

// Step by step

Windows PowerShell

Third Edition



Ed Wilson

Windows PowerShell Step by Step, Third Edition

Ed Wilson

PUBLISHED BY
Microsoft Press
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2015 by Ed Wilson

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2014922916
ISBN: 978-0-7356-7511-7

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at mspinput@microsoft.com. Please tell us what you think of this book at <http://aka.ms/tellpress>.

This book is provided “as-is” and expresses the author’s views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at www.microsoft.com on the “Trademarks” webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Acquisitions and Developmental Editor: Karen Szall

Project Editor: Rosemary Caperton

Editorial Production: Online Training Solutions, Inc. (OTSI)

Technical Reviewer: Brian Wilhite; Technical Review services provided by Content Master, a member of CM Group, Ltd.

Copyeditor: Kathy Krause (OTSI)

Indexer: Susie Carr (OTSI)

Cover: Twist Creative • Seattle

To Teresa: you make life an adventure.

—ED WILSON

Contents at a glance

	<i>Introduction</i>	<i>xix</i>
CHAPTER 1	Overview of Windows PowerShell 5.0	1
CHAPTER 2	Using Windows PowerShell cmdlets	23
CHAPTER 3	Understanding and using Windows PowerShell providers	65
CHAPTER 4	Using Windows PowerShell remoting and jobs	109
CHAPTER 5	Using Windows PowerShell scripts	137
CHAPTER 6	Working with functions	179
CHAPTER 7	Creating advanced functions and modules	217
CHAPTER 8	Using the Windows PowerShell ISE	259
CHAPTER 9	Working with Windows PowerShell profiles	275
CHAPTER 10	Using WMI	291
CHAPTER 11	Querying WMI	313
CHAPTER 12	Remoting WMI	341
CHAPTER 13	Calling WMI methods on WMI classes	361
CHAPTER 14	Using the CIM cmdlets	375
CHAPTER 15	Working with Active Directory	395
CHAPTER 16	Working with the AD DS module	431
CHAPTER 17	Deploying Active Directory by using Windows PowerShell	459
CHAPTER 18	Debugging scripts	473
CHAPTER 19	Handling errors	511
CHAPTER 20	Using the Windows PowerShell workflow	547
CHAPTER 21	Managing Windows PowerShell DSC	565
CHAPTER 22	Using the PowerShell Gallery	581
	<i>Appendix A: Windows PowerShell scripting best practices</i>	<i>591</i>
	<i>Appendix B: Regular expressions quick reference</i>	<i>599</i>
	<i>Index</i>	<i>603</i>

Contents

<i>Introduction</i>	<i>xix</i>
Chapter 1 Overview of Windows PowerShell 5.0	1
Understanding Windows PowerShell	1
Using cmdlets	3
Installing Windows PowerShell	3
Deploying Windows PowerShell to down-level operating systems . .	3
Using command-line utilities	4
Security issues with Windows PowerShell	6
Controlling execution of Windows PowerShell cmdlets	6
Confirming actions.	7
Suspending confirmation of cmdlets	8
Working with Windows PowerShell.	10
Accessing Windows PowerShell.	10
Configuring the Windows PowerShell console.	11
Supplying options for cmdlets	11
Working with the help options.	12
Exploring commands: Step-by-step exercises	19
Chapter 1 quick reference	22
Chapter 2 Using Windows PowerShell cmdlets	23
Understanding the basics of cmdlets	23
Using the <i>Get-ChildItem</i> cmdlet.	24
Obtaining a directory listing	24
Formatting a directory listing by using the <i>Format-List</i> cmdlet . .	26

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can improve our books and learning resources for you. To participate in a brief survey, please visit:
<http://aka.ms/tellpress>

Using the <i>Format-Wide</i> cmdlet	27
Formatting a directory listing by using <i>Format-Table</i>	29
Formatting output with <i>Out-GridView</i>	31
Taking advantage of the power of <i>Get-Command</i>	36
Searching for cmdlets by using wildcard characters	36
Using the <i>Get-Member</i> cmdlet	44
Using the <i>Get-Member</i> cmdlet to examine properties and methods	45
Using the <i>New-Object</i> cmdlet	50
Creating and using the <i>wshShell</i> object	50
Using the <i>Show-Command</i> cmdlet	52
Windows PowerShell cmdlet naming helps you learn	54
Windows PowerShell verb grouping	55
Windows PowerShell verb distribution	55
Creating a Windows PowerShell profile	57
Working with cmdlets: Step-by-step exercises	59
Chapter 2 quick reference	63

Chapter 3 Understanding and using Windows PowerShell providers

65

Understanding Windows PowerShell providers	65
Understanding the alias provider	66
Understanding the certificate provider	69
Understanding the environment provider	76
Understanding the filesystem provider	80
Understanding the function provider	85
Using the registry provider to manage the Windows registry	87
The two registry drives	88
The short way to create a new registry key	95
Dealing with a missing registry property	98
Understanding the variable provider	99
Exploring Windows PowerShell providers: Step-by-step exercises	103
Chapter 3 quick reference	107

Chapter 4 Using Windows PowerShell remoting and jobs 109

Understanding Windows PowerShell remoting	109
Classic remoting	109
WinRM	114
Using Windows PowerShell jobs	122
Using Windows PowerShell remoting and jobs: Step-by-step exercises	132
Chapter 4 quick reference	135

Chapter 5 Using Windows PowerShell scripts 137

Why write Windows PowerShell scripts?	137
The fundamentals of scripting	139
Running Windows PowerShell scripts.	139
Turning on Windows PowerShell scripting support.	140
Transitioning from command line to script.	143
Manually running Windows PowerShell scripts	145
Understanding variables and constants.	148
Using the <i>While</i> statement	154
Constructing the <i>While</i> statement in Windows PowerShell.	154
A practical example of using the <i>While</i> statement.	156
Using special features of Windows PowerShell.	157
Using the <i>Do...While</i> statement	157
Using the range operator	158
Operating over an array	158
Casting to ASCII values	159
Using the <i>Do...Until</i> statement	160
Comparing the Windows PowerShell <i>Do...Until</i> statement with VBScript.	160
Using the Windows PowerShell <i>Do</i> statement	161
The <i>For</i> statement.	162
Using the <i>For</i> statement	163
Using the <i>Foreach</i> statement	164
Exiting the <i>Foreach</i> statement early	166

Using the <i>If</i> statement	168
Using assignment and comparison operators	169
Evaluating multiple conditions	170
The <i>Switch</i> statement	171
Using the <i>Switch</i> statement	172
Controlling matching behavior	174
Creating multiple folders: Step-by-step exercises	174
Chapter 5 quick reference	177

Chapter 6 Working with functions 179

Understanding functions	179
Using functions to provide ease of code reuse	186
Including functions in the Windows PowerShell environment	188
Using dot-sourcing	188
Using dot-sourced functions	190
Adding help for functions	191
Using a <i>here-string</i> object for help	192
Using two input parameters	194
Using a type constraint in a function	198
Using more than two input parameters	200
Using functions to encapsulate business logic	202
Using functions to provide ease of modification	204
Understanding filters	209
Creating a function: Step-by-step exercises	213
Chapter 6 quick reference	216

Chapter 7 Creating advanced functions and modules 217

The <i>[cmdletbinding]</i> attribute	217
Easy verbose messages	218
Automatic parameter checks	219
Adding support for the <i>-WhatIf</i> switch parameter	222
Adding support for the <i>-Confirm</i> switch parameter	223
Specifying the default parameter set	224

The <i>Parameter</i> attribute	224
The <i>Mandatory</i> parameter property	225
The <i>Position</i> parameter property	226
The <i>ParameterSetName</i> parameter property	227
The <i>ValueFromPipeline</i> property	228
The <i>HelpMessage</i> property	229
Understanding modules	230
Locating and loading modules	230
Installing modules	235
Creating a module	246
Creating an advanced function and installing a module:	
Step-by-step exercises	253
Chapter 7 quick reference	257

Chapter 8 Using the Windows PowerShell ISE 259

Running the Windows PowerShell ISE	259
Navigating the Windows PowerShell ISE	260
Working with the script pane	263
Using tab expansion and IntelliSense	264
Working with Windows PowerShell ISE snippets	266
Using Windows PowerShell ISE snippets to create code	266
Creating new Windows PowerShell ISE snippets	268
Removing user-defined Windows PowerShell ISE snippets	269
Using the Commands add-on and snippets: Step-by-step exercises ..	270
Chapter 8 quick reference	274

Chapter 9 Working with Windows PowerShell profiles 275

Six different Windows PowerShell profiles	275
Understanding the six Windows PowerShell profiles	276
Examining the <i>\$profile</i> variable	276
Determining whether a specific profile exists	278
Creating a new profile	279
Design considerations for profiles	279
Using one or more profiles	281

Using the All Users, All Hosts profile	283
Using your own file	284
Grouping similar functionality into a module	285
Where to store the profile module	285
Creating and adding functionality to a profile:	
Step-by-step exercises	286
Chapter 9 quick reference	289
Chapter 10 Using WMI	291
Understanding the WMI model	292
Working with objects and namespaces	292
Listing WMI providers	297
Working with WMI classes	298
Querying WMI	301
Obtaining service information: Step-by-step exercises	306
Chapter 10 quick reference	312
Chapter 11 Querying WMI	313
Alternate ways to connect to WMI	313
Returning selective data from all instances	321
Selecting multiple properties	322
Choosing specific instances	325
Using an operator	327
Shortening the syntax	330
Working with software: Step-by-step exercises	332
Chapter 11 quick reference	339
Chapter 12 Remoting WMI	341
Using WMI against remote systems	341
Supplying alternate credentials for the remote connection.	342
Using Windows PowerShell remoting to run WMI	345
Using CIM classes to query WMI classes	346

Working with remote results.	348
Reducing data via Windows PowerShell parameters.	352
Reducing data via WQL query.	353
Running WMI jobs	355
Using Windows PowerShell remoting and WMI:	
Step-by-step exercises	357
Chapter 12 quick reference	360

Chapter 13 Calling WMI methods on WMI classes 361

Using WMI cmdlets to execute instance methods	361
Using the <i>Terminate</i> method directly.	363
Using the <i>Invoke-WmiMethod</i> cmdlet	365
Using the <i>[wmi]</i> type accelerator	366
Using WMI cmdlets to work with static methods	367
Executing instance methods: Step-by-step exercises.	370
Chapter 13 quick reference	373

Chapter 14 Using the CIM cmdlets 375

Using the CIM cmdlets to explore WMI classes.	375
Using the <i>Get-CimClass</i> cmdlet and the <i>-ClassName</i>	
parameter	375
Finding WMI class methods	377
Filtering classes by qualifier	379
Retrieving WMI instances	383
Reducing returned properties and instances	383
Cleaning up output from the command	384
Working with associations.	385
Retrieving WMI instances: Step-by-step exercises	392
Chapter 14 quick reference	394

Chapter 15 Working with Active Directory 395

Creating objects in Active Directory	395
Creating an OU	395
ADSI providers	397
LDAP names	399
Creating users	405
What is user account control?	408
Working with users	409
Creating multiple OUs: Step-by-step exercises	423
Chapter 15 quick reference	429

Chapter 16 Working with the AD DS module 431

Understanding the Active Directory module	431
Installing the Active Directory module	431
Getting started with the Active Directory module	433
Using the Active Directory module	433
Finding the FSMO role holders	435
Discovering Active Directory	439
Renaming Active Directory sites	442
Managing users	443
Creating a user	446
Finding and unlocking Active Directory user accounts	447
Finding disabled users	449
Finding unused user accounts	451
Updating Active Directory objects: Step-by-step exercises	454
Chapter 16 quick reference	457

Chapter 17 Deploying Active Directory by using Windows PowerShell 459

Using the Active Directory module to deploy a new forest	459
Adding a new domain controller to an existing domain	465
Adding a read-only domain controller	468

Installing domain controller prerequisites and adding to a forest: Step-by-step exercises	470
Chapter 17 quick reference	472

Chapter 18 Debugging scripts 473

Understanding debugging in Windows PowerShell	473
Understanding the three different types of errors	473
Using the <i>Set-PSDebug</i> cmdlet	479
Tracing the script	479
Stepping through the script	483
Enabling strict mode	488
Debugging the script	492
Setting breakpoints	492
Setting a breakpoint on a line number	492
Setting a breakpoint on a variable	495
Setting a breakpoint on a command	499
Responding to breakpoints	501
Listing breakpoints	503
Enabling and disabling breakpoints	504
Deleting breakpoints	504
Debugging a function: Step-by-step exercises	505
Chapter 18 quick reference	509

Chapter 19 Handling errors 511

Handling missing parameters	511
Creating a default value for a parameter	512
Making the parameter mandatory	513
Limiting choices	514
Using <i>PromptForChoice</i> to limit selections	514
Using <i>Test-Connection</i> to identify computer connectivity	516
Using the <i>-contains</i> operator to examine the contents of an array	517
Using the <i>-contains</i> operator to test for properties	519

Handling missing rights	521
Using an attempt-and-fail approach	522
Checking for rights and exiting gracefully	522
Handling missing WMI providers	523
Handling incorrect data types	532
Handling out-of-bounds errors	536
Using a boundary-checking function	536
Placing limits on the parameter	537
Using <i>Try...Catch...Finally</i>	538
Catching multiple errors	541
Using <i>PromptForChoice</i> to limit selections and using <i>Try...Catch...Finally</i> : Step-by-step exercises	544
Chapter 19 quick reference	546

Chapter 20 Using the Windows PowerShell workflow 547

Why use workflows?	547
Workflow requirements	548
A simple workflow	548
Parallel PowerShell	549
Workflow activities	552
Windows PowerShell cmdlets as activities	553
Disallowed core cmdlets	554
Non-automatic cmdlet activities	554
Parallel activities	555
Checkpointing Windows PowerShell workflow	556
Understanding checkpoints	556
Placing checkpoints	556
Adding checkpoints	556
Adding a sequence activity to a workflow	559
Creating a workflow and adding checkpoints: Step-by-step exercises	561
Chapter 20 quick reference	563

Chapter 21 Managing Windows PowerShell DSC	565
Understanding Desired State Configuration	565
The DSC process.	566
Configuration parameters	568
Setting dependencies	570
Controlling configuration drift.	571
Modifying environment variables	573
Creating a DSC configuration and adding a dependency:	
Step-by-step exercises	576
Chapter 21 quick reference.	580
 Chapter 22 Using the PowerShell Gallery	 581
Exploring the PowerShell Gallery.	581
Configuring and using PowerShell Get.	583
Installing a module from the PowerShell Gallery	585
Configuring trusted installation locations	586
Uninstalling a module	586
Searching for and installing modules from the PowerShell Gallery:	
Step-by-step exercises	587
Chapter 22 quick reference.	589
 <i>Appendix A: Windows PowerShell scripting best practices</i>	 591
<i>Appendix B: Regular expressions quick reference</i>	599
 <i>Index</i>	 603
<i>About the author</i>	631

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can improve our books and learning resources for you. To participate in a brief survey, please visit:

<http://aka.ms/tellpress>

Introduction

Windows PowerShell is the de facto management standard for Windows administrators. As part of the Microsoft Engineering Common Criteria, Windows PowerShell management hooks are built into all server-based products, including Microsoft SQL Server, Exchange, System Center, and SharePoint. Knowledge of, and even expertise in, this technology is no longer “nice to know”—it is essential, and it often appears as a required skill set in open job notices. *Windows PowerShell Step by Step, Third Edition*, offers a solid footing for the IT pro trying to come up to speed on this essential management technology.

Who should read this book

This book exists to help IT pros come up to speed quickly on the exciting Windows PowerShell 5.0 technology. *Windows PowerShell Step by Step, Third Edition* is specifically aimed at several audiences, including:

- **Windows networking consultants** Anyone who wants to standardize and to automate the installation and configuration of Microsoft .NET networking components.
- **Windows network administrators** Anyone who wants to automate the day-to-day management of Windows or .NET networks.
- **Microsoft Certified Solutions Experts (MCSEs) and Microsoft Certified Trainers (MCTs)** Windows PowerShell is a key component of many Microsoft courses and certification exams.
- **General technical staff** Anyone who wants to collect information or configure settings on Windows machines.
- **Power users** Anyone who wants to obtain maximum power and configurability of their Windows machines, either at home or in an unmanaged desktop workplace environment.

Assumptions

This book expects that you are familiar with the Windows operating system; therefore, basic networking terms are not explained in detail. The book does not expect you to have any background in programming, development, or scripting. All elements related to these topics, as they arise, are fully explained.

This book might not be for you if...

Not every book is aimed at every possible audience. This is not a Windows PowerShell 5.0 reference book; therefore, extremely deep, esoteric topics are not covered. Although some advanced topics are covered, in general the discussion starts with beginner topics and proceeds through an intermediate depth. If you have never seen a computer and have no idea what a keyboard or a mouse is, this book definitely is not for you.

Organization of this book

This book can be divided into three parts. The first part explores the Windows PowerShell command line. The second discusses Windows PowerShell scripting. The third part covers more advanced Windows PowerShell techniques, in addition to the use of Windows PowerShell in various management scenarios. This three-part structure is somewhat artificial and is not actually delimited by “part” pages, but it is a useful way to approach a rather long book.

A better way to approach the book would be to think of it as a big sampler box of chocolates. Each chapter introduces new experiences, techniques, and skills. Though the book is not intended to be an advanced-level book on computer programming, it is intended to provide a foundation that you could use to progress to advanced levels of training if you find an area that you see as especially suited to your needs. So if you fall in love with Windows PowerShell Desired State Configuration, remember that Chapter 21, “Managing Windows PowerShell DSC,” is only a sample of what you can do with this technology. Indeed, some Windows PowerShell MVPs are almost completely focused on this one aspect of Windows PowerShell.

Finding your best starting point in this book

The different sections of *Windows PowerShell Step by Step, Third Edition*, cover a wide range of technologies. Depending on your needs and your existing understanding of Microsoft tools, you might want to focus on specific areas of the book. Use the following table to determine how best to proceed through the book.

If you are	Follow these steps
New to Windows PowerShell	Focus on Chapters 1–3 and 5–9, or read through the entire book in order.
An IT pro who knows the basics of Windows PowerShell and only needs to learn how to manage network resources	Briefly skim Chapters 1–3 if you need a refresher on the core concepts. Read up on the new technologies in Chapters 4, 14, and 20–22.
Interested in Active Directory	Read Chapters 15–17.
Interested in Windows PowerShell Scripting	Read Chapters 5–8, 18, and 19.
Familiar with Windows PowerShell 3.0	Read Chapter 1, skim Chapters 8 and 18, and read Chapters 20–22.
Familiar with Windows PowerShell 4.0	Read Chapter 1, skim Chapters 8, 18, and 21, and read Chapter 22.

All of the book’s chapters include two hands-on labs that let you try out the concepts just learned.

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

- Windows 10, Windows 7, Windows Server 2012 R2, Windows Server 2012, Windows Server 2008 R2, or Windows Server 2008 with Service Pack 2.
- Computer that has a 1.6 GHz or faster processor (2 GHz recommended)
- 1 GB (32-bit) or 2 GB (64-bit) RAM
- 3.5 GB of available hard disk space
- 5400 RPM hard disk drive
- DirectX 9 capable video card running at 1024 x 768 or higher-resolution display
- Internet connection to download software or chapter examples

Depending on your Windows configuration, you might require Local Administrator rights to run certain commands.

Downloads: Scripts

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample scripts can be downloaded from the following page:

<http://aka.ms/PS3E/files>

Follow the instructions to download the PS3E_675117_Scripts.zip file.

Installing the scripts

Follow these steps to install the scripts on your computer so that you can use them with the exercises in this book.

1. Unzip the PS3E_675117_Scripts.zip file that you downloaded from the book's website.
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

Using the scripts

The folders created by unzipping the file are named for each chapter from the book that contains scripts.

Acknowledgments

I'd like to thank the following people: my editors Kathy Krause and Jaime Odell from OTSI, for turning the book into something resembling English and steering me through the numerous Microsoft stylisms; my technical reviewer and good friend Brian Wilhite, Microsoft PFE, whose attention to detail kept me from looking foolish; Jason Walker from Microsoft Consulting Services, and Gary Siepser and Ashley McGlone, both from Microsoft PFE, who reviewed my outline and made numerous suggestions with regard to completeness. Lastly, I want to acknowledge my wife, Teresa Wilson, Windows PowerShell MVP (aka the Scripting Wife), who read every page and made numerous suggestions that will be of great benefit to beginning scripters.

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

<http://aka.ms/PS3E/errata>

If you discover an error that is not already listed, please submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at:

msspinput@microsoft.com

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to:

<http://support.microsoft.com>

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

<http://aka.ms/mspressfree>

Check back often to see what is new!

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

<http://aka.ms/tellpress>

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *<http://twitter.com/MicrosoftPress>*.

Overview of Windows PowerShell 5.0

After completing this chapter, you will be able to

- Understand the basic use and capabilities of Windows PowerShell.
- Install Windows PowerShell.
- Use basic command-line utilities inside Windows PowerShell.
- Use Windows PowerShell help.
- Run basic Windows PowerShell cmdlets.
- Get help on basic Windows PowerShell cmdlets.

The release of Windows PowerShell 5.0 continues to offer real power to the Windows network administrator. Combining the power of a full-fledged scripting language with access to command-line utilities, Windows Management Instrumentation (WMI), and even Microsoft Visual Basic Scripting Edition (VBScript), Windows PowerShell provides real power and ease. The implementation of hundreds of cmdlets and advanced functions provides a rich ecosystem that makes sophisticated changes as simple as a single line of easy-to-read code. As part of the Microsoft Common Engineering Criteria, Windows PowerShell is the management solution for the Windows platform.

Understanding Windows PowerShell

Perhaps the biggest obstacle for a Windows network administrator in migrating to Windows PowerShell 5.0 is understanding what Windows PowerShell actually is. In some respects, it is a replacement for the venerable CMD (command) shell. In fact, on Windows Server–based computers running Server Core, it is possible to replace the CMD shell with Windows PowerShell so that when the server starts up, it uses Windows PowerShell as the interface.

As shown here, after Windows PowerShell launches, you can use *cd* to change the working directory, and then use *dir* to produce a directory listing in exactly the same way you would perform these tasks from the CMD shell.

```
PS C:\Windows\System32> cd\  
PS C:\> dir
```

Directory: C:\

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d-----	7/10/2015 7:07 PM		FSO
d-----	7/9/2015 5:24 AM		PerfLogs
d-r---	7/9/2015 6:59 AM		Program Files
d-r---	7/10/2015 7:27 PM		Program Files (x86)
d-r---	7/10/2015 7:18 PM		Users
d-----	7/10/2015 6:00 PM		Windows

```
PS C:\>
```

You can also combine traditional CMD interpreter commands with other utilities, such as *fsutil*. This is shown here.

```
PS C:\> md c:\test
```

Directory: C:\

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d-----	7/11/2015 11:14 AM		test

```
PS C:\> fsutil file createnew c:\test\myfile.txt 1000  
File c:\test\myfile.txt is created  
PS C:\> cd c:\test  
PS C:\test> dir
```

Directory: C:\test

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a----	7/11/2015 11:14 AM	1000	myfile.txt

```
PS C:\test>
```

The preceding two examples show Windows PowerShell being used in an interactive manner. Interactivity is one of the primary features of Windows PowerShell, and you can begin to use Windows PowerShell interactively by opening a Windows PowerShell prompt and entering commands. You can enter the commands one at a time, or you can group them together like a batch file. I will discuss this later because you will need more information to understand it.

Using cmdlets

In addition to using Windows console applications and built-in commands, you can also use the *cmdlets* (pronounced *commandlets*) that are built into Windows PowerShell. Cmdlets can be created by anyone. The Windows PowerShell team creates the core cmdlets, but many other teams at Microsoft were involved in creating the hundreds of cmdlets that were included with Windows 10. They are like executable programs, but they take advantage of the facilities built into Windows PowerShell, and therefore are easy to write. They are not scripts, which are uncompiled code, because they are built using the services of a special Microsoft .NET Framework namespace. Windows PowerShell 5.0 comes with about 1,300 cmdlets on Windows 10, and as additional features and roles are added, so are additional cmdlets. These cmdlets are designed to assist the network administrator or consultant to take advantage of the power of Windows PowerShell without having to learn a scripting language. One of the strengths of Windows PowerShell is that cmdlets use a standard naming convention that follows a verb-noun pattern, such as *Get-Help*, *Get-EventLog*, or *Get-Process*. The cmdlets that use the *get* verb display information about the item on the right side of the dash. The cmdlets that use the *set* verb modify or set information about the item on the right side of the dash. An example of a cmdlet that uses the *set* verb is *Set-Service*, which can be used to change the start mode of a service. All cmdlets use one of the standard verbs. To find all of the standard verbs, you can use the *Get-Verb* cmdlet. In Windows PowerShell 5.0, there are nearly 100 approved verbs.

Installing Windows PowerShell

Windows PowerShell 5.0 comes with Windows 10 Client. You can download the Windows Management Framework 5.0 package, which contains updated versions of Windows Remote Management (WinRM), WMI, and Windows PowerShell 5.0, from the Microsoft Download Center. Because Windows 10 comes with Windows PowerShell 5.0, there is no Windows Management Framework 5.0 package available for download—it is not needed. In order to install Windows Management Framework 5.0 on Windows 7, Windows 8.1, Windows Server 2008 R2, Windows Server 2012, and Windows Server 2012 R2, they all must be running the .NET Framework 4.5.

Deploying Windows PowerShell to down-level operating systems

After Windows PowerShell is downloaded from <http://www.microsoft.com/downloads>, you can deploy it to your enterprise by using any of the standard methods.

Here are few of the methods that you can use to accomplish Windows PowerShell deployment:

- Create a Microsoft Systems Center Configuration Manager package and advertise it to the appropriate organizational unit (OU) or collection.
- Create a Group Policy Object (GPO) in Active Directory Domain Services (AD DS) and link it to the appropriate OU.
- Approve the update in Software Update Services (SUS), when available.
- Add the Windows Management Framework 5.0 packages to a central file share or webpage for self-service.

If you are not deploying to an entire enterprise, perhaps the easiest way to install Windows PowerShell is to download the package and step through the wizard.



Note To use a command-line utility in Windows PowerShell, launch Windows PowerShell by choosing Start | Run | PowerShell. At the Windows PowerShell prompt, enter in the command to run.

Using command-line utilities

As mentioned earlier, command-line utilities can be used directly within Windows PowerShell. The advantages of using command-line utilities in Windows PowerShell, as opposed to simply running them in the CMD interpreter, are the Windows PowerShell pipelining and formatting features. Additionally, if you have batch files or CMD files that already use existing command-line utilities, you can easily modify them to run within the Windows PowerShell environment. The following procedure illustrates adding *ipconfig* commands to a text file.

Running *ipconfig* commands

1. Start Windows PowerShell by choosing Start | Run | PowerShell. The Windows PowerShell prompt opens by default at the root of your user folder—for example, C:\Users\Ed.
2. Enter the command **ipconfig /all**. This is shown here.

```
PS C:\> ipconfig /all
```

3. Pipeline the result of *ipconfig /all* to a text file. This is illustrated here.

```
PS C:\> ipconfig /all >ipconfig.txt
```

4. Open Notepad to view the contents of the text file, as follows.

```
PS C:\> notepad ipconfig.txt
```

Entering a single command into Windows PowerShell is useful, but at times you might need more than one command to provide troubleshooting information or configuration details to assist with setup issues or performance problems. This is where Windows PowerShell really shines. In the past, you would have either had to write a batch file or enter the commands manually. This is shown in the `TroubleShoot.bat` script that follows.

TroubleShoot.bat

```
ipconfig /all >C:\tshoot.txt
route print >>C:\tshoot.txt
hostname >>C:\tshoot.txt
net statistics workstation >>C:\tshoot.txt
```

Of course, if you entered the commands manually, you had to wait for each command to complete before entering the subsequent command. In that case, it was always possible to lose your place in the command sequence, or to have to wait for the result of each command. Windows PowerShell eliminates this problem. You can now enter multiple commands on a single line, and then leave the computer or perform other tasks while the computer produces the output. No batch file needs to be written to achieve this capability.



Tip Use multiple commands on a single Windows PowerShell line. Enter each complete command, and then use a semicolon to separate the commands.

The following exercise describes how to run multiple commands.

Running multiple commands

1. Open Windows PowerShell by choosing Start | Run | PowerShell. The Windows PowerShell prompt opens by default at the root of your user folder.
2. Enter the **ipconfig /all** command. Pipeline the output to a text file called *Tshoot.txt* by using the redirection arrow (`>`). This is the result.

```
ipconfig /all >tshoot.txt
```

3. On the same line, use a semicolon to separate the *ipconfig /all* command from the *route print* command. Append the output from the command to a text file called *Tshoot.txt* by using the redirect-and-append arrow (`>>`). Here is the command so far.

```
ipconfig /all >tshoot.txt; route print >>tshoot.txt
```

4. On the same line, use a semicolon to separate the *route print* command from the *hostname* command. Append the output from the command to a text file called *Tshoot.txt* by using the redirect-and-append arrow. The command up to this point is shown here.

```
ipconfig /all >tshoot.txt; route print >>tshoot.txt; hostname >>tshoot.txt
```

5. On the same line, use a semicolon to separate the *hostname* command from the *net statistics workstation* command. Append the output from the command to a text file called *Tshoot.txt* by using the redirect-and-append arrow. The completed command looks like the following.

```
ipconfig /all >tshoot.txt; route print >>tshoot.txt; hostname >>tshoot.txt; net statistics workstation >>tshoot.txt
```

Security issues with Windows PowerShell

As with any tool as versatile as Windows PowerShell, there are bound to be some security concerns. Security, however, was one of the design goals in the development of Windows PowerShell.

When you launch Windows PowerShell, it opens in the root of your user folder; this ensures that you are in a directory where you will have permission to perform certain actions and activities. This is far safer than opening at the root of the drive, or even opening in system root.

The running of scripts is disabled by default and can be easily managed through Group Policy. It can also be managed on a per-user or per-session basis.

Controlling execution of Windows PowerShell cmdlets

Have you ever opened a CMD interpreter prompt, entered a command, and pressed Enter so that you could find out what it does? What if that command happened to be *Format C:*? Are you sure you want to format your C drive? This section covers some parameters that can be supplied to cmdlets that allow you to control the way they execute. Although not all cmdlets support these parameters, most of those included with Windows PowerShell do. The three switch parameters you can use to control execution are *-WhatIf*, *-Confirm*, and *suspend*. *Suspend* is not really a switch parameter that is supplied to a cmdlet, but rather is an action you can take at a confirmation prompt, and is therefore another method of controlling execution.



Note To use *-WhatIf* at a Windows PowerShell prompt, enter the cmdlet. Type the *-WhatIf* switch parameter after the cmdlet. This only works for cmdlets that change system state. Therefore, there is no *-WhatIf* parameter for cmdlets like *Get-Process* that only display information.

Windows PowerShell cmdlets that change system state (such as *Set-Service*) support a *prototype mode* that you can enter by using the *-WhatIf* switch parameter. The developer decides to implement *-WhatIf* when developing the cmdlet; however, the Windows PowerShell team recommends that developers implement *-WhatIf*. The use of the *-WhatIf* switch parameter is shown in the following procedure.

Using *-WhatIf* to prototype a command

1. Open Windows PowerShell by choosing Start | Run | PowerShell. The Windows PowerShell prompt opens by default at the root of your user folder.
2. Start an instance of Notepad.exe. Do this by entering **notepad** and pressing the Enter key. This is shown here.

```
notepad
```

3. Identify the Notepad process you just started by using the *Get-Process* cmdlet. Type enough of the process name to identify it, and then use a wildcard asterisk (*) to avoid typing the entire name of the process, as follows.

```
Get-Process notep*
```

4. Examine the output from the *Get-Process* cmdlet, and identify the process ID. The output on my machine is shown here. Note that, in all likelihood, the process ID used by your instance of Notepad.exe will be different from the one on my machine.

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
114	8	1544	8712	...54	0.00	3756	notepad

5. Use *-WhatIf* to find out what would happen if you used *Stop-Process* to stop the process ID you obtained in step 4. This process ID is found under the Id column in your output. Use the *-Id* parameter to identify the Notepad.exe process. The command is as follows.

```
Stop-Process -id 3756 -whatif
```

6. Examine the output from the command. It tells you that the command will stop the Notepad process with the process ID that you used in your command.

```
What if: Performing the operation "Stop-Process" on target "notepad (3756)".
```

Confirming actions

As described in the previous section, you can use *-WhatIf* to prototype a cmdlet in Windows PowerShell. This is useful for finding out what a cmdlet would do; however, if you want to be prompted before the execution of the cmdlet, you can use the *-Confirm* parameter.

Confirming the execution of cmdlets

1. Open Windows PowerShell, start an instance of Notepad.exe, identify the process, and examine the output, just as in steps 1 through 4 in the previous exercise.
2. Use the `-Confirm` parameter to force a prompt when using the `Stop-Process` cmdlet to stop the Notepad process identified by the `Get-Process note*` command. This is shown here.

```
Stop-Process -id 3756 -confirm
```

The `Stop-Process` cmdlet, when used with the `-Confirm` parameter, displays the following confirmation prompt.

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (3756)".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):
```

3. Enter **y** and press Enter. The Notepad.exe process ends. The Windows PowerShell prompt returns to the default, ready for new commands, as shown here.

```
PS C:\>
```



Tip To suspend cmdlet confirmation, at the confirmation prompt from the cmdlet, enter **s** and press Enter.

Suspending confirmation of cmdlets

The ability to prompt for confirmation of the execution of a cmdlet is extremely useful and at times might be vital to assisting in maintaining a high level of system uptime. There might be times when you enter a long command and then remember that you need to check on something else first. For example, you might be in the middle of stopping a number of processes, but you need to view details on the processes to ensure that you do not stop the wrong one. For such eventualities, you can tell the confirmation that you would like to suspend execution of the command.

Suspending execution of a cmdlet

1. Open Windows PowerShell, start an instance of Notepad.exe, identify the process, and examine the output, just as in steps 1 through 4 in the “Using `-WhatIf` to prototype a command” exercise. The output on my machine is shown following. Note that in all likelihood, the process ID used by your instance of Notepad.exe will be different from the one on my machine.

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	-----	-----
39	2	944	400	29	0.05	3576	notepad

2. Use the `-Confirm` parameter to force a prompt when using the `Stop-Process` cmdlet to stop the Notepad process identified by the `Get-Process note*` command. This is illustrated here.

```
Stop-Process -id 3576 -confirm
```

The `Stop-Process` cmdlet, when used with the `-Confirm` parameter, displays the following confirmation prompt.

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (3576)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):
```

3. To suspend execution of the `Stop-Process` cmdlet, enter `s.` and then a double-arrow prompt appears, as follows.

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

4. Use the `Get-Process` cmdlet to obtain a list of all the running processes that begin with the letter `n`. The syntax is as follows.

```
Get-Process n*
```

On my machine, two processes appear, the Notepad process I launched earlier and another process. This is shown here.

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
269	168	4076	2332	...98	0.19	1632	NisSrv
114	8	1536	8732	...54	0.02	3576	notepad

5. Return to the previous confirmation prompt by entering `exit`.

Again, the confirmation prompt appears as follows.

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (3576)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):
```

6. Enter `y` and press Enter to stop the Notepad process. There is no further confirmation. The prompt now displays the default Windows PowerShell prompt, as shown here.

```
PS C:\>
```

Working with Windows PowerShell

This section goes into detail about how to access Windows PowerShell and configure the Windows PowerShell console.

Accessing Windows PowerShell

After Windows PowerShell is installed on a down-level system, it becomes available for immediate use. However, pressing the Windows logo key on the keyboard and pressing R to bring up a *run* dialog box—or using the mouse to choose Start | Run | PowerShell all the time—will become time-consuming and tedious. (This is not quite as big a problem on Windows 10, where you can just enter **PowerShell** on the Start screen.) On Windows 10, I pin both Windows PowerShell and the Windows PowerShell ISE to both the Start screen and the taskbar. On Windows Server 2012 R2 running Server Core, I replace the CMD prompt with the Windows PowerShell console. For me and the way I work, this is ideal, so I wrote a script to do it. This script can be called through a log-on script to automatically deploy the shortcut on the desktop. On Windows 10, the script adds both the Windows PowerShell ISE and the Windows PowerShell console to both the Start screen and the taskbar. On Windows 7, it adds both to the taskbar and to the Start menu. The script only works for US English-language operating systems. To make it work in other languages, change the value of *\$pinToStart* and *\$pinToTaskBar* to the equivalent values in the target language.



Note Using Windows PowerShell scripts is covered in Chapter 5, “Using Windows PowerShell scripts.” See that chapter for information about how the script works and how to actually run the script.

The script is called `PinToStart.ps1`, and is as follows.

`PinToStart.ps1`

```
$pinToStart = "Pin to Start"
```

```
$file = @((Join-Path -Path $PSHOME -childpath "PowerShell.exe"),  
          (Join-Path -Path $PSHOME -childpath "powershell_ise.exe") )  
Foreach($f in $file)  
{ $path = Split-Path $f  
  $shell=New-Object -com "Shell.Application"  
  $folder=$shell.Namespace($path)  
  $item = $folder.parsename((Split-Path $f -leaf))  
  $verbs = $item.verbs()  
  foreach($v in $verbs)  
  { if($v.Name.Replace("&", "") -match $pinToStart){ $v.DoIt() } }
```

Configuring the Windows PowerShell console

Many items can be configured for Windows PowerShell. These items can be stored in a `PSConsole` file. To export the console configuration file, use the `Export-Console` cmdlet, as shown here.

```
PS C:\> Export-Console myconsole
```

The `PSConsole` file is saved in the current directory by default and has an extension of `.psc1`. The `PSConsole` file is saved in XML format. A generic console file is shown here.

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>5.0.10224.0</PSVersion>
  <PSSnapIns />
</PSConsoleFile>
```

Controlling Windows PowerShell launch options

1. Launch Windows PowerShell without the banner by using the `-NoLogo` argument. This is shown here.

```
PowerShell -nologo
```

2. Launch a specific version of Windows PowerShell by using the `-Version` argument. This is shown here.

```
PowerShell -version 3
```

3. Launch Windows PowerShell using a specific configuration file by specifying the `-PSConsoleFile` argument, as follows.

```
PowerShell -psconsolefile myconsole.psc1
```

4. Launch Windows PowerShell, execute a specific command, and then exit by using the `-Command` argument. The command itself must be prefixed by an ampersand (&) and enclosed in braces. This is shown here.

```
Powershell -command "& {Get-Process}"
```

Supplying options for cmdlets

One of the useful features of Windows PowerShell is the standardization of the syntax in working with cmdlets. This vastly simplifies the learning of Windows PowerShell and language constructs. Table 1-1 lists the common parameters. Keep in mind that some cmdlets cannot implement some of these parameters. However, if these parameters are used, they will be interpreted in the same manner for all cmdlets, because the Windows PowerShell engine itself interprets the parameters.

TABLE 1-1 Common parameters

Parameter	Meaning
-WhatIf	Tells the cmdlet to not execute, but to tell you what would happen if the cmdlet were to run.
-Confirm	Tells the cmdlet to prompt before executing the command.
-Verbose	Instructs the cmdlet to provide a higher level of detail than a cmdlet not using the verbose parameter.
-Debug	Instructs the cmdlet to provide debugging information.
-ErrorAction	Instructs the cmdlet to perform a certain action when an error occurs. Allowed actions are <i>Continue</i> , <i>Ignore</i> , <i>Inquire</i> , <i>SilentlyContinue</i> , <i>Stop</i> , and <i>Suspend</i> .
-ErrorVariable	Instructs the cmdlet to use a specific variable to hold error information. This is in addition to the standard <i>\$Error</i> variable.
-OutVariable	Instructs the cmdlet to use a specific variable to hold the output information.
-OutBuffer	Instructs the cmdlet to hold a certain number of objects before calling the next cmdlet in the pipeline.



Note To get help on any cmdlet, use the *Get-Help <cmdletname>* cmdlet. For example, use *Get-Help Get-Process* to obtain help with using the *Get-Process* cmdlet.

Working with the help options

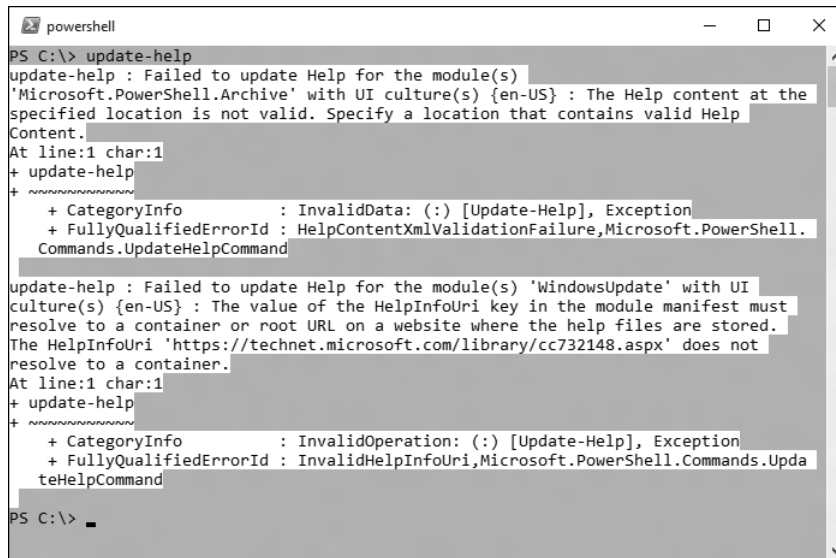
One of the first commands to run when you are opening Windows PowerShell for the first time is the *Update-Help* cmdlet. This is because Windows PowerShell does not include help files with the product, as of Windows PowerShell version 3. This does not mean that no help presents itself—it does mean that help beyond simple syntax display requires an additional download.

A default installation of Windows PowerShell 5.0 contains numerous modules that vary from installation to installation, depending upon the operating system features and roles selected. In fact, Windows PowerShell 5.0 installed on Windows 7 workstations contains far fewer modules and cmdlets than are available on a similar Windows 10 workstation. This does not mean that all is chaos, however, because the essential Windows PowerShell cmdlets—the *core* cmdlets—remain unchanged from installation to installation. The difference between installations is because additional features and roles often install additional Windows PowerShell modules and cmdlets.

The modular nature of Windows PowerShell requires additional consideration when you are updating help. Simply running *Update-Help* does not update all of the modules loaded on a particular system. In fact, some modules might not support updatable help at all—these generate an error when you attempt to update help. The easiest way to ensure that you update all possible help is to use both the *-Module* parameter and the *-Force* switch parameter. The command to update help for all installed modules (those that support updatable help) is shown here.

```
Update-Help -Module * -Force
```

The result of running the *Update-Help* cmdlet on a typical Windows 10 client system is shown in Figure 1-1.



```
PS C:\> update-help
update-help : Failed to update Help for the module(s) 'Microsoft.PowerShell.Archive' with UI culture(s) {en-US} : The Help content at the specified location is not valid. Specify a location that contains valid Help Content.
At line:1 char:1
+ update-help
+ ~~~~~
+ CategoryInfo          : InvalidData: (:) [Update-Help], Exception
+ FullyQualifiedErrorId : HelpContentXmlValidationFailure,Microsoft.PowerShell.Commands.UpdateHelpCommand

update-help : Failed to update Help for the module(s) 'WindowsUpdate' with UI culture(s) {en-US} : The value of the HelpInfoUri key in the module manifest must resolve to a container or root URL on a website where the help files are stored. The HelpInfoUri 'https://technet.microsoft.com/library/cc732148.aspx' does not resolve to a container.
At line:1 char:1
+ update-help
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Update-Help], Exception
+ FullyQualifiedErrorId : InvalidHelpInfoUri,Microsoft.PowerShell.Commands.UpdateHelpCommand

PS C:\>
```

FIGURE 1-1 Errors appear when you attempt to update help files that do not support updatable help.

One way to update help and not receive a screen full of error messages is to run the *Update-Help* cmdlet and suppress the errors altogether. This technique is shown here.

```
Update-Help -Module * -Force -ea 0
```

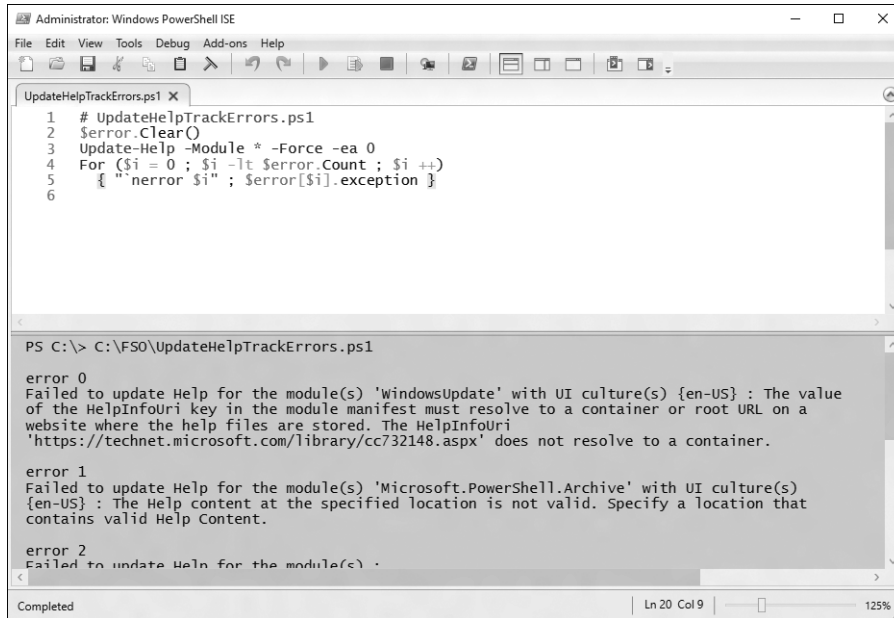
The problem with this approach is that you can never be certain that you have actually received updated help for everything you wanted to update. A better approach is to hide the errors during the update process, but also to display errors after the update completes. The advantage to this approach is the ability to display cleaner errors. The *UpdateHelpTrackErrors.ps1* script illustrates this technique. The first thing the *UpdateHelpTrackErrors.ps1* script does is empty the error stack by calling the *clear* method. Next, it calls the *Update-Help* module with both the *-Module* parameter and the *-Force* switch parameter. In addition, it uses the *-ErrorAction* parameter (*ea* is an alias for this parameter) with a value of 0 (zero). A 0 value means that errors will not be displayed when the command runs. The script concludes by using a *For* loop to walk through the errors and by displaying the error exceptions. The complete *UpdateHelpTrackErrors.ps1* script is shown here.

```
UpdateHelpTrackErrors.ps1
$error.Clear()
Update-Help -Module * -Force -ea 0
For ($i = 0 ; $i -lt $error.Count ; $i ++){
    "`nerror $i" ; $error[$i].exception }
```



Note For information about writing Windows PowerShell scripts and about using the *For* loop, see Chapter 5.

When the `UpdateHelpTrackErrors` script runs, a progress bar is shown, indicating the progress as the updatable help files update. When the script is finished, any errors appear in order. The script and associated errors are shown in Figure 1-2.



```
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
UpdateHelpTrackErrors.ps1 X
1 # UpdateHelpTrackErrors.ps1
2 $Error.Clear()
3 Update-Help -Module * -Force -ea 0
4 For ($i = 0 ; $i -lt $Error.Count ; $i +=)
5 { "error $i" ; $Error[$i].exception }
6

PS C:\> C:\FSO\UpdateHelpTrackErrors.ps1

error 0
Failed to update Help for the module(s) 'windowsUpdate' with UI culture(s) {en-US} : The value
of the HelpInfoUri key in the module manifest must resolve to a container or root URL on a
website where the help files are stored. The HelpInfoUri
'https://technet.microsoft.com/library/cc732148.aspx' does not resolve to a container.

error 1
Failed to update Help for the module(s) 'Microsoft.PowerShell.Archive' with UI culture(s)
{en-US} : The Help content at the specified location is not valid. Specify a location that
contains valid Help Content.

error 2
Failed to update Help for the module(s) .

Completed | Ln 20 Col 9 | 125%
```

FIGURE 1-2 Cleaner error output from updatable help is generated by the `UpdateHelpTrackErrors` script.

You can also determine which modules receive updated help by running the *Update-Help* cmdlet with the *-Verbose* switch parameter. Unfortunately, when you do this, the output scrolls by so fast that it is hard to see what has actually updated. To solve this problem, redirect the verbose output to a text file. In the command that follows, all modules attempt to update *help*. The verbose messages redirect to a text file named *updatedhelp.txt* in a folder named *fso* off the root.

```
Update-Help -module * -force -verbose 4>>c:\fso\updatedhelp.txt
```

Windows PowerShell has a high level of discoverability; that is, to learn how to use Windows PowerShell, you can simply use Windows PowerShell. Online help serves an important role in assisting in this discoverability. The help system in Windows PowerShell can be entered by several methods.

To learn about using Windows PowerShell, use the *Get-Help* cmdlet as follows.

Get-Help Get-Help

This command prints out help about the Get-Help cmdlet. The output from this cmdlet is illustrated here:

NAME

Get-Help

SYNOPSIS

Displays information about Windows PowerShell commands and concepts.

SYNTAX

```
Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
[-Full] [-Functionality <String[]>] [-Path <String>] [-Role <String[]>]
[<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
[-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -Detailed
[<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
[-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -Examples
[<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
[-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -Online
[<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
[-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -Parameter
<String> [<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>]
[-Functionality <String[]>] [-Path <String>] [-Role <String[]>] -ShowWindow
[<CommonParameters>]
```

DESCRIPTION

The Get-Help cmdlet displays information about Windows PowerShell concepts and commands, including cmdlets, functions, CIM commands, workflows, providers, aliases and scripts.

To get help for a Windows PowerShell command, type "Get-Help" followed by the command name, such as: Get-Help Get-Process. To get a list of all help topics on your system, type: Get-Help *. You can display the entire help topic or use the parameters of the Get-Help cmdlet to get selected parts of the topic, such as the syntax, parameters, or examples.

Conceptual help topics in Windows PowerShell begin with "about_", such as "about_Comparison_Operators". To see all "about_" topics, type: Get-Help about_*. To see a particular topic, type: Get-Help about_<topic-name>, such as Get-Help about_Comparison_Operators.

To get help for a Windows PowerShell provider, type "Get-Help" followed by the provider name. For example, to get help for the Certificate provider, type: `Get-Help Certificate`.

In addition to "Get-Help", you can also type "help" or "man", which displays one screen of text at a time, or "<cmdlet-name> -?", which is identical to Get-Help but works only for commands.

Get-Help gets the help content that it displays from help files on your computer. Without the help files, Get-Help displays only basic information about commands. Some Windows PowerShell modules come with help files. However, beginning in Windows PowerShell 3.0, the modules that come with Windows do not include help files. To download or update the help files for a module in Windows PowerShell 3.0, use the Update-Help cmdlet.

You can also view the help topics for Windows PowerShell online in the TechNet Library. To get the online version of a help topic, use the Online parameter, such as: `Get-Help Get-Process -Online`. You can read all of the help topics beginning at: <http://go.microsoft.com/fwlink/?LinkID=107116>.

If you type "Get-Help" followed by the exact name of a help topic, or by a word unique to a help topic, Get-Help displays the topic contents. If you enter a word or word pattern that appears in several help topic titles, Get-Help displays a list of the matching titles. If you enter a word that does not appear in any help topic titles, Get-Help displays a list of topics that include that word in their contents.

Get-Help can get help topics for all supported languages and locales. Get-Help first looks for help files in the locale set for Windows, then in the parent locale (such as "pt" for "pt-BR"), and then in a fallback locale. Beginning in Windows PowerShell 3.0, if Get-Help does not find help in the fallback locale, it looks for help topics in English ("en-US") before returning an error message or displaying auto-generated help.

For information about the symbols that Get-Help displays in the command syntax diagram, see `about_Command_Syntax`. For information about parameter attributes, such as Required and Position, see `about_Parameters`.

TROUBLESHOOTING NOTE: In Windows PowerShell 3.0 and 4.0, Get-Help cannot find About topics in modules unless the module is imported into the current session. This is a known issue. To get About topics in a module, import the module, either by using the Import-Module cmdlet or by running a cmdlet in the module.

RELATED LINKS

Online Version: <http://go.microsoft.com/fwlink/p/?linkid=289584>
Updatable Help Status Table (<http://go.microsoft.com/fwlink/?LinkID=270007>)
`Get-Command`
`Get-Member`
`Get-PSDrive`
`about_Command_Syntax`
`about_Comment_Based_Help`
`about_Parameters`

REMARKS

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

The good thing about help with Windows PowerShell is that it not only displays help about cmdlets, which you would expect, but it also has three levels of display: normal, detailed, and full. Additionally, you can obtain help about concepts in Windows PowerShell. This last feature is equivalent to having an online instruction manual. To retrieve a listing of all the conceptual help articles, use the *Get-Help about** command, as follows.

```
Get-Help about*
```

Suppose you do not remember the exact name of the cmdlet you want to use, but you remember it was a *get* cmdlet. You can use a wildcard, such as an asterisk (*), to obtain the name of the cmdlet. This is shown here.

```
Get-Help get*
```

This technique of using a wildcard operator can be extended further. If you remember that the cmdlet was a *get* cmdlet, and that it started with the letter *p*, you can use the following syntax to retrieve the cmdlet you're looking for.

```
Get-Help get-p*
```

Suppose, however, that you know the exact name of the cmdlet, but you cannot exactly remember the syntax. For this scenario, you can use the *-Examples* switch parameter. For example, for the *Get-PSDrive* cmdlet, you would use *Get-Help* with the *-Examples* switch parameter, as follows.

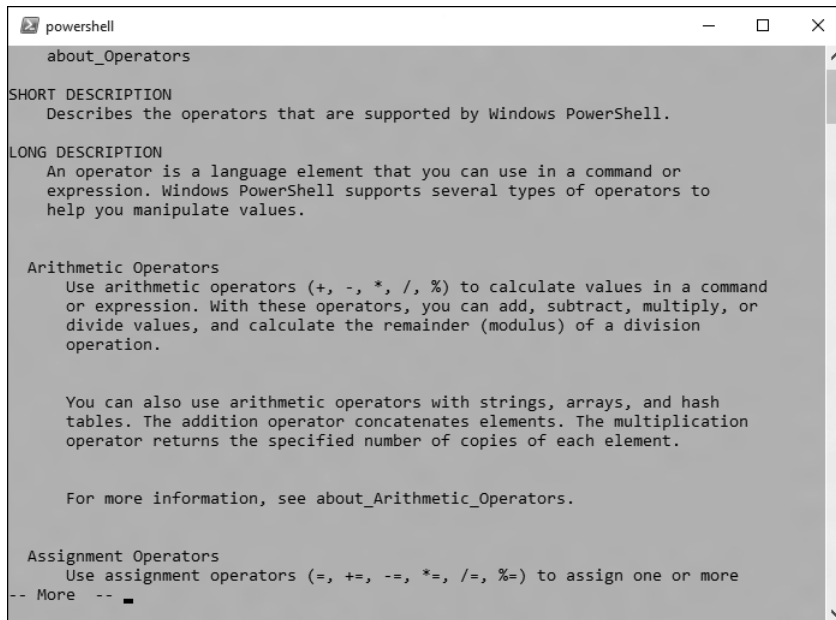
```
Get-Help Get-PSDrive -examples
```

To view help displayed one page at a time, you can use the *Help* function. The *Help* function passes your input to the *Get-Help* cmdlet, and pipelines the resulting information to the *more.com* utility. This causes output to display one page at a time in the Windows PowerShell console. This is useful if you want to avoid scrolling up and down to view the help output.



Note Keep in mind that in the Windows PowerShell ISE, the pager does not work, and therefore you will find no difference in output between *Get-Help* and *Help*. In the ISE, both *Get-Help* and *Help* behave the same way. However, it is likely that if you are using the Windows PowerShell ISE, you will use *Show-Command* for your help instead of relying on *Get-Help*.

This formatted output is shown in Figure 1-3.



```
powershell

about_Operators

SHORT DESCRIPTION
    Describes the operators that are supported by Windows PowerShell.

LONG DESCRIPTION
    An operator is a language element that you can use in a command or
    expression. Windows PowerShell supports several types of operators to
    help you manipulate values.

    Arithmetic Operators
    Use arithmetic operators (+, -, *, /, %) to calculate values in a command
    or expression. With these operators, you can add, subtract, multiply, or
    divide values, and calculate the remainder (modulus) of a division
    operation.

    You can also use arithmetic operators with strings, arrays, and hash
    tables. The addition operator concatenates elements. The multiplication
    operator returns the specified number of copies of each element.

    For more information, see about_Arithmetic_Operators.

    Assignment Operators
    Use assignment operators (=, +=, -=, *=, /=, %=) to assign one or more
    -- More --
```

FIGURE 1-3 Use *Help* to display information one page at a time.

Getting tired of typing *Get-Help* all the time? After all, it is eight characters long. The solution is to create an alias to the *Get-Help* cmdlet. An alias is a shortcut keystroke combination that will launch a program or cmdlet when entered. In the “Creating an alias for the *Get-Help* cmdlet” procedure, you will assign the *Get-Help* cmdlet to the G+H key combination.



Note When creating an alias for a cmdlet, confirm that it does not already have an alias by using *Get-Alias*. Use *New-Alias* to assign the cmdlet to a unique keystroke combination.

Creating an alias for the *Get-Help* cmdlet

1. Open Windows PowerShell by choosing Start | Run | PowerShell. The Windows PowerShell prompt opens by default at the root of your user folder.
2. Retrieve an alphabetic listing of all currently defined aliases, and inspect the list for one assigned to either the *Get-Help* cmdlet or the keystroke combination G+H. The command to do this is as follows.

```
Get-Alias | sort
```

3. After you have determined that there is no alias for the *Get-Help* cmdlet and that none is assigned to the G+H keystroke combination, review the syntax for the *New-Alias* cmdlet. Use the *-Full* switch parameter to the *Get-Help* cmdlet. This is shown here.

```
Get-Help New-Alias -full
```

4. Use the *New-Alias* cmdlet to assign the G+H keystroke combination to the *Get-Help* cmdlet. To do this, use the following command.

```
New-Alias gh Get-Help
```

Exploring commands: Step-by-step exercises

In the following exercises, you'll explore the use of command-line utilities in Windows PowerShell. You will find that it is as easy to use command-line utilities in Windows PowerShell as in the CMD interpreter; however, by using such commands in Windows PowerShell, you gain access to new levels of functionality.

Using command-line utilities

1. Open Windows PowerShell by choosing Start | Run | PowerShell. The Windows PowerShell prompt opens by default at the root of your user folder.
2. Change to the root of C:\ by entering **cd c:** inside the Windows PowerShell prompt.

```
cd c:\
```

3. Obtain a listing of all the files in the root of C:\ by using the *dir* command.

```
dir
```

4. Create a directory off the root of C:\ by using the *md* command.

```
md mytest
```

5. Obtain a listing of all files and folders off the root that begin with the letter *m*.

```
dir m*
```

6. Change the working directory to the Windows PowerShell working directory. You can do this by using the *Set-Location* command, as follows.

```
Set-Location $psHOME
```

7. Obtain a listing of memory counters related to the available bytes by using the *typeperf.exe* command. This command is shown here.

```
typeperf "\memory\available bytes"
```

8. After a few counters have been displayed in the Windows PowerShell window, press Ctrl+C to break the listing.
9. Display the current startup configuration by using the *bcdedit* command (note that you must run this command with admin rights).

```
bcdedit
```

10. Change the working directory back to the C:\Mytest directory you created earlier.

```
Set-Location c:\mytest
```

11. Create a file named *mytestfile.txt* in the C:\Mytest directory. Use the *fsutil* utility, and make the file 1,000 bytes in size. To do this, use the following command.

```
fsutil file createnew mytestfile.txt 1000
```

12. Obtain a directory listing of all the files in the C:\Mytest directory by using the *Get-ChildItem* cmdlet.

13. Print the current date by using the *Get-Date* cmdlet.

14. Clear the screen by using the *cls* command.

15. Print a listing of all the cmdlets built into Windows PowerShell. To do this, use the *Get-Command* cmdlet.

16. Use the *Get-Command* cmdlet to get the *Get-Alias* cmdlet. To do this, use the *-Name* parameter while supplying *Get-Alias* as the value for the parameter. This is shown here.

```
Get-Command -name Get-Alias
```

This concludes the step-by-step exercise. Exit Windows PowerShell by entering **exit** and pressing Enter.

In the following exercise, you'll use various help options to obtain assistance with various cmdlets.

Obtaining help

1. Open Windows PowerShell by choosing Start | Run | PowerShell. The Windows PowerShell prompt opens by default at the root of your user folder.
2. Use the *Get-Help* cmdlet to obtain help about the *Get-Help* cmdlet. Use the command *Get-Help Get-Help* as follows.

```
Get-Help Get-Help
```

3. To obtain detailed help about the *Get-Help* cmdlet, use the *-Detailed* switch parameter, as follows.

```
Get-Help Get-Help -detailed
```

4. To retrieve technical information about the *Get-Help* cmdlet, use the *-Full* switch parameter. This is shown here.

```
Get-Help Get-Help -full
```

5. If you only want to obtain a listing of examples of command usage, use the *-Examples* switch parameter, as follows.

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

6. Obtain a listing of all the informational help topics by using the *Get-Help* cmdlet and the *about* noun with the asterisk (*) wildcard operator. The code to do this is shown here.

```
Get-Help about*
```

7. Obtain a listing of all the help topics related to *get* cmdlets. To do this, use the *Get-Help* cmdlet, and specify the word *get* followed by the wildcard operator, as follows.

```
Get-Help get*
```

8. Obtain a listing of all the help topics related to *set* cmdlets. To do this, use the *Get-Help* cmdlet, followed by the *set* verb, followed by the asterisk wildcard. This is shown here.

```
Get-Help set*
```

This concludes this exercise. Exit Windows PowerShell by entering **exit** and pressing Enter.

Chapter 1 quick reference

To	Do this
Use an external command-line utility	Enter the name of the command-line utility while inside Windows PowerShell.
Use multiple external command-line utilities sequentially	Separate each command-line utility with a semicolon on a single Windows PowerShell line.
Obtain a list of running processes	Use the <i>Get-Process</i> cmdlet.
Stop a process	Use the <i>Stop-Process</i> cmdlet and specify either the name or the process ID parameter.
Model the effect of a cmdlet before actually performing the requested action	Use the <i>-WhatIf</i> switch parameter.
Instruct Windows PowerShell to start up, run a cmdlet, and then exit	Use the <i>PowerShell</i> command while prefixing the cmdlet with & and enclosing the name of the cmdlet in braces.
Prompt for confirmation before stopping a process	Use the <i>Stop-Process</i> cmdlet while specifying the <i>-Confirm</i> parameter.

Working with functions

After completing this chapter, you will be able to

- Understand functions.
- Use functions to provide ease of reuse.
- Use functions to encapsulate logic.
- Use functions to provide ease of modification.

There are clear-cut guidelines that can be used to design functions. These guidelines can be used to ensure that functions are easy to understand, easy to maintain, and easy to troubleshoot. This chapter examines the reasons for the scripting guidelines and provides examples of both good and bad code design.

Understanding functions

In Windows PowerShell, functions have moved to the forefront as the primary programming element used when writing Windows PowerShell scripts. This is not necessarily due to improvements in functions per se, but rather to a combination of factors, including the maturity of Windows PowerShell script writers. In Windows PowerShell 1.0, functions were not well understood, perhaps due to the lack of clear documentation as to their use, purpose, and application.

Microsoft Visual Basic Scripting Edition (VBScript) included both subroutines and functions. According to the classic definitions, a subroutine was used to encapsulate code that would do things like write to a database or create a Microsoft Word document. Functions, on the other hand, were used to return a value. An example of a classic VBScript function is one that converts a temperature from Fahrenheit to Celsius. The function receives a value in Fahrenheit and returns the value in Celsius. The classic function always returns a value—if it does not, a subroutine should be used instead.



Note Needless to say, the concepts of functions and subroutines were a bit confusing for many VBScript writers. A common question I used to receive when teaching VBScript classes was, “When do I use a subroutine and when do I use a function?” After expounding the classic definition, I would then show them that you could actually write a subroutine that would behave like a function. Next, I would write a function that acted like a subroutine. It was great fun, and the class loved it. The Windows PowerShell team has essentially done the same thing. There is no confusion over when to use a subroutine and when to use a function, because there are no subroutines in Windows PowerShell—only functions.

To create a function in Windows PowerShell, you begin with the *Function* keyword, followed by the name of the function. As a best practice, use the Windows PowerShell verb-noun combination when creating functions. Pick the verb from the standard list of Windows PowerShell verbs to make your functions easier to remember. It is a best practice to avoid creating new verbs when there is an existing verb that can easily do the job.

An idea of the verb coverage can be obtained by using the *Get-Command* cmdlet and pipelining the results to the *Group-Object* cmdlet. This is shown here.

```
Get-Command -CommandType cmdlet | Group-Object -Property Verb |  
Sort-Object -Property count -Descending
```

When the preceding command is run, the resulting output is as follows. This command was run on Windows 10 and includes cmdlets from the default modules. As shown in the listing, *Get* is used the most by the default cmdlets, followed distantly by *Set*, *New*, and *Remove*.

Count	Name	Group
----	----	----
107	Get	{Get-Acl, Get-Alias, Get-AppLockerFileInformation...
49	Set	{Set-Acl, Set-Alias, Set-AppBackgroundTaskResourc...
37	New	{New-Alias, New-AppLockerPolicy, New-CertificateN...
29	Remove	{Remove-AppxPackage, Remove-AppxProvisionedPackag...
17	Add	{Add-AppxPackage, Add-AppxProvisionedPackage, Add...
15	Export	{Export-Alias, Export-BinaryMiLog, Export-Certifi...
14	Disable	{Disable-AppBackgroundTaskDiagnosticLog, Disable-...
14	Enable	{Enable-AppBackgroundTaskDiagnosticLog, Enable-Co...
12	Import	{Import-Alias, Import-BinaryMiLog, Import-Certifi...
11	Invoke	{Invoke-CimMethod, Invoke-Command, Invoke-DscReso...
10	Clear	{Clear-Content, Clear-EventLog, Clear-History, Cl...
10	Test	{Test-AppLockerPolicy, Test-Certificate, Test-Com...
9	Write	{Write-Debug, Write-Error, Write-EventLog, Write...
9	Start	{Start-BitsTransfer, Start-DscConfiguration, Star...
8	Register	{Register-ArgumentCompleter, Register-CimIndicati...
7	Out	{Out-Default, Out-File, Out-GridView, Out-Host...}
6	Stop	{Stop-Computer, Stop-DtcDiagnosticResourceManager...
6	ConvertTo	{ConvertTo-Csv, ConvertTo-Html, ConvertTo-Json, C...
5	Update	{Update-FormatData, Update-Help, Update-List, Upd...
5	Format	{Format-Custom, Format-List, Format-SecureBootUEF...
5	ConvertFrom	{ConvertFrom-Csv, ConvertFrom-Json, ConvertFrom-S...

4 Wait	{Wait-Debugger, Wait-Event, Wait-Job, Wait-Process}
4 Unregister	{Unregister-Event, Unregister-PackageSource, Unre...
3 Rename	{Rename-Computer, Rename-Item, Rename-ItemProperty}
3 Receive	{Receive-DtcDiagnosticTransaction, Receive-Job, R...
3 Move	{Move-AppxPackage, Move-Item, Move-ItemProperty}
3 Suspend	{Suspend-BitsTransfer, Suspend-Job, Suspend-Service}
3 Show	{Show-Command, Show-ControlItem, Show-EventLog}
3 Debug	{Debug-Job, Debug-Process, Debug-Runspace}
3 Complete	{Complete-BitsTransfer, Complete-DtcDiagnosticTra...
3 Select	{Select-Object, Select-String, Select-Xml}
3 Resume	{Resume-BitsTransfer, Resume-Job, Resume-Service}
3 Save	{Save-Help, Save-Package, Save-WindowsImage}
2 Unblock	{Unblock-File, Unblock-Tpm}
2 Split	{Split-Path, Split-WindowsImage}
2 Undo	{Undo-DtcDiagnosticTransaction, Undo-Transaction}
2 Restart	{Restart-Computer, Restart-Service}
2 Resolve	{Resolve-DnsName, Resolve-Path}
2 Send	{Send-DtcDiagnosticTransaction, Send-MailMessage}
2 Convert	{Convert-Path, Convert-String}
2 Use	{Use-Transaction, Use-WindowsUnattend}
2 Disconnect	{Disconnect-PSSession, Disconnect-WSMan}
2 Join	{Join-DtcDiagnosticResourceManager, Join-Path}
2 Exit	{Exit-PSHostProcess, Exit-PSSession}
2 Enter	{Enter-PSHostProcess, Enter-PSSession}
2 Copy	{Copy-Item, Copy-ItemProperty}
2 Expand	{Expand-WindowsCustomDataImage, Expand-WindowsImage}
2 Measure	{Measure-Command, Measure-Object}
2 Connect	{Connect-PSSession, Connect-WSMan}
2 Mount	{Mount-AppxVolume, Mount-WindowsImage}
2 Dismount	{Dismount-AppxVolume, Dismount-WindowsImage}
1 Pop	{Pop-Location}
1 Trace	{Trace-Command}
1 Uninstall	{Uninstall-Package}
1 Checkpoint	{Checkpoint-Computer}
1 Tee	{Tee-Object}
1 Unprotect	{Unprotect-CmsMessage}
1 Where	{Where-Object}
1 Switch	{Switch-Certificate}
1 Compare	{Compare-Object}
1 Limit	{Limit-EventLog}
1 Install	{Install-Package}
1 Protect	{Protect-CmsMessage}
1 Optimize	{Optimize-WindowsImage}
1 ForEach	{ForEach-Object}
1 Find	{Find-Package}
1 Initialize	{Initialize-Tpm}
1 Group	{Group-Object}
1 Reset	{Reset-ComputerMachinePassword}
1 Repair	{Repair-WindowsImage}
1 Sort	{Sort-Object}
1 Restore	{Restore-Computer}
1 Push	{Push-Location}
1 Publish	{Publish-DscConfiguration}
1 Confirm	{Confirm-SecureBootUEFI}
1 Read	{Read-Host}

A function is not required to accept any parameters. In fact, many functions do not require input to perform their job in the script. Let's use an example to illustrate this point. A common task for network administrators is obtaining the operating system version. Script writers often need to do this to ensure that their script uses the correct interface or exits gracefully. It is also quite common that one set of files would be copied to a desktop running one version of the operating system, and a different set of files would be copied for another version of the operating system. The first step in creating a function is to come up with a name. Because the function is going to retrieve information, in the listing of cmdlet verbs shown earlier, the best verb to use is *Get*. For the noun portion of the name, it is best to use something that describes the information that will be obtained. In this example, a noun of *OperatingSystemVersion* makes sense. An example of such a function is shown in the *Get-OperatingSystemVersion.ps1* script. The *Get-OperatingSystemVersion* function uses Windows Management Instrumentation (WMI) to obtain the version of the operating system. In this basic form of the function, you have the function keyword followed by the name of the function, and a script block with code in it, which is delimited by braces. This pattern is shown here.

```
Function Function-Name
{
    #insert code here
}
```

In the *Get-OperatingSystemVersion.ps1* script, the *Get-OperatingSystemVersion* function is at the top of the script. It uses the *Function* keyword to define the function, followed by the name, *Get-OperatingSystemVersion*. The script block opens, followed by the code, and then the script block closes. The function uses the *Get-CimInstance* cmdlet to retrieve an instance of the *Win32_Operating-System* WMI class. Because this WMI class only returns a single instance, the properties of the class are directly accessible. The *version* property is the one you'll work with, so use parentheses to force the evaluation of the code inside. The returned management object is used to emit the version value. The braces are used to close the script block. The operating system version is returned to the code that calls the function. In this example, a string that writes *This OS is version* is used. A subexpression is used to force evaluation of the function. The version of the operating system is returned to the place where the function was called. This is shown here.

```
Get-OperatingSystemVersion.ps1

Function Get-OperatingSystemVersion
{
    (Get-CimInstance -Class Win32_OperatingSystem).Version
} #end Get-OperatingSystemVersion

"This OS is version $(Get-OperatingSystemVersion)"
```

Now let's look at choosing the cmdlet verb. In the earlier listing of cmdlet verbs, there is one cmdlet that uses the verb *Read*. It is the *Read-Host* cmdlet, which is used to obtain information from the command line. This would indicate that the verb *Read* is not used to describe reading a file. There is no verb called *Display*, and the *Write* verb is used in cmdlet names such as *Write-Error* and *Write-Debug*, both of which do not really seem to have the concept of displaying information. If you were writing a function that would read the content of a text file and display statistics about that file, you might call the function *Get-TextStatistics*. This is in keeping with cmdlet names such as *Get-Process*

and *Get-Service*, which include the concept of emitting their retrieved content within their essential functionality. The *Get-TextStatistics* function accepts a single parameter called *path*. The interesting thing about parameters for functions is that when you pass a value to the parameter, you use a hyphen. When you refer to the value inside the function, it is a variable such as *\$path*. To call the *Get-TextStatistics* function, you have a couple of options. The first is to use the name of the function and put the value inside parentheses. This is shown here.

```
Get-TextStatistics("C:\fso\mytext.txt")
```

This is a natural way to call the function, and it works when there is a single parameter. It does not work when there are two or more parameters. Another way to pass a value to the function is to use the hyphen and the parameter name. This is shown here.

```
Get-TextStatistics -path "C:\fso\mytext.txt"
```

Note from the previous example that no parentheses are required. You can also use positional arguments when passing a value. In this usage, you omit the name of the parameter entirely and simply place the value for the parameter following the call to the function. This is illustrated here.

```
Get-TextStatistics "C:\fso\mytext.txt"
```



Note The use of positional parameters works well when you are working from the command line and want to speed things along by reducing the typing load. However, it can be a bit confusing to rely on positional parameters, and in general I tend to avoid them—even when working at the command line. This is because I often copy my working code from the console directly into a script, and as a result, I would need to retype the command a second time to get rid of aliases and unnamed parameters. With the improvements in tab expansion, I feel that the time saved by using positional parameters or partial parameters does not sufficiently warrant the time involved in retyping commands when they need to be transferred to scripts. The other reason for always using named parameters is that it helps you to be aware of the exact command syntax.

One additional way to pass a value to a function is to use partial parameter names. All that is required is enough of the parameter name to disambiguate it from other parameters. This is illustrated here.

```
Get-TextStatistics -p "C:\fso\mytext.txt"
```

The complete text of the *Get-TextStatistics* function is shown here.

Get-TextStatistics Function

```
Function Get-TextStatistics($path)
{
    Get-Content -path $path |
    Measure-Object -line -character -word
}
```

Between Windows PowerShell 1.0 and Windows PowerShell 2.0, the number of verbs grew from 40 to 60. In Windows PowerShell 5.0, the number of verbs remained consistent at 98. The list of approved verbs is shown here.

Add	Clear	Close	Copy	Enter	Exit	Find
Format	Get	Hide	Join	Lock	Move	New
Open	Optimize	Pop	Push	Redo	Remove	Rename
Reset	Resize	Search	Select	Set	Show	Skip
Split	Step	Switch	Undo	Unlock	Watch	Backup
Checkpoint	Compare	Compress	Convert	ConvertFrom	ConvertTo	Dismount
Edit	Expand	Export	Group	Import	Initialize	Limit
Merge	Mount	Out	Publish	Restore	Save	Sync
Unpublish	Update	Approve	Assert	Complete	Confirm	Deny
Disable	Enable	Install	Invoke	Register	Request	Restart
Resume	Start	Stop	Submit	Suspend	Uninstall	Unregister
Wait	Debug	Measure	Ping	Repair	Resolve	Test
Trace	Connect	Disconnect	Read	Receive	Send	Write
Block	Grant	Protect	Revoke	Unblock	Unprotect	Use

After the function has been named, you should specify any parameters the function might require. The parameters are contained within parentheses. In the *Get-TextStatistics* function, the function accepts a single parameter: *-path*. When you have a function that accepts a single parameter, you can pass the value to the function by placing the value for the parameter inside parentheses. This is known as calling a function like a method, and is disallowed when you use *Set-StrictMode* with the *Latest* value for the *-Version* parameter. The following command generates an error when the latest strict mode is in effect—otherwise, it is a permissible way to call a function.

```
Get-TextLength("C:\fso\test.txt")
```

The path *C:\fso\test.txt* is passed to the *Get-TextStatistics* function via the *-path* parameter. Inside the function, the string *C:\fso\test.txt* is contained in the *\$path* variable. The *\$path* variable lives only within the confines of the *Get-TextStatistics* function. It is not available outside the scope of the function. It is available from within child scopes of the *Get-TextStatistics* function. A *child scope* of *Get-TextStatistics* is one that is created from within the *Get-TextStatistics* function. In the *Get-TextStatisticsCallChildFunction.ps1* script, the *Write-Path* function is called from within the *Get-TextStatistics* function. This means the *Write-Path* function will have access to variables that are created within the *Get-TextStatistics* function. This is the concept of *variable scope*, which is extremely important when working with functions. As you use functions to separate the creation of objects, you must always be aware of where the objects get created, and where you intend to use them. In the *Get-TextStatisticsCallChildFunction*, the *\$path* variable does not obtain its value until it is passed to the function. It therefore lives within the *Get-TextStatistics* function. But because the *Write-Path* function is called from within the *Get-TextStatistics* function, it inherits the variables from that scope. When you call a function from within another function, variables created within the parent function are available to the child function. This is shown in the *Get-TextStatisticsCallChildFunction.ps1* script, which follows.

```
Get-TextStatisticsCallChildFunction.ps1
Function Get-TextStatistics($path)
{
    Get-Content -path $path |
```

```
Measure-Object -line -character -word
Write-Path
}

Function Write-Path()
{
    "Inside Write-Path the `$path variable is equal to $path"
}

Get-TextStatistics("C:\fso\test.txt")
"Outside the Get-TextStatistics function `$path is equal to $path"
```

Inside the *Get-TextStatistics* function, the *\$path* variable is used to provide the path to the *Get-Content* cmdlet. When the *Write-Path* function is called, nothing is passed to it. But inside the *Write-Path* function, the value of *\$path* is maintained. Outside both of the functions, however, *\$path* does not have any value. The output from running the script is shown here.

Lines	Words	Characters	Property
-----	-----	-----	-----
3	41	210	

```
Inside Write-Path the $path variable is equal to C:\fso\test.txt
Outside the Get-TextStatistics function $path is equal to
```

You will then need to open and close a script block. A pair of opening and closing braces is used to delimit the script block on a function. As a best practice, when writing a function, I will always use the *Function* keyword, and type in the name, the input parameters, and the braces for the script block at the same time. This is shown here.

```
Function My-Function
{
    #insert code here
}
```

In this manner, I make sure I do not forget to close the braces. Trying to identify a missing brace within a long script can be somewhat problematic, because the error that is presented does not always correspond to the line that is missing the brace. For example, suppose the closing brace is left off the *Get-TextStatistics* function, as shown in the *Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBrace.ps1* script. An error will be generated, as shown here.

```
Missing closing '}' in statement block.
At C:\Scripts\Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1:28
char:1
```

The problem is that the position indicator of the error message points to the first character on line 28. Line 28 happens to be the first blank line after the end of the script. This means that Windows PowerShell scanned the entire script looking for the closing brace. Because it did not find it, it states that the error is at the end of the script. If you were to place a closing brace on line 28, the error in this example would go away, but the script would not work. The *Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1* script is shown here, with a comment that indicates where the missing closing brace should be placed.

Get-TextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBrace.ps1

```
Function Get-TextStatistics($path)
{
    Get-Content -path $path |
    Measure-Object -line -character -word
    Write-Path
    # Here is where the missing brace goes

Function Write-Path()
{
    "Inside Write-Path the `$path variable is equal to $path"
}
Get-TextStatistics("C:\fso\test.txt")
Write-Host "Outside the Get-TextStatistics function `$path is equal to $path"
```

One other technique to guard against the problem of the missing brace is to add a comment to the closing brace of each function.

Using functions to provide ease of code reuse

When scripts are written using well-designed functions, it makes it easier to reuse them in other scripts, and to provide access to these functions from within the Windows PowerShell console. To get access to these functions, you will need to *dot-source* the containing script by placing a dot in front of the path to the script when you call it, and put the functions in a module or load them via the profile. An issue with dot-sourcing scripts to bring in functions is that often the scripts might contain global variables or other items you do not want to bring into your current environment.

An example of a useful function is the *ConvertToMeters.ps1* script because it converts feet to meters. There are no variables defined outside the function, and the function itself does not use the *Write-Host* cmdlet to break up the pipeline. The results of the conversion will be returned directly to the calling code. The only problem with the *ConvertToMeters.ps1* script is that when it is dot-sourced into the Windows PowerShell console, it runs and returns the data because all executable code in the script is executed. The *ConvertToMeters.ps1* script is shown here.

ConvertToMeters.ps1

```
Function Script:ConvertToMeters($feet)
{
    "$feet feet equals $($feet*.31) meters"
} #end ConvertToMeters
$feet = 5
ConvertToMeters -Feet $feet
```

With well-written functions, it is trivial to collect them into a single script—you just cut and paste. When you are done, you have created a function library.

When pasting your functions into the function library script, pay attention to the comments at the end of the function. The comments at the closing brace for each function not only point to the end of the script block, they also provide a nice visual indicator for the end of each function. This

can be helpful when you need to troubleshoot a script. An example of such a function library is the `ConversionFunctions.ps1` script, which is shown here.

`ConversionFunctions.ps1`

```
Function Script:ConvertToMeters($feet)
{
    "$feet feet equals $($feet*.31) meters"
} #end ConvertToMeters

Function Script:ConvertToFeet($meters)
{
    "$meters meters equals $($meters * 3.28) feet"
} #end ConvertToFeet

Function Script:ConvertToFahrenheit($celsius)
{
    "$celsius celsius equals $((1.8 * $celsius) + 32 ) fahrenheit"
} #end ConvertToFahrenheit

Function Script:ConvertToCelsius($fahrenheit)
{
    "$fahrenheit fahrenheit equals $( ( ($fahrenheit - 32)/9)*5 ) celsius"
} #end ConvertToCelsius

Function Script:ConvertToMiles($kilometer)
{
    "$kilometer kilometers equals $( ($kilometer *.6211) ) miles"
} #end convertToMiles

Function Script:ConvertToKilometers($miles)
{
    "$miles miles equals $( ($miles * 1.61) ) kilometers"
} #end convertToKilometers
```

One way to use the functions from the `ConversionFunctions.ps1` script is to use the dot-sourcing operator to run the script so that the functions from the script are part of the calling scope. To dot-source the script, you use the dot-source operator (the period, or dot symbol), followed by a space, followed by the path to the script containing the functions you want to include in your current scope. (Dot-sourcing is covered in more depth in the following section.) After you do this, you can call the function directly, as shown here.

```
PS C:\> . C:\scripts\ConversionFunctions.ps1
PS C:\> convertToMiles 6
6 kilometers equals 3.7266 miles
```

All of the functions from the dot-sourced script are available to the current session. This can be demonstrated by creating a listing of the function drive, as shown here.

```
PS C:\> dir function: | Where { $_.name -like 'conv*' } |
Format-Table -Property name, definition -AutoSize
```

Name	Definition
----	-----
ConvertToMeters	param(\$feet) "\$feet feet equals \$(\$feet*.31) meters"...

```

ConvertToFeet      param($meters) "$meters meters equals $($meters * 3.28) feet"...
ConvertToFahrenheit param($celsius) "$celsius celsius equals $((1.8 * $celsius) + 32 )
fahrenheit"...
ConvertToCelsius   param($fahrenheit) "$fahrenheit fahrenheit equals $( ( ($fahrenheit -
32)/9)*5 ) celsius..."
ConvertToMiles     param($kilometer) "$kilometer kilometers equals $( ($kilometer *.6211) )
miles"...
ConvertToKilometers param($miles) "$miles miles equals $( ($miles * 1.61) ) kilometers"...

```

Including functions in the Windows PowerShell environment

In Windows PowerShell 1.0, you could include functions from previously written scripts by dot-sourcing the script. The use of a module, which was introduced in Windows PowerShell 2.0, offers greater flexibility than dot-sourcing because you can create a *module manifest*, which specifies exactly which functions and programming elements will be imported into the current session.

Using dot-sourcing

This technique of dot-sourcing still works in Windows PowerShell 5.0, and it offers the advantage of simplicity and familiarity. In the TextFunctions.ps1 script shown following, two functions are created. The first function is called *New-Line*, and the second is called *Get-TextStats*. The TextFunctions.ps1 script is shown here.

TextFunctions.ps1

```

Function New-Line([string]$stringIn)
{
    "-" * $stringIn.length
} #end New-Line

Function Get-TextStats([string[]]$textIn)
{
    $textIn | Measure-Object -Line -word -char
} #end Get-TextStats

```

The *New-Line* function creates a string of hyphen characters as long as the length of the input text. This is helpful when you want an underline that is sized to the text, for text separation purposes. An example of using the *New-Line* text function in this manner is shown here.

CallNew-LineTextFunction.ps1

```

Function New-Line([string]$stringIn)
{
    "-" * $stringIn.length
} #end New-Line

Function Get-TextStats([string[]]$textIn)
{
    $textIn | Measure-Object -Line -word -char
} #end Get-TextStats

# *** Entry Point to script ***
"This is a string" | ForEach-Object {$_ ; New-Line $_}

```

When the script runs, it returns the following output.

```
This is a string
-----
```

Of course, this is a bit inefficient and limits your ability to use the functions. If you have to copy the entire text of a function into each new script you want to produce, or edit a script each time you want to use a function in a different manner, you dramatically increase your workload. If the functions were available all the time, you might be inclined to use them more often. To make the text functions available in your current Windows PowerShell console, you need to dot-source the script containing the functions into your console, put it in a module, or load it via your profile. You will need to use the entire path to the script unless the folder that contains the script is in your search path. The syntax to dot-source a script is so easy that it actually becomes a stumbling block for some people who are expecting some complex formula or cmdlet with obscure parameters. It is none of that—just a period (dot), followed by a space, followed by the path to the script that contains the function. This is why it is called dot-sourcing: you have a dot and the source (path) to the functions you want to include. This is shown here.

```
PS C:\> . C:\fso\TextFunctions.ps1
```

After you have included the functions in your current console, all the functions in the source script are added to the Function drive. This is shown in Figure 6-1.

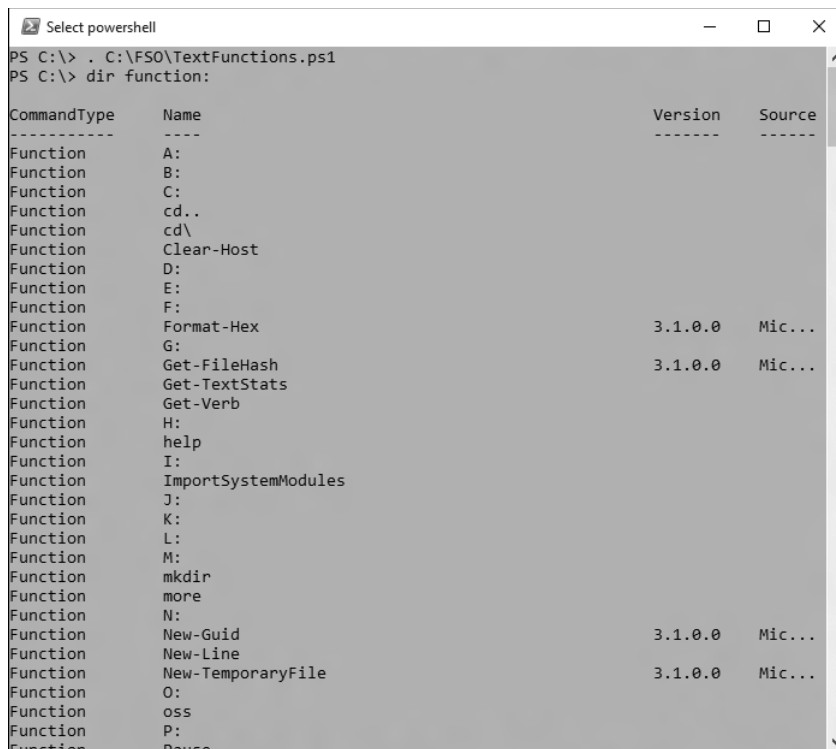


FIGURE 6-1 Functions from a dot-sourced script are available via the Function drive.

Using dot-sourced functions

After the functions have been introduced to the current console, you can incorporate them into your normal commands. This flexibility should also influence the way you write the function. If the functions are written so they will accept pipelined input and do not change the system environment—by adding global variables, for example—you will be much more likely to use the functions, and they will be less likely to conflict with either functions or cmdlets that are present in the current console.

As an example of using the *New-Line* function, consider the fact that the *Get-CimInstance* cmdlet allows the use of an array of computer names for the *-ComputerName* parameter. In this example, BIOS information is obtained from two separate workstations. This is shown here.

```
PS C:\> Get-CimInstance win32_bios -ComputerName dc1, c10
```

```
SMBIOSBIOSVersion : 090006
Manufacturer       : American Megatrends Inc.
Name               : BIOS Date: 05/23/12 17:15:53 Ver: 09.00.06
SerialNumber       : 5198-1332-9667-8393-5778-4501-39
Version            : VRTUAL - 5001223
PSComputerName     : c10
```

```
SMBIOSBIOSVersion : Hyper-V UEFI Release v1.0
Manufacturer       : Microsoft Corporation
Name               : Hyper-V UEFI Release v1.0
SerialNumber       : 3601-6926-9922-0181-5225-8175-58
Version            : VRTUAL - 1
PSComputerName     : dc1
```

You can improve the display of the information returned by *Get-CimInstance* by pipelining the output to the *New-Line* function so that you can underline each computer name as it comes across the pipeline. You do not need to write a script to produce this kind of display. You can enter the command directly into the Windows PowerShell console. The first thing you need to do is to dot-source the *TextFunctions.ps1* script. This makes the functions directly available in the current Windows PowerShell console session. You then use the same *Get-CimInstance* query you used earlier to obtain BIOS information via WMI from two computers. Pipeline the resulting management objects to the *ForEach-Object* cmdlet. Inside the script block section, you use the *\$_* automatic variable to reference the current object on the pipeline and retrieve the *PSComputerName* property. You send this information to the *New-Line* function so the server name is underlined, and you display the BIOS information that is contained in the *\$_* variable.

The command to import the *New-Line* function into the current Windows PowerShell session and use it to underline the server names is shown here.

```
PS C:\> . C:\fso\TextFunctions.ps1
PS C:\> Get-CimInstance win32_bios -ComputerName dc1, c10 | ForEach-Object { $_.PSComputerName
; New-Line $_.PSComputerName ; $_}
```

The results of using the *New-Line* function are shown in Figure 6-2.

```

PS C:\> Get-CimInstance win32_bios -ComputerName dc1, c10 | ForEach-Object { $_.PSComputerName ; New-Line $_.PSComputerName ; $_ }
c10
---
SMBIOSBIOSVersion : 090006
Manufacturer       : American Megatrends Inc.
Name               : BIOS Date: 05/23/12 17:15:53 Ver: 09.00.06
SerialNumber       : 5198-1332-9667-8393-5778-4501-39
Version            : VRTUAL - 5001223
PSComputerName     : c10

dc1
---
SMBIOSBIOSVersion : Hyper-V UEFI Release v1.0
Manufacturer       : Microsoft Corporation
Name               : Hyper-V UEFI Release v1.0
SerialNumber       : 3601-6926-9922-0181-5225-8175-58
Version            : VRTUAL - 1
PSComputerName     : dc1

PS C:\>

```

FIGURE 6-2 Functions that are written to accept pipelined input find an immediate use in your daily work routine.

The *Get-TextStats* function from the *TextFunctions.ps1* script provides statistics based upon an input text file or text string. After the *TextFunctions.ps1* script is dot-sourced into the current console, the statistics it returns when the function is called are word count, number of lines in the file, and number of characters. An example of using this function is shown here.

```
Get-TextStats "This is a string"
```

When the *Get-TextStats* function is used, the following output is produced.

Lines	Words	Characters	Property
-----	-----	-----	-----
1	4	16	

In this section, the use of functions was discussed. The reuse of functions could be as simple as copying the text of the function from one script into another script. It is easier, however, to dot-source the function than to reuse it. This can be done from within the Windows PowerShell console or from within a script.

Adding help for functions

When you dot-source functions into the current Windows PowerShell console, one problem is introduced. Because you are not required to open the file that contains the function to use it, you might be unaware of everything the file contains within it. In addition to functions, the file could contain variables, aliases, Windows PowerShell drives, or any number of other things. Depending on what you are actually trying to accomplish, this might or might not be an issue. The need sometimes arises, however, to have access to help information about the features provided by the Windows PowerShell script.

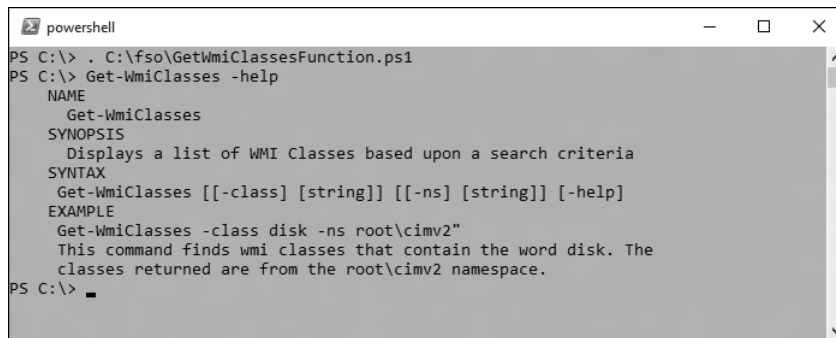
Using a *here-string* object for help

In Windows PowerShell 1.0, you could solve this problem by adding a *help* parameter to the function and storing the help text within a *here-string* object. You can also use this approach in Windows PowerShell 5.0, but as shown in Chapter 7, “Creating advanced functions and modules,” there is a better approach to providing help for functions. The classic *here-string* approach for help is shown in the `GetWmiClassesFunction.ps1` script, which follows. The first step that needs to be done is to define a switch parameter named `$help`. The second step involves creating and displaying the results of a *here-string* object that includes help information. The `GetWmiClassesFunction.ps1` script is shown here.

`GetWmiClassesFunction.ps1`

```
Function Get-WmiClasses(
    $class=($paramMissing=true),
    $ns="root\cimv2",
    [switch]$help
)
{
    If($help)
    {
        $helpstring = @"
NAME
    Get-WmiClasses
SYNOPSIS
    Displays a list of WMI Classes based upon a search criteria
SYNTAX
    Get-WmiClasses [[-class] [string]] [[-ns] [string]] [-help]
EXAMPLE
    Get-WmiClasses -class disk -ns root\cimv2"
        This command finds wmi classes that contain the word disk. The
        classes returned are from the root\cimv2 namespace.
"@
        $helpString
        break #exits the function early
    }
    If($local:paramMissing)
    {
        throw "USAGE: Get-WmiClasses -class <class type> -ns <wmi namespace>"
    } #local:paramMissing
    "`nClasses in $ns namespace ...."
    Get-WmiObject -namespace $ns -list |
    Where-Object {
        $_.name -match $class -and `
        $_.name -notlike 'cim*'
    }
    #
} #end get-wmiclasses
```

The *here-string* technique works pretty well for providing function help if you follow the cmdlet help pattern. This is shown in Figure 6-3.



```
PS C:\> . C:\fso\GetWmiClassesFunction.ps1
PS C:\> Get-WmiClasses -help
NAME
    Get-WmiClasses
SYNOPSIS
    Displays a list of WMI Classes based upon a search criteria
SYNTAX
    Get-WmiClasses [[-class] [string]] [[-ns] [string]] [-help]
EXAMPLE
    Get-WmiClasses -class disk -ns root\cimv2"
    This command finds wmi classes that contain the word disk. The
    classes returned are from the root\cimv2 namespace.
PS C:\>
```

FIGURE 6-3 Manually created help can mimic the look of core cmdlet help.

The drawback with manually creating help for a function is that it is tedious, and as a result, only the most important functions receive help information when you use this methodology. This is unfortunate, because it then requires the user to memorize the details of the function contract. One way to work around this is to use the *Get-Content* cmdlet to retrieve the code that was used to create the function. This is much easier than searching for the script that was used to create the function and opening it up in Notepad. To use the *Get-Content* cmdlet to display the contents of a function, you enter *Get-Content* and supply the path to the function. All functions available to the current Windows PowerShell environment are available via the Function Windows PowerShell drive. You can therefore use the following syntax to obtain the content of a function.

```
PS C:\> Get-Content Function:\Get-WmiClasses
```

The technique of using *Get-Content* to read the text of the function is shown in Figure 6-4.

An easier way to add help, by using comment-based help, is discussed in Chapter 7. Comment-based help, although more complex than the method discussed here, offers a number of advantages—primarily due to the integration with the Windows PowerShell help subsystem. When you add comment-based help, users of your function can access your help in exactly the same manner as for any of the core Windows PowerShell cmdlets.

```

PS C:\> Get-Content Function:\Get-WmiClasses
param($class=($paramMissing=$true), $ns="root\cimv2", [switch]$help)

If($help)
{
    $helpstring = @"
    NAME
        Get-WmiClasses
    SYNOPSIS
        Displays a list of WMI Classes based upon a search criteria
    SYNTAX
        Get-WmiClasses [[-class] [string]] [[-ns] [string]] [-help]
    EXAMPLE
        Get-WmiClasses -class disk -ns root\cimv2"
    This command finds wmi classes that contain the word disk. The
    classes returned are from the root\cimv2 namespace.
"@
    $helpString
    break #exits the function early
}
If($local:paramMissing)
{
    throw "USAGE: getwmi2 -class <class type> -ns <wmi namespace>"
} # $local:paramMissing
""nClasses in $ns namespace ..."
Get-WmiObject -namespace $ns -list |
Where-Object {
    $_.name -match $class -and `
    $_.name -notlike 'cim*'
}

#
PS C:\>

```

FIGURE 6-4 The *Get-Content* cmdlet can retrieve the contents of a function.

Using two input parameters

To create a function that uses multiple input parameters, you use the *Function* keyword, specify the name of the function, use a variable for each input parameter, and then define the script block within the braces. The pattern is shown here.

```

Function My-Function($Input1,$Input2)
{
    #Insert Code Here
}

```

An example of a function that takes multiple parameters is the *Get-FreeDiskSpace* function, which is shown in the *Get-FreeDiskSpace.ps1* script at the end of this section.

The *Get-FreeDiskSpace.ps1* script begins with the *Function* keyword and is followed by the name of the function and the two input parameters. The input parameters are placed inside parentheses, as shown here.

```

Function Get-FreeDiskSpace($drive,$computer)

```

Inside the function's script block, the *Get-FreeDiskSpace* function uses the *Get-WmiObject* cmdlet to query the *Win32_LogicalDisk* WMI class. It connects to the computer specified in the *\$computer* parameter, and it filters out only the drive that is specified in the *\$drive* parameter. When the function

is called, each parameter is specified as *-drive* and *-computer*. In the function definition, the variables *\$drive* and *\$computer* are used to hold the values supplied to the parameters.

After the data from WMI is retrieved, it is stored in the *\$driveData* variable. The data that is stored in the *\$driveData* variable is an instance of the *Win32_LogicalDisk* class. This variable contains a complete instance of the class. The members of this class are shown in Table 6-1.

TABLE 6-1 Members of the *Win32_LogicalDisk* class

Name	Member type	Definition
<i>Chkdsk</i>	Method	<i>System.Management.ManagementBaseObject Chkdsk(System.Boolean FixErrors, System.Boolean VigorousIndexCheck, System.Boolean SkipFolderCycle, System.Boolean ForceDismount, System.Boolean RecoverBadSectors, System.Boolean OkToRunAtBootUp)</i>
<i>Reset</i>	Method	<i>System.Management.ManagementBaseObject Reset()</i>
<i>SetPowerState</i>	Method	<i>System.Management.ManagementBaseObject SetPowerState(System.UInt16 PowerState, System.String Time)</i>
<i>Access</i>	Property	<i>System.UInt16 Access {get;set;}</i>
<i>Availability</i>	Property	<i>System.UInt16 Availability {get;set;}</i>
<i>BlockSize</i>	Property	<i>System.UInt64 BlockSize {get;set;}</i>
<i>Caption</i>	Property	<i>System.String Caption {get;set;}</i>
<i>Compressed</i>	Property	<i>System.Boolean Compressed {get;set;}</i>
<i>ConfigManagerErrorCode</i>	Property	<i>System.UInt32 ConfigManagerErrorCode {get;set;}</i>
<i>ConfigManagerUserConfig</i>	Property	<i>System.Boolean ConfigManagerUserConfig {get;set;}</i>
<i>CreationClassName</i>	Property	<i>System.String CreationClassName {get;set;}</i>
<i>Description</i>	Property	<i>System.String Description {get;set;}</i>
<i>DeviceID</i>	Property	<i>System.String DeviceID {get;set;}</i>
<i>DriveType</i>	Property	<i>System.UInt32 DriveType {get;set;}</i>
<i>ErrorCleared</i>	Property	<i>System.Boolean ErrorCleared {get;set;}</i>
<i>ErrorDescription</i>	Property	<i>System.String ErrorDescription {get;set;}</i>
<i>ErrorMethodology</i>	Property	<i>System.String ErrorMethodology {get;set;}</i>
<i>FileSystem</i>	Property	<i>System.String FileSystem {get;set;}</i>
<i>FreeSpace</i>	Property	<i>System.UInt64 FreeSpace {get;set;}</i>
<i>InstallDate</i>	Property	<i>System.String InstallDate {get;set;}</i>
<i>LastErrorCode</i>	Property	<i>System.UInt32 LastErrorCode {get;set;}</i>
<i>MaximumComponentLength</i>	Property	<i>System.UInt32 MaximumComponentLength {get;set;}</i>
<i>MediaType</i>	Property	<i>System.UInt32 MediaType {get;set;}</i>
<i>Name</i>	Property	<i>System.String Name {get;set;}</i>
<i>NumberOfBlocks</i>	Property	<i>System.UInt64 NumberOfBlocks {get;set;}</i>
<i>PNPDeviceID</i>	Property	<i>System.String PNPDeviceID {get;set;}</i>

Name	Member type	Definition
<i>PowerManagementCapabilities</i>	Property	<i>System.UInt16[] PowerManagementCapabilities {get;set;}</i>
<i>PowerManagementSupported</i>	Property	<i>System.Boolean PowerManagementSupported {get;set;}</i>
<i>ProviderName</i>	Property	<i>System.String ProviderName {get;set;}</i>
<i>Purpose</i>	Property	<i>System.String Purpose {get;set;}</i>
<i>QuotasDisabled</i>	Property	<i>System.Boolean QuotasDisabled {get;set;}</i>
<i>QuotasIncomplete</i>	Property	<i>System.Boolean QuotasIncomplete {get;set;}</i>
<i>QuotasRebuilding</i>	Property	<i>System.Boolean QuotasRebuilding {get;set;}</i>
<i>Size</i>	Property	<i>System.UInt64 Size {get;set;}</i>
<i>Status</i>	Property	<i>System.String Status {get;set;}</i>
<i>StatusInfo</i>	Property	<i>System.UInt16 StatusInfo {get;set;}</i>
<i>SupportsDiskQuotas</i>	Property	<i>System.Boolean SupportsDiskQuotas {get;set;}</i>
<i>SupportsFileBasedCompression</i>	Property	<i>System.Boolean SupportsFileBasedCompression {get;set;}</i>
<i>SystemCreationClassName</i>	Property	<i>System.String SystemCreationClassName {get;set;}</i>
<i>SystemName</i>	Property	<i>System.String SystemName {get;set;}</i>
<i>VolumeDirty</i>	Property	<i>System.Boolean VolumeDirty {get;set;}</i>
<i>VolumeName</i>	Property	<i>System.String VolumeName {get;set;}</i>
<i>VolumeSerialNumber</i>	Property	<i>System.String VolumeSerialNumber {get;set;}</i>
<i>__CLASS</i>	Property	<i>System.String __CLASS {get;set;}</i>
<i>__DERIVATION</i>	Property	<i>System.String[] __DERIVATION {get;set;}</i>
<i>__DYNASTY</i>	Property	<i>System.String __DYNASTY {get;set;}</i>
<i>__GENUS</i>	Property	<i>System.Int32 __GENUS {get;set;}</i>
<i>__NAMESPACE</i>	Property	<i>System.String __NAMESPACE {get;set;}</i>
<i>__PATH</i>	Property	<i>System.String __PATH {get;set;}</i>
<i>__PROPERTY_COUNT</i>	Property	<i>System.Int32 __PROPERTY_COUNT {get;set;}</i>
<i>__RELPATH</i>	Property	<i>System.String __RELPATH {get;set;}</i>
<i>__SERVER</i>	Property	<i>System.String __SERVER {get;set;}</i>
<i>__SUPERCLASS</i>	Property	<i>System.String __SUPERCLASS {get;set;}</i>
<i>PSStatus</i>	Property set	<i>PSStatus {Status, Availability, DeviceID, StatusInfo}</i>
<i>ConvertFromDateTime</i>	Script method	<i>System.Object ConvertFromDateTime();</i>
<i>ConvertToDateTime</i>	Script method	<i>System.Object ConvertToDateTime();</i>

When you have the data stored in the *\$driveData* variable, you will want to print some information to the user of the script. The first thing to do is print the name of the computer and the name of the drive. To do this, you can place the variables inside double quotation marks. Double quotation marks

denote expanding strings, and variables placed inside double quotation marks emit their value, not their name. This is shown here.

```
"$computer free disk space on drive $drive"
```

The next thing you will want to do is format the data that is returned. To do this, use the Microsoft .NET Framework format strings to specify two decimal places. You will need to use a subexpression to prevent the unraveling of the WMI object inside the expanding-string double quotation marks. The subexpression uses the dollar sign and a pair of parentheses to force the evaluation of the expression before returning the data to the string. This is shown here.

Get-FreeDiskSpace.ps1

```
Function Get-FreeDiskSpace($drive,$computer)
{
    $driveData = Get-WmiObject -class win32_LogicalDisk `
    -computername $computer -filter "Name = '$drive'"
    "
    $computer free disk space on drive $drive
    "${0:n2}" -f ($driveData.FreeSpace/1MB)) MegaBytes
    "
}
```

```
Get-FreeDiskSpace -drive "C:" -computer "C10"
```

Obtaining specific WMI data

Though storing the complete instance of the object in the *\$driveData* variable is a bit inefficient due to the amount of data it contains, in reality the class is rather small, and the ease of using the *Get-WmiObject* cmdlet is usually worth the wasteful methodology. If performance is a primary consideration, the use of the *[wmi]* type accelerator would be a better solution. To obtain the free disk space by using this method, you would use the following syntax.

```
([wmi]"Win32_LogicalDisk.DeviceID='c:').FreeSpace
```

To put the preceding command into a usable function, you would need to substitute the hard-coded drive letter for a variable. In addition, you would want to modify the class constructor to receive a path to a remote computer. The newly created function is contained in the *Get-DiskSpace.ps1* script, shown here.

Get-DiskSpace.ps1

```
Function Get-DiskSpace($drive,$computer)
{
    ([wmi]"\\$computer\root\cimv2:Win32_LogicalDisk.DeviceID='$drive'").FreeSpace
}
Get-DiskSpace -drive "C:" -computer "Office"
```

After you have made the preceding changes, the code only returns the value of the *FreeSpace* property from the specific drive. If you were to send the output to *Get-Member*, you would find that you have an integer. This technique is more efficient than storing an entire instance of the *Win32_LogicalDisk* class and then selecting a single value.

Using a type constraint in a function

When you are accepting parameters for a function, it might be important to use a type constraint to ensure that the function receives the correct type of data. To do this, you place the name of the type you want inside brackets in front of the input parameter. This constrains the data type and prevents the entry of an incorrect type of data. Frequently used type accelerators are shown in Table 6-2.

TABLE 6-2 Data type aliases

Alias	Type
[int]	32-bit signed integer
[long]	64-bit signed integer
[string]	Fixed-length string of Unicode characters
[char]	Unicode 16-bit character
[bool]	True/false value
[byte]	8-bit unsigned integer
[double]	Double-precision 64-bit floating-point number
[decimal]	128-bit decimal value
[single]	Single-precision 32-bit floating-point number
[array]	Array of values
[xml]	XML object
[hashtable]	Hashtable object (similar to a dictionary object)

In the *Resolve-ZipCode* function, which is shown in the following *Resolve-ZipCode.ps1* script, the *\$zip* input parameter is constrained to allow only a 32-bit signed integer for input. (Obviously, the *[int]* type constraint would eliminate most of the world's postal codes, but the web service the script uses only resolves US-based postal codes, so it is a good addition to the function.)

In the *Resolve-ZipCode* function, the first thing that is done is to use a string that points to the WSDL (Web Services Description Language) for the web service. Next, the *New-WebServiceProxy* cmdlet is used to create a new web service proxy for the ZipCode service. The WSDL for the ZipCode service defines a method called the *GetInfoByZip* method. It will accept a standard US-based postal code. The results are displayed as a table. The *Resolve-ZipCode.ps1* script is shown here.

Resolve-ZipCode.ps1

```
#Requires -Version 5.0
Function Resolve-ZipCode([int]$zip)
{
    $URI = "http://www.webservices.net/uszip.asmx?WSDL"
    $zipProxy = New-WebServiceProxy -uri $URI -namespace WebServiceProxy -class ZipClass
    $zipProxy.getinfobyzip($zip).table
} #end Get-ZipCode
```

Resolve-ZipCode 28273

When you use a type constraint on an input parameter, any deviation from the expected data type will generate an error similar to the one shown here.

```
Resolve-ZipCode : Cannot process argument transformation on parameter 'zip'. Cannot convert
value "COW" to type "System
.Int32". Error: "Input string was not in a correct format."
At C:\Users\ed\AppData\Local\Temp\tmp3351.tmp.ps1:22 char:16
+ Resolve-ZipCode <<<< "COW"
+ ~~~~~
+ CategoryInfo          : InvalidData: (:) [Resolve-ZipCode],
ParameterBindin...mationException
+ FullyQualifiedErrorId : ParameterArgumentTransformationError,Resolve-ZipCode
```

Needless to say, such an error could be distracting to the users of the function. One way to handle the problem of confusing error messages is to use the *Trap* keyword. In the `DemoTrapSystemException.ps1` script, the *My-Test* function uses *[int]* to constrain the *\$myinput* variable to accept only a 32-bit unsigned integer for input. If such an integer is received by the function when it is called, the function will return the string *It worked*. If the function receives a string for input, an error will be raised, similar to the one shown previously.

Rather than display a raw error message, which most users and many IT professionals find confusing, it is a best practice to suppress the display of the error message, and perhaps inform the user that an error condition has occurred and provide more meaningful and direct information that the user can then relay to the help desk. Many times, IT departments will display such an error message, complete with either a local telephone number for the appropriate help desk, or even a link to an internal webpage that provides detailed troubleshooting and corrective steps the user can perform. You could even provide a webpage that hosted a script that the user could run to fix the problem. This is similar to the “Fix it for me” webpages Microsoft introduced.

When an instance of a *System.SystemException* class occurs (when a system exception occurs), the *Trap* statement will trap the error, rather than allowing it to display the error information on the screen. If you were to query the *\$error* variable, you would find that the error had in fact occurred and was actually received by the error record. You would also have access to the *ErrorRecord* class via the *\$_* automatic variable, which means that the error record has been passed along the pipeline. This gives you the ability to build a rich error-handling solution. In this example, the string *error trapped* is displayed, and the *Continue* statement is used to continue the script execution on the next line of code. In this example, the next line of code that is executed is the *After the error* string. When the `DemoTrapSystemException.ps1` script is run, the following output is shown.

```
error trapped
After the error
```

The complete DemoTrapSystemException.ps1 script is shown here.

```
DemoTrapSystemException.ps1
Function My-Test([int]$myinput)
{

    "It worked"
} #End my-test function
# *** Entry Point to Script ***

Trap [SystemException] { "error trapped" ; continue }
My-Test -myinput "string"
"After the error"
```

Using more than two input parameters

When using more than two input parameters, I consider it a best practice to modify the way the function is structured. This not only makes the function easier to read, it also permits cmdlet binding. In the basic function pattern shown here, the function accepts three input parameters. When you consider the default values and the type constraints, you can tell that the parameters begin to become long. Moving them to the inside of the function body highlights the fact that they are input parameters, and it makes them easier to read, understand, and maintain. It also allows for decorating the parameters with attributes.

```
Function Function-Name
{
    Param(
        [int]$Parameter1,
        [String]$Parameter2 = "DefaultValue",
        $Parameter3
    )
    #Function code goes here
} #end Function-Name
```

An example of a function that uses three input parameters is the *Get-DirectoryListing* function. With the type constraints, default values, and parameter names, the function signature would be rather cumbersome to include on a single line. This is shown here.

```
Function Get-DirectoryListing ([String]$Path,[String]$Extension = "txt",[Switch]$Today)
```

If the number of parameters were increased to four, or if a default value for the *-Path* parameter was wanted, the signature would easily scroll to two lines. The use of the *Param* statement inside the function body also provides the ability to specify input parameters to a function.



Note The use of the *Param* statement inside the function body is often regarded as a personal preference. It requires additional work, and often leaves the reader of the script wondering why this was done. When there are more than two parameters, visually the *Param* statement stands out, and it is obvious why it was done in this particular manner. But, as will be shown in Chapter 7, using the *Param* statement is the only way to gain access to advanced function features such as cmdlet binding, parameter attributes, and other powerful features of Windows PowerShell.

Following the *Function* keyword, the name of the function, and the opening script block, the *Param* keyword is used to identify the parameters for the function. Each parameter must be separated from the others by a comma. All the parameters must be surrounded with a set of parentheses. If you want to assign a default value for a parameter, such as the extension *.txt* for the *Extension* parameter in the *Get-DirectoryListing* function, you perform a straight value assignment followed by a comma.

In the *Get-DirectoryListing* function, the *Today* parameter is a switch parameter. When it is supplied to the function, only files written to since midnight on the day the script is run will be displayed. If it is not supplied, all files matching the extension in the folder will be displayed. The *Get-DirectoryListing-Today.ps1* script is shown here.

Get-DirectoryListingToday.ps1

```
Function Get-DirectoryListing
{
    Param(
        [String]$Path,
        [String]$Extension = "txt",
        [Switch]$Today
    )
    If($Today)
    {
        Get-ChildItem -Path $path\* -include *.$Extension |
        Where-Object { $_.LastWriteTime -ge (Get-Date).Date }
    }
    ELSE
    {
        Get-ChildItem -Path $path\* -include *.$Extension
    }
} #end Get-DirectoryListing

# *** Entry to script ***
Get-DirectoryListing -p c:\fso -t
```



Note As a best practice, you should avoid creating functions that have a large number of input parameters. It is very confusing. When you find yourself creating a large number of input parameters, you should ask if there is a better way to do things. It might be an indicator that you do not have a single-purpose function. In the *Get-DirectoryListing* function, I have a switch parameter that will filter the files returned by the ones written to today. If I were writing the script for production use, instead of just to demonstrate multiple function parameters, I would have created another function called something like *Get-FilesByDate*. In that function, I would have a *Today* switch, and a *Date* parameter to allow a selectable date for the filter. This separates the data-gathering function from the filter/presentation function. See the “Using functions to provide ease of modification” section later in this chapter for more discussion of this technique.

Using functions to encapsulate business logic

There are two kinds of logic with which script writers need to be concerned. The first is program logic, and the second is business logic. *Program logic* includes the way the script works, the order in which things need to be done, and the requirements of code used in the script. An example of program logic is the requirement to open a connection to a database before querying the database.

Business logic is something that is a requirement of the business, but not necessarily a requirement of the program or script. The script can often operate just fine regardless of the particulars of the business rule. If the script is designed properly, it should operate perfectly fine no matter what gets supplied for the business rules.

In the *BusinessLogicDemo.ps1* script, a function called *Get-Discount* is used to calculate the discount to be granted to the total amount. One good thing about encapsulating the business rules for the discount into a function is that as long as the contract between the function and the calling code does not change, you can drop any kind of convoluted discount schedule that the business decides to come up with into the script block of the *Get-Discount* function—including database calls to determine on-hand inventory, time of day, day of week, total sales volume for the month, the buyer’s loyalty level, and the square root of some random number that is used to determine an instant discount rate.

So, what is the contract with the function? The contract with the *Get-Discount* function says, “If you give me a rate number as a type of *system.double* and a total as an integer, I will return to you a number that represents the total discount to be applied to the sale.” As long as you adhere to that contract, you never need to modify the code.

The *Get-Discount* function begins with the *Function* keyword and is followed by the name of the function and the definition for two input parameters. The first input parameter is the *\$rate* parameter, which is constrained to be of type *system.double* (which will permit you to supply decimal numbers). The second input parameter is the *\$total* parameter, which is constrained to be of type *system.integer*,

and therefore will not allow decimal numbers. In the script block, the value of the *-total* parameter is multiplied by the value of the *-rate* parameter. The result of this calculation is returned to the pipeline.

The *Get-Discount* function is shown here.

```
Function Get-Discount([double]$rate,[int]$total)
{
    $rate * $total
} #end Get-Discount
```

The entry point to the script assigns values to both the *\$total* and *\$rate* variables, as shown here.

```
$rate = .05
$total = 100
```

The variable *\$discount* is used to hold the result of the calculation from the *Get-Discount* function. When calling the function, it is a best practice to use the full parameter names. It makes the code easier to read and will help make it immune to unintended problems if the function signature ever changes.

```
$discount = Get-Discount -rate $rate -total $total
```



Note The signature of a function consists of the order and names of the input parameters. If you typically supply values to the signature via positional parameters, and the order of the input parameters changes, the code will fail, or worse yet, produce inconsistent results. If you typically call functions via partial parameter names, and an additional parameter is added, the script will fail due to difficulty with the disambiguation process. Obviously, you take this into account when first writing the script and the function, but months or years later, when you are making modifications to the script or calling the function via another script, the problem can arise.

The remainder of the script produces output for the screen. The results of running the script are shown here.

```
Total: 100
Discount: 5
Your Total: 95
```

The complete text of the *BusinessLogicDemo.ps1* script is shown here.

```
BusinessLogicDemo.ps1
Function Get-Discount([double]$rate,[int]$total)
{
    $rate * $total
} #end Get-Discount

$rate = .05
$total = 100
```

```
$discount = Get-Discount -rate $rate -total $total
"Total: $total"
"Discount: $discount"
"Your Total: $($total-$discount)"
```

Business logic does not have to be related to business purposes. Business logic is anything that is arbitrary that does not affect the running of the code. In the `FindLargeDocs.ps1` script, there are two functions. The first function, *Get-Doc*, is used to find document files (files with an extension of *.doc*, *.docx*, or *.dot*) in a folder that is passed to the function when it is called. The *-Recurse* switch parameter, when used with the *Get-ChildItem* cmdlet, causes the function to look in the present folder, and within child folders. This function is a stand-alone function and has no dependency on any other functions.

The *LargeFiles* piece of code is a filter. A filter is a kind of special-purpose function that uses the *Filter* keyword rather than the *Function* keyword when it is created. (For more information on filters, see the “Understanding filters” section later in this chapter.) The `FindLargeDocs.ps1` script is shown here.

```
FindLargeDocs.ps1
Function Get-Doc($path)
{
    Get-ChildItem -Path $path -include *.doc,*.docx,*.dot -recurse
} #end Get-Doc

Filter LargeFiles($size)
{
    $_ |
    Where-Object { $_.length -ge $size }
} #end LargeFiles

Get-Doc("C:\FS0") | LargeFiles 1000
```

Using functions to provide ease of modification

It is a truism that a script is never completed. There is always something else to add to a script—a change that will improve it, or additional functionality that someone requests. When a script is written as one long piece of inline code, without recourse to functions, it can be rather tedious and error-prone to modify.

An example of an inline script is the `InLineGetIPDemo.ps1` script. The first line of code uses the *Get-WmiObject* cmdlet to retrieve the instances of the *Win32_NetworkAdapterConfiguration* WMI class that IP enabled. The results of this WMI query are stored in the *\$IP* variable. This line of code is shown here.

```
$IP = Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
```

When the WMI information has been obtained and stored, the remainder of the script prints information to the screen. The *IPAddress*, *IPSubNet*, and *DNSServerSearchOrder* properties are all stored in an array. For this example, you are only interested in the first IP address, and you therefore print

element 0, which will always exist if the network adapter has an IP address. This section of the script is shown here.

```
"IP Address: " + $IP.IPAddress[0]
"Subnet: " + $IP.IPSubNet[0]
"GateWay: " + $IP.DefaultIPGateway
"DNS Server: " + $IP.DNSServerSearchOrder[0]
"FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
```

When the script is run, it produces output similar to the following.

```
IP Address: 192.168.2.5
Subnet: 255.255.255.0
GateWay: 192.168.2.1
DNS Server: 192.168.2.1
FQDN: w8client1.nwtraders.com
```

The complete `InLineGetIPDemo.ps1` script is shown here.

`InLineGetIPDemo.ps1`

```
$IP = Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
"IP Address: " + $IP.IPAddress[0]
"Subnet: " + $IP.IPSubNet[0]
"GateWay: " + $IP.DefaultIPGateway
"DNS Server: " + $IP.DNSServerSearchOrder[0]
"FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
```

With just a few modifications to the script, a great deal of flexibility can be obtained. The modifications, of course, involve moving the inline code into functions. As a best practice, a function should be narrowly defined and should encapsulate a single thought. Though it would be possible to move the entire previous script into a function, you would not have as much flexibility. There are two thoughts or ideas that are expressed in the script. The first is obtaining the IP information from WMI, and the second is formatting and displaying the IP information. It would be best to separate the gathering and the displaying processes from one another, because they are logically two different activities.

To convert the `InLineGetIPDemo.ps1` script into a script that uses a function, you only need to add the *Function* keyword, give the function a name, and surround the original code with a pair of braces. The transformed script is now named `GetIPDemoSingleFunction.ps1` and is shown here.

`GetIPDemoSingleFunction.ps1`

```
Function Get-IPDemo
{
    $IP = Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
    "IP Address: " + $IP.IPAddress[0]
    "Subnet: " + $IP.IPSubNet[0]
    "GateWay: " + $IP.DefaultIPGateway
    "DNS Server: " + $IP.DNSServerSearchOrder[0]
    "FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
} #end Get-IPDemo

# *** Entry Point To Script ***
```

`Get-IPDemo`

If you go to all the trouble to transform the inline code into a function, what benefit do you derive? By making this single change, your code will become

- Easier to read
- Easier to understand
- Easier to reuse
- Easier to troubleshoot

The script is easier to read because you do not really need to read each line of code to understand what it does. You can tell that there is a function that obtains the IP address, and it is called from outside the function. That is all the script does.

The script is easier to understand because you can tell there is a function that obtains the IP address. If you want to know the details of that operation, you read that function. If you are not interested in the details, you can skip that portion of the code.

The script is easier to reuse because you can dot-source the script, as shown here. When the script is dot-sourced, all the executable code in the script is run.

As a result, because each of the scripts prints information, the following is displayed.

```
IP Address: 192.168.2.5
Subnet: 255.255.255.0
GateWay: 192.168.2.1
DNS Server: 192.168.2.1
FQDN: C10.nwtraders.com
```

```
C10 free disk space on drive C:
48,767.16 MegaBytes
```

```
This OS is version 10.0
```

The DotSourceScripts.ps1 script is shown following. As you can tell, it provides you with a certain level of flexibility to choose the information required, and it also makes it easy to mix and match the required information. If each of the scripts had been written in a more standard fashion, and the output had been more standardized, the results would have been more impressive. As it is, three lines of code produce an exceptional amount of useful output that could be acceptable in a variety of situations.

DotSourceScripts.ps1

```
. C:\Scripts\GetIPDemoSingleFunction.ps1
. C:\Scripts\Get-FreeDiskSpace.ps1
. C:\Scripts\Get-OperatingSystemVersion.ps1
```

A better way to work with the function is to think about the things the function is actually doing. In the FunctionGetIPDemo.ps1 script, there are two functions. The first connects to WMI, which returns a management object. The second function formats the output. These are two completely unrelated

tasks. The first task is data gathering, and the second task is the presentation of the information. The `FunctionGetIPDemo.ps1` script is shown here.

```
FunctionGetIPDemo.ps1

Function Get-IPObject
{
    Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
} #end Get-IPObject

Function Format-IPOutput($IP)
{
    "IP Address: " + $IP.IPAddress[0]
    "Subnet: " + $IP.IPSubNet[0]
    "GateWay: " + $IP.DefaultIPGateway
    "DNS Server: " + $IP.DNSServerSearchOrder[0]
    "FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
} #end Format-IPOutput

# *** Entry Point To Script

$ip = Get-IPObject
Format-IPOutput -ip $ip
```

By separating the data-gathering and the presentation activities into different functions, you gain additional flexibility. You could easily modify the *Get-IPObject* function to look for network adapters that were not IP enabled. To do this, you would need to modify the *-Filter* parameter of the *Get-WmiObject* cmdlet. Because most of the time you would actually be interested only in network adapters that are IP enabled, it would make sense to set the default value of the input parameter to *\$true*. By default, the behavior of the revised function is exactly as it was prior to modification. The advantage is that you can now use the function and modify the objects returned by it. To do this, you supply *\$false* when calling the function. This is illustrated in the `Get-IPObjectDefaultEnabled.ps1` script.

```
Get-IPObjectDefaultEnabled.ps1

Function Get-IPObject([bool]$IPEnabled = $true)
{
    Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $IPEnabled"
} #end Get-IPObject

Get-IPObject -IPEnabled $False
```

By separating the gathering of the information from the presentation of the information, you gain flexibility not only in the type of information that is garnered, but also in the way the information is displayed. When you are gathering network adapter configuration information from a network adapter that is not enabled for IP, the results are not as impressive as for one that is enabled for IP. You might therefore decide to create a different display to list only the pertinent information. Because the function that displays the information is different from the one that gathers the information, a change can easily be made to customize the information that is most germane. The *Begin* section of the function is run once during the execution of the function. This is the perfect place to create a header for the output data. The *Process* section executes once for each item on the pipeline, which in

this example will be each of the non-IP-enabled network adapters. The *Write-Host* cmdlet is used to easily write the data out to the Windows PowerShell console. The backtick-*t* character combination (*`t*) is used to produce a tab.



Note The *`t* character is a string character, and as such it works with cmdlets that accept string input.

The *Get-IPObjectDefaultEnabledFormatNonIPOutput.ps1* script is shown here.

Get-IPObjectDefaultEnabledFormatNonIPOutput.ps1

```
Function Get-IPObject([bool]$IPEnabled = $true)
{
    Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $IPEnabled"
} #end Get-IPObject

Function Format-NonIPOutput($IP)
{
    Begin { "Index #   Description" }
    Process {
        ForEach ($i in $ip)
        {
            Write-Host $i.Index `t $i.Description
        } #end ForEach
    } #end Process
} #end Format-NonIPOutPut

$ip = Get-IPObject -IPEnabled $False
Format-NonIPOutput($ip)
```

You can use the *Get-IPObject* function to retrieve the network adapter configuration, and you can use the *Format-NonIPOutput* and *Format-IPOutput* functions in a script to display the IP information as specifically formatted output, as shown in the *CombinationFormatGetIPDemo.ps1* script shown here.

CombinationFormatGetIPDemo.ps1

```
Function Get-IPObject([bool]$IPEnabled = $true)
{
    Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $IPEnabled"
} #end Get-IPObject

Function Format-IPOutput($IP)
{
    "IP Address: " + $IP.IPAddress[0]
    "Subnet: " + $IP.IPSubNet[0]
    "Gateway: " + $IP.DefaultIPGateway
    "DNS Server: " + $IP.DNSServerSearchOrder[0]
    "FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
} #end Format-IPOutput

Function Format-NonIPOutput($IP)
{
    Begin { "Index #   Description" }
```

```

Process {
    ForEach ($i in $ip)
    {
        Write-Host $i.Index `t $i.Description
    } #end ForEach
} #end Process
} #end Format-NonIPOutPut

# *** Entry Point ***
$IPEnabled = $false
$ip = Get-IPObject -IPEnabled $IPEnabled
If($IPEnabled) { Format-IPOutput($ip) }
ELSE { Format-NonIPOutput($ip) }

```

Understanding filters

A filter is a special-purpose function. It is used to operate on each object in a pipeline and is often used to reduce the number of objects that are passed along the pipeline. Typically, a filter does not use the *Begin* or the *End* parameters that a function might need to use. So a filter is often thought of as a function that only has a *Process* block. Many functions are written without using the *Begin* or *End* parameters, but filters are never written in such a way that they use the *Begin* or the *End* parameters. The biggest difference between a function and a filter is a bit subtler, however. When a function is used inside a pipeline, it actually halts the processing of the pipeline until the first element in the pipeline has run to completion. The function then accepts the input from the first element in the pipeline and begins its processing. When the processing in the function is completed, it then passes the results along to the next element in the script block. A function runs once for the pipelined data. A filter, on the other hand, runs once for each piece of data passed over the pipeline. In short, a filter will stream the data when in a pipeline, and a function will not. This can make a big difference in the performance. To illustrate this point, let's examine a function and a filter that accomplish the same things.

In the `MeasureAddOneFilter.ps1` script, which follows, an array of 50,000 elements is created by using the `1..50000` syntax. (In Windows PowerShell 1.0, 50,000 was the maximum size of an array created in this manner. In Windows PowerShell 5.0, this ceiling has a maximum size of an `[Int32]` (2,146,483,647). The use of this size is dependent upon memory. This is shown here.

```

PS C:\> 1..[Int32]::MaxValue
Array dimensions exceeded supported range.
At line:1 char:1
+ 1..[Int32]::MaxValue
+ ~~~~~
+ CategoryInfo          : OperationStopped: (:) [], OutOfMemoryException
+ FullyQualifiedErrorId : System.OutOfMemoryException

```

The array is then pipelined into the `AddOne` filter. The filter prints out the string *add one filter* and then adds the number 1 to the current number on the pipeline. The length of time it takes to run the command is then displayed. On my computer, it takes about 2.6 seconds to run the `MeasureAddOne-Filter.ps1` script.

MeasureAddOneFilter.ps1

```
Filter AddOne
{
    "add one filter"
    $_ + 1
}
```

```
Measure-Command { 1..50000 | addOne }
```

The function version is shown following. In a similar fashion to the MeasureAddOneFilter.ps1 script, it creates an array of 50,000 numbers and pipelines the results to the *AddOne* function. The string *Add One Function* is displayed. An automatic variable is created when pipelining input to a function. It is called *\$input*. The *\$input* variable is an enumerator, not just a plain array. It has a *moveNext* method, which can be used to move to the next item in the collection. Because *\$input* is not a plain array, you cannot index directly into it—*\$input[0]* would fail. To retrieve a specific element, you use the *\$input.current* property. When I run the following script, it takes 4.3 seconds on my computer (that is almost twice as long as the filter).

MeasureAddOneFunction.ps1

```
Function AddOne
{
    "Add One Function"
    While ($input.moveNext())
    {
        $input.current + 1
    }
}
```

```
Measure-Command { 1..50000 | addOne }
```

What was happening that made the filter so much faster than the function in this example? The filter runs once for each item on the pipeline. This is shown here.

```
add one filter
2
add one filter
3
add one filter
4
add one filter
5
add one filter
6
```

The DemoAddOneFilter.ps1 script is shown here.

DemoAddOneFilter.ps1

```
Filter AddOne
{
    "add one filter"
    $_ + 1
}
```

```
1..5 | addOne
```


The *AddOne* function runs to completion once for all the items in the pipeline. This effectively stops the processing in the middle of the pipeline until all the elements of the array are created. Then all the data is passed to the function via the *\$input* variable at one time. This type of approach does not take advantage of the streaming nature of the pipeline, which in many instances is more memory-efficient.

```
Add One Function
2
3
4
5
6
```

The DemoAddOneFunction.ps1 script is shown here.

```
DemoAddOneFunction.ps1
Function AddOne
{
    "Add One Function"
    While ($input.MoveNext())
    {
        $input.current + 1
    }
}
```

1..5 | addOne

To close this performance issue between functions and filters when used in a pipeline, you can write your function so that it behaves like a filter. To do this, you must explicitly call out the *Process* block. When you use the *Process* block, you are also able to use the *\$_* automatic variable instead of being restricted to using *\$input*. When you do this, the script will look like DemoAddOneR2Function.ps1, the results of which are shown here.

```
add one function r2
2
add one function r2
3
add one function r2
4
add one function r2
5
add one function r2
6
```

The complete DemoAddOneR2Function.ps1 script is shown here.

```
DemoAddOneR2Function.ps1
Function AddOneR2
{
    Process {
        "add one function r2"
        $_ + 1
    }
} #end AddOneR2
```

1..5 | addOneR2

What does using an explicit *Process* block do to the performance? When run on my computer, the function takes about 2.6 seconds, which is virtually the same amount of time taken by the filter. The `MeasureAddOneR2Function.ps1` script is shown here.

```
MeasureAddOneR2Function.ps1
Function AddOneR2
{
    Process {
        "add one function r2"
        $_ + 1
    }
} #end AddOneR2

Measure-Command {1..50000 | addOneR2 }
```

Another reason for using filters is that they visually stand out, and therefore improve readability of the script. The typical pattern for a filter is shown here.

```
Filter FilterName
{
    #insert code here
}
```

The *HasMessage* filter, found in the `FilterHasMessage.ps1` script, begins with the *Filter* keyword, and is followed by the name of the filter, which is *HasMessage*. Inside the script block (the braces), the `$_` automatic variable is used to provide access to the pipeline. It is sent to the *Where-Object* cmdlet, which performs the filter. In the calling script, the results of the *HasMessage* filter are sent to the *Measure-Object* cmdlet, which tells the user how many events in the application log have a message attached to them. The `FilterHasMessage.ps1` script is shown here.

```
FilterHasMessage.ps1
Filter HasMessage
{
    $_ |
    Where-Object { $_.message }
} #end HasMessage

Get-WinEvent -LogName Application | HasMessage | Measure-Object
```

Although the filter has an implicit *Process* block, this does not prevent you from using the *Begin*, *Process*, and *End* script blocks explicitly. In the `FilterToday.ps1` script, a filter named *IsToday* is created. To make the filter a stand-alone entity with no external dependencies required (such as the passing of a *DateTime* object to it), you need the filter to obtain the current date. However, if the call to the *Get-Date* cmdlet was done inside the *Process* block, the filter would continue to work, but the call to *Get-Date* would be made once for each object found in the input folder. So, if there were 25 items in the folder, the *Get-Date* cmdlet would be called 25 times. When you have something that you want to occur only once in the processing of the filter, you can place it in a *Begin* block. The *Begin* block is called only once, whereas the *Process* block is called once for each item in the pipeline. If you wanted any post-processing to take place (such as printing a message stating how many files were found today), you would place the relevant code in the *End* block of the filter.

The FilterToday.ps1 script is shown here.

```
FilterToday.ps1
Filter IsToday
{
    Begin {$dte = (Get-Date).Date}
    Process { $_ |
        Where-Object { $_.LastWriteTime -ge $dte }
    }
}

Get-ChildItem -Path C:\fso | IsToday
```

Creating a function: Step-by-step exercises

In this exercise, you'll explore the use of the *Get-Verb* cmdlet to find permissible Windows PowerShell verbs. You will also use *Function* keyword and create a function. After you have created the basic function, you'll add additional functionality to the function in the next exercise.

Creating a basic function

1. Start the Windows PowerShell ISE.
2. Use the *Get-Verb* cmdlet to obtain a listing of approved verbs.
3. Select a verb that would be appropriate for a function that obtains a listing of files by date last modified. In this case, the appropriate verb is *Get*.
4. Create a new function named *Get-FilesByDate*. The code to do this is shown here.

```
Function Get-FilesByDate
{
}

```

5. Add four command-line parameters to the function. The first parameter is an array of file types, the second is for the month, the third parameter is for the year, and the last parameter is an array of file paths. This portion of the function is shown here.

```
Param(
    [string[]]$fileTypes,
    [int]$month,
    [int]$year,
    [string[]]$path)

```

6. Following the *Param* portion of the function, add the code to perform a recursive search of paths supplied via the *\$path* variable. Limit the search to include only file types supplied via the *\$filetypes* variable. This portion of the code is shown here.

```
Get-ChildItem -Path $path -Include $filetypes -Recurse |
```

7. Add a *Where-Object* clause to limit the files returned to the month of the *lastwritetime* property that equals the month supplied via the command line, and the year supplied via the command line. This portion of the function is shown here.

```
Where-Object {  
    $_.lastwritetime.month -eq $month -AND $_.lastwritetime.year -eq $year }
```

8. Save the function in a .ps1 file named Get-FilesByDate.ps1.
9. Run the script containing the function inside the Windows PowerShell ISE.
10. In the command pane, call the function and supply appropriate parameters for the function. One such example of a command line is shown here.

```
Get-FilesByDate -fileTypes *.docx -month 5 -year 2012 -path c:\data
```

The completed function is shown here.

```
Function Get-FilesByDate  
{  
    Param(  
        [string[]]$fileTypes,  
        [int]$month,  
        [int]$year,  
        [string[]]$path)  
    Get-ChildItem -Path $path -Include $filetypes -Recurse |  
    Where-Object {  
        $_.lastwritetime.month -eq $month -AND $_.lastwritetime.year -eq $year }  
    } #end function Get-FilesByDate
```

This concludes this step-by-step exercise.

In the following exercise, you will add additional functionality to your Windows PowerShell function. In this additional functionality you will include a default value for the file types and make the *\$month*, *\$year*, and *\$path* parameters mandatory.

Adding additional functionality to an existing function

1. Start the Windows PowerShell ISE.
2. Open the Get-FilesByDate.ps1 script (created in the previous exercise) and use the Save As feature of the Windows PowerShell ISE to save the file with a new name of Get-FilesByDateV2.ps1.
3. Create an array of default file types for the *\$filetypes* input variable. Assign the array of file types to the *\$filetypes* input variable. Use array notation when creating the array of file types. For this exercise, use *.doc and *.docx. The command to do this is shown here.

```
[string[]]$fileTypes = @(".doc", "*.docx"),
```

4. Use the `[Parameter(Mandatory=$true)]` parameter tag to make the `$month` parameter mandatory. The tag appears just above the input parameter in the `param` portion of the script. Do the same thing for the `$year` and `$path` parameters. The revised portion of the `param` section of the script is shown here.

```
[Parameter(Mandatory=$true)]  
[int]$month,  
[Parameter(Mandatory=$true)]  
[int]$year,  
[Parameter(Mandatory=$true)]  
[string[]]$path)
```

5. Save and run the function. Call the function without assigning a value for the path. An input box should appear prompting you to enter a path. Enter a single path residing on your system, and press Enter. A second prompt appears (because the `$path` parameter accepts an array). Simply press Enter a second time. An appropriate command line is shown here.

```
Get-FilesByDate -month 10 -year 2011
```

6. Now run the function and assign a path value. An appropriate command line is shown here.

```
Get-FilesByDate -month 10 -year 2011 -path c:\data
```

7. Now run the function and look for a different file type. In the example shown here, I look for Microsoft Excel documents.

```
Get-FilesByDate -month 10 -year 2011 -path c:\data -fileTypes *.xlsx,*.xls
```

The revised function is shown here.

```
Function Get-FilesByDate  
{  
    Param(  
        [string[]]$fileTypes = @(".DOC","*.DOCX"),  
        [Parameter(Mandatory=$true)]  
        [int]$month,  
        [Parameter(Mandatory=$true)]  
        [int]$year,  
        [Parameter(Mandatory=$true)]  
        [string[]]$path)  
        Get-ChildItem -Path $path -Include $fileTypes -Recurse |  
        Where-Object {  
            $_.lastwritetime.month -eq $month -AND $_.lastwritetime.year -eq $year }  
    } #end function Get-FilesByDate
```

This concludes the exercise.

Chapter 6 quick reference

To	Do this
Create a function	Use the <i>Function</i> keyword, and provide a name and a script block.
Reuse a Windows PowerShell function	Dot-source the file containing the function.
Constrain a data type	Use a type constraint in brackets and place it in front of the variable or data to be constrained.
Provide input to a function	Use the <i>Param</i> keyword and supply variables to hold the input.
To use a function	Load the function into memory.
To store a function	Place the function in a script file.
To name a function	Use <i>Get-Verb</i> to identify an appropriate verb, and use the verb-noun naming convention.

Index

Symbols

\040 escape sequence 600
\$\$ variable 148
\$^ variable 148
\$_ variable 75, 148
\$? variable 148
? alias 111
* (asterisk) wildcard character 69
` (backtick) character 144
= (equal sign) operator 168
! (exclamation point) 80
> (greater-than) operator 327
< (less-than) operator 327
.(period) character 601
| (pipe) character 24, 144, 319
? (question mark) character 377

A

\a escape sequence 599
abstract classes, querying 299
abstract WMI class 382
access control list (ACL) 359
AccountsWithNoRequiredPassword.ps1 139
ACL 369
-Action parameter 498
Active Directory
 See also ADSI (Active Directory Service Interfaces)
 binding 400
 committing changes 401, 429
 creating objects 395, 396
 installing RSAT 432
 modifying user properties 410
 overwriting fields 429
 user account control values 408, 409

Active Directory Domain Services (AD DS)
 See AD DS (Active Directory Domain Services)
Active Directory Management Gateway Service (ADMGS) 431
Active Directory module
 deploying forests 459–465
 importing 433, 434
 installing 431–433
 loading automatically 434
 remote sessions 434
 verifying presence of 433
Active Directory Service Interfaces (ADSI) *See* ADSI (Active Directory Service Interfaces)
Active Directory sites, renaming 442, 443, 457
activities, workflow 552
AD DS (Active Directory Domain Services)
 adding features 460, 472
 assigning IP addresses 460, 472
 changing passwords 456
 creating computer accounts 443
 creating users 446, 447
 deploying 459
 deployment tools 460
 installing tools 397, 398
 prerequisites 459
 renaming computers 460
 renaming sites 442, 443, 457
 restarting computers 461, 472
 setting passwords 457
 unlocking accounts 457
 verifying roles and features 462, 472
AD DS and AD LDS Tools 397, 398
Add-ADFeatures.ps1 463

AddAdPrereqs.ps1

- AddAdPrereqs.ps1 461
 - Add-Computer cmdlet 110
 - addfeature job 463
 - Add-History cmdlet 554
 - AddOne function 211
 - Add-PSSnapin cmdlet 554
 - Add-RegistryValue function 480
 - address pages, creating 412–414
 - AddTwoError.ps1 490, 491
 - Add-WindowsFeature cmdlet 397, 398, 431, 432, 460, 468, 472
 - ADMGs (Active Directory Management Gateway Service) 431
 - [ADSI] accelerator 396
 - ADSI (Active Directory Service Interfaces)
 - See also* Active Directory
 - ADSI Edit 397, 398
 - AdsPath 396
 - attribute types 396
 - binding 400–405
 - connecting to objects 400–405
 - connecting to Windows NT 397
 - creating computer accounts 407, 408
 - creating groups 406, 407
 - creating objects 395, 396
 - creating users 405
 - deleting users 422
 - providers 397–399
 - ADSI Edit 397, 398
 - AdsPath 396
 - alias object, exposing properties 69
 - alias provider 66–67, 69
 - aliases
 - See also* commands
 - avoiding in scripts 592
 - best practices 593
 - canonical 592
 - case sensitivity 81
 - compatibility 592
 - creating 69, 593
 - creating for Get-Help 18, 19
 - creating new 69
 - data types 152, 153, 198
 - definition 18
 - finding 37, 45
 - listing all 59, 67, 107
 - types of 592
 - user-defined 592
 - using description property in 593
 - using to retrieve syntaxes 43
 - working with 66
 - AllowPasswordReplicationAccountName parameter 468
 - All Users, All Hosts profile 283
 - altering system state using the WhatIf parameter 74
 - Archive resource provider 565, 570
 - \$args variable 219
 - arguments
 - detecting extra in functions 221
 - eliminating 326
 - limiting returned data set 326
 - passing multiple to functions 220
 - [array] alias 198
 - array objects 55
 - arrays
 - creating for computer names 133
 - evaluating 173
 - indexing 238
 - turning text files into 429
 - using -contains operator to examine contents 517–519
 - ASCII values 159
 - \$ASCII variable 329
 - AsJob parameter 355, 356, 358, 360
 - assignment operators 169, 170
 - association classes 381, 385
 - AutoSize parameter 335, 393
- ## B
- \b escape sequence 599
 - BadScript.ps1 484, 502
 - basename script property 238
 - basicFunctions.psm1 247
 - binary byte array security descriptor (binary SD) 369
 - binding 400
 - binding string 400
 - BIOS information, retrieving from remote systems 118, 121
 - [bool] alias 198

- Boolean values 546
- boundary-checking functions 536–538
- braces, delimiting script blocks 185, 186
- Break statement 166, 167
- breakpoints
 - See also* debugging
 - access modes 495
 - currently enabled 503
 - debugging commands 501
 - deleting 496, 504, 505, 509
 - disabling 504
 - enabling 504
 - listing 503, 504, 509
 - pipelining results 503
 - responding to 501, 502
 - setting 492
 - setting on commands 499–501, 509
 - setting on first line 492
 - setting on line numbers 492–494, 509
 - setting on read operations 496
 - setting on variables 495–499, 509
 - tracking status of 504
- browsing classes 312
- business logic 202–204
- bypass parameter 143
- [byte] alias 198

C

- C attribute 400
- Calculator 51
- calling instance methods 365
- canonical aliases 592
- case sensitivity
 - aliases 81
 - file names 85
 - variables 84
- Catch block 538, 539
- \cC escape sequence 600
- contains operator 517
- certificate provider
 - and the file system model 69
 - and Windows 10 66
 - capabilities 69
 - identifying expired certificates 75
 - listing certificates 69
 - searching expiring certificates 75
 - searching for certificates 74
 - using MMC 69
- certificates
 - expired 75
 - searching for specific 74
 - viewing properties 72
- Certificates Microsoft Management Console (MMC) 69
- changing registry property values 97
- [char] alias 198
- [character_group] character pattern 601
- [^character_group] character pattern 601
- character patterns in regular expressions 601
- Check-AllowedValue function 536
- checkpoints
 - adding to workflows 556, 562
 - configuring 556
 - creating 552
 - disabling 558
 - placing 556
 - setting at activity levels 558
- CheckPoint-Workflow cmdlet 563
- CheckPoint-Workflow workflow activity 552, 558
- child scope 184
- ChoiceDescription class 514
- CIM class qualifiers 380
- CIM cmdlets 363
 - See also* CIM (Common Information Model) 375
 - combining parameters 381
 - default WMI namespace 375
 - and tab expansion 375
- CIM (Common Information Model)
 - See also* CIM cmdlets
 - checking configurations 571, 572
 - namespaces 375
 - sessions, creating 348
 - querying WMI classes 346–348
- CimClassMethods property 378
- CimClassName property 378
- CimClassQualifiers property 380
- CimSession parameter 347

classes

classes

- abstract, querying 299
- browsing 312
- common 298
- core 298
- direct querying 299
- displaying 302
- dynamic 298–300
- finding 298
- identifying which to use 299
- information about 312
- listing 298
- properties, retrieving 312
- querying 299, 346–348
- referencing 302
- searching for 298
- types of 298
- ClassName parameter 293, 297, 353, 381, 383, 394
- cleaning up output 384
- Clear-EventLog cmdlet 110
- Clear-History cmdlet 554
- Clear-Host cmdlet 60
- clear method 13
- Clear-Variable cmdlet 554
- ClientLoadableCLSID property 526
- client operating systems, managing 341
- CLSID property 526, 528
- CMD (command) shell 76
- CMD interpreter 2, 76
- CMD prompt, running inside Windows
 - PowerShell console 76
- cmdlet binding, enabling for functions 218
- [cmdletbinding] attribute 217–225, 257, 476
- cmdlets
 - See also* commands
 - adding logic to workflows 549
 - aliases 18
 - common parameters 11, 12
 - confirming execution 7, 8
 - debugging 492
 - default parameter sets 224
 - disallowed from workflows 554
 - finding 36
 - finding properties of 37
 - getting help 12, 21

- impersonating users 113
- information about 3
- naming 3, 54–57
- non-automatic activities 554
- prototype mode 7
- remoting 109–111
- retrieving syntax of 43
- returning methods for 48
- returning objects 44
- selecting from a list 52
- sorting 46
- spelling out names 592
- standard verbs 3
- suspending 8, 9
- tab completion 24
- verb-noun naming convention 54
- workflow activities 553

CN attribute 400

code

- See also* scripts
- downloading samples xxii
- formatting 594–597
- wrapping to next line 324

collections, looping through 167, 177

color of fonts, changing 333

columns 32, 36

-Columns parameter 28

COM-based objects 61, 62

CombinationFormatGetIPDemo.ps1 208

-Command argument 11

command lines, wrapping 350

-Command parameter 499

commandline property 350

command-line utilities 4–6, 19, 20, 22

commands

- See also* aliases; cmdlets
- building in Windows PowerShell ISE 260
- copying to Clipboard 53
- creating in Windows PowerShell ISE 274
- editing in Windows PowerShell ISE 262
- executing in parallel 549
- finding 36–44, 53, 262
- getting details of 36–44
- listing history of used 338, 339
- moving the insertion point 62
- recursive 294

- retrieving 336
- running as different user 113
- running as jobs 135
- running from script pane 263
- running from session history 339
- running ipconfig 4, 5
- running multiple 5
- running on remote systems 135
- running sequentially 559
- running single 120–122
- running via Commands add-on 262
- setting breakpoints on 499–501, 509
- Commands add-on 260, 264, 270–272
- comments 593, 594
- common classes 298
- Common Information Model (CIM)
 - cmdlets *See* CIM cmdlets
- ComObject parameter 50, 51
- comparison operators 169, 170
- compatibility aliases 592
- Complete-Transaction cmdlet 554
- computer accounts, creating with ADSI 407, 408
- computer connectivity 516, 546
- computer names, creating an array of 133
- computer parameter 195
- \$computer variable 195
- ComputerName parameter 111, 112, 301, 347, 512, 546
- computers, checking for valid WMI class 533
- concatenation operators 145
- Concurrency property 526
- ConfigurationData parameter 568
- configuration drift 571, 572
- Configuration keyword 566, 580
- ConfigurationNamingContext property 442
- configurations
 - calling 569, 574
 - checking 571
 - controlling drift 571, 572
 - creating DSC scripts 580
 - creating using DSC 566–568
 - parameters 568–570
 - running multiple times 571
 - setting dependencies 570–572
- Confirm switch parameter 6–9, 23, 22, 223, 224, 445
- ConfirmImpact property 224
- connection pooling 548
- connection throttling 548
- Connect-PSession cmdlet 110
- Connect-WSMan cmdlet 110
- constants
 - See also* variables
 - best practices 597
 - creating 177
 - definition 153
 - naming 597
 - referring to 153
- consumers 292
- contains operator 514, 517
- Continue cmdlet 501
- ConversionFunctions.ps1 187
- ConvertFrom-String 599
- copying text 72
- core classes 298
- Count parameter 516
- count property 55, 106, 128
- countryCode attribute 413
- country/region codes 413, 429
- CreateAdditionalDC.ps1 467
- Create method 396
- CreateMultipleUsers.ps1 418
- CreateOU.ps1 396
- CreateReadOnlyDomainController.ps1 469
- CreateRegistryKey.ps1 480, 481
- creating
 - aliases 69
 - folders and files 82
 - registry drives 88
 - registry keys 93, 95
 - temporary environment variables 78
 - text files 107
- Credential parameter 112, 342, 347
- credentials
 - administrator account 345
 - alternate 132, 342, 344
 - remote connections 112, 343–345
- Current User, All Hosts profile 279, 289
- CurrentUserAllHosts property 279
- custom error actions and namespaces 295

D

- \d character pattern 601
- data
 - evaluating using operators 327
 - reducing 352, 353
 - WMI, filtering 360
- data output, controlling 303
- data sets 301
- data type aliases 152, 153
- data types, constraining 216
- date, finding current 339
- date object, assigning 333
- datetime type 347
- DC attribute 400
- Debug switch parameter 12, 476, 478
- debugging
 - See also* breakpoints; errors; scripts
 - bypassing commands 487
 - cmdlets 492, 501
 - functions 505, 506, 509
 - logic errors 478, 479
 - quitting 493
 - run-time errors 474–478
 - scripts 507–509
 - setting breakpoints 492–500
 - stepping over functions 502
 - stepping through scripts 483–488, 509
 - suspending script execution 486
 - syntax errors 473, 474
 - syntax parser 474
 - trace levels 480–483
 - tracing scripts 479–483
 - turning off stepping 488
- Debug-Process cmdlet 554
- DebugRemoteWMISSession.ps1 476
- [decimal] alias 198
- default
 - parameter sets, specifying 224
 - registry drives 88
 - registry key value, assigning 96
 - Windows PowerShell prompt 76
 - WMI namespace 375
 - WMI namespaces, finding 312
- default property 90
- Default statement 172
- DefaultMachineName property 526
- DefaultParameterSetName property 224
- Definition parameter 46, 59, 157
- definition property 38, 39
- Delete method 423
- deleting
 - breakpoints 496, 504, 505, 509
 - directories 107
 - expired certificates 75
 - folders 177
 - users 422
 - Windows PowerShell ISE snippets 269, 270, 274
- DemoAddOneR2Function.ps1 211
- DemoBreakFor.ps1 167
- DemoDoWhile.ps1 158
- DemoForEach.ps1 165
- DemoForLoop.ps1 163
- DemoForWithoutInitOrRepeat.ps1 163, 164
- DemofIfElseIfElse.ps1 170
- Demof.ps1 168
- DemoSwitchArray.ps1 173
- DemoSwitchCase.ps1 172
- DemoSwitchMultiMatch.ps1 173
- DemoTrapSystemException.ps1 199
- DemoWhileLessThan.ps1 154
- dependencies
 - adding DSC resource 578–580
 - setting for file resources 574
- DependsOn keyword 570
- deprecated qualifiers 381
- deprecated WMI classes, finding 381
- Descending switch parameter 35
- Description parameter 268
- Desired State Configuration (DSC) *See* DSC (Desired State Configuration)
- DestinationPath parameter 566
- dir command 24
- direct querying 299
- directories, deleting 107
- directory listings 24–29
- directory parameter 81
- DirectoryInfo object 44
- DirectoryListWithArguments.ps1 138
- Disable-PSBreakpoint cmdlet 492, 554
- Disconnect-WSMan cmdlet 110

- \$discount variable 203
- Discover switch parameter 436
- disk drives 318, 319
- \$Disk variable 318, 319
- DisplayName parameter 307, 442
- distinguishedname attribute 446
- DNS servers
 - adding as domain controllers 466
 - adding roles 472
 - assigning to DNS clients 465
 - installing features 460, 463
 - renaming 466
 - restarting 466
 - viewing features/roles 472
- Do keyword 161
- Do statement 161
- domain controllers
 - adding as read-only 468, 469
 - adding to existing domains 465–467
 - adding to forests 464, 471, 472
 - checking on remote machines 441
 - connecting to 442
 - finding 436
 - prerequisites 459, 470, 471
- domain password policy 440
- domains 397, 399
- dot-source operator 187
- DotSourceScripts.ps1 206
- dot-sourcing 186, 188, 189
- dotted notation 39, 228, 278
- [double] alias 198
- Do...Until statement 160
- DoWhileAlwaysRuns.ps1 161
- Do...While statement 157–160
 - casting to ASCII values 159
 - operating over arrays 158, 159
 - using the range operator 158
- drift, configurations 571, 572
- drive-and-file-system analogy 65
- drive parameter 195
- \$drive variable 195
- \$driveData variable 195
- drives
 - changing 337
 - creating 240
 - global scope 241

- DriveType property 318, 319
- DSC (Desired State Configuration)
 - adding resource dependencies 578
 - calling configurations 574
 - compiling MOF 574
 - configuration parameters 568, 569
 - controlling drift 571, 572
 - creating configurations 566, 567, 576–578, 580
 - creating scripts 566, 580
 - definition 565
 - importing resource modules 573
 - modifying environment variables 573–576
 - resource provider properties 565, 566
 - running against remote servers 568
 - setting dependencies 570–572, 574
 - showing available resource members 573
 - specifying configuration location 580
 - starting IntelliSense to display resource members 573
 - starting the configuration process 574
 - viewing existing resources 580
 - viewing progress 580
- \$dteDiff variable 333
- \$dteEnd variable 333
- \$dteStart variable 333
- dynamic classes 298–300
- dynamic qualifiers 382
- dynamic WMI classes, finding 394

E

- \e escape sequence 600
- ea alias 143 *See also* -ErrorAction parameter
- echo command 76
- Else keyword 170
- else statement 516
- enabled property 447, 526
- Enable-PSBreakpoint cmdlet 492, 554
- Enable-PSRemoting cmdlet 114, 115, 135, 345
- enabling QuickEdit mode 72
- Encoding parameter 329
- EndlessDoUntil.ps1 161, 162
- Enter-PSSession cmdlet 110, 118, 119, 132, 135, 439, 554
- enumeration values 526

EnumNetworkDrives method

- EnumNetworkDrives method 63
- environment provider 104
 - and environment variables 77–79
- Environment resource provider 565, 573
- environment variables
 - creating 78, 573
 - modifying 573–576
 - on computer, listing 335
 - removing 79
 - renaming 79
 - viewing new 575
- \$env:PSModulePath variable 230
- eq operator 169
- equals argument 310
- error handling
 - adding 404
 - incorrect data types 532–536, 546
 - limiting choices 514–521
 - missing parameters 511–514
 - missing rights 521–523
 - missing WMI providers 523–532
 - Try...Catch...Finally 538–541, 545, 546
- error stacks, clearing 534
- ErrorAction parameter 12, 13, 98, 143, 144
- errors
 - See also* debugging
 - Access Denied 295
 - capturing 539
 - creating objects 401–404
 - logic 478, 479
 - remote connections 112
 - remote procedure call (RPC) 342
 - run-time 474–478
 - scripts 143, 185
 - scripts, ignoring 295
 - suppressing messages 199
 - syntax 473, 474
 - system exceptions 199
 - trapping 199
 - WinRM (Windows Remote Management) 117
 - workflows 551
- ErrorVariable parameter 12
- escape sequences, in regular expressions 599
- est-ParameterSet function 227
- Examples switch parameter 17

- execution policies for scripts 177
 - retrieving 142
 - setting for current user 142
 - setting for entire machine 142
 - turning on options 140, 141
- exit command 132
- Exit statement 167
- Exit-PSSession cmdlet 554
- ExpandEnvironmentStrings method 51
- expanding strings 155, 163
- Export-Alias cmdlet 554
- Export-Clixml cmdlet 348
- Export-Console cmdlet 11, 554
- exposing properties of an Alias object 69

F

- \f escape sequence 600
- feedback, providing xxiv
- file names, case sensitivity 85
- File parameter 81, 82
- File resource provider 565
- FileInfo object 44
- FilePath argument 329
- files in folders, listing 63
- filesystem provider 80–86
- \$File variable 328, 329
- Filter keyword 204, 212
- Filter parameter 323, 324, 331, 351, 353, 360, 383, 394, 527
- filter strings 33
- \$Filter variable 326
- FilterHasMessage.ps1 212
- filtering
 - columns 36
 - data 351, 360
 - output 306–308
 - using CPU time 35
- filters 209–213
 - adding to tables 33
 - definition 204
 - options 33
- Finally block 539, 546
- Find-Module cmdlet 583, 584
- [firstCharacter-lastCharacter] character pattern 601

- Flexible Single Master Operation (FSMO)
 - roles 435–438, 457
- folders
 - deleting 176, 177
 - listing files in 63
- fonts, changing color 333
- Force parameter 81, 95
- For keyword 162
- For loop 13
- For statement, and endless loops 164
- Force switch parameter 12, 46, 95, 115, 574
- ForceDiscover switch parameter 436
- Foreach keyword 550
- ForEach-Object cmdlet 144, 165, 177, 295, 393, 394
- ForEach -Parallel workflow activity 552
- Foreach statement 165, 166
- ForegroundColor parameter 333
- ForEndlessLoop.ps1 164
- forests
 - adding domain controllers 464, 471, 472
 - creating 472
 - deploying 459–465
- Format-List cmdlet 26, 72, 77, 78, 99, 303, 315, 317, 326, 328–330, 339, 385, 394, 398, 584
- Format-Table cmdlet 29–31, 146, 303, 319, 323, 331, 384, 385, 392
- Format-Wide cmdlet 27, 63
- formatting code, best practices 594, 596
- forscripting registry key 569
- freespace property 319
- From statement 320
- FSMO role holders 435–438, 457
- ft alias 384 *See also* Format-Table cmdlet
- FullyQualifiedErrorId property 402
- Function keyword 180, 182, 185, 194, 202, 216
- function provider 85–86
 - capabilities 85
 - file system-based model 85
 - listing all functions on system 86
- FunctionGetIPDemo.ps1 206
- functions
 - adding functionality to 214, 215
 - adding help 191–194
 - adding -WhatIf support 222, 223
 - advanced 217
 - automatic parameter checks 219–221
 - business logic 202–204
 - calling like methods 184
 - checking number of arguments in 220
 - choosing verbs 182, 184
 - [cmdletbinding] attribute 217–225
 - comment-based help 193
 - complete using Windows PowerShell ISE snippets 266, 267
 - copying into modules 246
 - creating 182, 213–215, 253–256
 - creating with Windows PowerShell ISE snippets 266
 - debugging 505, 506, 509
 - default parameter sets 224
 - delimiting script blocks 185
 - detecting extra arguments 221
 - displaying contents of 193
 - dot-sourced 190, 191
 - enabling cmdlet binding 218
 - enabling strict mode for 490
 - filters 204, 209–213
 - formatting 596
 - getting help 251
 - including in scripts 591, 592
 - library script 186
 - listing all 107
 - modifying cmdlet behavior using 26
 - modifying scripts 205
 - multiple input parameters 194
 - naming 182, 216, 594
 - parameters 183, 184
 - passing multiple arguments 220
 - passing values to 183
 - pipelined input 190, 191
 - positional parameters 183
 - promoting readability of 595
 - providing input to 216
 - reusing 186–188, 216
 - script cmdlets 217
 - signatures 203
 - storing 216
 - suppressing error messages 199
 - tracing features 529
 - understanding 179–186
 - using 216

functions (*continued*)

using comments 594

using type constraints 198

variable scope 184

verb-noun combinations 180

verbose messages 218, 219

G

gal alias 46, 59

gc alias 157

gci alias 70, 337 *See also* Get-ChildItem cmdlet

gcim alias 301, 334 *See also* Get-CimInstance cmdlet

gcm alias 37, 43

-ge operator 169

get verb 54

Get-Acl cmdlet 369

Get-ADDefaultDomainPasswordPolicy cmdlet 440

Get-ADDomain cmdlet 439, 440, 457

Get-ADDomainController cmdlet 436, 437, 441

Get-ADForest cmdlet 439, 457

Get-ADObject cmdlet 437, 442, 457

Get-ADOrganizationalUnit cmdlet 446

Get-ADRootDSE cmdlet 442

Get-ADUser cmdlet 446

Get-Alias cmdlet 18, 45, 59, 157, 317, 336, 554

Get-AllowedComputerAndProperty.ps1 520

Get-AllowedComputerAndProperty.ps1 521

Get-AllowedComputer function 518, 519

Get-AllowedComputer.ps1 519

Get-BiosInformationDefaultParam.ps1 513

Get-BiosInformation.ps1 512

Get-ChildItem cmdlet 24, 59, 67, 79, 100, 239, 335

listing all aliases 107

listing all available properties 103

listing all certificates 103

listing all functions 107

listing certificates 70

listing of environment variables 79

listing registry keys 91, 107

listing variables 100, 107

on the currentuser store 75

pipelining results 67

searching for software 92

Get-Choice function 515

Get-ChoiceFunction.ps1 515

Get-CimAssociatedInstance cmdlet 385, 388, 390, 394

array indexing 388

errors 388

finding types of classes returned 394

inputobject parameter 388

pipelining to Get-Member cmdlet 385, 390

Get-CimClass cmdlet 298, 299, 312, 375, 380, 392, 393

finding WMI classes 375

wildcards 375, 379

Get-CimInstance cmdlet 294, 297, 301, 312, 314, 315, 323, 339, 346, 348, 360, 383, 385, 393

reducing instances returned 394

reducing properties returned 394

wildcards 376, 385

Get-Command cmdlet 36–44

Get-ComputerInfo function 248, 251

GetComputerInfoWorkflow.ps1 551

Get-Content cmdlet 84, 157, 177, 193, 518

Get-Counter cmdlet 110

Get-Credential cmdlet 132, 343, 345, 346, 356, 357

Get-Date cmdlet 333, 339

Get-Discount function 202

Get-Doc function 204

GetDrivesCheckAllowedValue.ps1 537

GetDrivesValidRange.ps1 538

Get-DscResource cmdlet 580

Get-EventLog cmdlet 110

Get-ExecutionPolicy cmdlet 141, 142, 177

Get-FileSystemDrives function 241

GetFolderPath method 280

Get-FreeDiskSpace function 194

Get-FreeDiskSpace.ps1 194

Get-Help cmdlet 12, 15, 26, 69, 99, 109

creating an alias for 19

listing cmdlets 99

Get-History cmdlet 336, 339, 554

Get-HotFix cmdlet 110

Get-InstalledModule cmdlet 589

Get-IPObjDefaultEnabledFormatNonIP-Output.ps1 208

Get-IPObjDefaultEnabled.ps1 207

Get-IseSnippet cmdlet 269
 Get-Item cmdlet 89, 105
 listing environment variables 78, 105
 viewing registry key values 89
 Get-ItemProperty cmdlet 89, 90, 150, 313, 314
 accessing registry key values 90
 viewing registry key values 89
 Get-Job cmdlet 124, 128, 135, 356, 357
 Get-Location cmdlet 88
 Get-Member cmdlet 44–49, 59, 67, 300, 308, 367, 385, 387, 390, 394
 Get-Module cmdlet 230, 243, 433
 Get-MyModule function 242, 243, 431
 Get-NetAdapter cmdlet 460, 472
 Get-NetConnectionProfile function 233
 Get-OperatingSystemVersion function 236
 Get-OperatingSystemVersion.ps1 182
 Get-Process cmdlet 9, 12, 22, 31, 110, 322, 323
 Get-PSBreakpoint cmdlet 492, 496, 503, 554
 Get-PsCallStack cmdlet 501
 Get-PSCallStack cmdlet 492, 554
 Get-PSDrive cmdlet 17, 77, 88, 103
 Get-PSProvider cmdlet 66, 67
 Get-PSSession cmdlet 110, 119
 Get-PSSnapin cmdlet 554
 GetRandomFileName method 83
 Get-Service cmdlet 110, 306, 307
 Get-TextStats function 191
 Get-Transaction cmdlet 554
 Get-ValidWmiClass function 534
 Get-Variable cmdlet 101, 554
 Get-Verb cmdlet 3, 54
 Get-WimObject cmdlet 385
 Get-WindowsFeature cmdlet 397, 398, 432, 460, 472
 Get-WinEvent cmdlet 110
 Get-WinFeatureServersWorkflow.ps1 559
 GetWmiClassesFunction.ps1 192
 Get-WmiInformation function 535
 Get-WmiObject cmdlet 110, 295, 298, 342, 345, 360, 361, 365, 366, 367
 Get-WmiProvider function 526, 531
 Get-WSManInstance cmdlet 110
 ghy alias 338
 gi alias 78 *See also* Get-Item cmdlet
 global security group 444

gm alias 81 *See also* Get-Member cmdlet
 gps alias 31
 grave accent character *See* ` (backtick) character
 grids *See* tables
 group alias 55
 group-and-dot 363, 364
 Group-Object cmdlet 55
 Group Policy, configuring WMI 341
 Group resource provider 565
 groups, creating with ADSI 406, 407
 See also security groups
 -GroupScope parameter 444
 gsv alias 33
 gwmi alias 361, 367

H

handle property 330
 [hashtable] alias 198
 -Height parameter 52
 help
 adding for functions 191
 comment-based 193
 here-string objects 192
 specific parameters 229
 using functions 251
 Help cmdlet 501
 help files
 suppressing errors during update 13
 Update-Help cmdlet 12
 updating 12
 Help function 17
 -help parameter 192
 help system
 entering 14–19
 levels of display 17
 output, displaying 17
 using wildcards 17
 HelpMessage parameter property 229, 257
 here-string object 192
 hierarchical namespaces 292
 Hit Variable breakpoint 496
 HKCR drives, checking for 529
 home directories, listing 327
 HostingModel property 526

I

- icm alias 314, 345
- contains operator 517
- Id parameter 7
- identifying properties of directories 81
- identifying the Certificate drive 103
- IdentifyServiceAccounts.ps1 script 328
- identity parameter 436, 444, 450
- If statement 98, 164, 166, 243, 516
 - assignment operators 169, 170
 - comparison operators 168–170
 - evaluating arrays 173
 - evaluating multiple conditions 170
- IfIndex property 472
- ihy alias 338
- impersonation levels 314
- ImpersonationLevel property 527
- Import-Alias cmdlet 554
- Import-Module cmdlet 233, 433
- includemanagementtools parameter 472
- index numbers, finding 472
- InitializationReentrancy property 527
- InitializationTimeoutInterval property 527
- InitializeAsAdminFirst property 527
- InlineScript activity 554, 560
- input parameters
 - computer 294
 - localhost 294
 - namespace 294
 - root 293
 - using more than two 200–202
- InputObject parameter 48, 308, 394
- Install-ADDomainController cmdlet 468
- Install-ADDSDomainController cmdlet 466
- Install-ADDSEForest cmdlet 472
- InstallationPolicy parameter 589
- InstallDns parameter 466
- Install-Module cmdlet 589
- instance methods
 - calling 365–366
 - definition 361
 - executing 361
 - finding relative path 373
 - terminating 363, 370

- InstanceName parameter 568
- [int] alias 198
- IntelliSense 264, 573
- Internet Explorer zone 141
- InvocationInfo property 402
- Invoke-CimMethod cmdlet 311
- Invoke-Command cmdlet 110, 120, 121, 135, 314, 345, 346, 356, 357
- Invoke-History cmdlet 339, 554
- Invoke-Item cmdlet 73
- Invoke-WmiMethod cmdlet 110, 365
- Invoke-WSManAction cmdlet 110
- [io.path] class 83
- IP addresses, assigning 472
- ipconfig commands, running 4, 5
- ise alias 279
- ItemType parameter 83, 107

J

- jobs
 - cleaning up 127
 - completion notification 127
 - creating 134
 - IDs 123, 135
 - keeping data from 128–131
 - monitoring 129
 - naming 124
 - pipelining objects 128
 - receiving results 134, 135
 - removing completed 124
 - retrieving WMI results 360
 - running commands as 122
 - starting new 128
 - status 127, 135
 - stopping 128
 - storing returned objects 124, 126
 - WMI 355–357, 359
- Join-Path cmdlet 238, 295, 530

K

- Keep switch parameter 123, 128, 356
- key parameter 481

L

- l attribute 413
- LastWriteTime property 31, 60
- LDAP
 - See also* RDN (relative distinguished name)
 - naming convention 399
 - provider 397
- le operator 169
- Length property 31
- like operator 169
- Limit-EventLog cmdlet 110
- limiting choices 514
 - for parameter values 521
 - using -contains operator 517–521
 - using PromptForChoice 514, 515, 544, 545
 - using Test-Connection to identify computer connectivity 516
- line parameter 492
- List cmdlet 501
- list parameter 141, 298
- ListAvailable switch parameter 231, 234, 433
- listing
 - aliases 107
 - environment variables 77
 - functions 86, 107
 - mapped drives 63
 - registry keys 91, 107
 - variables defined in a session 107
- ListNamePathShare.ps1 script 322
- ListProcessesSortResults.ps1 138
- ListShares.ps1 script 320, 322
- ListSpecificShares.ps1 script 325
- literal quotation marks and default
 - property 90
- literal strings 155, 156
- local computer shortcut name 312
- LockedOut parameter 447, 457
- Log resource provider 566
- logging
 - adding 324
 - service accounts 328, 329
- logic errors 478, 479
- [long] alias 198
- Loop keyword 155
- looping through collections 167, 177
- lt operator 169

M

- Managed Object Format (MOF) *See* MOF (Managed Object Format)
- Mandatory parameter property 225, 257
- MandatoryParameter.ps1 513
- mandatory parameters 513
- mapped drives, listing 63
- marque 565
- match operator 87, 169, 599
- matching 172–174
- Maximum parameter 339
- md alias 83 *See also* mkdir function
- MeasureAddOneR2Function.ps1 212
- Measure-Object cmdlet 54, 319, 339
- Members parameter 444
- membertype attribute 81
- MemberType parameter 46, 47, 81
- Method member types 195
- MethodName parameter 311, 394
- method notation 490
- methods
 - definition 377
 - examining 45
 - listing all available 63
 - PromptForChoice 514, 544, 545
 - retrieving with wildcards 48
- Microsoft Management Console (MMC)
 - renaming Active Directory sites 442
 - starting 399
- Microsoft.PowerShellISE_profile.ps1 279
- Microsoft.PowerShell_profile.ps1 279
- Minimum parameter 339
- missing registry properties 98
- missing rights 522
- missing WMI providers
 - checking for installation 524–532
 - connecting to namespaces 523
 - information about 523
- mkdir function 83
- MMC (Certificates Microsoft Management Console) 69
- Mode parameter 495
- modifying registry property values 97
- ModifySecondPage.ps1 412
- ModifyUserProperties.ps1 410

module manifest

- module manifest 188
- module parameter 12, 13, 250, 433
- \$modulePath variable 238–240
- modules
 - copying files into directories 239
 - copying functions into 246
 - copying to module stores 248
 - creating 246–253, 256, 257
 - creating drives 240, 241
 - creating subdirectories 239
 - definition 230
 - dependencies 242–244
 - directory 230, 235
 - downloading from PowerShell Get 586
 - expanding names 233
 - exported commands 250
 - exporting 253
 - finding in PowerShell Gallery 582, 587, 589
 - finding installed 587, 589
 - folder locations 235
 - folder naming 236
 - grouping profile information 285
 - importing 252
 - installing 66, 235–246, 248, 252, 253, 256, 257
 - installing from PowerShell Gallery 588, 589
 - installing from PowerShell Get 585, 586
 - listing 235
 - listing available 230–232, 239
 - loading 233, 234
 - locating 230–233
 - locations 230, 240
 - names 234
 - netconnection 233
 - packaging workflows 547
 - passing to functions 244
 - paths 238
 - PowerShellGet 583
 - retrieving paths 237
 - searching by contributor 584
 - searching descriptions 585
 - shared 246
 - sorting by revision history 584
 - storing profiles 285, 286
 - uninstalling 586, 589

- uninstalling from PowerShell Gallery 588, 589
- using from shares 244–246
- using in profiles 282
- wildcard patterns 233, 234
- MOF (Managed Object Format)
 - compiling 574
 - creating 566
 - definition 566
 - storing 569
- more.com utility 17
- Move-ADObject cmdlet 446
- mred alias 60
- mydocuments folder 280
- my-function function 479

N

- \n escape sequence 600
- Name parameter 69, 78, 83, 99, 150, 307, 444
- name parts 399
- name property 28, 31, 78, 295, 327, 527
- named parameters 226
- namespace input parameter 294
- Namespace parameter 293, 301
- namespaces
 - on computer, listing 312
 - custom error actions and 295
 - default 312, 313
 - default WMI value 375
 - hierarchical 292
 - information about 296
 - installed, list of 296
 - listing classes 312
 - nesting 294
 - and objects 293–295
 - organizing 293, 294
 - properties 295
 - providers, listing 312
- naming
 - constants 597
 - functions 594
 - variables 594, 597
- naming conventions
 - cmdlets 3
 - LDAP 399

- nouns 54
- verbs 54
- NDS provider 397
- ne operator 169
- nesting namespaces 294
- netconnection module 233
- network adapters, finding index numbers 472
- New-ADGroup cmdlet 444
- New-ADOrganizationalUnit cmdlet 443
- New-ADUser cmdlet 446, 457
- New-Alias cmdlet 18, 554
- New-CimSession cmdlet 347, 348, 360
- Newest parameter 129, 135
- New-EventLog cmdlet 110
- New-IseSnippet cmdlet 268
- New-Item cmdlet 69, 289
 - creating aliases 69
 - creating and assigning values to registry keys 96
 - creating environment variables 78
 - creating text files 107
- New-Line function 188, 190
- New-ModuleDrive function 241
- New-ModulesDrive.ps1 241
- NewName parameter 79
- New-NetIPAddress cmdlet 460, 472
- New-Object cmdlet 50–52
- NewPassword parameter 446
- New-PSDrive cmdlet 88, 240, 530
- New-PSSession cmdlet 110, 119
- New-TimeSpan cmdlet 333, 339
- New-Variable cmdlet 177, 329, 554
- New-WSManInstance cmdlet 110
- Next keyword 162
- node 566
- Node command 580
- NoExit parameter 146
- NoLogo argument 11
- nonterminating errors 522
- notafter property 75
- notlike operator 169
- notmatch operator 87, 169, 599
- Noun parameter 43
- nouns, naming convention 54
- NWCOMPAT provider 397

O

- O attribute 400
- Object Editor 528
- objects
 - See also* OU (organizational unit)
 - COM-based 61, 62
 - definition 44
 - deserialized 124–127
 - errors 401–404
 - and namespaces 293–295
 - renaming 443
 - retrieving member information 44
 - retrieving values of 339
 - storing in variables 50, 124, 127
- Off parameter 488, 492
- operating systems, retrieving version numbers 236
- OperationTimeoutInterval property 527
- operators
 - assignment 169, 170
 - comparison 169, 170
 - using 327–329
- Option parameter 153
- organizational unit (OU) *See* OU (organizational unit)
- OtherTelephone attribute 410
- OU attribute 400
- OU (organizational unit)
 - See also* objects
 - [ADSI] accelerator 396
 - creating from text files 424
 - creating on remote machine 443
 - creating using ADSI 395, 396
 - moving users to 446
 - storing user accounts 446
- OutBuffer parameter 12
- Out-File cmdlet 328, 329
- Out-GridView cmdlet 31–36, 315, 554
- Out-Null cmdlet 239
- out-of-bound errors
 - placing limits on parameters 537, 538
 - using boundary-checking functions 536, 537
- output
 - filtering/sorting 306–308
 - formatting 26, 27, 30, 31–36

-OutputPath parameter

- output (*continued*)
 - grouping by size 28
 - paged, producing 339
 - pipelining 59
 - reducing 351, 360
 - self-updating in filtered tables 34
 - sorting/filtering 306–308
 - wide, producing 63
- OutputPath parameter 568
- OutVariable parameter 12
- overwriting registry keys 95

P

- Package resource provider 566
- paged output, producing 339
- parallel script blocks 553
- parallel workflow activities 552, 555, 559
- param keyword 568
- Param keyword 201, 216, 217
- param statement 512
- parameter attribute 224, 225
- Parameter parameter 109
- parameter sets 227, 257
- parameters
 - assigning default values 512, 568
 - assigning positions 257
 - automatic checks 219–221
 - checking value validity 532
 - commonly used 12
 - configurations 568, 569
 - identifying 201
 - input, using more than two 200–202
 - making mandatory 257
 - mandatory 513, 514, 546
 - missing 229, 257, 511–513, 521
 - missing values 512, 546
 - named 226
 - passing multiple 490
 - placing limits on 537, 538
 - positional 97, 183
 - required for Windows PowerShell ISE snippets 268
 - specifying for functions 184
 - supplying values for 53
 - Windows PowerShell, reducing data 352

- ParameterSetName parameter property 227, 257
- PassThru parameter 144
- passwords
 - See also* security
 - changing 456
 - creating secure strings 446
 - resetting 446, 457
- path parameter 70, 78, 79, 105, 107, 150, 183, 184, 444
- path strings, converting to rich types 593
- \$path variable 183–185
- paths 238
- patterns 299 *See also* wildcards
- pause function 87
- PerLocaleInitialization property 527
- permissions, remote callers 342
- persistence 556 *See also* checkpoints
- PerUserInitialization property 527
- PING commands, and Windows 8 client systems 117
- PinToStart.ps1 10
- pipeline 228
- pipelined data, displaying in tables 31–36
- \p{name} character pattern 601
- Pop-Location cmdlet 94
- pop-up boxes, producing 62, 63
- Popup method 62
- position message 143
- Position parameter property 226, 257
- positional parameters 97, 183
- postalCode attribute 413
- postOfficeBox attribute 413
- PowerShell Gallery
 - configuring as trusted installation 589
 - configuring installation policy 586
 - finding 581
 - installing modules from 585, 588, 589
 - searching for modules 582, 587
 - uninstalling modules 588, 589
 - wildcards 585
- PowerShell Get
 - configuring and using 583–585
 - configuring as trusted location 586
 - downloading modules 586
 - finding installed modules 587

- installing modules 586
- installing required file 583
- PowerShellGet module 583
- processes
 - running 322–324
 - stopping 22
- process lists, sorting 35
- profile.ps1 279
- profiles 275, 276
 - adding functionality 288, 289
 - All Users, All Hosts 283, 289
 - checking for specific 278, 289
 - cleaning up 285
 - creating 58, 59, 279, 286, 287, 289
 - Current User, All Hosts 279, 289
 - definition 57
 - determining types to use 280
 - directory location 280
 - editing 289
 - grouping information into modules 285
 - ISE vs. console 280, 281
 - locations 280
 - mydocuments folder location 280
 - names 279, 280
 - opening for editing 279
 - paths 275, 289
 - single vs. multiple 281, 282
 - storing information in files 284, 285
 - storing modules 285, 286
 - types of 275
 - usage patterns 280
 - using files 284, 285
 - using modules 282
 - using multiple 281
 - viewing all for current host 277, 278
- program logic 202
- PromptForChoice method 514
- properties
 - added by CIM cmdlets 317
 - of classes 312
 - definition 38, 39, 377
 - displaying 302
 - examining 45
 - finding for cmdlets 37
 - hidden files/folders 46
 - listing all available 103
 - removing empty 319
 - resource providers 565, 566
 - retrieving 315–317
 - selecting multiple 322–324
 - selecting specific 321
 - spacing/capitalization 322
 - using -contains operator to test for 519–521
 - and variables 385
- Property member types 195–197
- Property parameter 28, 38, 77, 303, 312, 330, 339, 351, 353, 360, 383, 384, 394
- Property set member types 196
- property sets 303
- ProtectedFromAccidentalDeletion parameter 444
- prototype mode 7
- providers
 - class IDs 529
 - DCOM registration 529
 - definition 65
 - handling missing 523–532
 - installing 297
 - LDAP 397
 - listing 297
 - listing installed 312
 - in namespaces, listing 312
 - NDS 397
 - NWCOMPAT 397
 - searching for 527, 528
 - searching registry for 529
 - system template class 297
 - WinNT 397
 - WMI Microsoft Installer (MSI) 332
- providing feedback xxiv
- proxy function 26
- \$PSCmdlet variable 227
- PSComputerName parameter 555
- PSComputerName property 346
- PSConsoleFile argument 11
- PSDesiredStateConfiguration module 573
- PSGallery *See* PowerShell Gallery
- psiscontainer property 75
- PSModulePath variable 237, 433
- PSPersist parameter 552, 563
- Pure property 527
- Put method 405, 429

Q

- qualifier names and tab expansion 382
- qualifier queries and wildcards 382
- QualifierName parameter 299, 300
- queries
 - against remote computers 294
 - limiting results 325
 - particular classes 320
 - results 301
 - select * 320
 - suppressing 445
 - WQL, reducing data with 353
 - WQL, using 360
- Query parameter 320, 321, 353
- \$Query variable 327, 328, 330, 332
- querying
 - classes 346
 - using classes 299
 - direct 299
 - remote systems 346–348
- querying abstract WMI classes 382
- QuickEdit mode 72
- Quiet switch parameter 516, 546
- quotation marks 324
 - environment variables 51
 - string values 325

R

- \r escape sequence 599
- range operator 158
- \$rate variable 202
- RDN (relative distinguished name)
 - See also* LDAP
 - as name part 399
 - attribute types 400
 - definition 396
 - verifying 400
- reading and writing for files 84
- ReadUserInfoFromReg.ps1 149
- Receive-Job cmdlet 110, 123, 127, 135, 355, 356, 360
- Receive-PSSession cmdlet 110
- Recurse switch parameter 63, 70, 84, 104, 204, 239
- recursive commands 294

- recursive listings, using custom functions 294
- reducing data
 - with Windows PowerShell parameters 352
 - with WQL queries 353
- reducing returned instances 383
- reducing returned properties 383
- referencing classes 302
- RegExTab.ps1 600
- Register-WmiEvent cmdlet 110
- registry
 - backing up 94
 - changing property values 97
 - editing 94
 - finding all drives 88
 - keys, checking for 529
 - searching for providers 529
 - setting missing property values 98
 - storing current location 94
- registry drives
 - checking for 529
 - creating 530
 - removing 530, 531
- registry keys
 - accessing stored values 90
 - creating 93–95
 - creating and assigning values 96
 - forscripting 569
 - listing from a registry hive 107
 - overwriting existing 95
 - setting default values 96
 - testing for properties 93, 94, 98
 - viewing stored values 89
- registry provider
 - capabilities 88, 90
 - creating registry drives 88
 - creating registry keys 93
 - default drives 88
 - listing registry keys 91
 - retrieving registry values 89
 - searching for software 92
 - setting default value for registry keys 96
- Registry resource provider 566, 569
- regular expressions
 - character patterns 601
 - escape sequences 599, 600
 - places to use 599

- relative distinguished name (RDN) *See* RDN (relative distinguished name)
- remote caller permissions 342
- remote computers, querying 294
- remote connections
 - alternate credentials 132
 - cmdlet errors 112
 - creating sessions 118–120
 - exiting 119
 - impersonating users 113
 - multiple 120
 - security 112, 339
 - specifying credentials for 112
 - stored sessions 119
 - using WinRM 114–118
- remote machines
 - changing working directory 118
 - checking domain controllers 441
 - checking domain password policy 440
 - configuring Windows PowerShell 114, 115
 - creating OUs (organizational unit) 443
 - entering PS sessions 439
 - importing Active Directory module 439
 - multiple connections 119
 - obtaining domain information 439
 - retrieving BIOS information 118
 - running commands against multiple 121
 - verifying operating systems 439
- remote procedure call (RPC) error 342
- Remote Server Administration Tools (RSAT) 431
- remote sessions
 - alternate credentials 132
 - capturing output from 118, 119
 - creating 135
 - loading Active Directory module 434
 - storing in a variable 119
- RemoteWMI_SessionNoDebug.ps1 476
- remoting
 - alternate credentials 342
 - bandwidth 348
 - cmdlets 109–111
 - configuring 135
 - connection errors 342
 - creating a session 118–120
 - discovering Active Directory 439–442
 - logged-on users 345
 - multiple connections 343
 - required ports 345
 - retrieving information 357, 358
 - specifying credentials 112
 - storing credentials 343
 - user permissions 342
 - using native WMI 345
 - WMI disadvantages 345
 - running WMI 345, 346
- remotejob type 356
- Remove-ADGroupMember cmdlet 445
- Remove-Computer cmdlet 110
- Remove-EventLog cmdlet 110
- Remove-Item cmdlet 75, 80, 84, 107, 177
- Remove-Job cmdlet 124, 556
- Remove-PSBreakpoint cmdlet 492, 496, 504, 554
- Remove-PSDrive cmdlet 105, 530
- Remove-PSSession cmdlet 110, 119
- Remove-PSSnapin cmdlet 554
- RemoveUserFromGroup.ps1 445
- Remove-Variable cmdlet 554
- Remove-WmiObject cmdlet 110
- Remove-WSManInstance cmdlet 110
- removing an environment variable 79
- removing PS drive mapping 105
- Rename-ADObject cmdlet 443, 457
- Rename-Computer cmdlet 110, 460
- Rename-Item cmdlet 79, 599
- renaming environment variables 79
- Repeat cmdlet 501
- Replace operator 599
- ReplicationSourceDC parameter 466
- #requires statement 242
- Reset parameter 446
- Resolve-ZipCode function 198
- Resolve-ZipCode.ps1 198
- Restart-Computer cmdlet 110, 461, 472
- restricted execution policy 522
- ResultClassName parameter 394
- ResultSetSize parameter 449
- RetrieveAndSortServiceState.ps1 146
- retrieving registry values 89
- retrieving specific variables 101
- retrieving WMI association classes 393
- return codes 363

- Root/Cimv2 375
- RPC error 342
- rsat-ad-tools 433, 460
- RSAT (Remote Server Administration Tools)
 - 431, 432
- run method 51
- running processes 22, 322–324
- run-time errors 474–478

S

- \s character pattern 601
- sAMAccountName attribute 405, 406
- script blocks
 - braces 185
 - definition 155
 - delimiting on functions 185
 - InlineScript 554
 - running statements 552
- script cmdlet 217
- script execution policy, setting 459
- Script method member types 196
- script parameter 492
- script property 35
- Script resource provider 566
- ScriptBlock parameter 133, 135
- ScriptFolderConfig.ps1 567
- ScriptFolderVersion.ps1 569
- scripting support, enabling 240
- scripts
 - See also* code; debugging
 - accessing Windows PowerShell with 10
 - adding error handling 404
 - avoiding aliases in 592
 - best practices 591–598
 - breaking lines of code 144
 - business logic 202–204
 - business rules 478
 - bypassing execution policies 143
 - calling configurations 569
 - constants 153, 154
 - creating 139
 - creating multiple folders 174–176
 - debugging 507–509
 - deleting multiple folders 176, 177
 - dot-sourcing 186–188

- downloading samples xxii
- enabling 57
- ending 167
- errors 143, 295, 473–479
- execution policies 140–143, 177
- function library 186
- impersonation levels 339
- including functions in 591, 592
- incorrect data types 532–536
- logic errors 478
- missing parameters 511–513, 521
- missing rights 521–523
- missing WMI providers 523–532
- modifying 204–207
- nonterminating errors 522
- profiles 57, 284
- program logic 202
- promoting readability of 593, 594
- quotation marks 139, 140
- reasons for 137–139
- referring to constants 153
- restricted execution policy 522
- reusing 186–188
- running 139, 140
- running faster 295
- running inside Windows PowerShell 147
- running manually 145–148
- running outside Windows PowerShell 148
- run-time errors 474–478
- signing 69
- simplifying 314
- singularizing strings 150
- skipping past errors 144
- sorting data 146
- status of services 146
- stepping through 483–489, 509
- stopping processes 144, 145
- storing profile information 284, 285
- strict mode 479, 488–493
- strings 150–152, 155, 156
- support options 140, 141
- suppressing queries 445
- suspending execution of 486
- syntax errors 473, 474
- syntax parser 474
- terminating errors 522

- timer, adding 333
- tracing 479–483
- use-case scenario 511
- using canonical aliases in 592
- using comments 593
- variables 144, 148–153
- SDDL 369
- SDDLToBinarySD method 369
- Search-ADAccount cmdlet 447, 457
- SearchBase parameter 451
- searching
 - for classes 298
 - for certificates 74, 75
 - for software 92
- security
 - See also* passwords
 - controlling execution of cmdlets 6, 7
 - remote connections 112, 339
- security groups 444, 445
- Security Descriptor Definition Language (SDDL) *See* SDDL
- security identifier (SID) *See* SID (security identifier)
- SecurityDescriptor property 527
- select * query 320
- Select statement 325
- selecting specific data 321
- Select-Object cmdlet 36, 297, 301, 303, 304, 310, 312, 315, 319, 322, 344, 378, 380, 394
 - Unique switched parameter 394
- Select-String cmdlet 302, 599
- sequence activity 559
- Sequence keyword 559, 563
- Sequence workflow activity 553
- sequences 562
- ServerManager module 397
- service accounts 327–329
- Service resource provider 566
- Set-ADAccountPassword cmdlet 446, 457
- Set-Alias cmdlet 554
- Set-Content cmdlet 85
- Set-DNSClientServerAddress cmdlet 465
- Set-ExecutionPolicy cmdlet 140, 177, 240, 268, 459, 522
- SetInfo() method 396, 405
- Set-Item cmdlet 96
- Set-ItemProperty cmdlet 97, 98, 482
- Set-Location cmdlet 67, 88, 118, 150, 335, 337
 - and complete drive names 68
 - changing location of registry drives 88
 - changing working location 94
 - switching PS drives 68
 - working with aliases 66
- Set-PropertyItem cmdlet 97
- Set-PSBreakpoint cmdlet 492, 554
- Set-PSDebug cmdlet 479, 509, 554
- Set-PSRepository cmdlet 586, 589
- Set-Service cmdlet 110
- SetServicesConfig.ps1 571
- Set-StrictMode cmdlet 490, 491, 554
- Set-TraceMode cmdlet 554
- Set-Variable cmdlet 102, 153, 554
- set verb 54
- Set-WmiInstance cmdlet 110
- Set-WSManInstance cmdlet 111
- shares
 - listing 327
 - maximum connections 322
 - reviewing 320
- ShellId variable 101
- shortcut keystroke combination 18
- shortcut name, using for local computer 312
- Should object 599
- Show-Command cmdlet 52–54
- Show-EventLog cmdlet 111
- SID (security identifier) 387
- signing scripts 69
- SimpleTypingError.ps1 489
- SimpleTypingErrorNotReported.ps1 490
- [single] alias 198
- singularizing strings 150
- sl alias 67, 70, 118, 337
 - See also* Set-Location cmdlet
- snap-ins 65, 66
- Snippets directory 268
- software, finding installed 332
- sort alias 55, 78 *See also* Sort-Object cmdlet
- sort order in tables 32
- sorting output 306–308
- Sort-Object cmdlet 55, 77, 146, 297, 306, 322, 327
- Split method 238

Split statement

- Split statement 599
- Start-DscConfiguration cmdlet 567, 574, 580
- Start-Job cmdlet 122, 128, 135, 360
- startName property 327
- Start-Service cmdlet 308
- Start-Transaction cmdlet 554
- Start-Transcript cmdlet 58, 118, 281, 554
- static methods 367
 - and double colons 369
 - definition 361
 - finding 368, 373
 - Invoke cmdlet 373
 - security 369
 - WMI 373
 - [wmiclass] type accelerator 369
- st attribute 413
- Status property 33
- Step parameter 485, 486, 509
- Step-Into cmdlet 501
- Step-Out cmdlet 501
- Step-Over cmdlet 501
- Stop-Computer cmdlet 111
- Stop-Job cmdlet 128
- StopNotepad.ps1 143
- StopNotepadSilentlyContinue.ps1 144
- stopping processes 223
- Stop-Process cmdlet 7–9, 22, 144, 223
- Stop (Quit) cmdlet 501
- Stop-Service cmdlet 308
- Stop-Transcript cmdlet 554
- Street attribute 400
- streetAddress attribute 413
- strict mode 479, 488–493
- Strict parameter 489
- [string] alias 198
- string characters 208
- string values 325
- strings 151, 152
 - See also* variables
 - breaking into arrays 238
 - concatenating 150
 - expanding 155, 163
 - literal 155, 156
 - singularizing 150
- subexpressions 534
- subject property 74
- subroutines 180
- SupportsExplicitShutdown property 527
- SupportsExtendedStatus property 527
- SupportsQuotas property 527
- SupportsSendStatus property 527
- SupportsShutdown property 527
- SupportsThrottling property 527
- Suspend-Workflow workflow activity 553
- Switch keyword 172
- switch parameters 53
- Switch statement 599
 - defining default condition 172
 - matching 172–174
- Switch_DebugRemoteWMI_Session.ps1 477
- switching PS drives 68
- syntax
 - retrieving 43
 - shortening 330, 331
- syntax errors in scripts 473, 474
- syntax parser 474
- Syntax switch parameter 43
- system classes 524, 526
- system properties
 - __Path 365
 - __RelPath 365, 366
 - removing 339
- system requirements xxi
- System.Boolean property types 526, 527
- system.DirectoryServices.DirectoryEntry object 396
- System.Int32 property types 526, 527
- System.IO.DirectoryInfo object 82
- System.IO.FileInfo class 238
- System.IO.FileInfo objects 82
- System.String class 238
- System.String property types 526, 527
- System.SystemException class 199
- System.UInt32 property types 527

T

- \t escape sequence 599
- tab completion 24, 46, 51
- tab expansion 393
 - and qualifier names 382
 - and CIM cmdlets 375

- tables
 - adding filters 33
 - displaying pipelined data 31
 - sorting column data 32
- TargetObject property 402
- telephone settings 416–418
- temp variable 82
- template files, creating 596
- Terminate method 361, 364
- terminating errors 522
- terminating instance methods 363
 - directly 363, 373
 - in Windows PowerShell 2.0 364
 - using WMI 364, 373
 - Win32_Process WMI class 370
 - [wmi] type accelerator 366
- Test-ComputerPath.ps1 516
- Test-Connection cmdlet 111, 476, 514, 546
- Test-DscConfiguration function 571
- Test-Mandatory function 225
- Test-ModulePath function 236, 239
- Test-Path cmdlet 93, 98, 236, 278, 289, 480, 529
 - determining if a registry key exists 98
 - registry key property 98
- Test-PipedValueByPropertyName function 228
- TestTryCatchFinally.ps1 539
- TestTryMultipleCatchFinally.ps1 541
- Test-ValueFromRemainingArguments
 - function 228
- Test-WSMan cmdlet 111
- text files
 - creating new 107
 - reading 177
 - turning into arrays 429
- Text parameter 268
- TextFunctions.ps1 188
- Then keyword 168
- throttling 548
- time, finding current 339
- timers, adding to scripts 333
- Title parameter 268
- Today parameter 201
- \$total variable 202
- totalSeconds property 333
- trace levels 480–483, 487
- Trace parameter 479

- Trace-Command cmdlet 554
- tracing features, implementing in
 - functions 529
- tracing scripts 479–483
- Trap keyword 199
- trusted locations 586
- Try block 538, 539
- Try...Catch...Finally 546
 - catching multiple errors 541–543
 - catching specific errors 542
 - using 538–541, 545, 546
- type accelerators
 - [wmi] 366
 - [wmiiclass] 369
- type constraints 198, 216

U

- \u0020 escape sequence 600
- UID attribute 400
- underlining, sizing to text 188
- Undo-Transaction cmdlet 554
- uninitialized variables 489, 491
- Uninstall-Module cmdlet 589
- Unique switch parameter 394
- universal security group, creating 444
- UnloadTimeout properties 527
- Unlock-ADAccount cmdlet 448, 457
- unprotect verb 54
- Update-Help cmdlet 12, 13, 99
- UpdateHelpTrackErrors.ps1 13, 14
- updates, errata, and book support xxiii
- url attribute 410
- use verb 54
- UseADCmdletsToCreateOuComputerAndUser.ps1
 - 444
- use-case scenario 511
- User Account Control (UAC) 521
- user account control values 408, 409
- User resource provider 566
- UserAccountControl attribute 408
- user-defined aliases 592
- username property 63
- users
 - adding to security groups 444
 - assigning passwords 446, 457

Use-Transaction cmdlet

users (*continued*)

- creating 405, 446, 447, 457
- creating address pages 412
- creating multiple 418, 419
- creating multivalued 425–429
- currently logged on 63
- deleting 422, 423, 429
- enabling accounts 446, 447
- finding disabled accounts 449–451
- finding unused accounts 451–454
- locked accounts 447, 448, 457
- managing 443–445
- modifying organizational settings 420–422
- modifying profile settings 414–416
- modifying properties 410
- modifying telephone settings 416–418
- moving to OUs 446
- removing from security groups 445
- retrieving properties 452
- running as different 113
- unlocking accounts 447, 457

Use-Transaction cmdlet 554

V

- \v escape sequence 600
- value parameter 69, 78, 96, 481
- ValueFromPipelineByPropertyName
 - property 228
- ValueFromPipeline property 228
- ValueFromRemainingArguments parameter
 - property 228
- variable provider 99–101
- variable scope 184
- variables
 - See also* constants; strings
 - automatic 148, 149
 - best practices 597
 - breakpoint access modes 495
 - case sensitivity 84
 - computer environment, listing 335
 - constraints 152
 - creating 177
 - data type aliases 152, 153
 - definition 148
 - listing 100, 107

- naming 594, 597
- printing info for 196
- retrieving 101
- returned job objects as 126
- scripts 144, 148–153
- setting breakpoints on 495–499, 509
- storing objects in 127
- storing returned objects in 124
- strings 150–152
- uninitialized 489, 491
- Windows environment 334–339
- verb-noun combinations 180
- verb-noun naming convention 54
- verbose messages 218, 219, 257
- verbose output, directing to text files 14
- Verbose switch parameter 12, 14, 218, 235, 526, 528, 574, 580
- \$VerbosePreference variable 218
- verbs

- approved list of 184
- checking authorized 234
- displaying 56
- distribution of 55–57
- finding patterns 55
- get 54
- getting list of 54
- grouping 55
- in naming convention 54
- set 54
- unapproved 235
- unprotect 54
- use 54

- verifying old executable files 75
- Version argument 11
- version property 182, 527

W

- \w character pattern 601
- Wait switch parameter 574
- WaitForAll resource provider 566
- WaitForAny resource provider 566
- WaitForSome resource provider 566
- Wait-Job cmdlet 127
- WbemTest 367
- Wend keyword 155

- WhatIf switch parameter 6, 7, 12, 22, 74, 84, 222, 223, 257
- whenCreated property 452
- where alias *See* Where-Object cmdlet
- Where clause 325, 326
- Where method 87
- Where-Object cmdlet 60, 66, 67, 81, 111, 153, 310, 599
- While loop 154, 156
- WhileReadLine.ps1 156
- While statement 162
 - constructing 154, 155
 - using 156
- white space, finding in files 601, 602
- whoami command 132
- Width parameter 52
- wildcard patterns 233, 234, 299
- wildcards 382
 - and qualifier queries 382
 - finding classes 298, 299
 - finding cmdlets 36
 - finding installed modules using 587
 - PowerShell Gallery 585
 - using in help 17
 - using to retrieve methods 48
- [wmi] accelerators 197
- WIM (Windows Information Model) 355–357
- Win32_Bios class 315, 347, 383, 523
- Win32_ComputerSystem WMI class 315
- Win32_Environment WMI class 334
- Win32_LoggedOnUser class 344, 345
- Win32_LogicalDisk class 195–197, 318
- Win32_LogonSession class 385
- Win32_PingStatus class 516
- Win32_PNPEntity WMI class 394
- Win32_Process class 385
- Win32_Product class 525, 528
- Win32_Service class 356, 384
- Win32_Share class 320, 321
- Win32_SystemAccount class 385, 387
- Win32_UserAccount class 385, 387, 388
- Win32_VideoController WMI class 393
- window size, controlling 52
- Windows 8, PING command errors 117
- Windows 10 Client 3
- Windows directory, finding path to 51

- Windows Management Instrumentation Tester (WbemTest) *See* WbemTest
- Windows Management Instrumentation (WMI) *See* WMI (Windows Management Instrumentation)
- Windows PowerShell
 - accessing 10
 - case sensitivity 24
 - changing working directory 2
 - classic remoting 109
 - code wrapping 324
 - configuring on remote machines 114, 115
 - configuring the console 11
 - deploying 3, 4
 - displaying verbs 56
 - DSC (Desired State Configuration) 565
 - help files 12–19
 - installing 3
 - interactivity 3
 - launch options 11
 - producing directory listings 2
 - running as different user 113, 114
 - running single commands 120–122
 - security issues 6–9
 - transcript tool 118
 - using command-line utilities 4–6
 - verb distribution 55–57
 - verb grouping 55
- Windows PowerShell console, configuring 11
- Windows PowerShell ISE
 - building commands 260
 - calling WMI methods 270–272
 - Commands add-on 260
 - editing commands 262
 - finding commands 262
 - IntelliSense 264
 - locating commands 260
 - navigating 260–262
 - optimal screen resolution 262
 - reviewing commands 262
 - running commands from script pane 263, 274
 - snippets *See* Windows PowerShell ISE snippets
 - Snippets directory 268
 - starting 259

Windows PowerShell ISE snippets

- Windows PowerShell ISE (*continued*)
 - starting from Windows 10 259
 - turning off Commands add-on 264
- Windows PowerShell ISE snippets 266–270
 - completing functions 266, 267
 - creating code 266
 - creating functions 266
 - creating new 268, 274
 - definition 266
 - deleting 269, 270, 274
 - required parameters 268
 - using 272, 273
- Windows PowerShell profile
 - creating 58, 59
 - definition 57
- Windows PowerShell remoting
 - cmdlets 109–111
 - creating a session 118–120
 - credentials 342–344
 - native support for 109
 - previous versions 116
 - running WMI 345–347
- Windows Remote Management (WinRM)
 - See WinRM (Windows Remote Management)
- Windows service information 306–308
- WindowsFeature resource provider 566
- WindowsOptionalFeature resource provider 566
- WindowsProcess resource provider 566
- WinNT provider 397
- WinRM (Windows Remote Management)
 - accessing remote systems 114–118
 - and Windows 10 114, 115
 - definition 114
 - errors 117
 - Windows 8 client systems 117
- WMI (Windows Management Instrumentation)
 - case sensitivity 380
 - classes 298–300
 - commands, running on multiple computers 360
 - configuring using group policy 341
 - connecting 312
 - connecting to, default values 313, 339
 - consumers 292
 - deprecated classes 381
 - disadvantages of 345
 - dynamic classes 382
 - elements 293
 - evaluating return codes 363
 - filtering classes 379
 - finding classes 394
 - finding class methods 377–381
 - finding dynamic classes 394
 - finding installed software 332
 - information, retrieving 360
 - infrastructure 292
 - model, described 292
 - namespaces 296
 - obtaining specific data 197
 - providers 292
 - queries 294, 301–305
 - querying abstract WMI classes 382
 - and remoting 345
 - repository 292
 - resources 292
 - retrieving instances 392
 - retrieving results 360
 - scripts, simplifying 314
 - sections 292
 - service 292
 - service information, retrieving with 309–311
- WMI association classes
 - finding 385
 - retrieving 393
- WMI class methods, finding 377
- WMI classes
 - finding 394
 - Win32_BIOS 383
 - Win32_DisplayConfiguration 381
 - Win32_PNPEntity 394
 - Win32_Service 384
 - Win32_SystemAccount 387
 - Win32_UserAccount 387, 388
 - Win32_VideoController 393
- WMI instances, retrieving 383
- WMI Microsoft Installer (MSI) 332
- WMI query argument 326
- wmijob type 356
- workflow activities

- adding checkpoints 558
- core cmdlets as 553
- definition 552
- disallowed core cmdlets 554
- InlineScript 554
- list of 552
- non-automatic cmdlets 554
- parallel 555
- using CheckPoint-Workflow 558
- Windows PowerShell cmdlets as 553
- Workflow keyword 548, 563
- workflows
 - adding checkpoints 556, 562
 - adding logic with cmdlets 549
 - adding sequence activities 559, 560
 - adding sequences 562
 - checkpointing 556–559
 - creating 561, 563
 - creating blocks of sequential statements 553
 - creating checkpoints 552, 563
 - errors 551
 - handling interruptions with checkpoints 556
 - ordering 563
 - packaging in modules 547
 - performing parallel activities 548, 550
 - persistence points 547
 - placing checkpoints 556
 - reasons to use 547, 548
 - recovering 556
 - requirements 548
 - resuming 556

- running against remote computers 563
- running on remote servers 555
- running parallel statements 552
- running statements simultaneously 552
- syntax 549
- throttling 548
- writing 547, 548
- working with aliases 66
- working with directory listings 80
- WQL queries 331, 353, 360
- Wrap parameter 350
- Write-Debug cmdlet 476
- Write-EventLog cmdlet 111
- Write-Host cmdlet 332, 554
- Write-Verbose cmdlet 257
- wscript.shell 50
- wshShell object 63
 - creating a new instance 50, 51
 - program ID 51
- \$wshShell variable 51
- WS-Management protocol 114
- WSMan provider 66

X

- \x20 escape sequence 600
- [xml] alias 198

Z

- \$zip variable 198

About the author



ED WILSON is the Microsoft Scripting Guy and a well-known scripting expert. He writes the daily *Hey Scripting Guy!* blog. He has also spoken at TechEd and at the Microsoft internal TechReady conferences. He has written more than a dozen books, including nine on Windows scripting that were published by Microsoft Press. He has also contributed to nearly a dozen other books. His newest book with Microsoft Press is *Windows PowerShell Best Practices*. Ed holds more than 20 industry certifications, including Microsoft Certified Systems Engineer (MCSE) and Certified Information Systems Security Professional (CISSP). Prior to coming to work for Microsoft, he was a senior consultant for a Microsoft Gold Certified Partner, where he specialized in Active Directory design and Microsoft Exchange implementation. In his spare time, he is writing a mystery novel. For more about Ed, you can go to ewblog.edwilson.com/ewblog/.