
Cmdlet           Add-VBRHvCluster                       veeampssnapin
Cmdlet           Add-VBRHvHost                           veeampssnapin
Cmdlet           Add-VBRHvJobObject                     veeampssnapin
Cmdlet           Add-VBRHvProxy                         veeampssnapin
Cmdlet           Add-VBRHvReplicaJob                    veeampssnapin
Cmdlet           Add-VBRHvScvmm                         veeampssnapin
Cmdlet           Add-VBRJobObject                       veeampssnapin
Cmdlet           Add-VBRLinux                           veeampssnapin

```

Figure 15

You can save yourself some typing by using wildcards. You'll notice that I used the `-Module` parameter even though the Veeam commands are packaged as a PSSnapin. Get-Command is smart enough to know what you mean. In this situation you can use `-Module` or `-PSSnapin`. Personally, I prefer to learn and use one way to accomplish something, so I always use `-Module`.

Loaded snapins remain in your PowerShell session until you close it. If you always want to have commands from a given snapin available, add a line like this to your PowerShell profile script:

Add-PSSnapin -name VeeamPSSnapin

Provider

Perhaps one of the most misunderstood PowerShell terms is *provider*, which is also referenced as *psprovider*. A provider is a piece of software that exposes data in specialized storage. The provider is responsible for translating a cmdlet into something the underlying storage system understands. Out of the box, PowerShell includes providers for the file system, the registry, the certificate store, environmental variables, as well as a few special PowerShell-only features. You can use `Get-PSProvider` to display available providers.

PS C:\> get-psprovider

Name	Capabilities	Drives
----	-----	-----
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess, Credentials	{C, D, E, F...}
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transactions	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}
Certificate	ShouldProcess	{Cert}
WSMan	Credentials	{WSMan}

Providers expose their underlying technology through a special drive mapping called a *psdrive*. What this means for you is that you can use the same PowerShell commands to navigate these different storage systems. The provider translates the cmdlet, such as `dir` (`Get-ChildItem`), into something that the storage system understands. Some PSSnapins and modules may also add providers and create PSDrives when you load them into your PowerShell session.

Many of these PSDrives are created automatically when you start PowerShell. You can view them with the `Get-PSDrive` cmdlet.

```

PS C:\> get-psdrive
Name          Used (GB)    Free (GB)    Provider      Root
-----
Alias
C             342.99      110.98      FileSystem    C:\
Cert
D             102.38      9.41        FileSystem    D:\
E             696.10      235.41      FileSystem    E:\
Env
Function
G             428.76      36.99       FileSystem    G:\
HKCU
HKLM
Variable
WSMan
Z
    
```

Figure 16

As you can see in Figure 16, you should get PSDrives for all of your regular drives, like C and D as well as specialized ones like HKLM. The great thing is that for the most part you can use the same commands to navigate them all.

```

PS C:\work> dir z*

Directory: C:\work

Mode                LastWriteTime         Length Name
----                -
-a---             3/18/2013 12:22 PM         18069 zazu.gif

PS C:\work> cd HKLM:\Software\Microsoft\Windows\Shell
PS HKLM:\Software\Microsoft\Windows\Shell> dir

Hive: HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Shell

Name                Property
----                -
Associations

PS HKLM:\Software\Microsoft\Windows\Shell> cd hkcu:
PS HKCU:\> cd .\Software
PS HKCU:\Software>
    
```

Figure 17

Note that Get-PSDrive shows the drive names without the colon (:), although you need to use the colon when navigating the drive as I do in Figure 17. The PSDrive name is HKLM but when using it I use HKLM:. You can even create new PSDrive mappings and you aren't limited to single letters (see Figure 18).

PS C:\> New-PSDrive -Name Docs -PSProvider FileSystem -Root 'c:\users\jeff\documents'

```

Administrator: Windows PowerShell
PS C:\> New-PSDrive -Name Docs -PSProvider FileSystem -Root 'C:\users\jeff\Documents'

Name           Used (GB)  Free (GB)  Provider  Root
-----
Docs           110.98    110.98    FileSystem C:\users\jeff\Documents

PS C:\> dir docs:\*.pdf

Directory: C:\users\jeff\Documents

Mode                LastWriteTime         Length Name
----                -
-a---             2/26/2013  3:18 PM      474147 FunBook-Inside.pdf
-a---             3/5/2013  1:30 PM      314393 Guru Licensing Hicks.pdf
-a---             6/5/2013  7:36 AM      501992 Hertz-pccode.pdf
-a---             4/14/2013  5:57 PM       76943 Hicks-SQLSAT197AirFare.pdf
-a---             8/26/2013  8:31 PM      920903 MS2UT_MS2UB_User_Manual.pdf
-a---             3/5/2013  1:41 PM      121551 Newegg-ECUSB3S22.pdf
-a---             8/11/2013  4:51 PM      142759 SeagateReturn.pdf

PS C:\>

```

Figure 18

This PSDrive will last for as long as my PowerShell session is running. If I always want this drive mapping, then I would put the New-PSDrive expression in my PowerShell profile script.

You can learn more about providers from PowerShell help:

PS C:\> help about_Providers

The PowerShell paradigm

So what does this all mean? If you come from a VBScript or batch scripting background, many parts of PowerShell will feel familiar, albeit with different terms. Those of you coming from the Linux or Unix world might also feel slightly comfortable at a PowerShell prompt. But it is critical that you understand the PowerShell paradigm. If you try to force PowerShell into a preconceived mold, you may become frustrated and confused and ultimately you may ignore PowerShell—and this would be a shame.

When I run PowerShell training classes or speak at conferences, I always say that PowerShell is about objects in the pipeline. It is not about parsing text. PowerShell is a management tool for Microsoft Windows platforms and since Windows is based on the concept of objects, it only makes sense that PowerShell would do the same. By contrast, operating systems like Linux are primarily text based, so those management tools excel at text manipulation. Trying to compare something like bash to PowerShell is really comparing apples and oranges.

Objects

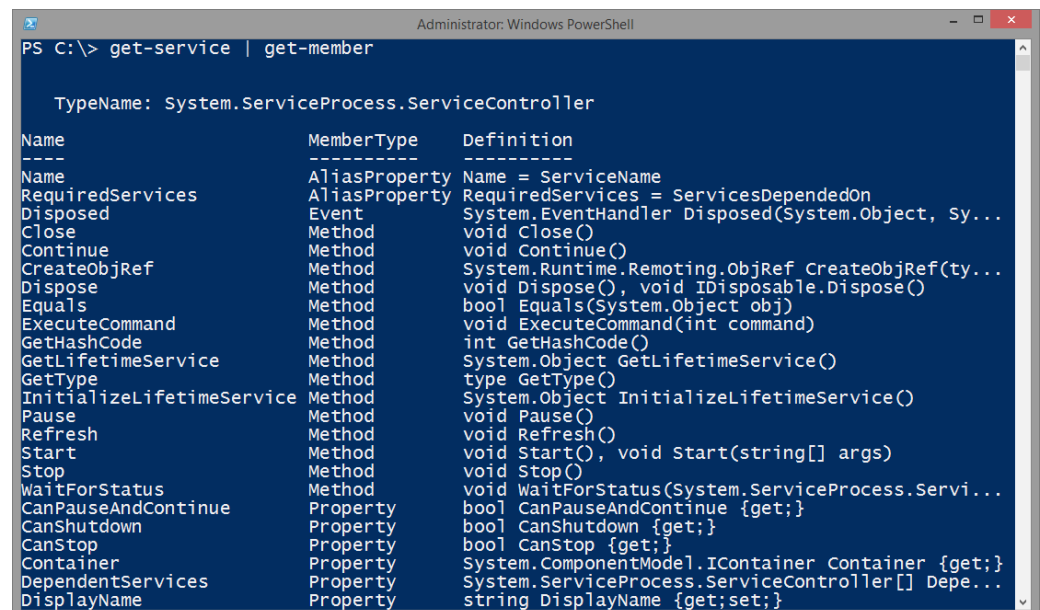
In PowerShell, everything is an object. Cmdlets and functions are designed to work with and emit objects. Of course you see text on the screen, but don't be misled. While your PowerShell command is running, objects are happening. An object is simply a software representation of something you are managing in PowerShell such as service, virtual machine or Active Directory user account. The object has properties that describe it. For example, a service has properties like Name, display name and status. Some of these properties can be modified, while others are read-only. Objects may also have methods, which are actions the object can do or that you can do to the object. Again, with services you could stop or start the service. And you might shut down a virtual machine or disable a user account.

This doesn't mean you have to be a .NET developer to use PowerShell. The whole point of cmdlets is that they do the work for you. You shouldn't have to worry about developer concepts, such as instantiating objects and invoking methods. Many of the core management tasks will have cmdlets that do all of that for you.

Get-Member

But still, it will often be useful to discover information about an object. To do that, use the Get-Member cmdlet. The easiest way to use this is to pipe a command to it.

PS C:\> get-service | get-member



```

Administrator: Windows PowerShell
PS C:\> get-service | get-member

TypeName: System.ServiceProcess.ServiceController
-----
Name           MemberType      Definition
-----
Name           AliasProperty  Name = ServiceName
RequiredServices AliasProperty  RequiredServices = ServicesDependedOn
Disposed       Event           System.EventHandler Disposed(System.Object, Sy...
Close         Method         void Close()
Continue       Method         void Continue()
CreateObjRef   Method         System.Runtime.Remoting.ObjRef CreateObjRef(ty...
Dispose        Method         void Dispose(), void IDisposable.Dispose()
Equals         Method         bool Equals(System.Object obj)
ExecuteCommand Method         void ExecuteCommand(int command)
GetHashCode    Method         int GetHashCode()
GetLifetimeService Method       System.Object GetLifetimeService()
GetType       Method         type GetType()
InitializeLifetimeService Method     System.Object InitializeLifetimeService()
Pause         Method         void Pause()
Refresh       Method         void Refresh()
Start         Method         void Start(), void Start(string[] args)
Stop         Method         void Stop()
WaitForStatus Method         void WaitForStatus(System.ServiceProcess.Servi...
CanPauseAndContinue Property      bool CanPauseAndContinue {get;}
CanShutdown  Property      bool CanShutdown {get;}
CanStop      Property      bool CanStop {get;}
Container    Property      System.ComponentModel.IContainer {get;}
DependentServices Property     System.ServiceProcess.ServiceController[] Depe...
DisplayName  Property      string DisplayName {get;set;}
  
```

Figure 19

In Figure 19 the `TypeName` is the type of object. Based on what you see, `Get-Service` uses an object with a type of `System.ServiceProcess.ServiceController`. You can see its methods and properties. PowerShell might add more methods and properties, including alias properties, to make the object easier for you to use. If you want to discover the object's native methods and properties, search for the type name that will most likely take you to an MSDN page.

It is important that you know how to use `Get-Member`, which has an alias of `gm`, because very often when you run a command, PowerShell has a default display configuration for any given object type. Don't assume that what you see in the output are the only properties, or even the property names. Use `Get-Member` to discover the actual property names because once you know the property name, you can use it—I'll demonstrate this in a bit.

The PowerShell pipeline

The concept of a pipeline is not unique to PowerShell. It exists in other shell languages. Like other languages, the vertical bar (`|`) is the pipeline character. In PowerShell this means you pass the output of one command as the input to the next command, assuming the second command knows what to do with the incoming objects. You can continue to pass objects through the pipeline, recognizing that the objects might even change. At the end of the pipeline, PowerShell will display whatever is left. You will often use expressions like this:

PS C:\> cmdletA -parameter Foo | cmdletB | cmdletC -parameter Bar

Let me walk you through the benefit of the PowerShell pipeline.

```
PS C:\> Get-Service

Status  Name                DisplayName
-----  -
Stopped AeLookupSvc        Application Experience
Stopped ALG                Application Layer Gateway Service
Stopped AppIDSvc       Application Identity
Stopped Appinfo        Application Information
Stopped Ar...          Management
Stopped a...           e Service
Running A...           Endpoint Builder
Running A...           )
Stopped A...           aller (AxInstSV)
Stopped BDESVC        BitLocker Drive Encryption Service
Running BFE           Base Filtering Engine
Running BITS          Background Intelligent Transfer Ser...
```




Figure 20

Figure 20 depicts a typical command. It might take a bit of clever text parsing of the output to get only the running services in other shells and language. But Get-Service is using objects that reflect the services on your computer, which means once you know how to use the object's properties, you can use the pipeline to filter (see Figure 21).

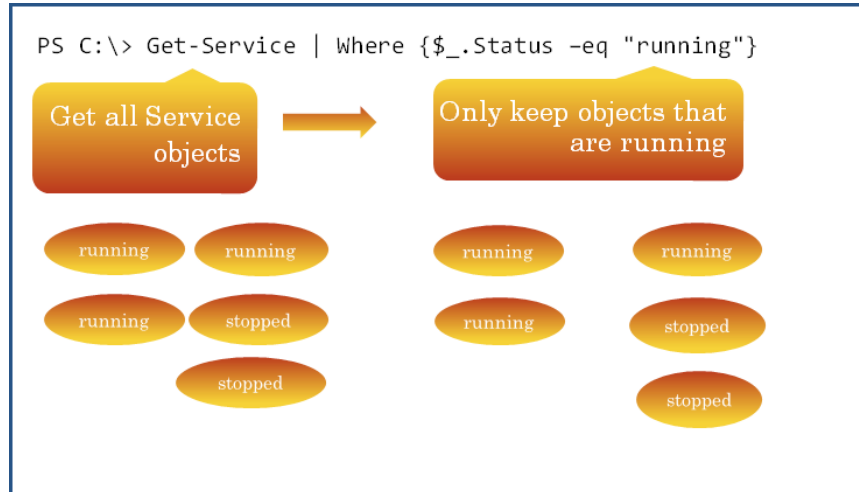


Figure 21

Get-Service will write objects to the pipeline, some with a status property of Running and some with a status of Stopped. These are piped to the Where-Object cmdlet, which evaluates the status property. If the property is equal to running, the object remains in the pipeline, otherwise it is discarded.

The end result is shown in Figure 22.

```
PS C:\> Get-Service | Where {$_.Status -eq "running"}

Status Name                DisplayName
-----
Running Appinfo              Application Information
Running AudioEndpointBu... Windows Audio Endpoint Builder
Running AudioSrv        Windows Audio
Running BFE              Base Filtering Engine
Running BITS             Background Intelligent Transfer Ser...
Running bthserv         Bluetooth Support Service
Running cmdAgent        COMODO Internet Security Helper Ser...
Running ConfigFree Gadg... ConfigFree Gadget Service
Running ConfigFree Service ConfigFree Service
Running CryptSvc        Cryptographic Services
...
```

Figure 22

By contrast, a comparable VBScript would be at least 20 lines of code and would not be formatted nearly as nicely. Not to mention that the script would most likely appear complex to someone unfamiliar with VBScript. In PowerShell, the expression is much clearer, even to someone who has never seen PowerShell before.

Bigger is not always better

That being said, don't feel that everything you do in PowerShell has to be constructed as one mammoth pipelined expression. It may be convenient to type something like this as one long pipelined expression:

```
PS C:\> dir c:\work -Recurse -File | Group-Object -Property Extension |
Select-Object -property Count,Name,@{Name="Size";Expression={ ($_.
group | Measure-Object -property length -sum).sum}} | Sort-Object -prop-
erty Size -Descending | Select-Object -first 5 | Format-Table -AutoSize
```

Count	Name	Size
28	.xml	80144355
2	.msi	6817280
4	.exe	1603382
17	.ps1	932675
9	.zip	712210

In a nutshell, this one-line command is used to display the top five file extensions under C:\work, formatted as a nice table. But that is a lot to type and as a beginner you are probably going to make a mistake. Instead it might be easier to break this mammoth pipelined expression down into smaller pieces using variables.

First, get all of the files in C:\work:

```
PS C:\> $files = dir c:\work -Recurse -File
```

Group the files by the extension property:

```
PS C:\> $grouped = $files | group extension
```

Select the necessary properties from the grouped objects, including a calculated property that shows the total size of each extension group:

```
PS C:\> $results = $grouped | select Count,Name,@
{Name="Size";Expression={ ($_.group | measure length -sum).sum}}
```

Finally, sort the results, select the first five objects and format as a table:

```
PS C:\> $results | sort size -desc | select -first 5 | ft -auto
```

You are still using the pipeline at each step. But now you can verify that each step works and build on each step. Let me also emphasize that you aren't really scripting here. You are working with file objects interactively from a PowerShell prompt. And although you may not fully understand every part of the commands, I'm betting you have a pretty good idea about what I am doing at each step, even though I am taking advantage of aliases, positional parameters and parameter shortcuts.

A Veeam example

Everything I've discussed so far applies to everything in PowerShell, not merely the out-of-the-box experience. The cmdlets I used in the example above, like Measure-Object and Select-Object, behave the same whether you are working with a file object or something else. As a demonstration, let's look at the Veeam PowerShell snapin from Veeam Backup & Replication™.

First, the commands need to be loaded into PowerShell. The Veeam cmdlets are packaged as a snapin.

PS C:\> add-pssnapin VeeamPSSnapin

Earlier I showed you how to list the commands in the snapin. In this example, I want to use PowerShell to get some information about my backups. I discover there is a cmdlet called Get-VBRBackup, so I ask PowerShell for help (see Figure 23).

PS C:\> help get-vbrbackup -showwindow

I'm using the PowerShell v4 help feature to display help in a pop-up window.

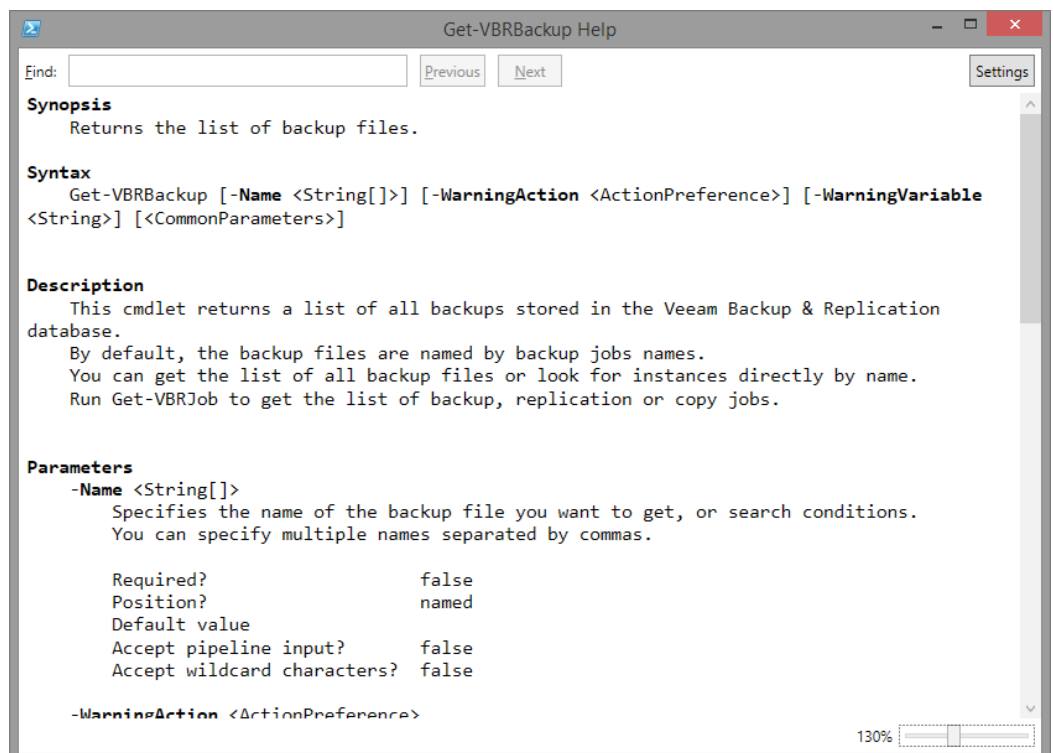


Figure 23

Scrolling down to examples, I can see how to use this command. I want to get all the backups and since I want to step through the process, I'll save the results of the command to a variable:

```
PS C:\> $vback = Get-VBRBackup
```

The variable, vback, is an array of some type of object and I can tell that there are 11 objects.

```
PS C:\> $vbackup.count
```

Or I could use the Measure-Object command, which will always give me a count:

```
PS C:\> $vback | measure-object
```

```
Count : 11
```

```
Average :
```

```
Sum :
```

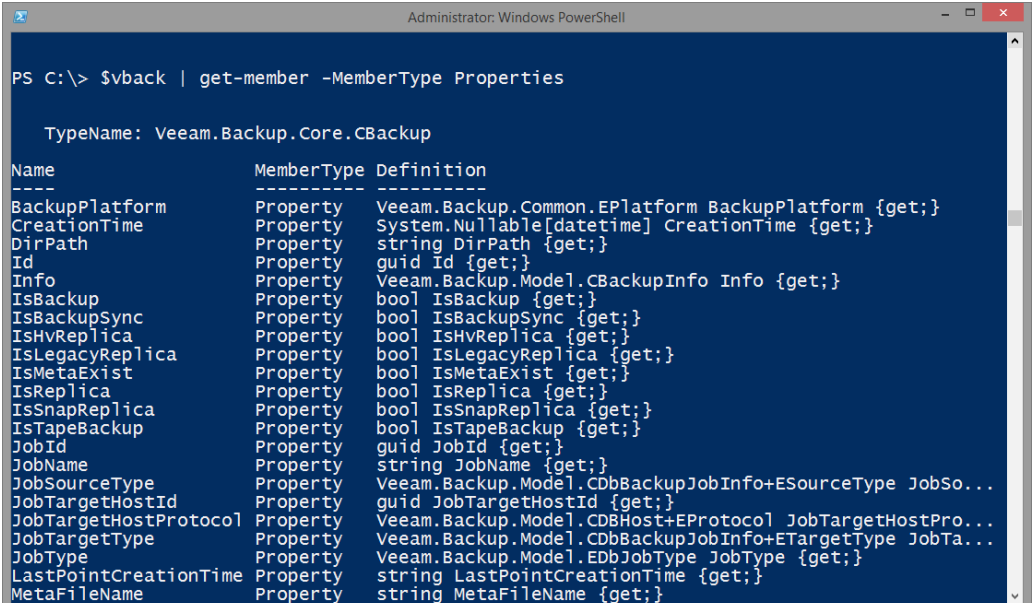
```
Maximum :
```

```
Minimum :
```

```
Property :
```

Since I know nothing about this type of object, I'm going to pipe the backup objects to Get-Member and look at only their properties (see Figure 24).

```
PS C:\> $vback | get-member -membertype properties
```



```
Administrator: Windows PowerShell

PS C:\> $vback | get-member -MemberType Properties

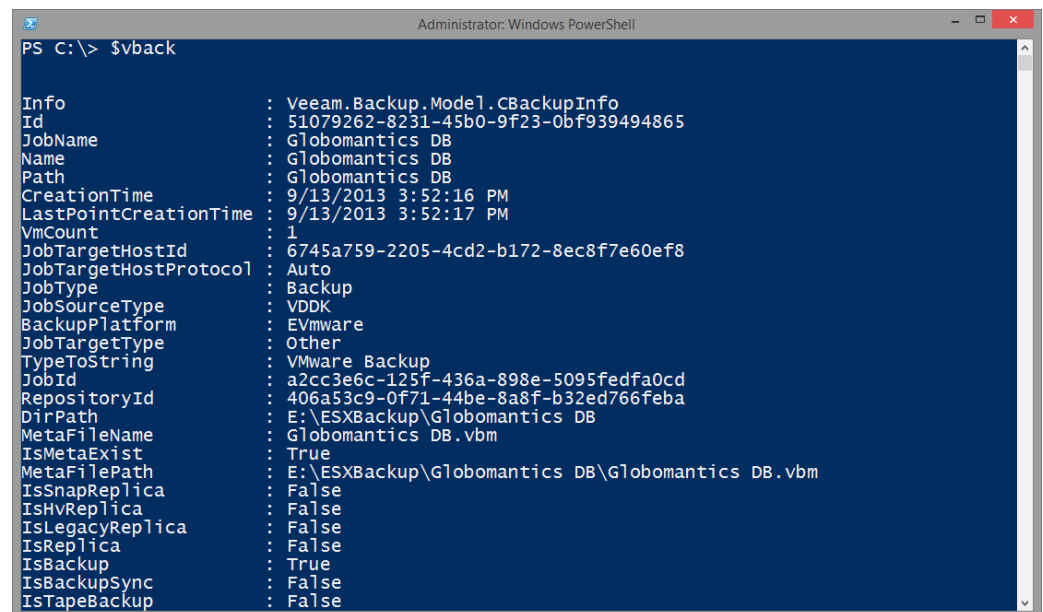
TypeName: Veeam.Backup.Core.CBackup

Name           MemberType Definition
-----
BackupPlatform Property  Veeam.Backup.Common.EPlatform BackupPlatform {get;}
CreationTime   Property  System.Nullable[datetime] CreationTime {get;}
DirPath        Property  string DirPath {get;}
Id             Property  guid Id {get;}
Info           Property  Veeam.Backup.Model.CBackupInfo Info {get;}
IsBackup       Property  bool IsBackup {get;}
IsBackupSync   Property  bool IsBackupSync {get;}
IsHvReplica    Property  bool IsHvReplica {get;}
IsLegacyReplica Property  bool IsLegacyReplica {get;}
IsMetaExist    Property  bool IsMetaExist {get;}
IsReplica      Property  bool IsReplica {get;}
IsSnapReplica  Property  bool IsSnapReplica {get;}
IsTapeBackup   Property  bool IsTapeBackup {get;}
JobId          Property  guid JobId {get;}
JobName        Property  string JobName {get;}
JobSourceType  Property  Veeam.Backup.Model.CDbBackupJobInfo+ESourceType JobSo...
JobTargetHostId Property  guid JobTargetHostId {get;}
JobTargetHostProtocol Property  Veeam.Backup.Model.CDbHost+EProtocol JobTargetHostPro...
JobTargetType  Property  Veeam.Backup.Model.CDbBackupJobInfo+ETargetType JobTa...
JobType        Property  Veeam.Backup.Model.EDbJobType JobType {get;}
LastPointCreationTime Property  string LastPointCreationTime {get;}
MetaFileName   Property  string MetaFileName {get;}
```

Figure 24

To see what is in the variable, I can simply type the variable name and PowerShell will write the contents to the pipeline (see Figure 25).

PS C:\> \$vback



```

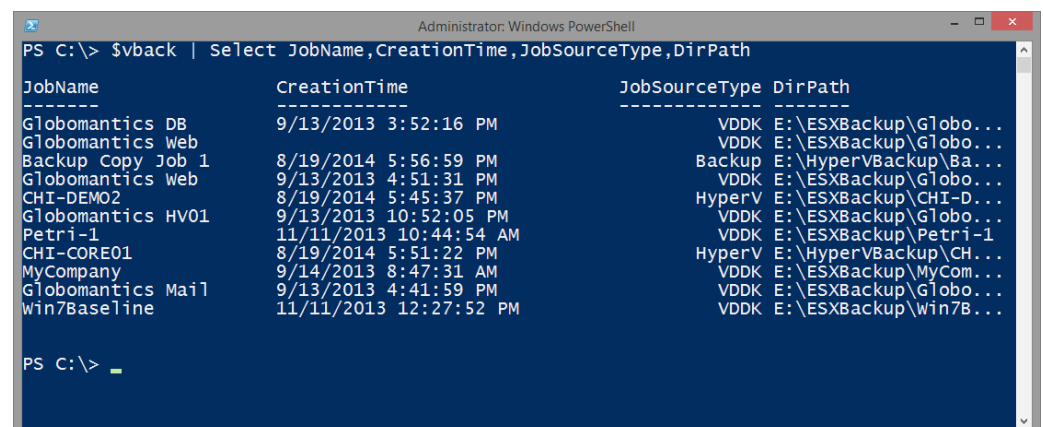
Administrator: Windows PowerShell
PS C:\> $vback

Info           : Veeam.Backup.Model.CBackupInfo
Id             : 51079262-8231-45b0-9f23-0bf939494865
JobName        : Globomantics DB
Name           : Globomantics DB
Path           : Globomantics DB
CreationTime   : 9/13/2013 3:52:16 PM
LastPointCreationTime : 9/13/2013 3:52:17 PM
VmCount        : 1
JobTargetHostId : 6745a759-2205-4cd2-b172-8ec8f7e60ef8
JobTargetHostProtocol : Auto
JobType        : Backup
JobSourceType   : VDDK
BackupPlatform : EVmware
JobTargetType   : Other
TypeToString    : VMware Backup
JobId           : a2cc3e6c-125f-436a-898e-5095fedfa0cd
RepositoryId    : 406a53c9-0f71-44be-8a8f-b32ed766feba
DirPath         : E:\ESXBackup\Globomantics DB
MetaFileName    : Globomantics DB.vbm
IsMetaExist     : True
MetaFilePath    : E:\ESXBackup\Globomantics DB\Globomantics DB.vbm
IsSnapReplica   : False
IsHvReplica     : False
IsLegacyReplica : False
IsReplica       : False
IsBackup        : True
IsBackupSync    : False
IsTapeBackup    : False
  
```

Figure 25

In this particular instance, PowerShell doesn't have any rules for this type of object so it displays all of the properties. This is helpful because now I can use Select-Object and select only the properties I am interested in (see Figure 26).

PS C:\> \$vback | select Jobname,CreationTime,JobSourceType,DirPath



```

Administrator: Windows PowerShell
PS C:\> $vback | select Jobname,CreationTime,JobSourceType,DirPath

JobName          CreationTime          JobSourceType  DirPath
-----
Globomantics DB   9/13/2013 3:52:16 PM VDDK           E:\ESXBackup\Globo...
Globomantics Web  8/19/2014 5:56:59 PM VDDK           E:\ESXBackup\Globo...
Backup Copy Job 1 8/19/2014 4:51:31 PM Backup        E:\HyperVBackup\Ba...
Globomantics Web  9/13/2013 4:51:31 PM VDDK           E:\ESXBackup\Globo...
CHI-DEMO2        8/19/2014 5:45:37 PM HyperV        E:\ESXBackup\CHI-D...
Globomantics HV01 9/13/2013 10:52:05 PM VDDK           E:\ESXBackup\Globo...
Petri-1          11/11/2013 10:44:54 AM VDDK           E:\ESXBackup\Petri-1
CHI-CORE01       8/19/2014 5:51:22 PM HyperV        E:\HyperVBackup\CH...
MyCompany        9/14/2013 8:47:31 AM VDDK           E:\ESXBackup\MyCom...
Globomantics Mail 9/13/2013 4:41:59 PM VDDK           E:\ESXBackup\Globo...
win7Baseline     11/11/2013 12:27:52 PM VDDK           E:\ESXBackup\win7B...

PS C:\> _
  
```

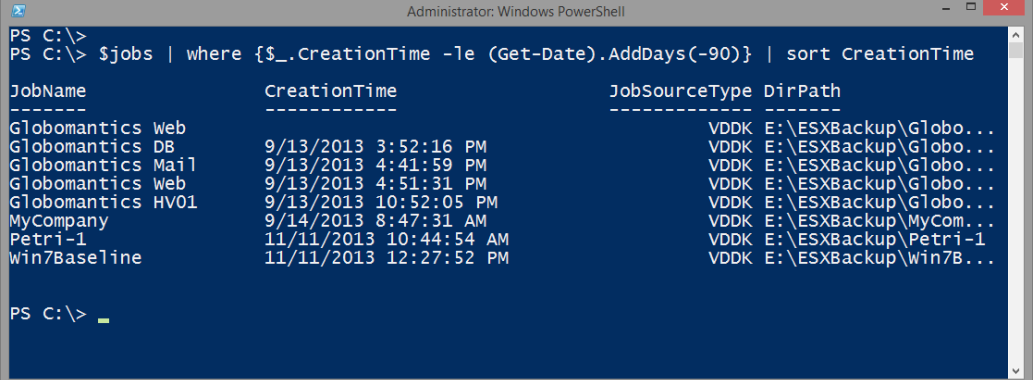
Figure 26

That's a good start. For now, I'm going to ignore the fact that I have a backup job with no creation time. My Veeam infrastructure has been set up and torn down several times in my lab, so I have probably introduced an artifact somewhere in the process. Let me rerun this command and save the results to a variable.

PS C:\> \$jobs = \$vback | select Jobname,CreationTime,JobSourceType,DirPath

Now I want to fine tune this. Perhaps I want to only see jobs older than 90 days in descending order (see Figure 27).

PS C:\> \$jobs | where {\$_.CreationTime -le (Get-Date).AddDays(-90)} | sort CreationTime



```

Administrator: Windows PowerShell
PS C:\>
PS C:\> $jobs | where {$_.CreationTime -le (Get-Date).AddDays(-90)} | sort CreationTime
-----
JobName                CreationTime                JobSourceType  DirPath
-----
Globomantics Web          9/13/2013 3:52:16 PM      VDDK           E:\ESXBackup\Globo...
Globomantics DB          9/13/2013 4:41:59 PM      VDDK           E:\ESXBackup\Globo...
Globomantics Mail        9/13/2013 4:51:31 PM      VDDK           E:\ESXBackup\Globo...
Globomantics Web          9/13/2013 10:52:05 PM     VDDK           E:\ESXBackup\Globo...
Globomantics HV01        9/14/2013 8:47:31 AM       VDDK           E:\ESXBackup\MyCom...
MyCompany                11/11/2013 10:44:54 AM    VDDK           E:\ESXBackup\Petri-1
Petri-1                  11/11/2013 12:27:52 PM    VDDK           E:\ESXBackup\Win7B...
Win7Baseline
PS C:\>

```

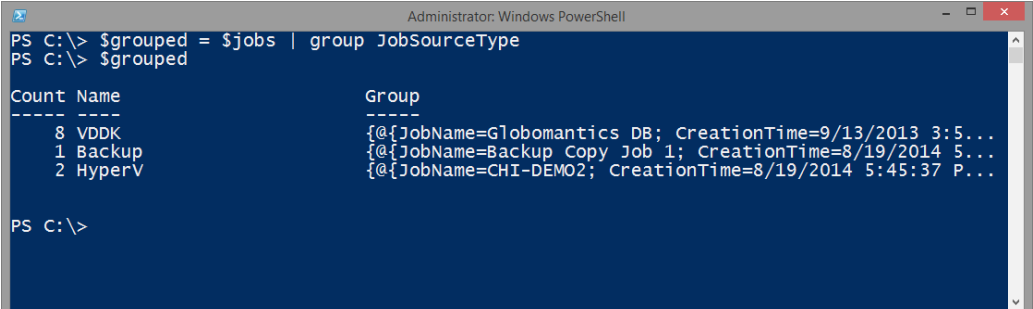
Figure 27

Perhaps my manager needs to see this. I can easily send the results to a text file.

PS C:\> \$jobs | where {\$_.CreationTime -le (Get-Date).AddDays(-90)} | sort CreationTime | out-file \\server01\veeam\oldbackups.txt

Because I have a mixed environment, it might be helpful to group the jobs based on their type (see Figure 28).

PS C:\> \$grouped = \$jobs | group JobSourceType



```

Administrator: Windows PowerShell
PS C:\> $grouped = $jobs | group JobSourceType
PS C:\> $grouped
-----
Count Name                Group
-----
      8 VDDK                {@{JobName=Globomantics DB; CreationTime=9/13/2013 3:5...
      1 Backup           {@{JobName=Backup Copy Job 1; CreationTime=8/19/2014 5...
      2 HyperV           {@{JobName=CHI-DEMO2; CreationTime=8/19/2014 5:45:37 P...
PS C:\>

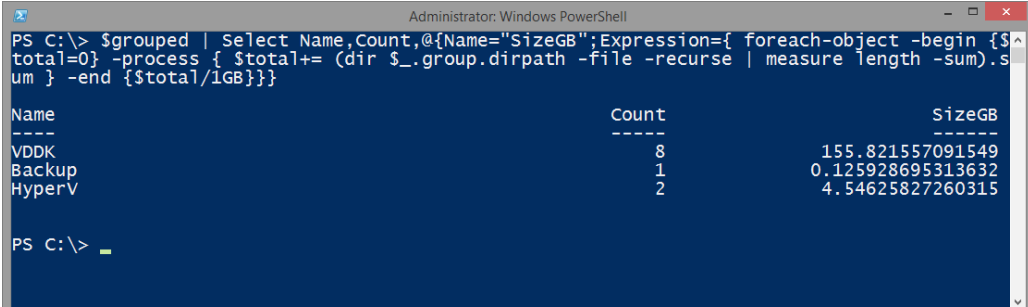
```

Figure 28

I now have a new type of object in \$grouped. The Group property is the collection of each job type. Because each object has a path, I could use the DIR command and Measure-Object to see how much space each backup is consuming on disk.

```
PS C:\> $grouped | Select Name,Count,@{Name="SizeGB";Expression={
foreach-object -begin {$total=0} -process { $total+= (dir $_.group.dirpath
-file -recurse | measure length -sum).sum } -end {$total/1GB}}
```

This is a little complicated, but the I am using ForEach-Object to get a directory size for each backup object in the group, totaling the sizes and then dividing by 1GB to get a value in GB (see Figure 29).



```
Administrator: Windows PowerShell
PS C:\> $grouped | Select Name,Count,@{Name="SizeGB";Expression={ foreach-object -begin {$
total=0} -process { $total+= (dir $_.group.dirpath -file -recurse | measure length -sum).s
um } -end {$total/1GB}}
```

Name	Count	SizeGB
VDDK	8	155.821557091549
Backup	1	0.125928695313632
HyperV	2	4.54625827260315

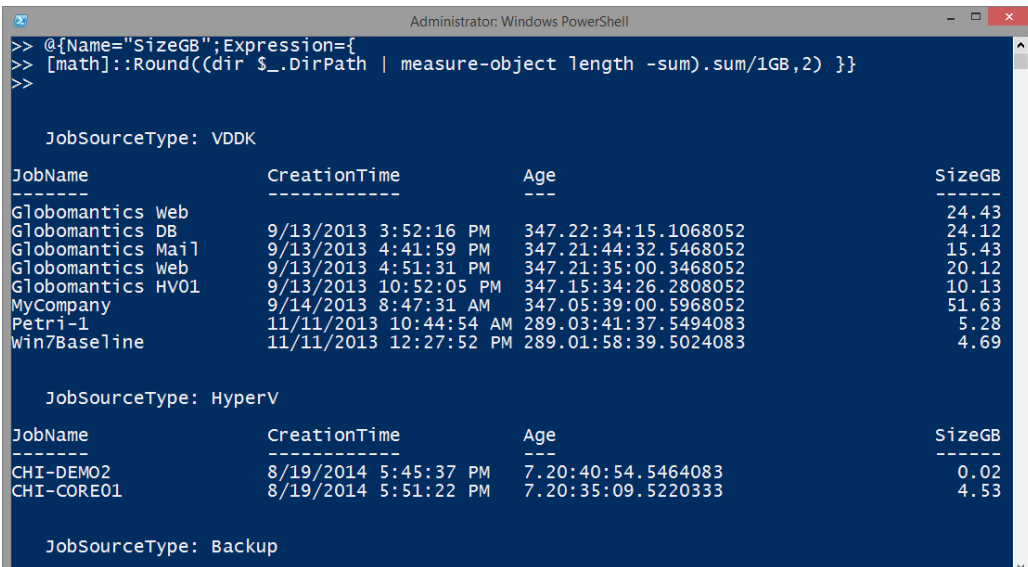
```
PS C:\> _
```

Figure 29

Another option is to create a better formatted report, grouping the results by job type (see Figure 30).

```
PS C:\> $jobs | Sort JobSourceType,CreationTime | Format-Ta-
ble -GroupBy JobSourceType -Property JobName,CreationTime,@
{Name="Age";Expression={{Get-Date} - $_.Creationtime}},@{Name="Siz
eGB";Expression={[math]::Round((dir $_.DirPath | measure-object length
-sum).sum/1GB,2) }}
```

I'll admit this might appear a bit daunting, but with time and experience it will become second nature.



```
Administrator: Windows PowerShell
>> @{Name="SizeGB";Expression={
>> [math]::Round((dir $_.DirPath | measure-object length -sum).sum/1GB,2) }}
>>
```

JobSourceType: VDDK			
JobName	CreationTime	Age	SizeGB
Globomantics Web			24.43
Globomantics DB	9/13/2013 3:52:16 PM	347.22:34:15.1068052	24.12
Globomantics Mail	9/13/2013 4:41:59 PM	347.21:44:32.5468052	15.43
Globomantics Web	9/13/2013 4:51:31 PM	347.21:35:00.3468052	20.12
Globomantics HV01	9/13/2013 10:52:05 PM	347.15:34:26.2808052	10.13
MyCompany	9/14/2013 8:47:31 AM	347.05:39:00.5968052	51.63
Petri-1	11/11/2013 10:44:54 AM	289.03:41:37.5494083	5.28
win7Baseline	11/11/2013 12:27:52 PM	289.01:58:39.5024083	4.69

JobSourceType: HyperV			
JobName	CreationTime	Age	SizeGB
CHI-DEMO2	8/19/2014 5:45:37 PM	7.20:40:54.5464083	0.02
CHI-CORE01	8/19/2014 5:51:22 PM	7.20:35:09.5220333	4.53

JobSourceType: Backup			
-----------------------	--	--	--

Figure 30

One important point about using the Format commands is that they must be the last step in your PowerShell expression. You can't do anything after a format command other than pipe it, using Out-File or Out-Printer. All of the examples I've shown you are not especially difficult by themselves. You should look at the full help and examples for every command I've used.

Next steps

As I mentioned at the beginning, it is impossible to learn PowerShell in 30 pages. The most important step is to learn how to use the help system in PowerShell. Once you know that, you can learn about all of the cmdlets and concepts I've discussed and with time, even the most complicated example will be easy to understand.

Learning PowerShell is like any other language. It has its own syntax, language, grammar and conventions. It takes time and practice to become fluent. To that end, you need to use PowerShell every day, even if you simply read the help for a random cmdlet and an "about" topic. The more you use PowerShell to solve your daily management tasks, the more proficient you will become and your learning will accelerate. However, be careful. Don't try to knock out an incredibly complex PowerShell one-liner or script on your first day. Start simple. I can't stress that enough.

To that end, you will most likely want additional tools and resources for learning PowerShell.

Further reading and resources

If you are intent on learning PowerShell from the ground up, then I recommend the book I co-wrote with fellow PowerShell MVP and industry expert, Don Jones: *Learn Windows PowerShell in a Month of Lunches*. That title should have a long shelf life. But since so many other training options come and go, I maintain a page on my blog with a list of essential resources, including books and training videos. Take a look at <http://jdhitsolutions.com/blog/essential-powershell-resources/>.

Finally, perhaps the best PowerShell resource for you is PowerShell.org. This site is run by a nonprofit, community-driven organization. The organization runs the Scripting Games and the PowerShell Summit. On this site you will also find an extremely active set of forums, free eBooks and all the PowerShell resources you will need to be successful. But once you've achieved some degree of fluency, I think you will wonder why you waited so long to learn and use PowerShell.

About the Author



Jeffery Hicks is a Microsoft MVP in Windows PowerShell, Microsoft Certified Professional and an IT veteran with almost 25 years of experience, much of it spent as an IT consultant specializing in Microsoft server technologies with an emphasis in automation and efficiency. He works today as an independent author, trainer and consultant. Jeff writes the popular Prof. PowerShell column for MPCMag.com, is a regular contributor to the Petri IT Knowledgebase, 4SysOps and the Altaro Hyper-V blog, as well as frequent speaker at technology conferences and user groups. If he isn't writing, Jeff is most likely recording training videos for companies like Pluralsight or hanging out in the forums at PowerShell.org.

Jeff's latest books are Learn PowerShell 3 in a Month of Lunches, Learn PowerShell Toolmaking in a Month of Lunches and PowerShell in Depth: An Administrators Guide.

You can keep up with Jeff at his blog <http://jdhitsolutions.com/blog> , on Twitter at twitter.com/jeffhicks and on Google Plus (<http://gplus.to/JeffHicks>)

Specialties: Improving management efficiency for IT Pros using automation tools like WIndows PowerShell. Experienced trainer and consultant for the IT Pro community.

About Veeam Software

Veeam® enables the Always-On Business™ by providing solutions that deliver Availability for the Modern Data Center™, which provides recovery time and point objectives (RTPO™) of less than 15 minutes for the majority of applications and data. Veeam recognizes the challenges in keeping a business up and running at all times and addresses them with solutions that provide high-speed recovery, data loss avoidance, verified protection, risk mitigation and complete visibility. [Veeam Backup & Replication™](#) leverages technologies that enable the modern data center, including VMware vSphere, Microsoft Hyper-V, NetApp storage, and HP 3PAR StoreServ and StoreVirtual Storage, to help organizations meet RTPOs, save time, mitigate risks, and dramatically reduce capital and operational costs. Veeam Availability Suite™ provides all of the benefits and features of Veeam Backup & Replication along with advanced monitoring, reporting and capacity planning for the backup infrastructure. [Veeam Management Pack™](#) extends Microsoft System Center monitoring to enterprise vSphere environments and also offers monitoring, reporting and capacity planning for the Veeam Backup & Replication infrastructure. The [Veeam Cloud Provider Program \(VCP\)](#) offers flexible monthly and perpetual licensing to meet the needs of hosting, managed service and cloud service providers. VCP currently has more than 4,500 service provider partners worldwide. Monthly rental is available in more than 70 countries from more than 50 Veeam aggregators.

Founded in 2006, Veeam currently has 25,000 ProPartners and more than 115,000 customers worldwide. Veeam's global headquarters are located in Baar, Switzerland, and the company has offices throughout the world.



AVAILABILITY™

for the Modern Data Center



To learn more, visit <http://www.veeam.com/backup>