Professional Expertise Distilled

PowerShell Troubleshooting Guide

Minimize debugging time and maximize troubleshooting efficiency by leveraging the unique features of the PowerShell language

Michael Shepard

[PACKT] PUBLISHING enterprise
professional expertise distilled

PowerShell Troubleshooting Guide

Minimize debugging time and maximize troubleshooting efficiency by leveraging the unique features of the PowerShell language

Michael Shepard

[PACKT] PUBLISHING enterprise
professional expertise distilled

BIRMINGHAM - MUMBAI

PowerShell Troubleshooting Guide

First published: November 2014

Production reference: 1211114

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78217-357-1

www.packtpub.com

Cover image by Michal Jasej (milak6@wp.pl)

Credits

Author
Michael Shepard

Reviewers
Christian Droulers
Rob Huelga
Steve Shilling

Acquisition Editor
Meeta Rajani

Content Development Editor
Adrian Raposo

Technical Editors
Tanvi Bhatt
Pragnesh Billimoria

Copy Editors
Simran Bhogal
Maria Gould
Ameesha Green

Project Coordinator
Kinjal Bari

Proofreaders
Simran Bhogal
Joel T. Johnson

Indexer
Monica Ajmera Mehta

Production Coordinator
Alwin Roy

Cover Work
Alwin Roy

About the Author

Michael Shepard has been working with computers since the early '80s, starting with an Apple II in school and a Commodore 64 at home. He started working in the IT industry in 1989 and has been working full-time since 1997. He has been working at Jack Henry & Associates, Inc. since 2000. His focus has changed over the years from being a database application developer to a DBA, an application admin, and is now a solutions architect. In his years as a DBA, he found PowerShell to be a critical component in creating the automation required to keep up with a growing set of servers and applications. He is active in the PowerShell community at Stack Overflow and projects at CodePlex. He has been blogging about PowerShell since 2009 at `http://powershellstation.com`.

I'd like to thank my employer, Jack Henry & Associates, Inc., for allowing me the freedom over the last few years to both learn and teach PowerShell. My wonderful wife, Stephanie, and my children, Simeon and Gwen, also deserve thanks for humoring me when I can't stop talking about PowerShell, and for giving me some breathing room to write.

About the Reviewers

Christian Droulers is a late-blooming software developer. He only started programming in college and has not stopped since. He's interested in beautiful, clean, and efficient code.

Steve Shilling has worked in the IT industry commercially since 1987, but started with computers in 1982 writing BASIC programs and debugging game programs written by others. He has broad knowledge about Unix, Windows, and Mainframe systems. He primarily lives in the Unix/Linux world automating systems for deployments and businesses, and has spent many years working in system administration, software development, training, and managing technical people. He remains in the technical field of expertise providing knowledge and experience to companies around the world to make their systems stable, reliable, and delivered on time. His experience has taken him through many different industries covering banking and finance, insurance services, betting exchanges, TV and media, retail, and others, allowing him to have a unique perspective of IT in business where most have only worked in one industry.

Steve works for TPS Services Ltd., which specializes in IT training and consultancy, life coaching, management training, and counselling. The IT part of TPS Services Ltd. specializes in Linux/Unix systems for small, medium, and large organizations, and the integration of Linux and Windows systems.

Table of Contents

Preface **1**

Chapter 1: PowerShell Primer **7**

- **Introduction** **7**
- **Cmdlets** **8**
  - The big three cmdlets 10
- **Functions** **13**
- **Scripts** **14**
- **Pipelines** **15**
  - How pipelines change the game 17
  - What's the fuss about sorting? 19
- **Variables** **20**
- **Modules** **21**
- **Further reading** **22**
- **Summary** **23**

Chapter 2: PowerShell Peculiarities **25**

- **PowerShell strings** **25**
  - String substitution 26
  - How string substitution goes wrong 28
  - Escaping in PowerShell strings 29
- **Function output** **29**
- **Pipeline processing** **32**
- **PowerShell error handling** **37**
  - The trap statement 37
  - try, catch, and finally statements 39
  - Non-terminating errors 40
- **Further reading** **42**
- **Summary** **43**

Chapter 3: PowerShell Practices 45

Filter left **45**
Format right **47**
Comment-based help **49**
Using Write-* cmdlets **53**
Write-Host 53
Output – the correct way 54
What about the other Write-* cmdlets? 55
Which Write should I use? 57
Write-Verbose 57
Write-Debug 58
Write-Warning 58
Write-Error 58
Write-Progress 59
Further reading **60**
Summary **61**

Chapter 4: PowerShell Professionalism 63

Naming conventions **63**
Cmdlet and function naming 64
Parameter naming 65
Module naming 66
Variable naming 67
Modularization **67**
Breaking a process into subtasks 67
Helper functions 68
Single responsibility principle 69
Don't repeat code 69
Understanding the process 70
Version control **71**
Using version control with PowerShell 73
Unit testing **73**
Rolling your own unit tests 74
Why is PowerShell testing difficult? 77
An introduction to Pester 78
Mocking with Pester 80
Further reading **82**
Summary **83**

Chapter 5: Proactive PowerShell 85

Error handling **85**
Error-handling guidelines 86

Error-handling techniques 86
Investigating cmdlet error behavior 88
Catch and release 89
CmdletBinding() 90
Common parameter support 91
SupportsShouldProcess 92
Parameter name validation 93
Parameter value validation 94
Pipeline input 96
Pipelines and function execution 99
Parameter type transformation 102
#REQUIRES statements 104
Set-StrictMode and Set-PSDebug -strict 105
Further reading 107
Summary 107
Chapter 6: Preparing the Scripting Environment 109
Validating operating system properties 109
Workstation/server version 110
Operating system version 111
Putting it all together 114
Validating service status 115
Validating disk and memory availability 116
Validating network connectivity 119
Using telnet 119
Using Test-NetConnection 120
Writing Test-NetConnection in downstream versions 120
Testing UDP and ICMP connectivity 122
Validating connectivity prior to implementation 123
Putting it all together 124
Further reading 127
Summary 127
Chapter 7: Reactive Practices – Traditional Debugging 129
Reading error messages 130
The color problem 130
Changing console colors 131
Changing ISE colors 132
PowerShell profiles 134
Error message content 134
Using Set-PSDebug 137

Debugging in the console **140**
Debugging in the ISE **145**
Event logs **147**
Listing event logs 149
Reading event logs 150
Writing to event logs 152
The PSDiagnostics module **153**
Using –confirm and –whatif **154**
Reducing input set **156**
Using Tee-Object to see intermediate results **158**
Replacing the foreach loop with the foreach-object cmdlet **159**
Further reading **165**
Summary **166**
Chapter 8: PowerShell Code Smells 167
Code smells **168**
Code smells, best practices, antipatterns, and technical debt 168
Language-agnostic code smells **169**
PowerShell-specific code smells **169**
Missing Param() statements **170**
Homegrown common parameters **172**
Unapproved verbs **173**
Accumulating output objects **174**
Sequences of assignment statements **175**
Using untyped or [object] parameters **176**
Static analysis tools – ScriptCop and Script Analyzer **180**
ScriptCop 180
Script Analyzer 182
Further reading **185**
Summary **186**
Index 187

Preface

PowerShell Troubleshooting Guide uses easy-to-understand examples to explain the PowerShell language, enabling you to spend more of your time writing code to solve the problems you face and less time agonizing over syntax and cryptic error messages. Beginning with the foundations of PowerShell, including functions, modules, and the pipeline, you will learn how to leverage the power built into the language to solve problems and avoid reinventing the wheel. Writing code in PowerShell can be fun, and once you've learned the techniques in this book, you will enjoy PowerShell more and more.

What this book covers

Chapter 1, *PowerShell Primer*, provides a brief introduction to some of the most important entities in the PowerShell language including cmdlets, functions, scripts, and modules. A special emphasis is placed on the importance of the pipeline in PowerShell operations.

Chapter 2, *PowerShell Peculiarities*, includes a number of features of the PowerShell language, which are unusual when compared with other mainstream programming languages. Examples of these topics are output from functions and non-terminating errors.

Chapter 3, *PowerShell Practices*, shows a few ways that the scripting experience in PowerShell can be improved, either in performance or in maintainability. A lengthy discussion of the various output cmdlets is included.

Chapter 4, *PowerShell Professionalism*, gives examples of practices that might not be as familiar to traditional system administrators but are common among professional developers. These practices will help scripters create more reliable products and be more confident when making changes to existing codebases.

Chapter 5, *Proactive PowerShell*, presents a number of practices that, when applied to code, will result in more flexible code with fewer bugs. In a sense, this is pre-emptive troubleshooting, where we create our code thoughtfully in order to reduce the need for troubleshooting later.

Chapter 6, *Preparing the Scripting Environment*, covers the idea of knowing the characteristics of the environment in which your scripts are running. We also spend some effort trying to weed out network connectivity issues.

Chapter 7, *Reactive Practices – Traditional Debugging*, shows how to perform traditional troubleshooting in PowerShell using the debugging features of the console and the ISE, along with other techniques. It wraps up with an example of how using the wrong PowerShell feature to perform an operation can lead to poor performance.

Chapter 8, *PowerShell Code Smells*, explains the concept of code smells (signs of poorly implemented code) and compares it with antipatterns, best practices, and technical debt. It then shows some ways that PowerShell code might begin to smell.

What you need for this book

Most of the examples in the book will work with PowerShell Version 2.0 and above. In places where a higher version of the engine is required, it will be indicated in the text. You should have no problems running the provided code on either a client installation (Windows 7 or greater) or a server installation (Windows Server 2008 R2 or greater).

Who this book is for

This book is intended for anyone who has some experience with PowerShell and would like to expand their understanding of the language design and features in order to spend less time troubleshooting their code. The examples are designed to be understood without needing any specific application knowledge (for example, Exchange, Active Directory, and IIS) in order to keep the intent clear. A few sections are aimed at system administrators specifically. This is due to the different skill set that most administrators bring to the table compared with developers. However, the points made are applicable to anyone using PowerShell.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `Get-Help` cmdlet has a number of switches that control precisely what help information is displayed."

A block of code is set as follows:

```
param($name)
    $PowerShellVersion=$PSVersionTable.PSVersion
    return "We're using $PowerShellVersion, $name!"
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#find largest 5 items in the directory tree
dir -recurse |
  tee-object -Variable Files |
  sort-object Length |
  tee-object -Variable SortedFiles |
  select-object -last 5
```

Any command-line input or output is written as follows:

```
Get-ChildItem "c:\program files" -include *.dll -recurse
```

New terms and **important** words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "It features a button labeled **Scan Script**, a gear button for options, and a grid for results."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
PowerShell Primer

This chapter will give you a very brief overview of the main features of the PowerShell language. By the end of the chapter, you will be familiar with the following topics:

- Cmdlets
- Functions
- Scripts
- Pipelines
- Variables
- Modules

Introduction

Windows PowerShell (or just PowerShell, for short) was introduced by Microsoft in late 2006 accompanied by little fanfare. In the last seven years, PowerShell has gone from being what might have seemed like a research project to what is now the mainstay of Windows automation and is included in every Windows operating system and most of the major Microsoft products including Exchange, System Center, SQL Server, SharePoint, and Azure.

PowerShell is often thought of as a command-line language, and that is an accurate (but incomplete) view. Working on the command line in PowerShell is a joy compared to MS-DOS batch files and most of the command-line tools that IT professionals are used to having at their fingertips work with no changes in the PowerShell environment. PowerShell is also a first-class scripting language where the knowledge you gain from the command line pays off big time. Unlike MS-DOS, PowerShell was designed from the beginning to be a powerful tool for scripting. Unlike VBScript, there is an interactive PowerShell console that allows you to iteratively develop solutions a bit at a time as you work your way through a sequence of objects, methods, and properties.

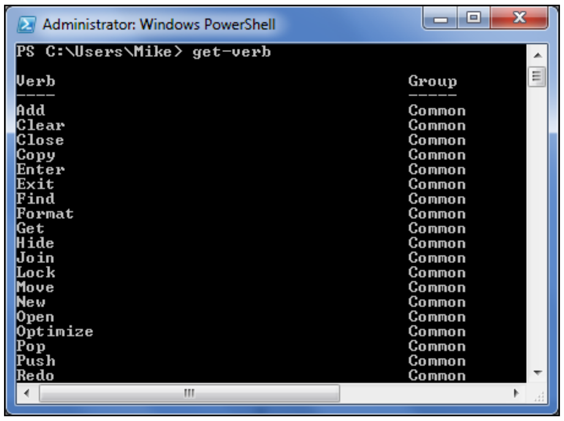
PowerShell includes several different elements that work together to create a very powerful and flexible ecosystem. While this chapter will give you an overview of several of these pieces, be aware that the PowerShell language is the subject of many books. For in-depth coverage of these topics, refer to *PowerShell In Practice* by Don Jones, Jeffery Hicks, and Richard Siddaway, *Manning Publications*.

Cmdlets

In PowerShell, a **cmdlet** (pronounced "command-let") describes a unit of functionality specific to PowerShell. In version 1.0 of PowerShell, the only way to create a cmdlet was by using managed (compiled) code, but 2.0 introduced **advanced functions**, which have the same capabilities as cmdlets. Built-in cmdlets exist to interact with the filesystem, services, processes, event logs, WMI, and other system objects. Some examples of cmdlets, which also show the flexibility in parameter passing, are shown as follows:

- `Get-ChildItem "c:\program files" -include *.dll -recurse`: This cmdlet outputs all `.dll` files under `c:\program files`
- `Get-EventLog Application -newest 5`: This cmdlet outputs the five most recent entries in the `Application` event log
- `Set-Content -path c:\temp\files.txt -value (dir c:\)`: This cmdlet writes a directory listing to a text file

Cmdlets are named with a two-part construction: verb-noun. Verbs in PowerShell describe the actions to be performed and come from a common list provided by Microsoft. These include `Get`, `Set`, `Start`, `Stop`, and other easy-to-remember terms. The `Get-Verb` cmdlet provides the list of approved verbs with some information on how the verbs can be grouped. The following screenshot shows the beginning of the list of verbs and their corresponding groups:



```
PS C:\Users\Mike> get-verb

Verb                                        Group
----                                        -----
Add                                         Common
Clear                                       Common
Close                                       Common
Copy                                        Common
Enter                                       Common
Exit                                        Common
Find                                        Common
Format                                      Common
Get                                         Common
Hide                                        Common
Join                                        Common
Lock                                        Common
Move                                        Common
New                                         Common
Open                                        Common
Optimize                                    Common
Pop                                         Common
Push                                        Common
Redo                                        Common
```

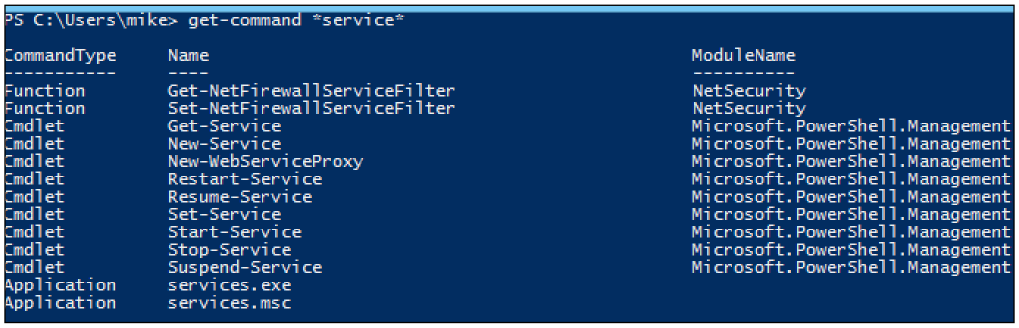
PowerShell nouns specify on which kind of objects the cmdlet operates. Examples of nouns are `Service`, `Process`, `File`, or `WMIObject` Unlike the list of verbs, there is no managed list of approved nouns. The reason for this is simple. With every new version of Windows, more and more cmdlets are being delivered which cover more and more of the operating system's features. An up-to-date reference for verbs along with guidance between similar or easily confused verbs can be found at `http://msdn.microsoft.com/en-us/library/ms714428.aspx`.

Putting nouns and verbs together, you get full cmdlet names such as `Get-Process` and `Start-Service`. By providing a list of verbs to choose from, the PowerShell team has gone a long way toward simplifying the experience for users. Without the guidance of a list such as this, cmdlet authors would often be forced to choose between several candidates for a cmdlet name. For instance, `Stop-Service` is the actual cmdlet name, but names such as `Kill-Service` and `Terminate-Service` would both convey the same effect. Knowing that `Stop` is the approved verb not only makes the decision simple, it also makes it simple to guess how one would terminate a process (as opposed to a service). The obvious answer would be `Stop-Process`.

Cmdlets each have their own set of parameters that allow values to be supplied on the command line or through a pipeline. Switch parameters also allow for on/off options without needing to pass a value. There is a large set of common parameters that can be used with all cmdlets. Cmdlets that modify the state of the system also generally allow the use of the `-Whatif` and `-Confirm` risk mitigation parameters. Common parameters and risk mitigation parameters are covered in detail in *Chapter 5*, *Proactive PowerShell*.

The big three cmdlets

When learning PowerShell, it's customary to emphasize three important cmdlets that are used to get PowerShell to give information about the environment and objects that are returned by the cmdlets. The first cmdlet is `Get-Command`. This cmdlet is used to get a list of matching cmdlets, scripts, functions, or executables in the current path. For instance, to get a list of commands related to services, the `Get-Command *service*` command would be a good place to start. The list displayed might look like this:



```
PS C:\Users\mike> get-command *service*

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Get-NetFirewallServiceFilter                       NetSecurity
Function        Set-NetFirewallServiceFilter                       NetSecurity
Cmdlet          Get-Service                                        Microsoft.PowerShell.Management
Cmdlet          New-Service                                        Microsoft.PowerShell.Management
Cmdlet          New-WebServiceProxy                                Microsoft.PowerShell.Management
Cmdlet          Restart-Service                                    Microsoft.PowerShell.Management
Cmdlet          Resume-Service                                     Microsoft.PowerShell.Management
Cmdlet          Set-Service                                        Microsoft.PowerShell.Management
Cmdlet          Start-Service                                      Microsoft.PowerShell.Management
Cmdlet          Stop-Service                                       Microsoft.PowerShell.Management
Cmdlet          Suspend-Service                                    Microsoft.PowerShell.Management
Application     services.exe
Application     services.msc
```

The thought behind listing `Get-Command` as the first cmdlet you would use is that it is used to discover the name of cmdlets. This is true, but in my experience you won't be using `Get-Command` for very long. The verb-noun naming convention combined with PowerShell's very convenient tab-completion feature will mean that as you get familiar with the language you will be able to guess cmdlet names quickly and won't be relying on `Get-Command`. It is useful though, and might show you commands that you didn't know existed. Another use for `Get-Command` is to figure out what command is executed. For instance, if you encountered the `Compare $a $b` command line and didn't know what the `Compare` command was, you could try the `Get-Command` command to find that `Compare` is an alias for `Compare-Object`.

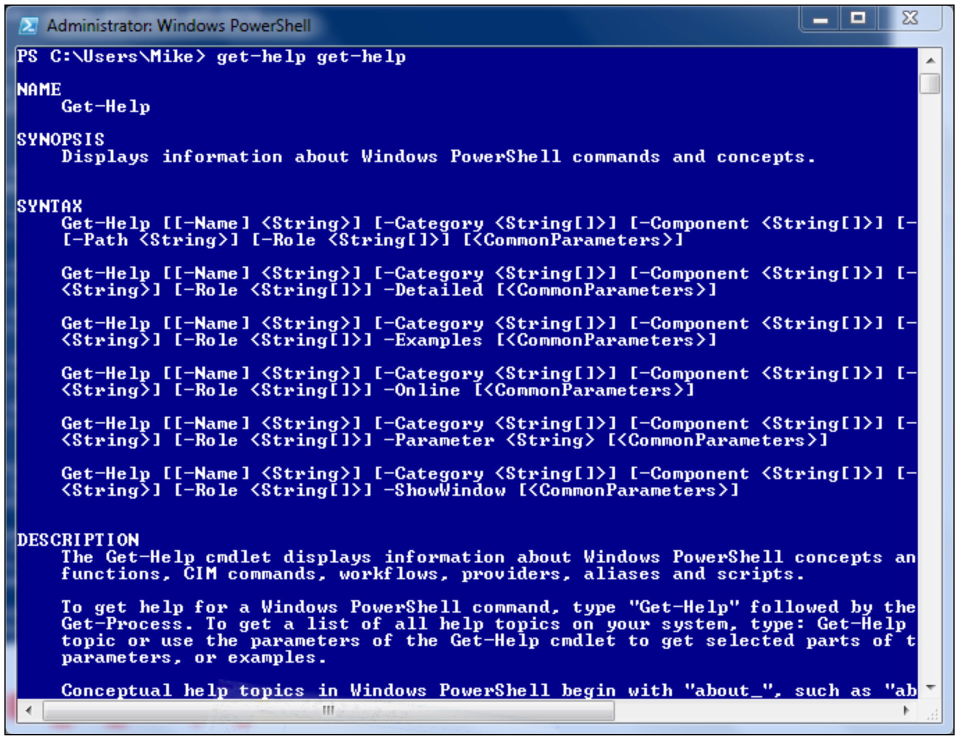
PowerShell provides aliases for two reasons. First, to provide aliases that are commands in other shells (such as `dir` or `ls`), which lead us to PowerShell cmdlets that perform similar functions. Secondly, to give abbreviations that are shorter and quicker to type for commonly used cmdlets (for example, `?` for `Where-Object` and `gsv` for `Get-Service`). In the PowerShell community, a best practice is to use aliases only in the command line and never in scripts. For that reason, I will generally not be using aliases in example scripts.

A similar trick can be used to find out where an executable is found: `Get-Command nslookup | Select-Object Path` returns the path `C:\Windows\system32\nslookup.exe`.

The second and probably most important cmdlet is `Get-Help`. `Get-Help` is used to display information in PowerShell's help system. The help system contains information about individual cmdlets and also contains general information about PowerShell-related topics. The cmdlet help includes syntax information about parameters used with each cmdlet, detailed information about cmdlet functionality, and it also often contains helpful examples illustrating common ways to use the cmdlet.

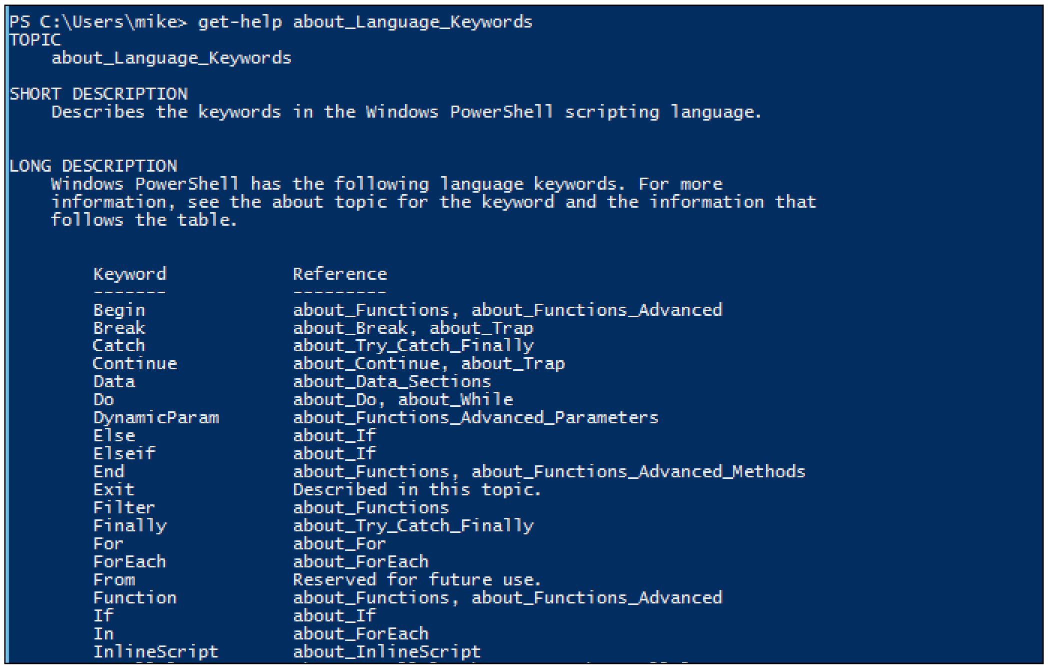
Pay attention to the help files. Sometimes, the problem you are having is because you are using a cmdlet or parameter differently than the designer intended. The examples in the help system might point you in the right direction.

The following screenshot shows the beginning of the help information for the `Get-Help` cmdlet:



```
PS C:\Users\Mike> get-help get-help

NAME
    Get-Help

SYNOPSIS
    Displays information about Windows PowerShell commands and concepts.


SYNTAX
    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>] [-
    [-Path <String>] [-Role <String[]>] [<CommonParameters>]

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

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

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

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

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


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

    To get help for a Windows PowerShell command, type "Get-Help" followed by the
    Get-Process. To get a list of all help topics on your system, type: Get-Help
    topic or use the parameters of the Get-Help cmdlet to get selected parts of t
    parameters, or examples.

    Conceptual help topics in Windows PowerShell begin with "about_", such as "ab
```

Another source of information in the help files are topics about the PowerShell language. The names of these help topics start with `about_`, and range from a few paragraphs to several pages of detailed information. In a few cases, the `about_` topics are more detailed than most books' coverage of them. The following screenshot shows the beginning of the `about_Language_Keywords` topic (the entire topic is approximately 13 pages long):

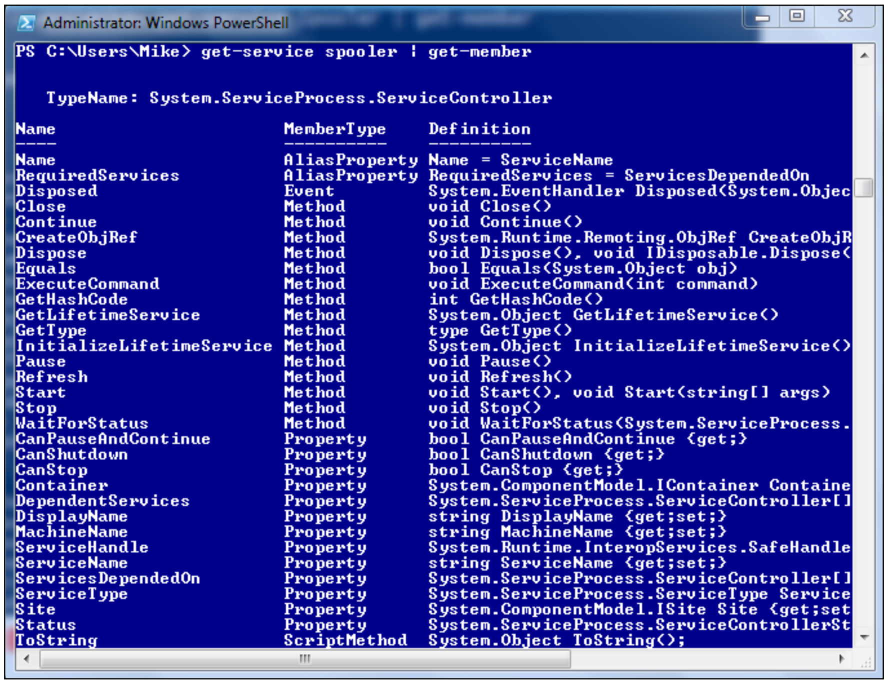


```
PS C:\Users\mike> get-help about_Language_Keywords
TOPIC
    about_Language_Keywords

SHORT DESCRIPTION
    Describes the keywords in the Windows PowerShell scripting language.


LONG DESCRIPTION
    Windows PowerShell has the following language keywords. For more
    information, see the about topic for the keyword and the information that
    follows the table.


        Keyword            Reference
        -------            ---------
        Begin              about_Functions, about_Functions_Advanced
        Break              about_Break, about_Trap
        Catch              about_Try_Catch_Finally
        Continue           about_Continue, about_Trap
        Data               about_Data_Sections
        Do                 about_Do, about_While
        DynamicParam       about_Functions_Advanced_Parameters
        Else               about_If
        Elseif             about_If
        End                about_Functions, about_Functions_Advanced_Methods
        Exit               Described in this topic.
        Filter             about_Functions
        Finally            about_Try_Catch_Finally
        For                about_For
        ForEach            about_ForEach
        From               Reserved for future use.
        Function           about_Functions, about_Functions_Advanced
        If                 about_If
        In                 about_ForEach
        InlineScript       about_InlineScript
```

The `Get-Help` cmdlet has a number of switches that control precisely what help information is displayed. The default display is somewhat brief and can be expanded by using the `-Full` or `-Detailed` switches. The `-Examples` switch displays the list of examples associated with the topic. The full help output can also be viewed in a pop-up window in PowerShell 3.0 or higher using the `-ShowWindow` switch.

PowerShell 3.0 and above do not ship with any help content. To view help in these systems you will need to use the `Update-Help` cmdlet in an elevated session.

The final member of the big three is `Get-Member`. In PowerShell, all output from commands comes in the form of objects. The `Get-Member` cmdlet is used to display the members (for example, properties, methods, and events) associated with a set of objects as well as the types of those objects. In general, you will pipe objects into `Get-Member` to see what you can do with those objects. An example involving services is shown as follows:



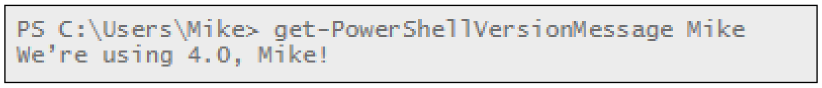
```
PS C:\Users\Mike> get-service spooler | get-member


   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
RequiredServices          AliasProperty RequiredServices = ServicesDependedOn
Disposed                  Event         System.EventHandler Disposed(System.Objec
Close                     Method        void Close()
Continue                  Method        void Continue()
CreateObjRef              Method        System.Runtime.Remoting.ObjRef CreateObjR
Dispose                   Method        void Dispose(), void IDisposable.Dispose(
Equals                    Method        bool Equals(System.Object obj)
ExecuteCommand            Method        void ExecuteCommand(int command)
GetHashCode               Method        int GetHashCode()
GetLifetimeService        Method        System.Object GetLifetimeService()
GetType                   Method        type GetType()
InitializeLifetimeService Method        System.Object InitializeLifetimeService()
Pause                     Method        void Pause()
Refresh                   Method        void Refresh()
Start                     Method        void Start(), void Start(string[] args)
Stop                      Method        void Stop()
WaitForStatus             Method        void WaitForStatus(System.ServiceProcess.
CanPauseAndContinue       Property      bool CanPauseAndContinue {get;}
CanShutdown               Property      bool CanShutdown {get;}
CanStop                   Property      bool CanStop {get;}
Container                 Property      System.ComponentModel.IContainer Containe
DependentServices         Property      System.ServiceProcess.ServiceController[]
DisplayName               Property      string DisplayName {get;set;}
MachineName               Property      string MachineName {get;set;}
ServiceHandle             Property      System.Runtime.InteropServices.SafeHandle
ServiceName               Property      string ServiceName {get;set;}
ServicesDependedOn        Property      System.ServiceProcess.ServiceController[]
ServiceType               Property      System.ServiceProcess.ServiceType Service
Site                      Property      System.ComponentModel.ISite Site {get;set
Status                    Property      System.ServiceProcess.ServiceControllerSt
ToString                  ScriptMethod  System.Object ToString();
```

Functions

Functions are similar to cmdlets and should follow the same naming conventions. Whereas cmdlets are compiled units of PowerShell functionality written in managed code like C#, functions are written using the PowerShell language. Starting with PowerShell 2.0, it has been possible to write advanced functions, which are very similar to cmdlets. It is possible to use common parameters and risk mitigation parameters with advanced functions. An example of a function is shown as follows:

```
function get-PowerShellVersionMessage{
param($name)
    $PowerShellVersion=$PSVersionTable.PSVersion
    return "We're using $PowerShellVersion, $name!"
}
```

Calling the function at the command line is straightforward, as shown in the following screenshot:



```
PS C:\Users\Mike> get-PowerShellVersionMessage Mike
We're using 4.0, Mike!
```

Scripts

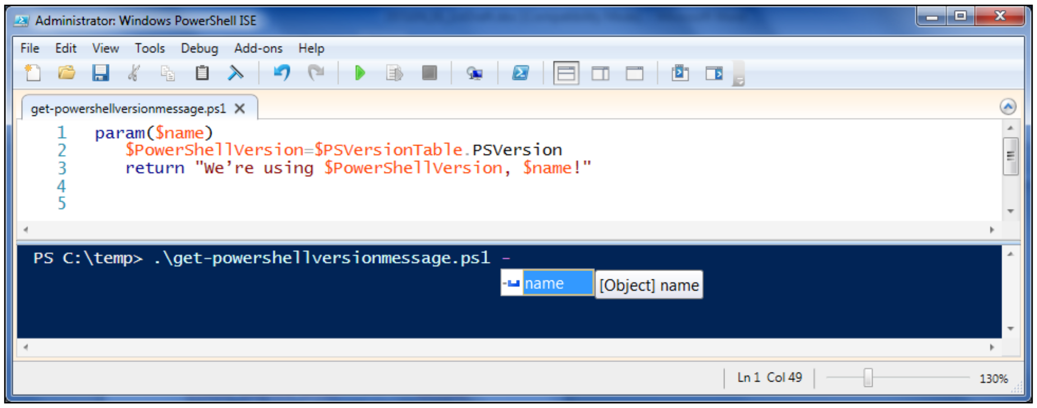
Scripts are simply files with a `.ps1` file extension, which contain PowerShell code. It is possible to parameterize a script file in the same way that you would a function using a `Param()` statement at the beginning of the file. If we were to store the following code in a file called `Get-PowerShellVersionMessage.ps1`, it would be roughly equivalent to the `Get-PowerShellVersionMessage` function in the previous section:

```
param($name)
    $PowerShellVersion=$PSVersionTable.PSVersion
    return "We're using $PowerShellVersion, $name!"
```

A security feature of PowerShell is that it won't run a script in the current directory without specifically referring to the directory, so calling this script would look like this:

```
.\get-powershellversionmessage -name Mike
```

The following screenshot shows the aforementioned code being stored in a file:



Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
get-powershellversionmessage.ps1
Ln 1 Col 49
130%

```
param($name)
   $PowerShellVersion=$PSVersionTable.PSVersion
   return "We're using $PowerShellVersion, $name!"
```

```
PS C:\temp> .\get-powershellversionmessage.ps1 -
                                               name    [Object] name
```

And the output would be (on the computer I'm using): **We're using PowerShell 4.0, Mike**.

> Depending on your environment, you might not be able to run scripts until you change the execution policy. The execution policy dictates whether scripts are allowed to be executed, where those scripts can be located, and whether they need to be digitally signed. Typically, I use `set-executionpolicy RemoteSigned` to allow local scripts without requiring signatures. For more information about execution policies, refer to `about_execution_policies`.

It is also possible to define multiple functions in a script. However, when doing so, it is important to understand the concept of scope. When a script or function is executed, PowerShell creates a new memory area for definitions (for example, variables and functions) that are created during the execution. When the script or function exits, that memory is destroyed, thereby removing the new definitions. Executing a script with multiple functions will not export those functions into the current scope. Instead, the script executes in its own scope and defines the functions in that scope. When the script execution is finished, the newly created scope is exited, removing the function definitions. To overcome this situation, the dot-source operator was created. To dot-source a file means to run the file, without creating a new scope in which to run. If there was a script file with function definitions called `myFuncs.ps1`, dot-sourcing the file would use the following syntax:

```
. .\myFuncs.ps1
```

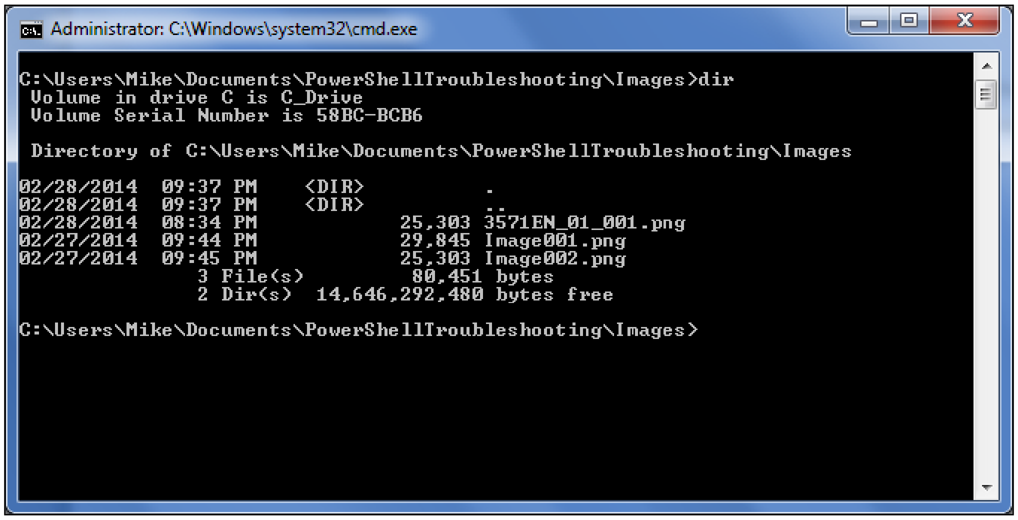
Note that there is a space after the first dot, and that since the script is in the current directory explicit use of the directory is required.

Pipelines

PowerShell expressions involving cmdlets and functions can be connected together using pipelines. Pipelines are not a new concept, and have been in DOS for a long time (and in Unix/Linux forever). The idea of a pipeline is similar to a conveyor belt in a factory. Materials on a conveyor belt move from one station to the next as workers or machinery work on the materials to connect, construct, or somehow modify the work in progress. In the same way, pipelines allow data to move from one command to the next, as the output of one command is treated as the input for the next. There is no practical limit to the number of commands that can be connected this way, but readability does keep command lines from continuing forever. It can be tempting to string more and more expressions together to create a single-line solution, but troubleshooting a pipeline evaluation can be tricky.

When working with long pipeline constructions, consider breaking the line into several expressions to make the execution clearer.

Prior to PowerShell, pipelines dealt with output and input in terms of text, passing strings from one program to the next regardless of what kind of information was being processed. PowerShell makes a major change to this paradigm by treating all input and output as objects. By doing this, PowerShell cmdlets are able to work with the properties, methods, and events that are exposed by the data rather than simply dealing with the string representation. The PowerShell community often refers to the methods used by string-based pipelines as parse-and-pray, which is named after the twin operations of string parsing based on an understanding of the text format and hoping that the format of the output doesn't ever change. An example, shown in the following screenshot, illustrates this quite well:



```
C:\Users\Mike\Documents\PowerShellTroubleshooting\Images>dir
 Volume in drive C is C_Drive
 Volume Serial Number is 58BC-BCB6

 Directory of C:\Users\Mike\Documents\PowerShellTroubleshooting\Images

02/28/2014  09:37 PM    <DIR>          .
02/28/2014  09:37 PM    <DIR>          ..
02/28/2014  08:34 PM            25,303 3571EN_01_001.png
02/27/2014  09:44 PM            29,845 Image001.png
02/27/2014  09:45 PM            25,303 Image002.png
               3 File(s)         80,451 bytes
               2 Dir(s)  14,646,292,480 bytes free

C:\Users\Mike\Documents\PowerShellTroubleshooting\Images>
```

It's easy to think of the output of the MS-DOS `dir` command as a sequence of files and folders, but if the output is carefully studied, something different becomes clear. There is a tremendous amount of other information provided:

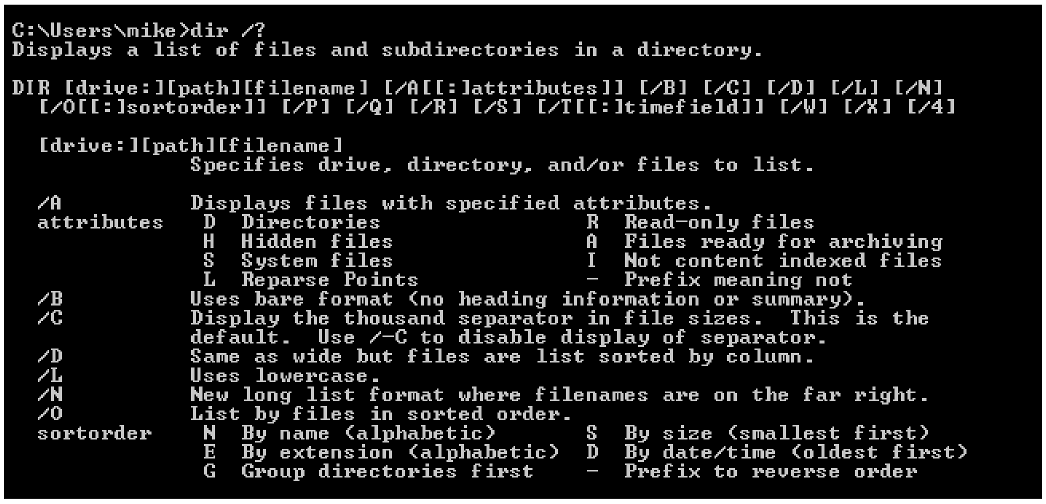
- Volume information
- Volume serial number
- A directory-level caption
- A list of files and folders
- A count of files

- The total size of those files
- The number of directories
- The space free on the drive

To work with this output and deal with, for instance, the file names, there's a tremendous amount of work that would need to be done to analyze the formatting of all of these elements. Also, there are several different formatting parameters that can be used with the MS-DOS `dir` command that would affect the output. By passing data between cmdlets as objects, all of this work is eliminated. The PowerShell `Get-ChildItem` cmdlet, which is similar to the MS-DOS `dir` command, outputs a sequence of file and directory objects if the current location is a filesystem location.

How pipelines change the game

To see how the choice of an object-oriented pipeline changes the way work is done, it is sufficient to look at the MS-DOS `dir` command. I am picking on the `dir` command because it has a simple function and everyone in IT has some level of experience with it. If you wanted to sort the output of a `dir` command, you would need to know what the parameters built into the command are. To do that, you'd do something like this:



```
C:\Users\mike>dir /?
Displays a list of files and subdirectories in a directory.

DIR [drive:][path][filename] [/A[[:]attributes]] [/B] [/C] [/D] [/L] [/N]
  [/O[[:]sortorder]] [/P] [/Q] [/R] [/S] [/T[[:]timefield]] [/W] [/X] [/4]

  [drive:][path][filename]
              Specifies drive, directory, and/or files to list.

  /A          Displays files with specified attributes.
  attributes   D  Directories                R  Read-only files
               H  Hidden files               A  Files ready for archiving
               S  System files               I  Not content indexed files
               L  Reparse Points             -  Prefix meaning not
  /B          Uses bare format (no heading information or summary).
  /C          Display the thousand separator in file sizes.  This is the
              default.  Use /-C to disable display of separator.
  /D          Same as wide but files are list sorted by column.
  /L          Uses lowercase.
  /N          New long list format where filenames are on the far right.
  /O          List by files in sorted order.
  sortorder    N  By name (alphabetic)       S  By size (smallest first)
               E  By extension (alphabetic)  D  By date/time (oldest first)
               G  Group directories first    -  Prefix to reverse order
```

It's clear that the designer of the command had sorting in mind, because there is a `/O` option with five different ways to sort (ten if you include reverse). That is helpful, but files have a lot more than five properties. What if you wanted to sort by more than one property? Ignoring those questions for a moment, does the collection of sorting options for this command help you at all if you were trying to sort the output of a different command (say an `ATTRIB` or `SET` command)? You might hope that the same developer wrote the code for the second command, or that they used the same specifications, but you would be disappointed. Even the simple operation of sorting output is either not implemented or implemented differently by MS-DOS commands.

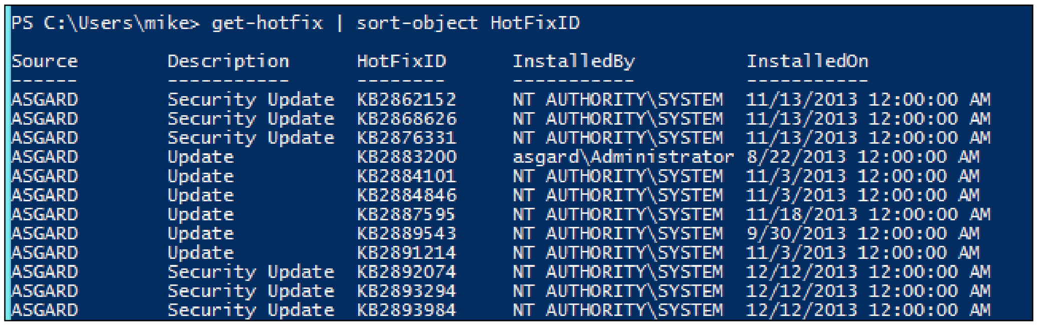
PowerShell takes an entirely different approach. If you were to look at the help for `Get-ChildItem`, you would find no provision for sorting at all. In fact, PowerShell cmdlets do not use parameters to supply sorting information. Instead, they use the object-oriented pipeline. MS-DOS developers needed to encode the sort parameters for the `dir` command inside the `dir` command itself is because that is the only place that the properties exist (including sorting criteria). Once the command has been executed, all that is left is text, and sorting text based on properties is a complex parse-and-pray operation (which we have already discussed). In PowerShell, however, the output of `Get-ChildItem` is a sequence of objects, so cmdlets downstream can still access the properties of the objects directly. Sorting in PowerShell is accomplished with the `Sort-Object` cmdlet, which is able to take a list of properties (among other things) on which to sort the sequence of objects that it receives as input. The following are some examples of sorting a directory listing in MS-DOS and also in PowerShell:

Sorting method	**DOS command**	**PowerShell equivalent**
Sort by filename	`DIR /O:N`	`Get-childitem | sort-object Name`
Sort by extension	`DIR /O:E`	`Get-ChildItem | Sort-object Extension`
Sort by size	`DIR /O:S`	`Get-ChildItem | Sort-object Size`
Sort by write date	`DIR /O:D`	`Get-ChildItem | Sort-object LastWriteTime`
Sort by creation date	`Out of luck`	`Get-ChildItem | Sort-object CreationTime`
Sort by name and size	`Out of luck`	`Get-ChildItem | Sort-object Name,Size`

It can be clearly seen by these examples that:

- PowerShell examples are longer
- PowerShell examples are easier to read (at least the sorting options)
- PowerShell techniques are more flexible

The most important thing about learning how to sort directory entries using `Sort-Object` is that sorting any kind of objects is done the exact same way. For instance, if you retrieved a list of applied hotfixes on the current computer using `Get-hotfix`, in order to sort it by HotFixID, you would issue the `Get-Hotfix | Sort-Object -Property HotFixID` command:



```
PS C:\Users\mike> get-hotfix | sort-object HotFixID

Source        Description      HotFixID      InstalledBy          InstalledOn
------        -----------      --------      -----------          -----------
ASGARD        Security Update  KB2862152     NT AUTHORITY\SYSTEM  11/13/2013 12:00:00 AM
ASGARD        Security Update  KB2868626     NT AUTHORITY\SYSTEM  11/13/2013 12:00:00 AM
ASGARD        Security Update  KB2876331     NT AUTHORITY\SYSTEM  11/13/2013 12:00:00 AM
ASGARD        Update           KB2883200     asgard\Administrator 8/22/2013 12:00:00 AM
ASGARD        Update           KB2884101     NT AUTHORITY\SYSTEM  11/3/2013 12:00:00 AM
ASGARD        Update           KB2884846     NT AUTHORITY\SYSTEM  11/3/2013 12:00:00 AM
ASGARD        Update           KB2887595     NT AUTHORITY\SYSTEM  11/18/2013 12:00:00 AM
ASGARD        Update           KB2889543     NT AUTHORITY\SYSTEM  9/30/2013 12:00:00 AM
ASGARD        Update           KB2891214     NT AUTHORITY\SYSTEM  11/3/2013 12:00:00 AM
ASGARD        Security Update  KB2892074     NT AUTHORITY\SYSTEM  12/12/2013 12:00:00 AM
ASGARD        Security Update  KB2893294     NT AUTHORITY\SYSTEM  12/12/2013 12:00:00 AM
ASGARD        Security Update  KB2893984     NT AUTHORITY\SYSTEM  12/12/2013 12:00:00 AM
```

Another point to note about sorting objects by referring to properties is that the sorting is done according to the type of the property. For instance, sorting objects by a numeric property would order the objects by the magnitude of the property values, not by the string representation of those values. That is, a value of 10 would sort after 9, not between 1 and 2. This is just one more thing that you don't have to worry about.

What's the fuss about sorting?

You might be asking, why is sorting such a big deal? You'd be correct; sorting is not necessarily a tremendously important concept. The point is, the method that the designers of PowerShell took with the pipeline (that is, using objects rather than strings) that allows this sorting method also allows other powerful operations such as filtering, aggregating, summarizing, and narrowing.

Filtering is the operation of selecting which (entire) objects in the pipeline will continue in the pipeline. Think of filtering like a worker who is inspecting objects on the conveyor belt, picking up objects that are bad and throwing them away (in the bit bucket). In the same way, objects that do not match the filter criteria are discarded and do not continue as output. Filtering in PowerShell is done via the `Where-Object` cmdlet and takes two forms. The first form is somewhat complicated to look at, and requires some explaining. We will start with an example such as the following:

```
Get-ChildItem | Where-object {$_.Size –lt 100}
```

Hopefully, even without an explanation, it is clear that the output would be a list of files that have a size less than 100. This form of the `Where-Object` cmdlet takes a piece of code as a parameter (called a scriptblock), which is evaluated for each object in the pipeline. If the script evaluates to true when the object in the pipeline is assigned to the special variable `$_`, the object will continue on the pipeline. If it evaluates to false, the object is discarded.

PowerShell 3.0 made a couple of changes to the `Where-Object` cmdlet. First, it added an easier-to-read option for the `$_` variable, called `$PSItem`. Using that, the previous command can be rewritten as follows:

```
Get-ChildItem | Where-object {$PSItem.Size –lt 100}
```

This is slightly more readable, but Version 3.0 also added a second form that simplifies it even more. If the script block is referring to a single property, a single operator, and a constant value, the simplified syntax can be used, shown as follows:

```
Where-Object Property Operator Value
```

Note that there are no braces indicating a scriptblock, and no `$_ or $PSItem`. The simplified syntax for our sample filter command is this:

```
Get-ChildItem | Where-Object Size –lt 100
```

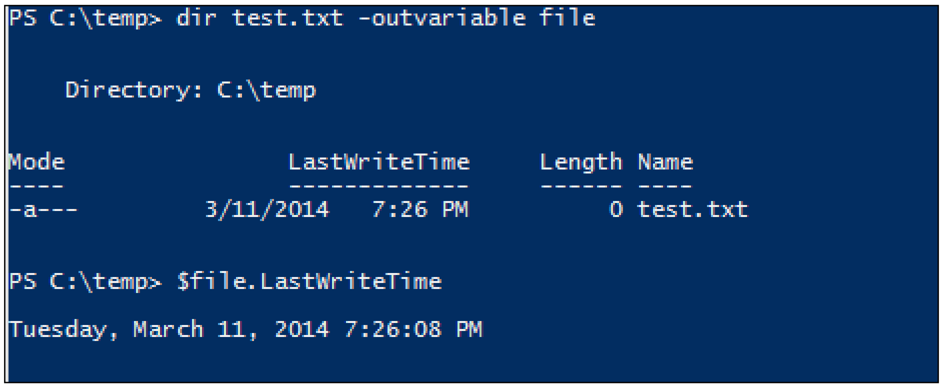
Variables

PowerShell variables, similar to variables in other programming languages, are names for data stored in memory. PowerShell variable references begin with a dollar sign and are created by assigning a value with the assignment operator (the equals sign). Unlike many programming languages, you do not need to define variables before using them or even specify what type of information the variable is going to point to. For instance, the following statements are all valid:

```
$var = 5
$anothervar = "Hello"
$files = dir c:\
```

Note that while the first two assignments were simple (integer and string constants), the third involved executing a pipeline (with a single statement) and storing the results of that pipeline in a variable. The command in the third line returns a collection of more than one kind of object (it has files and folders). Note that there is no special notation required to store a collection of objects.

Several common parameters in PowerShell take the name of a variable in order to store results of some kind in that variable. The `-ErrorVariable`, `-WarningVariable`, `-OutVariable` parameters, and (new in Version 4.0) `-PipelineVariable` parameter all follow this pattern. Also, all of the `*-Variable` cmdlets have a `-Name` parameter. These parameters are expecting the name of the variable rather than the contents of the variable. The name of the variable does not include the dollar sign. In the following screenshot, you can see that the `-outvariable` parameter was passed the `file` value, which caused a copy of the output to be stored in the variable called `file`:



```
PS C:\temp> dir test.txt -outvariable file


    Directory: C:\temp


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         3/11/2014   7:26 PM          0 test.txt


PS C:\temp> $file.LastWriteTime

Tuesday, March 11, 2014 7:26:08 PM
```

In short, referencing the content of the variable involves the dollar sign, but referencing the variable name does not.

Modules

In Version 1.0 of PowerShell, the only ways to group lists of functions were to either put script files for each function in a directory or to include several functions in a script file and use dot-sourcing to load the functions. Neither solution provided much in the way of functionality, though. Version 2.0 introduced the concept of modules. A PowerShell module usually consists of a folder residing in one of the directories listed in the `PSModulePath` environment variable and contains one of the following:

- A module file (`.psm1`) with the same name as the folder
- A module manifest (`.psd1`) with the same name as the folder
- A compiled assembly (`.dll`) with the same name as the folder

One tremendous advantage that modules have over scripts is that while every function in a script is visible when the script is run, visibility of functions (as well as variables and aliases) defined within a module can be controlled by using the `Export-ModuleMember` cmdlet.

The following module file, named `TroubleShooting.psm1`, re-implements the `Get-PowerShellMessage` function from earlier in the chapter using a helper function (`Get-Message`). Since only `Get-PowerShellVersionMessage` was exported, the helper function is not available after the module is imported but it is available to be called by the exported function.

```
function Get-Message{
param($ver,$name)
  return "We're using $ver, $name!"
}

function Get-PowerShellVersionMessage{
param($name)
  $version=$PSVersionTable.PSVersion
  $message=Get-Message $version $name
  return $message
}

Export-ModuleMember Get-PowerShellVersionMessage
```

Importing a module is accomplished by using the `Import-Module` cmdlet. Version 3.0 of PowerShell introduced the concept of automatic importing. With this feature enabled, if you refer to a cmdlet or function that does not exist, the shell looks in all of the modules that exist on the system for a matching name. If it finds one, it imports the module automatically. This even works with tab-completion. If you hit the *Tab* key, PowerShell will look for a cmdlet or function in memory that matches, but If it doesn't find one it will attempt to load the first module that has a function whose name matches the string you're trying to complete. Listing the cmdlets that have been loaded by a particular module is as simple as the `Get-Command –Module` module name.

Further reading

For more information, check out the following references:

- The Monad Manifesto at `http://blogs.msdn.com/b/powershell/archive/2007/03/19/monad-manifesto-the-origin-of-windows-powershell.aspx`

- Microsoft's approved cmdlet verb list at `http://msdn.microsoft.com/en-us/library/ms714428.aspx`
- `get-help get-command`
- `get-help get-verb`
- `get-help about_aliases`
- `get-help get-member`
- `get-help about_functions`
- `get-help about_functions_advanced`
- `get-help about_scripts`
- `get-help about_execution_policies`
- `get-help about_scopes`
- `get-help about_pipelines`
- `get-help where-object`
- `get-help about_variables`
- `get-help about_commonparameters`
- `get-help about_modules`

Summary

In this chapter, we have seen the main building blocks of PowerShell as a language. We have demonstrated how similar functionality can be implemented using a function, a script, and a module. An emphasis was placed on how PowerShell's use of an object-oriented pipeline gives tremendous advantages in terms of flexibility without re-implementing common features such as sorting and filtering in each function.

PowerShell provides an innovative programming and scripting experience. The next chapter will highlight various ways that the PowerShell language functions differently from other programming languages.

2
PowerShell Peculiarities

In many ways, PowerShell is different (as a language) than other traditional programming languages. Some of PowerShell's peculiarities will be presented in this chapter, as well as some guidance on how to avoid common pitfalls. Here are the topics we'll cover in this chapter:

- Strings (quoting, substitution, and escaping)
- Function return values
- Pipeline processing
- Error handling and non-terminating errors

PowerShell strings

In PowerShell, either double quotes or single quotes can be used to express string literals. For instance, the following values are the same:

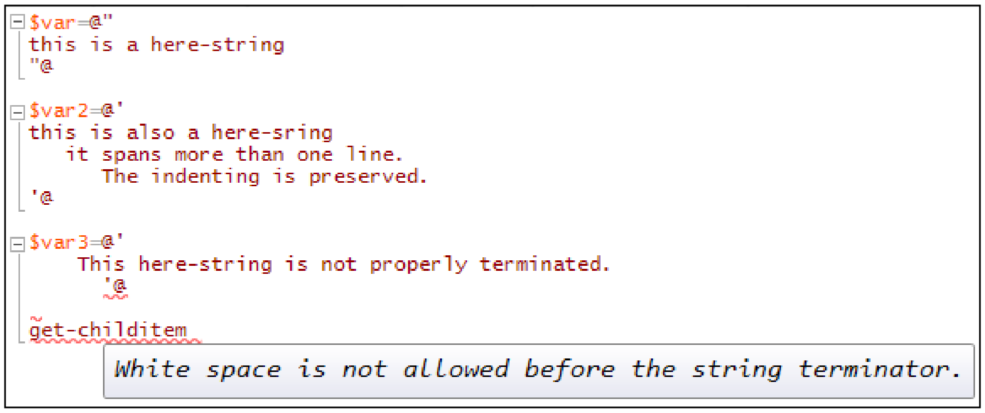
```
"HELLO WORLD"
'HELLO WORLD'
```

Using both kinds of quotes can be useful when quoting strings which themselves contain quotes, such as the following:

```
"I can't stop using PowerShell"
'He said, "I like using PowerShell" all day long'
```

If a single quote is needed in a single-quoted string, you can double the quote (for example, 'can't is a contraction'). The same technique allows the use of double-quotes in a double-quoted string. Strings written this way can be somewhat confusing to look at and it is easy to lose track of the number of quotes. A simpler method of including a double quote character in a string is to escape it with the backtick (`), also called a grave accent. The following string is an example: "the `" character is a double quote".

A peculiar kind of string in PowerShell is called a here-string. Here-strings allow strings to cross several lines and also to contain quotes of either kind without any doubling. Here-strings begin with either `@"` or `@'` at the end of a line and end with `"@` or `'@` respectively at the beginning of a line. A common error is to indent the closing punctuation (so that it is not at the beginning of a line) which causes the here-string to not be terminated. The syntax highlighting in the **integrated scripting environment** (**ISE**) will provide a good visual cue that something isn't quite right in this situation. The following illustration shows a couple of correctly formatted here-strings and one that isn't correctly terminated. Note that the text after the final here-string shows an error and the code hint explains the problem:



```
$var=@"
this is a here-string
"@

$var2=@'
this is also a here-sring
    it spans more than one line.
        The indenting is preserved.
'@

$var3=@'
     This here-string is not properly terminated.
        '@

get-childitem
```

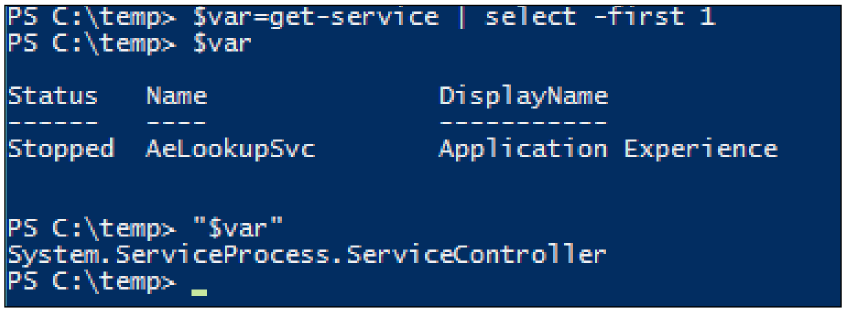
White space is not allowed before the string terminator.

String substitution

The main difference between single- and double-quoted strings (both normal strings and here-strings) is that single-quoted strings are static while double-quoted strings perform string substitution. Variable references contained in double-quoted strings are replaced with string representations of the contents of the variable. For example, if the variable `$name` contains the value `"Mike"`, the double-quoted string `"My name is $name!"` would become `"My name is Mike!"`.

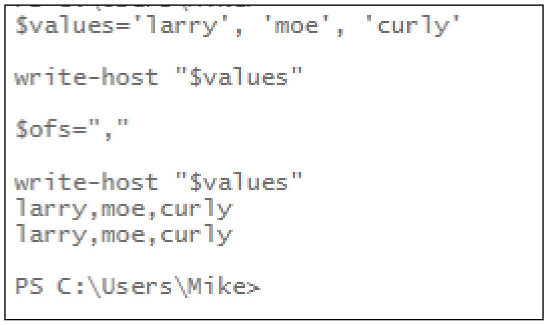
String substitution is a great timesaver. In many languages, embedding values in string output involves string concatenation, and code ends up with lots of expressions such as `"My name is "+$name+"!"`. If the desired output involves several variables, the expression will need to be broken down into more and more segments. However, embedding several variables in PowerShell is often as simple as including the variable names in the string.

For simple objects (such as strings, integers, and floating point numbers), the representation of the variable that is used in the string substitution is the same value that you would see if you used `Write-Host` (for example) to display the value. For complex objects, however, the value is the result of the object's `ToString()` method. This means, generally, the way `$var` is output is different from how `"$var"` is output, as shown in the following screenshot:



```
PS C:\temp> $var=get-service | select -first 1
PS C:\temp> $var

Status   Name               DisplayName
------   ----               -----------
Stopped  AeLookupSvc        Application Experience


PS C:\temp> "$var"
System.ServiceProcess.ServiceController
PS C:\temp> _
```

If a variable is an array, the value that is placed in a string is the value of each of the items in the array separated by the value of the built-in `$ofs` variable (which stands for output field separator). The default value of `$ofs` is a space, but it can be changed to create strings delimited by whatever is desired, as shown in the following screenshot:



```
$values='larry', 'moe', 'curly'

write-host "$values"

$ofs=","

write-host "$values"
larry,moe,curly
larry,moe,curly

PS C:\Users\Mike>
```

How string substitution goes wrong

String substitution is a simple concept, but there are a few common issues that people encounter with it. First, it is critical to realize that string substitution is only performed on double-quoted strings. For example, the string `'My name is $name!'` will not be changed in any way. A second common error is trying to embed something more complicated than a variable value in a string. For instance, if `$file` is a reference to a file, you might be tempted to use `"the file is $file.length bytes long"` and expect to have the length of the file replace `$file.length` in the string. The rule of string substitutions that the engine looks for a variable name and replaces it with a value. In this case, `$file` is the name of a variable and it will be replaced with the name of the file. The remainder of the string will be unchanged, as shown in the following code snippet:

```
$file=dir *.* | select -first 1

"the file is $file.length bytes long"
the file is C:\Users\Mike\.gradle.length bytes long
```

In order to include complicated expressions in a string, one approach is to use the subexpression operator `$()`; for example, `"the file is $($file.length) bytes long"`. Subexpressions in strings are not limited to property references, though. Any expression (including cmdlets) is allowed. The string `"the process started at $(Get-Date)"` is an example of using code in a string.

A second method to include complicated expressions in a string is to use the format operator, `-f`. Using the format operator involves preparing a string with placeholders numbered starting with zero and providing a list of values to be substituted. The previous example involving file lengths could be rewritten using the format operator as follows:

```
"the file is {0} bytes long" -f $file.length
```

Including more than one value is just as easy:

```
"the file {0} is {1} bytes long" -f $file.FullName,$file.length
```

There are several advantages to using the format operator over using string substitution. First, since the placeholders are short, the final string is much shorter in the code listing and will be easier to read on the screen and in a printout. Second, with the format operator, special formatting codes can be applied to the placeholders to format the values in specific ways. For example, formatting a date value in a long date format would use a placeholder such as `{0:D}`, and formatting a floating point value with two decimals would use `{0:N2}`. A reference for formatting codes can be found at `http://msdn.microsoft.com/en-us/library/26etazsy.aspx`.

Escaping in PowerShell strings

A common practice in programming languages is to use the backslash (\) as an escape character to allow special characters to be written in strings. PowerShell's integration as a scripting and command-line language necessitates that the backslash not be given any special meaning other than the traditional meaning as a path separator. Therefore, the backtick is used as the escape character. To include a dollar sign in a string without triggering substitution, you can escape the dollar sign with a backtick like so: `"the value of the variable `$var is $var"`. Notice that the first dollar sign is escaped with a backtick but the second one isn't, so the second `$var` will be replaced. This technique was mentioned earlier to include a literal double quote in a double-quoted string. An important point to remember is that substitution is not performed on single-quoted strings, so including an escaped single quote in a single-quoted string doesn't work.

The following is a table of allowed special characters in double-quoted strings:

Value	**Meaning**
`0	Null
`a	Alert (bell)
`b	Backspace
`f	Form feed
`n	New line
`r	Carriage return
`t	Horizontal tab
`v	Vertical tab

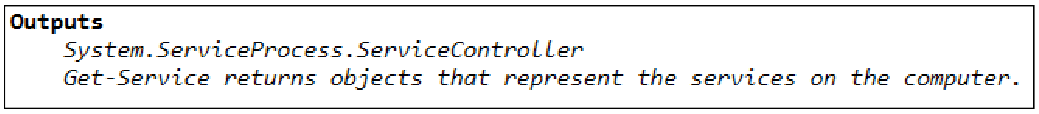
Function output

The example function included in *Chapter 1*, *PowerShell Primer*, used string substitution and returned the single result of that operation. As a reminder, here it is again:

```
function Get-PowerShellVersionMessage{
param($name)
    $PowerShellVersion=$PSVersionTable.PSVersion
    return "We're using $PowerShellVersion, $name!"
}
```

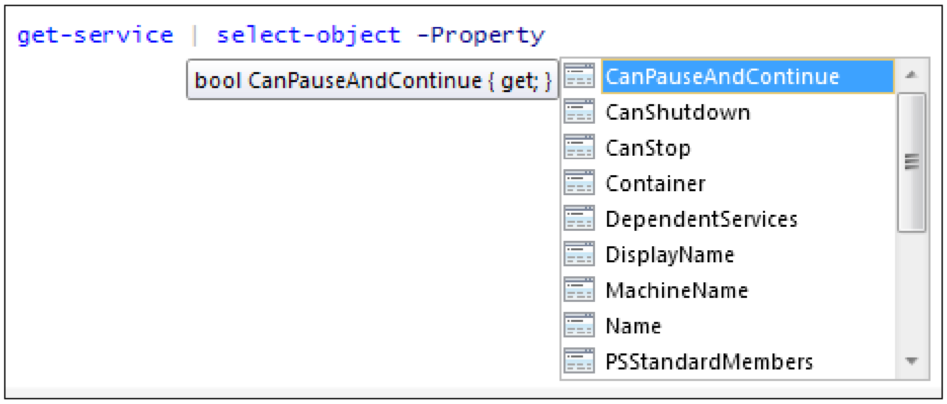
This pattern (that is, performing a calculation and returning the result) is common to procedural programming languages such as C#, Java, and Visual Basic. In PowerShell, however, functions are more complicated than the usage in this scenario in several ways.

In statically-typed languages, the type of a function (that is, the type of a value returned by the function) is either declared as part of the function definition or inferred by the compiler. In PowerShell, the only consideration of a type associated with the output of a function is in the help provided for the function. This output type can be used by a PowerShell host to guide IntelliSense in the environment, but does not place any restrictions on the function in any way. That is, there is no constraint on what types of objects a function outputs. For example, the help for `Get-Service` indicates that it outputs `System.Service.ServiceController` objects, as shown in the following screenshot:



```
Outputs
    System.ServiceProcess.ServiceController
    Get-Service returns objects that represent the services on the computer.
```

Because of this, when you use `Get-Service` in a pipeline, the environment can tell what properties are relevant, as shown in the following screenshot:



```
get-service | select-object -Property
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

As a specific example of the varying output types of a function in PowerShell, consider the `Get-ChildItem` cmdlet. A similar command in the .NET framework would be the `GetFiles` static method of the `System.IO.Directory` class, which returns an array of strings corresponding to the files in a directory. `Get-ChildItem`, on the other hand, generally outputs a list of `FileInfo` objects, but depending on the particular parameters used it might output nothing, a list of `FileInfo` objects, or a single `FileInfo` object. Consider the following code to see an example of this in action:

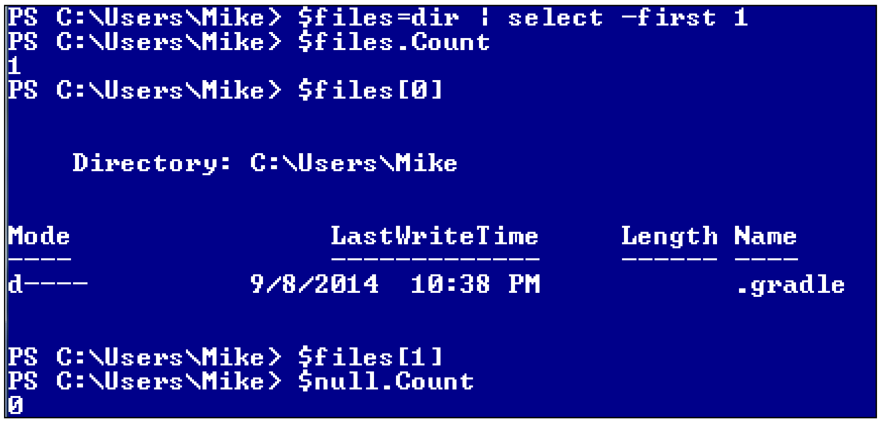
```
$files = Get-ChildItem -path *.exe

foreach ($file in $files){
   # do something interesting
}
```

The code follows a common pattern (that is, get a list of objects and use `foreach` to iterate through the list) and the intent of the code is plain. A problem arises in trying to interpret the `foreach` loop over a value that is not a list. If there are no `.exe` files in the current directory, `$files` will be empty. By this I mean that `$files` won't be an empty list; it will simply be `$null`. What about the situation where there is exactly one `.exe` file? Does it make sense to loop through a single file? Again, this is not a list with one file in it, it is a single `FileInfo` object.

In PowerShell Versions 1.0 and 2.0, the answer was to be more careful when storing lists in variables. By adding a type of `[array]` to `$files`, we have instructed the engine to make sure that what is stored in the variable is indeed an array. In this case, the zero and single object cases result in an empty list and a list containing one object. The loop now makes sense:

```
[array]$files = Get-ChildItem -path *.exe

foreach ($file in $files){
   # do something interesting
}
```

PowerShell Version 3.0 has a different approach to solving this problem. In this version, each non-null, non-collection object is given a `Count` property with a value of 1 and indexer that shows that the first item in the collection (at index 0) is the object itself. The built-in `$null` value has a `Count` property of 0. However, in the case of a null object, the indexer isn't really usable because the index needs to be less than the count (which is zero). These extra properties are added invisibly, that is, they do not show up in the output of `Get-Member`, but they can still be evaluated, as the following illustrates:



```
PS C:\Users\Mike> $files=dir | select -first 1
PS C:\Users\Mike> $files.Count
1
PS C:\Users\Mike> $files[0]


    Directory: C:\Users\Mike


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----          9/8/2014  10:38 PM            .gradle


PS C:\Users\Mike> $files[1]
PS C:\Users\Mike> $null.Count
0
```

Pipeline processing

Another way that functions in PowerShell are different from other languages is how they interact with the pipeline. Functions in PowerShell output objects that can be picked up as input to subsequent functions or cmdlets. This in and of itself is not unique, but the timing of the output is different from other languages.

Consider the process of obtaining a listing of all of the files on a drive. For some drives this operation will be very lengthy, possibly taking several minutes to complete. In PowerShell though, the list of files begin to appear almost instantly. The process can take a while to be complete, but the function (`Get-ChildItem`) will output file and folder objects as each directory is scanned rather than waiting to have all of them collected in a list and returning the list all at once. This feature of built-in cmdlets is something that might easily be taken for granted, but when writing functions, the concept of a return value needs to be carefully considered.

Although PowerShell includes a `return` keyword, the use of `return` is optional and omitting it is even considered by some to be a best practice. PowerShell functions will return a value that is included after a return statement, but that is not the only kind of output in a function.

The rule for function output is simple. Any value produced in a function that is not consumed is added to the output stream at the time the value is produced.

Consuming a value can be accomplished in many ways:

- You can assign the value to a variable
- You can use the value in an expression, as an argument to a cmdlet, function, or script
- You can pipe the value to `Out-Null`
- You can cast the value as `[void]`

With that understanding, consider the following PowerShell functions, which all return a single value:

```
function get-value1{
    return 1
}

function get-value2{
    1
    return
}

function get-value3{
    1
}

function get-value4{
    Write-Output 1
}
```

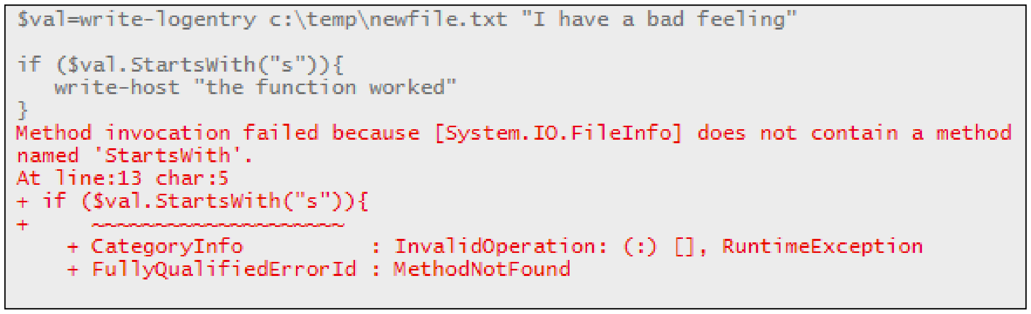
The first function uses the traditional method to output the single value and end the execution of the function. The second includes the value as an expression that is not used and is thus added to the output stream. The return statement in the second function ends the execution of the function, but does not add anything to the output stream. The third function outputs the value just as the second did, but does not include the superfluous return statement. The final version uses the `Write-Output` cmdlet to explicitly write the value to the output stream. The important point to understand is that values can be output from a function in more places than just the return statement. In fact, the return statement is not even needed to output values from a function.

When writing a function, it is extremely important to ensure that the values that are produced in the process of executing are consumed. In most cases, values will be consumed by the natural activities in your function. However, sometimes values are produced as a side effect of activities and make it into the output stream inadvertently. As an example, consider the following code:

```
function Write-Logentry{
param($filename,$entry)
    if(-not (test-path $filename)){
      New-Item -path $filename -itemtype File
    }
    Add-Content -path $filename -value $entry
    Write-Output "successful"
}
```

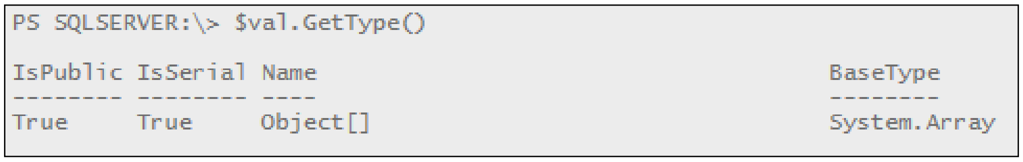
The intent of the code is to create a file if it doesn't exist, and then add text to that file. The problem comes in when the file is created. The `New-Item` cmdlet writes a `FileInfo` object to the output stream as well as creating the file. Since the code doesn't do anything with that value, the `FileInfo` object from the `New-Item` cmdlet is part of the output of the function. It is very common on Stack Overflow to see PowerShell questions that involve this kind of error. Using `New-Item` (or the `mkdir` proxy function for `New-Item`) is often the source of the extraneous object or objects. Other sources include the Add() methods in several .NET classes, which in addition to adding items to a collection, also returns the index of the newly added item.

Pinpointing this kind of error in a function is often confusing because the error message will almost never indicate that the function is the problem. The error message will be downstream from the function where the output is used. In the `Write-Logentry` example function, instead of a value of `"successful"`, the output could be an array of objects containing a `FileInfo` object and the value `"successful"`. Trying to compare the result for a good value might look as follows:



```
$val=write-logentry c:\temp\newfile.txt "I have a bad feeling"

if ($val.StartsWith("s")){
   write-host "the function worked"
}
Method invocation failed because [System.IO.FileInfo] does not contain a method
named 'StartsWith'.
At line:13 char:5
+ if ($val.StartsWith("s")){
+     ~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodNotFound
```

At first look, errors will seem like nonsense. What does PowerShell mean that it can't find a method called `StartsWith()`? Looking at the type of the variable shows that instead of the string that is expected, it contains an array, as shown in the following screenshot:



```
PS SQLSERVER:\> $val.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Similar errors will occur when expecting a numeric result and trying to do calculations on the value. Doing calculations with arrays is probably not going to work, and if it does, it will not work as intended.

It is possible to collect all of the values that are to be output in a variable and wait to output the values until the end of the function, but this is not recommended. One simple reason is that this requires the function writer to keep track of all of the objects and to have memory allocated for the entire collection. Using the output stream naturally, that is, writing to the output stream as objects become available, allows downstream PowerShell cmdlets to work with them while the rest of the objects are being discovered.

The following is a practical example:

```
function find-topProcess{
  param([string[]]$computername)

  $computername |
      foreach{ Get-WmiObject Win32_Process |
               sort-object WorkingSetSize -Descending |
               Select-Object -first 5 PSComputerName,
                                      Name,
                                      ProcessID,
                                      WorkingSetSize
        }
}
```

This function takes a list of computer names and outputs the five processes on each computer that use the highest amount of memory. It could have been written as the following:

```
function find-topProcessBad{
  param([string[]]$computername)

  $processes=$()
  $computername |
       foreach{ $processes+=Get-WMIObject Win32_Process |
                Sort-Object WorkingSetSize -Descending |
                Select-Object -first 5 PSComputerName,
                                       Name,
                                       ProcessID,
                                       WorkingSetSize
        }
   return $processes
}
```

The final output would be the same, in that the same values would be returned in the same order. On the other hand, the way the output is seen by downstream pipeline elements is very different. In the first case, as each computer is scanned, the list of processes is sent to the output stream and then the next computer is considered. There is no local storage in the function at all. In the second case, the processes from each computer are appended to a list. The downstream pipeline elements won't see any output from this function until all of the computers have been scanned. If only a few computer names are being passed in, there is little difference. But if the list is hundreds or thousands of names long, or if the network latency is high enough that it takes a long time to get each set of results, it may be several minutes until any output is delivered. If the function is being called at the command line, it may not be obvious that anything is happening.

Another implication of the second example is that all of the objects need to be stored in a list in memory. This example used a small list of small objects (five) so the effect might not be seen. If the function returned all processes with all of the properties associated with those objects, the memory usage would be quite high. Memory allocation times will also factor into execution time as well.

[To help keep memory usage and execution time lower, try to write objects to the output stream immediately rather than storing them in a collection to be returned all at once.]

PowerShell error handling

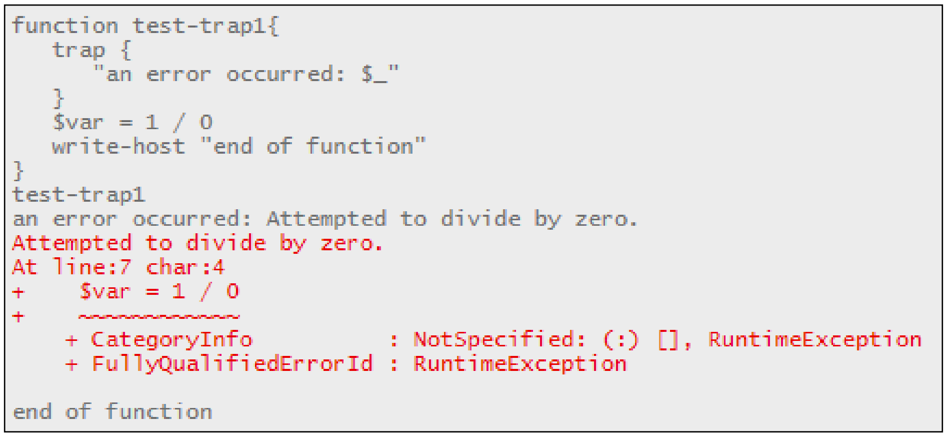
Discussions about error handling in PowerShell revolve around two things: the statements used in error handling, and what constitutes an error that needs to be handled.

The trap statement

PowerShell error handling had a rocky start. The Version 1.0 error handling statement was the trap statement. This statement is similar to the `ON ERROR GOTO` statement in Visual Basic 6. This was a functional way to do error handling, but it was not what most programming languages had been using for the last decade. If a trap statement is included in a scope, when an exception (called a **terminating error** in PowerShell terminology) occurs in that scope, the execution is stopped and the trap statement is executed. By default, trap statements handle any terminating error and can be written to only handle certain types of errors. In the scope of the trap statement, the `$_` special variable contains the error or exception that was caught.

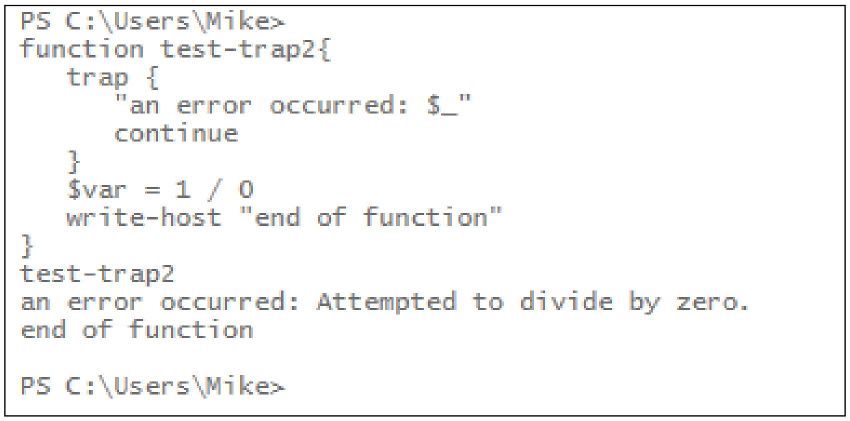
The default execution for a trap statement writes the error to the error stream (separate from the output stream) and continues the function after the statement that had the error. The `break` and `continue` keywords are used in a trap statement to either exit the function or to continue the execution without writing the error to the error stream.

The following screenshot shows an example of a trap statement without using `break` or `continue`. Notice that the error is written to the error stream and the execution of the function is resumed:



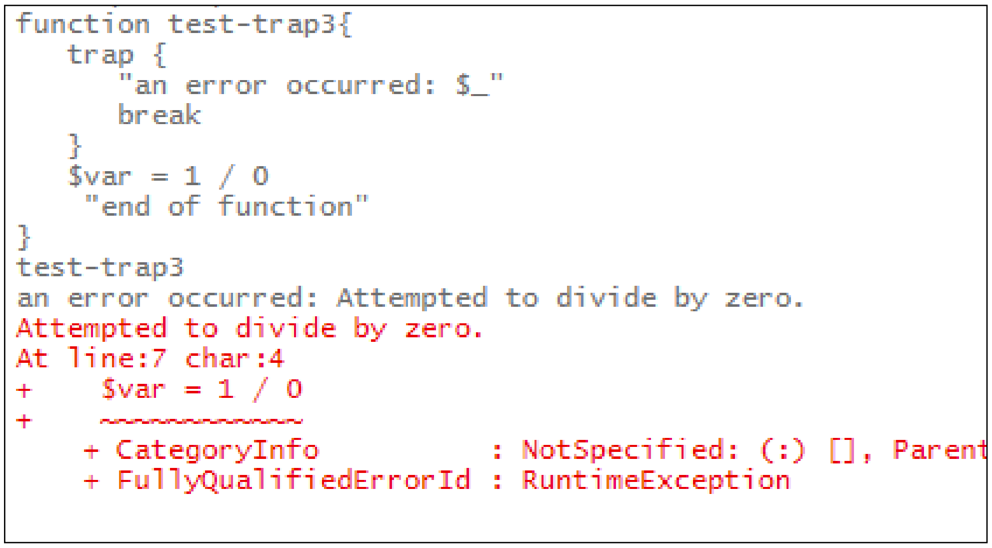
```
function test-trap1{
   trap {
      "an error occurred: $_"
   }
   $var = 1 / 0
   write-host "end of function"
}
test-trap1
an error occurred: Attempted to divide by zero.
Attempted to divide by zero.
At line:7 char:4
+    $var = 1 / 0
+    ~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], RuntimeException
    + FullyQualifiedErrorId : RuntimeException

end of function
```

The following screenshot shows an example of a trap statement using `continue`. Notice that the error is not written to the error stream and the execution of the function is resumed:



```
PS C:\Users\Mike>
function test-trap2{
   trap {
      "an error occurred: $_"
      continue
   }
   $var = 1 / 0
   write-host "end of function"
}
test-trap2
an error occurred: Attempted to divide by zero.
end of function

PS C:\Users\Mike>
```

Finally, an example of a trap statement using `break`. You will notice in the following screenshot that the error is written to the error stream and the execution of the function is not resumed:



```
function test-trap3{
   trap {
      "an error occurred: $_"
      break
   }
   $var = 1 / 0
    "end of function"
}
test-trap3
an error occurred: Attempted to divide by zero.
Attempted to divide by zero.
At line:7 char:4
+    $var = 1 / 0
+    ~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], Parent
    + FullyQualifiedErrorId : RuntimeException
```

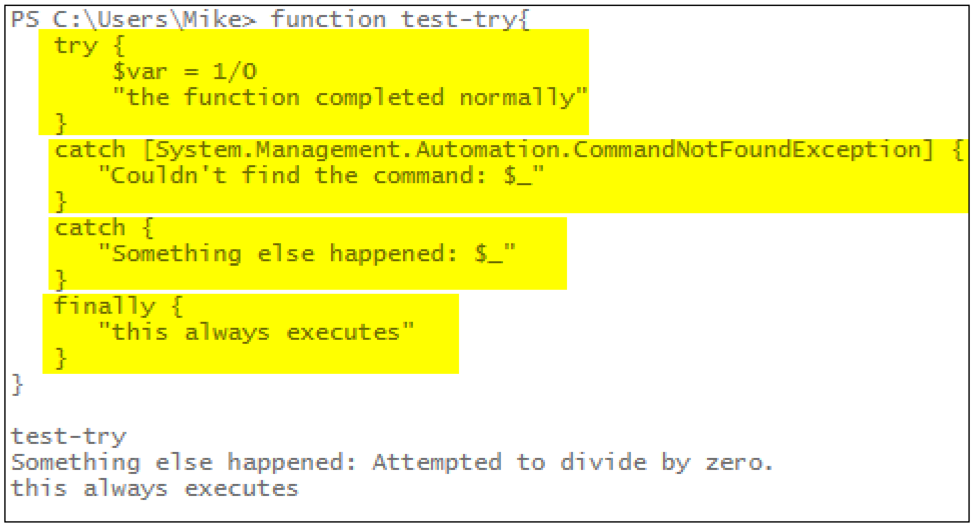
The reality of the trap statement is that it is rarely used except when only PowerShell Version 1.0 is available. The additional error handling features added in Version 2.0 are much more easily understood and less complicated.

> If you find yourself having trouble following the execution path of a trap statement, try to rewrite the function using `try`, `catch`, and `finally`.

try, catch, and finally statements

Version 2.0 of PowerShell introduced traditional `try`, `catch`, and `finally` statements to the language, which work in a similar fashion to structured error handling in languages such as C#, Java, and VB.NET. In a `try`, `catch`, and `finally` construct, code that might throw a terminating error is enclosed in a try block. One or more catch blocks will be present to handle specific classes of errors. An optional `finally` block can contain code that is always executed whether an exception is thrown in the `try` block or not. In this model, there is no concern about the code in the `try` block being resumed or not. Once a terminating error occurs, the execution immediately jumps to the `catch` blocks to be handled. The remaining code in the `try` block is not executed. As is the case in the trap statement, in the `catch` block, the special variable `$_` contains the error that occurred.

The next screenshot shows a simple example using `try`, `catch`, and `finally`. Note that there is a `catch` block for a specific type of error, but in this case the more general catch-all `catch` block was selected because the exception that is thrown does not match the specific exception type named in the first `catch` block.



```
PS C:\Users\Mike> function test-try{
   try {
       $var = 1/0
       "the function completed normally"
   }
   catch [System.Management.Automation.CommandNotFoundException] {
      "Couldn't find the command: $_"
   }
   catch {
      "Something else happened: $_"
   }
   finally {
      "this always executes"
   }
}

test-try
Something else happened: Attempted to divide by zero.
this always executes
```

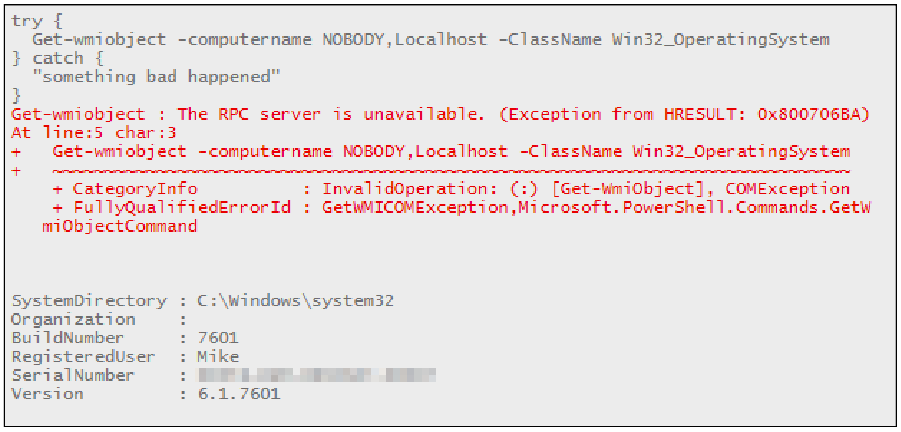
Non-terminating errors

While the `trap` statement might be considered peculiar because languages generally include some sort of structured error handling like `try`, `catch`, and `finally`, PowerShell error handling is peculiar for a completely different reason. In the previous sections, we were careful to use the terminology **terminating error** to describe situations that triggered the error handling capabilities. The reason for this is that PowerShell includes another type of error, not surprisingly called a **non-terminating error**, which is not found in other languages.

To understand non-terminating errors, it is helpful to consider a typical PowerShell script operating in a datacenter that retrieves performance counters from several thousand servers using CIM. With CIM, an administrator can issue a request to all of these servers using a single cmdlet. Anyone who has worked in a datacenter knows that with a large number of servers, there will always be some that are not working quite right for some reason. The reason might be storage-related, network-related, an OS issue, or maybe the servers are simply offline for maintenance. In a typical programming model, attempting to access these servers will result in an error. If the error stops the execution flow, the remainder of the servers will not be able to provide information. What is worse is that the information that has already been retrieved will be lost as well since the statement that is doing the retrieval encountered an error so no assignment can be made to a variable (or anything else, for that matter).

This situation is not ideal. What an administrator would want is to retrieve what information is available and accessible, but also be able to see what errors occurred in order to log them or respond to them in some way. To address this concern, PowerShell introduced non-terminating errors. By default, a non-terminating error is written to the error stream, but does not trigger error handling (`try`, `catch`, `finally`, or `trap`).

An example of a non-terminating error uses the `Get-WMIObject` to access localhost and a non-existent computer, as shown in the following screenshot:



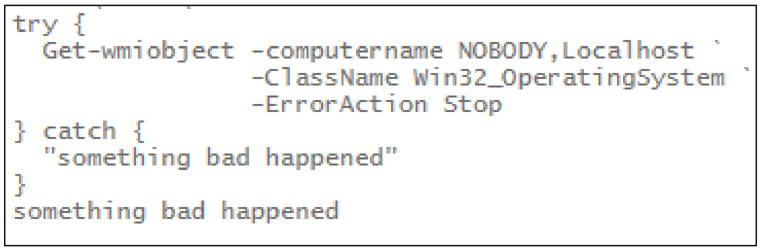
```
try {
  Get-wmiobject -computername NOBODY,Localhost -ClassName Win32_OperatingSystem
} catch {
  "something bad happened"
}
Get-wmiobject : The RPC server is unavailable. (Exception from HRESULT: 0x800706BA)
At line:5 char:3
+   Get-wmiobject -computername NOBODY,Localhost -ClassName Win32_OperatingSystem
+   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [Get-WmiObject], COMException
    + FullyQualifiedErrorId : GetWMICOMException,Microsoft.PowerShell.Commands.GetW
   miObjectCommand


SystemDirectory : C:\Windows\system32
Organization    :
BuildNumber     : 7601
RegisteredUser  : Mike
SerialNumber    : [redacted]
Version         : 6.1.7601
```

Note that an error occurred (stating that RPC wasn't available on the `NOBODY` computer), but that the execution of `Get-WMIObject` continued accessing the localhost computer. The error handling code did not execute because the error in question is a non-terminating error.

This is often the desired outcome as explained previously. On the other hand, what if it is appropriate to execute the error handling code if a non-terminating error occurs? Cmdlets provide a parameter called `-ErrorAction`, which can be used to override the default behavior. The default value for `-ErrorAction` is `Continue`, which means to write the error to the error stream, add it to the `$Error` collection, and continue executing the current statement. To force the cmdlet to treat non-terminating errors as exceptions (terminating errors), use the value `Stop`. The possible values for the `-ErrorAction` parameter are listed in the following table:

Value	**Meaning**
`Stop`	Treats any errors as terminating errors (that is, throws an exception).
`Continue`	Writes the error to the error stream, adds it to the `$Error` collection, and continues the execution.
`SilentlyContinue`	Adds the error to the `$Error` collection and continues execution. Does not write the error to the error stream.
`Inquire`	Asks the user whether to continue or not.
`Ignore`	Continues execution without writing to the error stream or recording the error.
`Suspend`	Opens a nested prompt for interactive debugging (refer to *Chapter 7, Reactive Practices – Traditional Debugging*).

The following screenshot illustrates the use of a non-terminating error as an exception:


```
try {
  Get-wmiobject -computername NOBODY,Localhost `
                -ClassName Win32_OperatingSystem `
                -ErrorAction Stop
} catch {
  "something bad happened"
}
something bad happened
```

In this case, since the `-ErrorAction` was set to `Stop`, the execution of the cmdlet was terminated as soon as an error (in this case, a non-terminating error) occurred. The catch statement was triggered, the error was not written to the output stream, but it was written to the `$Error` collection.

Further reading

You can go through the following references for more information on this topic:

- `get-help about_quoting_rules`
- `get-help about_PowerShell_Ise.exe`
- `get-help about_preference_variables`
- `get-help about_operators`
- `get-help about_escape_characters`
- `get-help about_return`
- `get-help about_trap`
- `get-help about_try_catch_finally`
- `get-help about_throw`
- Learn about formatting codes at `http://msdn.microsoft.com/en-us/library/26etazsy.aspx`

Summary

In this chapter, we looked at several aspects of the PowerShell language that are implemented in ways different from other popular programming languages such as C#, Java, and VB.NET. On the topic of strings, types of quotes, string substitution, and escaping special characters were covered. The discussion of PowerShell functions focused on the types and number of objects returned and how functions in PowerShell write objects to an output stream throughout the execution of the function rather than returning all of the values at the end of execution. The final topic focused on error handling methods in PowerShell, including the trap statement from PowerShell Version 1.0, the more advanced `try`, `catch`, and `finally` statements included from Version 2.0, and the difference between terminating errors and non-terminating errors.

The focus of the next chapter will be on practices that will help keep PowerShell code performing well and easy to debug.

3
PowerShell Practices

In this chapter, we will discuss a few practices that will help PowerShell scripts run faster and produce the results that are expected. We will also cover optional output to the user and documentation in terms of built-in help along with the following topics:

- Filter left
- Format right
- Comment-based help
- Output using `Write-*` cmdlets

Filter left

As discussed in *Chapter 1*, *PowerShell Primer*, pipelines are a central feature of PowerShell. Cmdlets can sometimes create a tremendous amount of data though, and pushing that data through a pipeline does have performance implications in terms of memory and processor usage. Filter left is the principle that objects should be filtered as early as possible in the pipeline. Since pipelines flow from left to right, the filter should be as far to the left as it can be.

For example, the following pipelines have the same results and to show the SQL Server datafiles (`*.mdf`) in order of size:

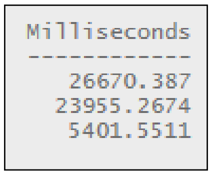
```
dir c:\ -recurse | sort-object -Property Size -descending | where-object Extension -eq '.mdf'

dir c:\ -recurse | where-object Extension -eq '.mdf' | sort-object -Property Size -descending

dir c:\ -recurse -include *.mdf | sort-object -Property Size -descending
```

Even though the results are the same, the execution is about as different as possible. The first pipeline collects all of the files on the disk into a collection, sorts that list, and then selects the `.mdf` files. The second passes all of the files on the disk again, but filters them before sorting. The third example only creates objects for the `.mdf` files and only sorts those objects.

Using the `Measure-Command` cmdlet to get the time each of these takes to execute reveals the very different performance. Note that I have formatted the statements to make them easier to fit on the page. The measurement code is as follows:

```
Measure-Command {dir d:\ -recurse |
                Sort-Object -Property Size -descending |
                Where-Object Extension -eq '.mdf'} |
                Select TotalMilliseconds

Measure-Command {dir d:\ -recurse |
                Where-Object Extension -eq '.mdf' |
                Sort-Object -Property Size -descending } |
                Select TotalMilliseconds

measure-command {dir d:\ -recurse -filter *.mdf |
                Sort-Object -Property Size -descending } |
                Select TotalMilliseconds
```

And the following screenshot shows the results:



```
Milliseconds
------------
   26670.387
  23955.2674
   5401.5511
```

Putting the filter on the extreme left caused the code to take about 80 percent less time in this instance. Filtering before sorting provided a slight benefit, but not nearly this striking.

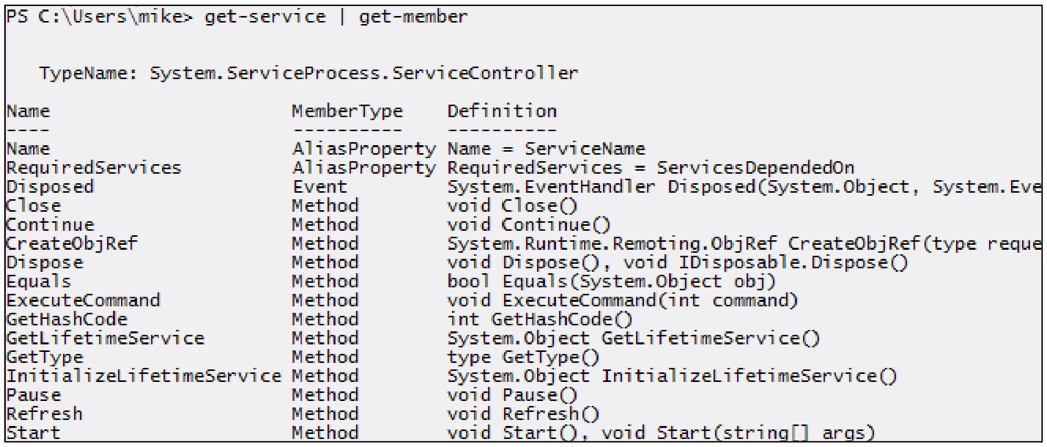
This seems like a simple example, but the principle is important. In production environments, there may be dozens or even hundreds of scripts running at the same time on a server, so the performance of each script is something that needs to be considered. When remoting is added into the mix the performance is magnified as the objects not only require memory but also network bandwidth. The difference between retrieving a list of all files on a server and pulling the single file that is needed is tremendous.

Filter left!

Make sure you keep the filtering parameters and cmdlets as far to the left in the pipeline as possible.

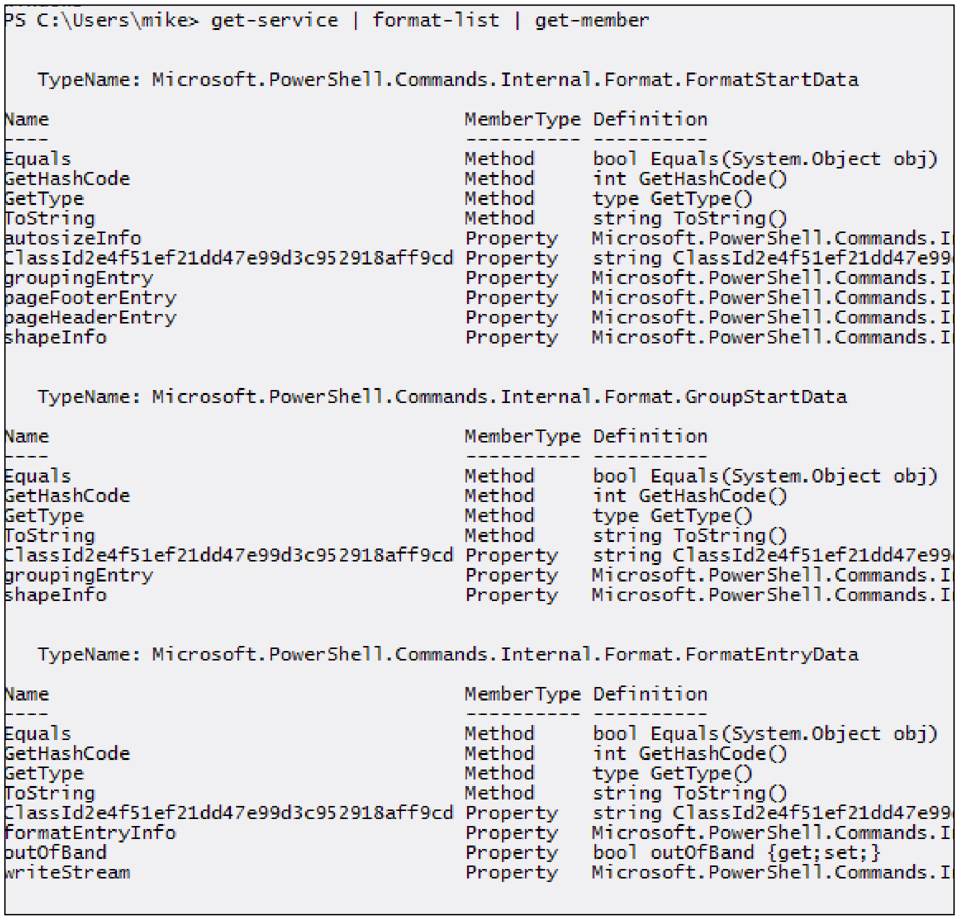
Format right

The next important practice in PowerShell is to format right. This means that any format cmdlets included in the pipeline should be at the far right of the pipeline. While filter left is primarily about efficiency, format right concerns the kinds of objects produced. To see this, have a look at the following output:



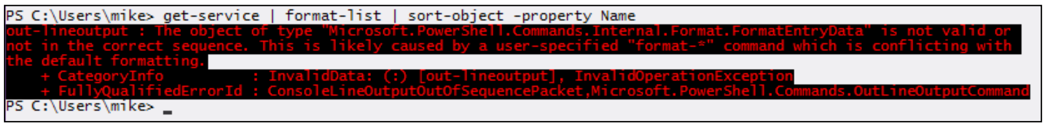
```
PS C:\Users\mike> get-service | get-member


   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
RequiredServices          AliasProperty RequiredServices = ServicesDependedOn
Disposed                  Event         System.EventHandler Disposed(System.Object, System.Eve
Close                     Method        void Close()
Continue                  Method        void Continue()
CreateObjRef              Method        System.Runtime.Remoting.ObjRef CreateObjRef(type reque
Dispose                   Method        void Dispose(), void IDisposable.Dispose()
Equals                    Method        bool Equals(System.Object obj)
ExecuteCommand            Method        void ExecuteCommand(int command)
GetHashCode               Method        int GetHashCode()
GetLifetimeService        Method        System.Object GetLifetimeService()
GetType                   Method        type GetType()
InitializeLifetimeService Method        System.Object InitializeLifetimeService()
Pause                     Method        void Pause()
Refresh                   Method        void Refresh()
Start                     Method        void Start(), void Start(string[] args)
```

First, you can see that the `Get-Service` cmdlet outputs the `ServiceController` objects. Piping those `ServiceController` objects to `Format-List`, though, produces a series of objects with indecipherable properties, as shown in the following screenshot:



```
PS C:\Users\mike> get-service | format-list | get-member


   TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatStartData

Name                                    MemberType Definition
----                                    ---------- ----------
Equals                                  Method     bool Equals(System.Object obj)
GetHashCode                             Method     int GetHashCode()
GetType                                 Method     type GetType()
ToString                                Method     string ToString()
autosizeInfo                            Property   Microsoft.PowerShell.Commands.I
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef21dd47e99
groupingEntry                           Property   Microsoft.PowerShell.Commands.I
pageFooterEntry                         Property   Microsoft.PowerShell.Commands.I
pageHeaderEntry                         Property   Microsoft.PowerShell.Commands.I
shapeInfo                               Property   Microsoft.PowerShell.Commands.I


   TypeName: Microsoft.PowerShell.Commands.Internal.Format.GroupStartData

Name                                    MemberType Definition
----                                    ---------- ----------
Equals                                  Method     bool Equals(System.Object obj)
GetHashCode                             Method     int GetHashCode()
GetType                                 Method     type GetType()
ToString                                Method     string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef21dd47e99
groupingEntry                           Property   Microsoft.PowerShell.Commands.I
shapeInfo                               Property   Microsoft.PowerShell.Commands.I


   TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatEntryData

Name                                    MemberType Definition
----                                    ---------- ----------
Equals                                  Method     bool Equals(System.Object obj)
GetHashCode                             Method     int GetHashCode()
GetType                                 Method     type GetType()
ToString                                Method     string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef21dd47e99
formatEntryInfo                         Property   Microsoft.PowerShell.Commands.I
outOfBand                               Property   bool outOfBand {get;set;}
writeStream                             Property   Microsoft.PowerShell.Commands.I
```

Once a formatting cmdlet has been executed, the only objects on the pipeline are PowerShell formatting objects which are only useful by the PowerShell host. As a general rule, the only cmdlets that can follow a formatting cmdlet in the pipeline are the output cmdlets that start with `Out-`. These output cmdlets are designed to interpret these formatting objects and render formatted output accordingly. Since pipelines always end in `Out-Default`, there's little danger of ever encountering one of these formatting objects unless something is included between the formatting cmdlet and the end of the pipeline, which is shown in the following screenshot:



```
PS C:\Users\mike> get-service | format-list | sort-object -property Name
out-lineoutput : The object of type "Microsoft.PowerShell.Commands.Internal.Format.FormatEntryData" is not valid or
not in the correct sequence. This is likely caused by a user-specified "format-*" command which is conflicting with
the default formatting.
    + CategoryInfo          : InvalidData: (:) [out-lineoutput], InvalidOperationException
    + FullyQualifiedErrorId : ConsoleLineOutputOutOfSequencePacket,Microsoft.PowerShell.Commands.OutLineOutputCommand
PS C:\Users\mike> _
```

Trying to sort the objects after formatting them is a common mistake. It is clear from this example that the `ServiceController` is no longer present, and that attempting to use the properties of the original objects are going to fail.

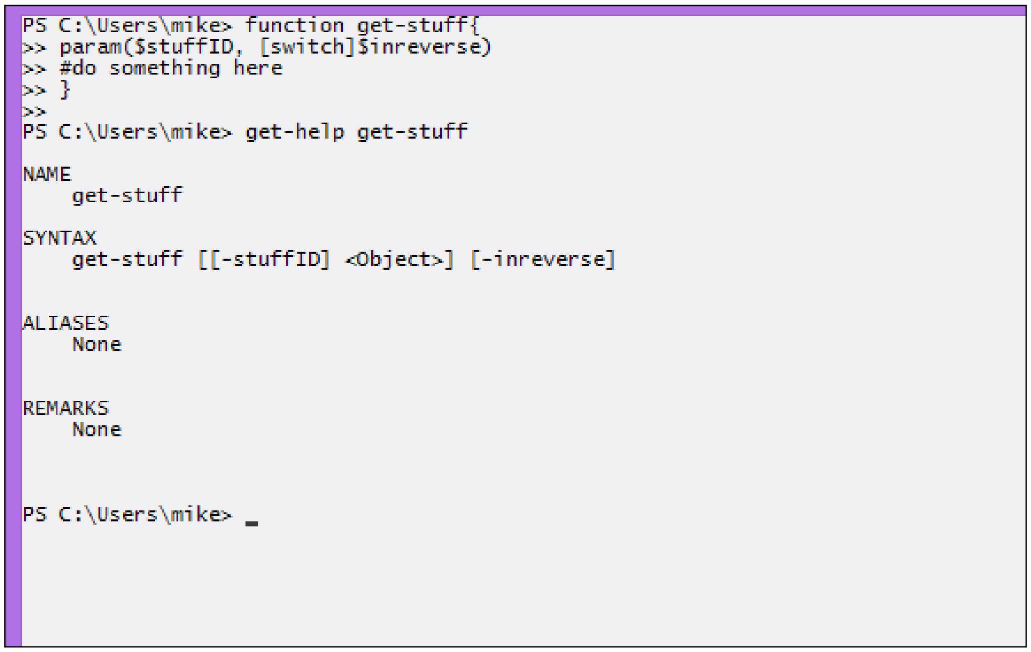
Because of the destructive nature of formatting cmdlets, the best practice is to not use them in functions or scripts unless the output of the script or function in question never needs to be manipulated programmatically. In other words, use formatting only if the output is intended for people to use rather than as input to other scripts or functions.

Avoid using formatting cmdlets in functions or scripts that may provide input for other functions or scripts. To show that the code uses formatting, consider naming the function or script with the verb `Show` (for example, `Show-ServiceInfo`).

Comment-based help

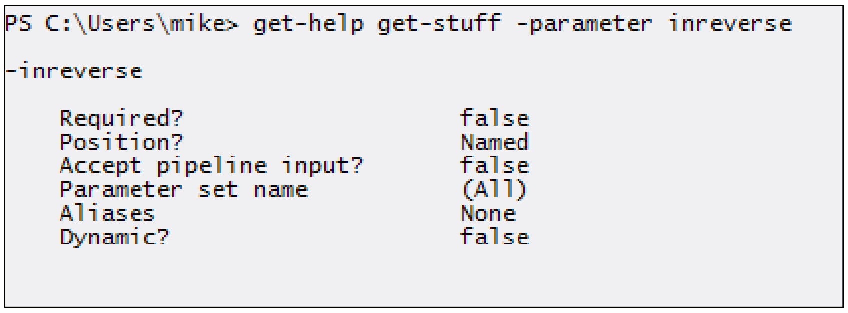
As mentioned in *Chapter 1*, *PowerShell Primer*, the help system in PowerShell is very useful. When writing functions and scripts in PowerShell, it is important to provide the same kind of documentation so users are able to use the code correctly.

In order to get the minimal amount of help available, the only requirement is to create a function. Take a look at the following screenshot, for example, shows the help content generated automatically for a simple function:



```
PS C:\Users\mike> function get-stuff{
>> param($stuffID, [switch]$inreverse)
>> #do something here
>> }
>>
PS C:\Users\mike> get-help get-stuff

NAME
    get-stuff

SYNTAX
    get-stuff [[-stuffID] <Object>] [-inreverse]


ALIASES
    None


REMARKS
    None



PS C:\Users\mike> _
```

There is even help for specific parameters, as shown in the following screenshot:



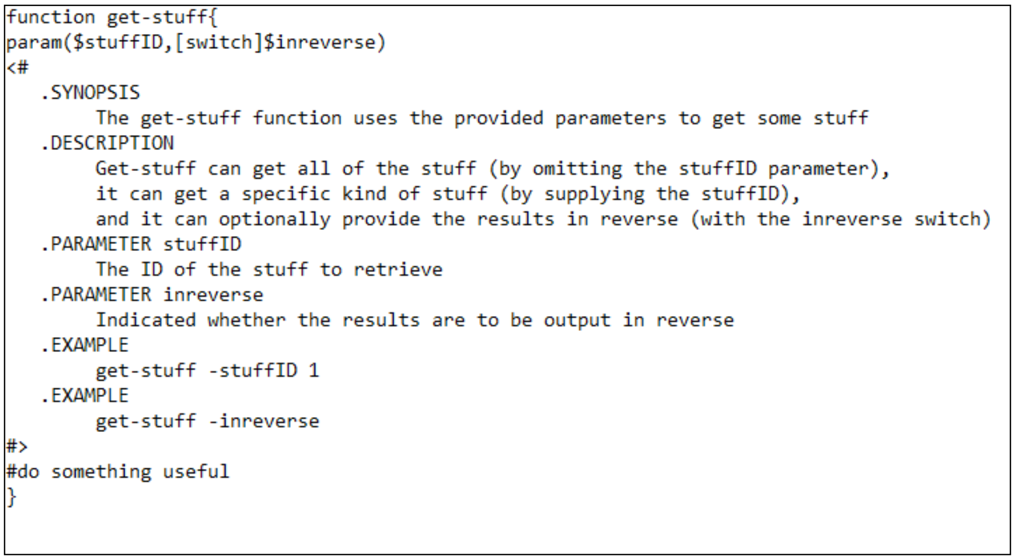
```
PS C:\Users\mike> get-help get-stuff -parameter inreverse

-inreverse

    Required?                    false
    Position?                    Named
    Accept pipeline input?       false
    Parameter set name           (All)
    Aliases                      None
    Dynamic?                     false
```

It should be clear that PowerShell is using the definition of the function to generate the help content. This is extremely helpful for several reasons, some of which are listed as follows:

- The syntax section is never out of sync with the definition. There is no need to worry about missing or misspelled parameters or whether the brackets are in the right places.
- It is not necessary to create a separate file (or any file at all) in order to have help for a function.
- Help for a function will be consistently displayed.

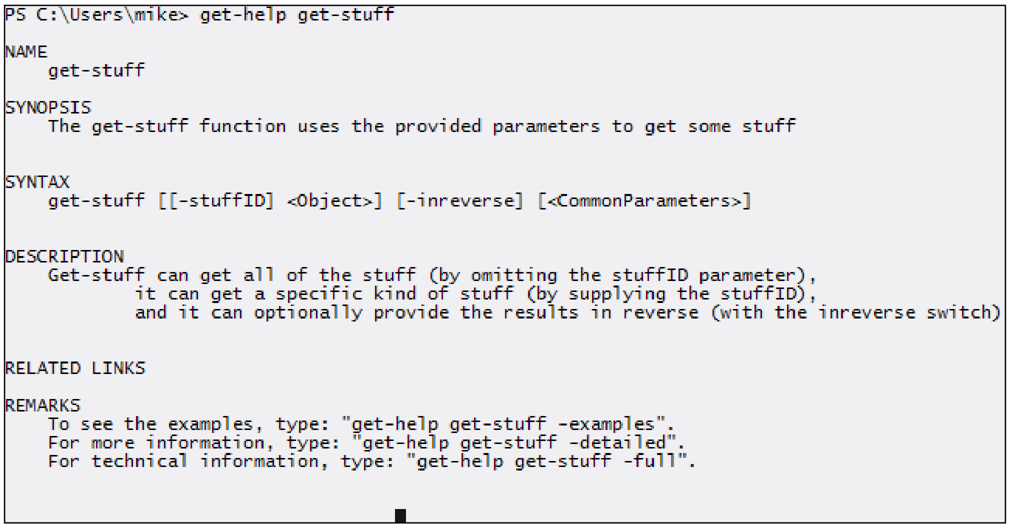
The help seen in the earlier examples is not nearly as complete as the help seen for cmdlets, such as `Get-ChildItem` or `Get-WMIObject`, so how does one provide the missing pieces of information? The answer is comment-based help.

Using comment-based help is a very simple process. The content for the help system is usually embedded in a function or immediately before the function in a comment or series of comments. The following screenshot illustrates this well:



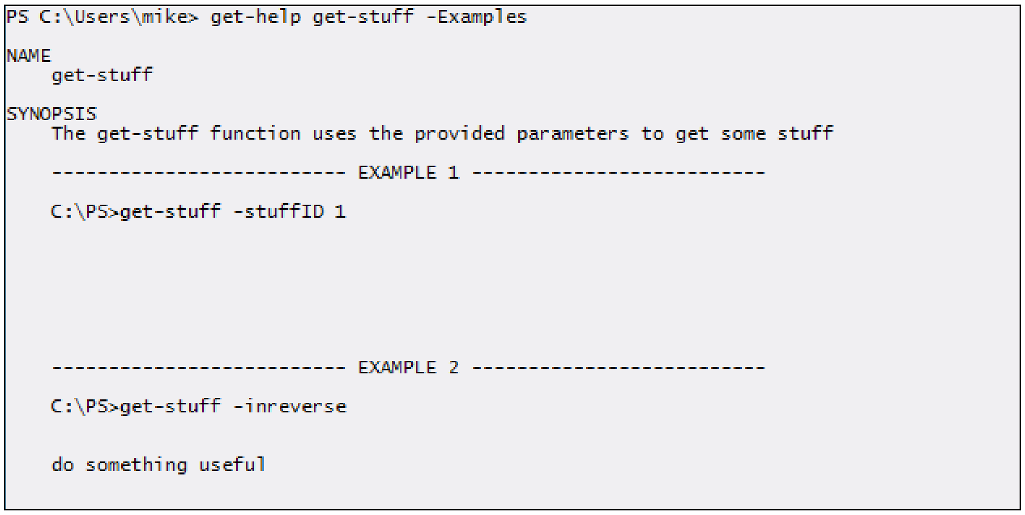
```
function get-stuff{
param($stuffID,[switch]$inreverse)
<#
   .SYNOPSIS
        The get-stuff function uses the provided parameters to get some stuff
   .DESCRIPTION
        Get-stuff can get all of the stuff (by omitting the stuffID parameter),
        it can get a specific kind of stuff (by supplying the stuffID),
        and it can optionally provide the results in reverse (with the inreverse switch)
   .PARAMETER stuffID
        The ID of the stuff to retrieve
   .PARAMETER inreverse
        Indicated whether the results are to be output in reverse
   .EXAMPLE
        get-stuff -stuffID 1
   .EXAMPLE
        get-stuff -inreverse
#>
#do something useful
}
```

Given this function definition with embedded comment-based help, `get-help` now returns a much more complete topic:



```
PS C:\Users\mike> get-help get-stuff

NAME
    get-stuff

SYNOPSIS
    The get-stuff function uses the provided parameters to get some stuff


SYNTAX
    get-stuff [[-stuffID] <Object>] [-inreverse] [<CommonParameters>]


DESCRIPTION
    Get-stuff can get all of the stuff (by omitting the stuffID parameter),
            it can get a specific kind of stuff (by supplying the stuffID),
            and it can optionally provide the results in reverse (with the inreverse switch)


RELATED LINKS

REMARKS
    To see the examples, type: "get-help get-stuff -examples".
    For more information, type: "get-help get-stuff -detailed".
    For technical information, type: "get-help get-stuff -full".
```

As stated in the comments at the end of this topic, the examples are visible by using `get-help` with the `-Examples` switch. The following screenshot illustrates this:



```
PS C:\Users\mike> get-help get-stuff -Examples

NAME
    get-stuff

SYNOPSIS
    The get-stuff function uses the provided parameters to get some stuff

    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>get-stuff -stuffID 1







    -------------------------- EXAMPLE 2 --------------------------

    C:\PS>get-stuff -inreverse


    do something useful
```

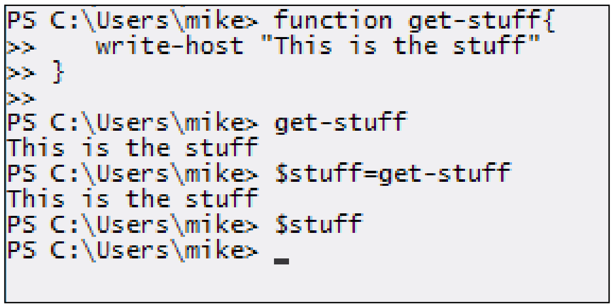
There is some flexibility in the precise placement of the comment as well as a tremendous variety of information that can be included. For complete details, refer to the `about_Comment_Based_Help` topic.

Using Write-* cmdlets

When writing PowerShell functions, it is often confusing to know exactly which cmdlet should be used to produce output. There are a number of cmdlets and methods that seem to do the same thing at first glance. The most important thing to remember in this context is that functions should always write objects to the output stream. In order to do this, there are several correct ways and one incorrect way that is frequently used.

Write-Host

PowerShell scripters who are beginners see the `Write-Host` cmdlet as a simple way to produce output. It is similar to a `PRINT` statement in many languages and if output to the console is the goal, it is a perfect fit. Unfortunately, the output is made solely to the console (or, in PowerShell terminology, the **host**). The following screenshot shows the main issue with `Write-Host`:



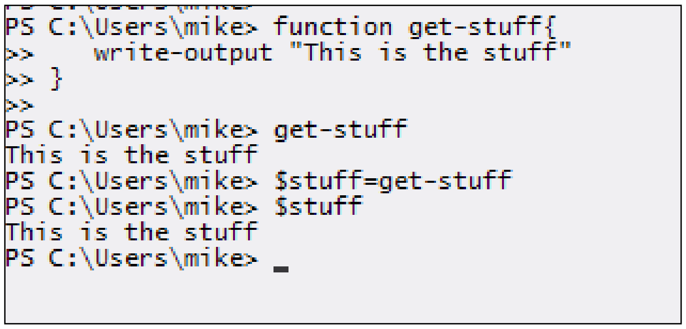
```
PS C:\Users\mike> function get-stuff{
>>    write-host "This is the stuff"
>> }
>>
PS C:\Users\mike> get-stuff
This is the stuff
PS C:\Users\mike> $stuff=get-stuff
This is the stuff
PS C:\Users\mike> $stuff
PS C:\Users\mike> _
```

Even though the function seems to output the string, attempting to capture the output in a variable shows that the value isn't actually output, but instead is simply text written to the host. Because of this, `Write-Host` is not a good fit for functions. Scenarios where `Write-Host` makes sense to use in a function are similar to those where formatting cmdlets are used (as discussed earlier in this chapter).

> `Write-Host` should only be used where output is to be read by a user and not ever by another function. This is one of the most universally accepted PowerShell best practices.

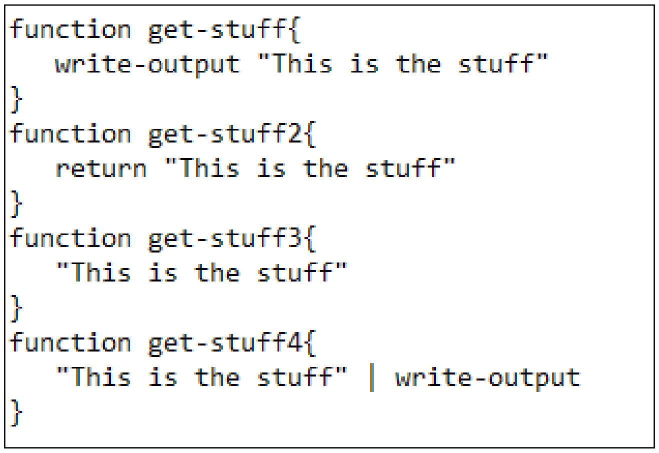
Output – the correct way

A cmdlet that at first glance looks similar to `Write-Host` is `Write-Output`. Rewriting the `get-stuff` function from the previous section using `Write-Output` produces the expected results, as shown in the following screenshot:



```
PS C:\Users\mike> function get-stuff{
>>    write-output "This is the stuff"
>> }
>>
PS C:\Users\mike> get-stuff
This is the stuff
PS C:\Users\mike> $stuff=get-stuff
PS C:\Users\mike> $stuff
This is the stuff
PS C:\Users\mike> _
```

Even though the output looks the same in the console, capturing the output in a variable is successful this time. The reason is that `Write-Output` writes the values of its parameters to the output stream rather than the host. Think back to the discussion about how a function returns values in *Chapter 1*, *PowerShell Primer*, and you might recall that there are several ways to return values from a function. All of these variations produce equivalent results:



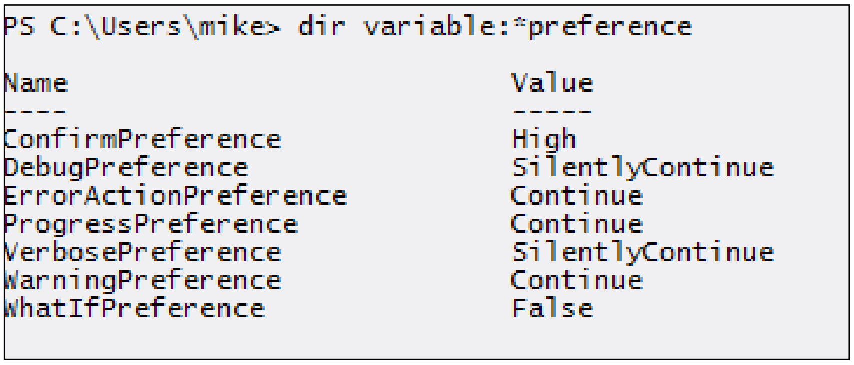
```
function get-stuff{
   write-output "This is the stuff"
}
function get-stuff2{
   return "This is the stuff"
}
function get-stuff3{
   "This is the stuff"
}
function get-stuff4{
   "This is the stuff" | write-output
}
```

What about the other Write-* cmdlets?

`Write-Host` and `Write-Output` are only two of several `Write-*` cmdlets. The other core cmdlets are listed as follows:

- `Write-Verbose`
- `Write-Warning`
- `Write-Debug`
- `Write-Progress`
- `Write-Error`

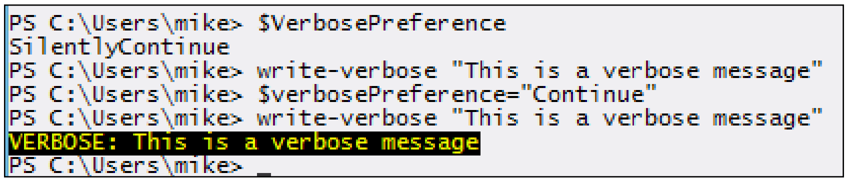
The first four cmdlets (`Write-Verbose`, `Write-Warning`, `Write-Debug`, and `Write-Progress`) are intended to be used to provide optional information about code execution. The output from these cmdlets is controlled by a set of preference variables. Preference variables are predefined by the host and used to indicate what action should be taken when one of these cmdlets is executed. The following screenshot shows the list preference variables with the PSDrive mechanism . The full list of predefined preference variables can be seen along with their values by listing matching names from the variable:



```
PS C:\Users\mike> dir variable:*preference

Name                           Value
----                           -----
ConfirmPreference              High
DebugPreference                SilentlyContinue
ErrorActionPreference          Continue
ProgressPreference             Continue
VerbosePreference              SilentlyContinue
WarningPreference              Continue
WhatIfPreference               False
```

PSDrives and **PSProviders** are mechanisms PowerShell uses to expose hierarchical storage. The `FileSystem` provider implements drives that match local and mapped drives. Other providers, such as the `Variable` and `Function` providers, give some visibility to PowerShell objects. In this example, we used the `dir` alias of `Get-ChildItem` to view a list of variables using the `Variable:` drive in the same way we would have looked at files in a folder.

We came across the `$errorActionPreference` variable in *Chapter 2, PowerShell Peculiarities*, in the context of error handling with non-terminating errors. The following shows an example of using `$VerbosePreference` to control the output from `Write-Verbose`:



```
PS C:\Users\mike> $VerbosePreference
SilentlyContinue
PS C:\Users\mike> write-verbose "This is a verbose message"
PS C:\Users\mike> $verbosePreference="Continue"
PS C:\Users\mike> write-verbose "This is a verbose message"
VERBOSE: This is a verbose message
PS C:\Users\mike> _
```

Since the value of `$VerbosePreference` was `SilentlyContinue`, no output was produced by the first invocation of the cmdlet. Changing the value to `Continue` allowed output from the second call. Note that the output is written to the host differently than output that comes from `Write-Host` or `Write-Output`. That is because `Write-Verbose` writes to its own stream. `Write-Debug`, `Write-Warning`, `Write-Error`, and `Write-Verbose` each have their own output streams, which are distinct from the standard object output stream.

Possible values for preference variables include: `Stop`, `Continue`, `SilentlyContinue`, `Ignore`, `Inquire`, and `Suspend`. Not all values are valid for all variables. The acceptable values are given in the following table:

Variable	**Potential values (default values are shown with quotes)**
`ErrorActionPreference`	`Stop`, `"Continue"`, `SilentlyContinue`, `Inquire`, and `Suspend`
`DebugPreference`	`Stop`, `Inquire`, `Continue`, and `"SilentlyContinue"`
`VerbosePreference`	`Stop`, `Inquire`, `Continue`, and `"SilentlyContinue"`
`WarningPreference`	`Stop`, `Inquire`, `"Continue"`, and `SilentlyContinue`
`ProgressPreference`	`Stop`, `Inquire`, `"Continue"`, and `SilentlyContinue`

`ConfirmPreference` and `WhatIfPreference` are not listed in the table because they deal with risk mitigation rather than output. Their usage will be explained in detail in *Chapter 7, Reactive Practices – Traditional Debugging*.

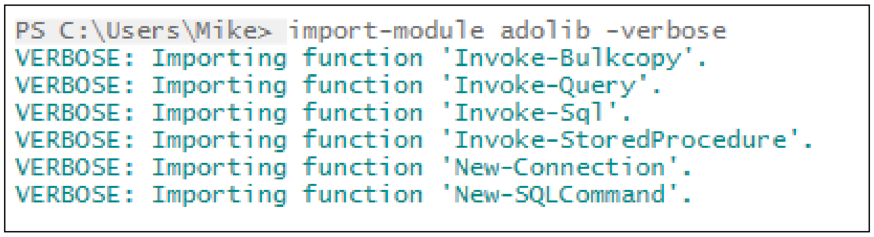
Preference variables are global in scope; that is, their value is used in all scopes unless they are overwritten in a local scope. Because of this, care should be given in changing these variables from their default values. Changing `$ErrorActionPreference` to `Stop`, for instance, might cause error handling in other functions or modules to stop working correctly.

Which Write should I use?

Except for `Write-Output` or its equivalents, which should always be used for function output, the precise use of the `Write-*` cmdlets is neither mandatory nor spelled out in the help content. The exact meaning of the verbose, warning, and debug streams is left to the user of the function or script. On the other hand, it is useful to have some guidelines for what kind of information should be communicated using these streams.

Write-Verbose

Verbose output should include non-technical information that will be valuable to the end user, enabling them to understand the functional processes that are taking place. For instance, using `Write-Verbose` to indicate when each step of a ten-step process is being started would be appropriate. If timing information is something that an end user might want to see, that might be included as well at a high level. It would probably not be usual practice to show timing information for individual iterations of a loop or low-level tasks. The following screenshot shows an example of built-in verbose output with `import-module`:



```
PS C:\Users\Mike> import-module adolib -verbose
VERBOSE: Importing function 'Invoke-Bulkcopy'.
VERBOSE: Importing function 'Invoke-Query'.
VERBOSE: Importing function 'Invoke-Sql'.
VERBOSE: Importing function 'Invoke-StoredProcedure'.
VERBOSE: Importing function 'New-Connection'.
VERBOSE: Importing function 'New-SQLCommand'.
```

Write-Debug

Debug output, as the name implies, is intended to be used to troubleshoot the code in question. Because of this, the content of debug output should provide insight into the implementation details of the operations. Unlike verbose output, which should be general information, debug output should be technical information including variable values, property values, counts, low-level timing information, and so on. The information provided in the debug stream should make what is happening in the code clear and why it is happening. Depending on the complexity of the function, the amount of debug information might be overwhelming, but the point is to provide enough details to help solve problems. Most end users will avoid turning on debug output unless they are working with the code, so the amount of output is not a problem.

Write-Warning

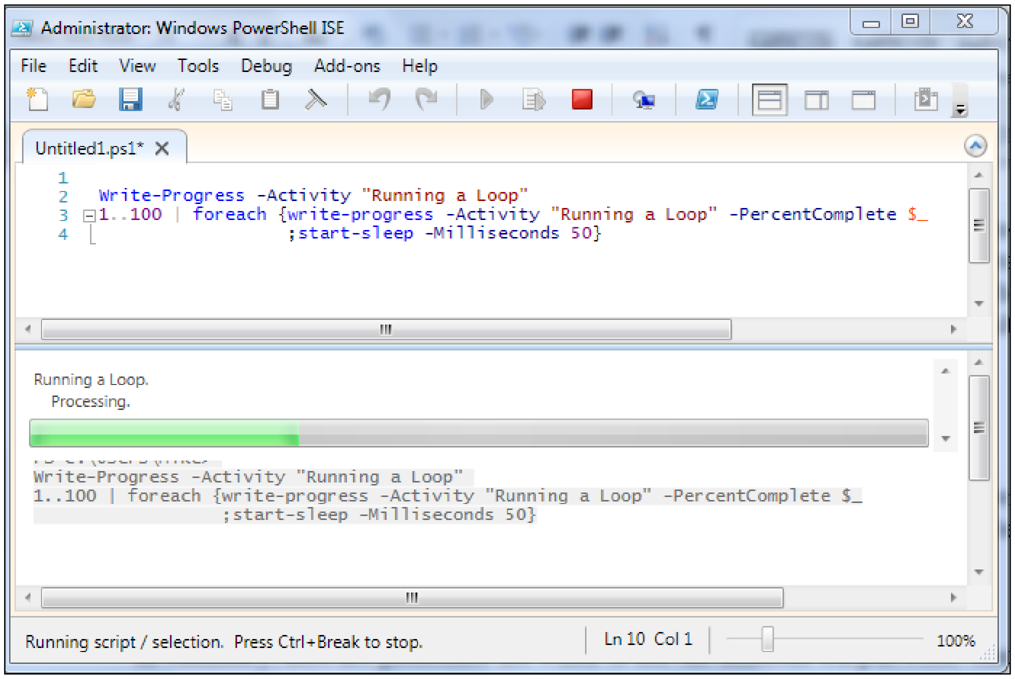
Warning output should always point out conditions that are not as expected, but are also not fatal. Some examples, depending on the operation, might be overwriting a file, no results from a query, stopping a service that is already stopped, or trying to delete a file that doesn't exist. None of these occurrences would necessarily mean that the code would need to stop, but might help the user correct something in the environment.

Write-Error

The error stream is for non-terminating errors. As discussed in *Chapter 2*, *PowerShell Peculiarities*, non-terminating errors indicate that something went wrong, but that the entire operation does not need to stop. By sending output to the error stream, a user could inspect the `$Error` collection to see the specifics about what needs to be followed up on or what results might be missing from the function output.

Write-Progress

The progress stream is where information about the process completion status is written. Operations that might take a long time to complete and have a known number of steps can send the completed percentage or other status information. The following screenshot shows an example using `Write-Progress` in the **Integrated Scripting Environment** (ISE):



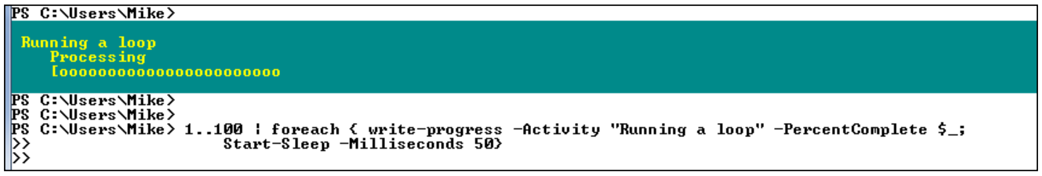
Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Untitled1.ps1*

```
Write-Progress -Activity "Running a Loop"
1..100 | foreach {write-progress -Activity "Running a Loop" -PercentComplete $_
                  ;start-sleep -Milliseconds 50}
```

```
Running a Loop.
   Processing.

Write-Progress -Activity "Running a Loop"
1..100 | foreach {write-progress -Activity "Running a Loop" -PercentComplete $_
                  ;start-sleep -Milliseconds 50}
```

Running script / selection. Press Ctrl+Break to stop. Ln 10 Col 1 100%

The following screenshot shows the output rendered in the console:



```
PS C:\Users\Mike>

 Running a loop
    Processing
    [oooooooooooooooooooooooo

PS C:\Users\Mike>
PS C:\Users\Mike>
PS C:\Users\Mike> 1..100 | foreach { write-progress -Activity "Running a loop" -PercentComplete $_;
>>                       Start-Sleep -Milliseconds 50}
>>
```

In summary, the following table gives the guidelines for when to use the different output `Write-*` cmdlets:

Cmdlet	**Type of information**
`Write-Output`	Objects
`Write-Host`	Formatted text
`Write-Verbose`	Nontechnical process information
`Write-Debug`	Technical implementation details
`Write-Warning`	Problems
`Write-Error`	Non-terminating errors
`Write-Progress`	Process completion information

Further reading

You can go through the following list of references for more information:

- `get-help measure-command`
- `get-help out-default`
- `get-help about_comment_based_help`
- `get-help get-help`
- `get-help write-host`
- `get-help write-output`
- `get-help write-debug`
- `get-help write-warning`
- `get-help write-progress`
- `get-help write-verbose`
- `get-help write-error`

Summary

This chapter dealt with practices that are specific to PowerShell. Filter left and format right are important principles to keep in mind in order to keep pipelines efficient and avoid losing properties that are needed. Comment-based help and the various `Write-*` cmdlets are crucial parts of the PowerShell environment that allow users to discover details about cmdlets and parameters and understand the operation of code.

The next chapter will introduce topics that, while common in development groups, might not be firmly established in the system administration groups. These *programming professionalism* topics include using naming standards, modularization, and unit testing.

4
PowerShell Professionalism

PowerShell scripters might not think of the process of writing a script as a development process, but some industry-standard development practices are appropriate to use with PowerShell, for example:

- Naming standards
- Source control
- Modularization (functional decomposition)
- Unit testing/mocking

Naming conventions

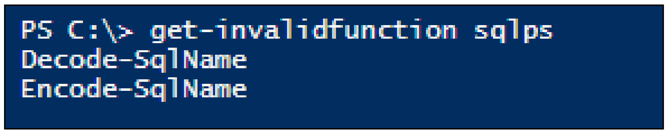
Naming conventions are not rocket science and they are certainly not unique to PowerShell. They are also not concrete rules that will cause your code to stop working if they are ignored. On the other hand, the designers of PowerShell began the language with a strong foundation of consistent naming which is one of the keys to its success. In the following sections, we will discuss several instances where naming conventions will improve your PowerShell experience.

Cmdlet and function naming

As discussed in *Chapter 1*, *PowerShell Primer*, built-in cmdlets are named with a verb-noun format using a verb from a predefined list of approved verbs. This format is not required for user-defined cmdlets, functions, advanced functions, or scripts, but is highly recommended. The only place where not having properly named functions will cause any kind of programmatic issue is when a module, including the code, is imported into a PowerShell session. A module exporting functions that are not correctly named will cause a warning when the module is imported. To disable the warning, the `–DisableNameChecking` switch can be used. `–Verbose` will allow us to see the individual function definitions being imported and will show which functions have issues with naming. The following function uses the verbose output to find incorrectly named functions:

```
function Get-InvalidFunction{
Param([string]$module)
if(Get-Module $module){
    Remove-Module $module -force
}
Import-Module $module -Force -Verbose *>&1 |
      Select-String "The '(.*)' command in the (.*)' module .*" |
      Where-Object {$_.Matches.Groups[2].Value -eq $module} |
      Foreach-Object { $_.matches.groups[1].Value}

}
```

When the `Get-InvalidFunction` function is used to find invalid function names in the SQLPS module, for instance, it shows two offending functions, shown as follows:

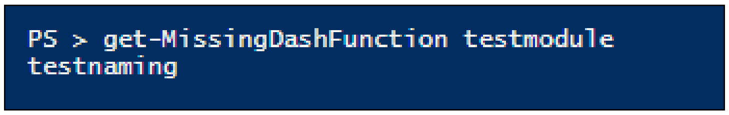


```
PS C:\> get-invalidfunction sqlps
Decode-SqlName
Encode-SqlName
```

A simple test shows that although PowerShell will warn about functions in modules that use improper verbs, it does not indicate a problem with functions that do not follow the verb-noun syntax. The following code snippet shows a function missing a dash, and a function to detect poorly named functions in modules:

```
function testnaming{
  Write-Host "No dash!"
}

function get-MissingDashFunction{
Param([string]$module)
if(get-module $module){
    Remove-Module $module -force
}
      Import-Module $module -force -Verbose *>&1 |
      Select-String "Importing function '(.*)'.*" |
      Where-Object {$_.Matches.Groups[1].Value -notlike "*-*"} |
      Foreach-Object { $_.matches.groups[1].Value}

}
```

The following screenshot illustrates the output showing the function that is missing the dash:



```
PS > get-MissingDashFunction testmodule
testnaming
```

A final thing to mention is that the convention for the noun in a function name is to be a singular noun. This sometimes leads to awkwardly named functions, such as `get-MissingDashFunction` in the previous example, but in order to stay consistent with delivered modules this is a convention that should be observed.

Parameter naming

There are no fixed rules for parameter naming, but we can learn from the authors of the built-in cmdlets and try to use their examples. For instance, looking at the parameters to the `get-childitem` cmdlet, we can see the following parameters (excluding common parameters):

- `Path`
- `LiteralPath`
- `Filter`
- `Include`
- `Exclude`
- `Recurse` (switch)
- `Force` (switch)
- `Name` (switch)
- `Attributes`
- `Directory` (switch)

- `File` (switch)
- `Hidden` (switch)
- `ReadOnly` (switch)
- `System` (switch)

The first thing to see is that `-Path` (and `-LiteralPath`) are the parameters to supply the identity of the child items. This is a good pattern to follow, instead of using other alternatives such as `$filename`, `$file`, `$inputfile`, and so on. Also note that like most of the `*-Item` cmdlets, the `Get-ChildItem` cmdlet provides parameter sets to allow both standard paths (including wildcards) and literal paths, which might include special characters that PowerShell would otherwise attempt to interpret. A final thing to see is that a number of switch parameters have been provided to make it simpler to get the specific results without requiring `Where-Object`. Obviously, not all possibilities are covered with switches, but most of the common cases have been addressed. Other filtering parameters include `-Filter`, `-Include`, and `-Exclude`.

As we write functions, we should try to think about using our functions and include parameters that make their use as fluid and effortless as possible. As we use our own functions (a practice known as **dogfooding**) we will invariably find things that don't flow quite as we'd like them to, so we will have an opportunity to improve them at that time.

Module naming

A survey of public PowerShell modules will reveal fairly quickly that there is no accepted naming convention for modules. Patterns include starting with Posh, starting with PS, using verb-noun formatting, or using single or compound words to name the module. With that out of the way, the following are a few guidelines that will help in this area:

- Use a company-specific prefix for private modules to avoid naming conflicts with public modules (for example, XYZActiveDirectory for a module of functions related to the ActiveDirectory implementation at company XYZ)
- Use a consistent naming scheme for your modules (for example, don't have some Posh modules and some verb-noun modules)
- Group functions into modules based on functionality (rather than project) and name the module according to that functionality

Following these simple guidelines will make your module names easier to remember and make it easier to find functions when you need to look at the source code.

Variable naming

Like modules, there is no accepted naming convention for variables. Again, some commonsense practices will help you keep your variable usage straight and less error-prone. The following is a list of some of these practices:

- Do not use a type prefix for variable names (for example, use `$FileName` instead of `$strFileName`).
- Do not overly abbreviate variable names. Tab completion makes using longer, descriptive variable names less painful.
- Do not reuse variables. PowerShell is garbage-collected, so any efficiency gained by avoiding new variables is miniscule compared to the risk of using a variable incorrectly.
- Avoid generic variable names like `$temp`, `$var`, `$obj`, and so on. Use the variable name to indicate what the variable is being used for or what is being stored.
- Use camel casing (initial capitals in each subsequent word contained in the variable name) to increase readability, for example, `$customerNumber`.

Modularization

Writing a script to accomplish a task can be a daunting process. For those of us that are administrators without any programming background, it might not be straightforward to even know where to start.

For very simple tasks, it might be possible to write the entire script in a single line-by-line flow. While this is possible for the shortest tasks, as we get more comfortable with scripting we will definitely be applying PowerShell to more complex problems. Writing complex scripts in a simple start-to-finish way is bound to cause difficulties. In the following sections, we will give some basic instruction on how to go from an idea to a workable script. We will use the task of copying a production database down to a development server as an example.

Breaking a process into subtasks

The first step is to break the task down into subtasks or steps. A common way to do this is to use a comment statement (starting with the hash character, #) for each step. For our example, it might look something like the following:

```
#Find the latest full backup file

#Figure out how much space is required to restore the database
```

```
#Make sure there is enough space for the restore

#Restore the database with _REFRESH suffix

#Wipe out database-level security on the _REFRESH database

#Export the original database's security as a sql statement

#Run the sql statement to assign the correct security to the _REFRESH
database

#Close all connections to the original database

#Rename the original database to _OLD

#Rename the _REFRESH database to drop the suffix

#Send an email to notify users of the completion
```

Describing a task in this kind of detail is a crucial step in automating. This level of information can be shared with users who might not be familiar with PowerShell for them to validate that what you are writing is going to accomplish the required result. More importantly, it gives us smaller items to work on writing so we can focus on very specific things rather than getting lost in the big picture.

Helper functions

One of the things we can do with this process outline is to look for obvious functions that we will need to write to help us accomplish the steps. In this example, it is clear that we are going to need to be able to communicate with the database. SQL Server comes with a PowerShell module called SQLPS. Using that module to do the low-level database communication, some examples of helper functions that might make sense are given as follows:

- Execute a SQL statement
- Get file sizes from a SQL backup file
- Restore a database

Note that these functions are going to help us write our script, but they are not limited in usefulness to our script alone. Because of this, it would make sense to have a module with these functions in it with a name like SQLServerTools. If we have identified common operations that we need to break out that were specific to this task, we might have helper functions embedded in the script. Remember, though, that reusability is a key benefit of scripting, so try to think in terms of writing more general-purpose functions.

Single responsibility principle

In programming, the single responsibility principle is one of the basic principles of object-oriented programming. This principle states that an object should have a single responsibility. PowerShell uses objects throughout, but most scripting is not considered to be object oriented. On the other hand, the principle can be applied to functions as well as objects. When we write a function, we should make sure that the function performs one operation. When it does more than one thing, it makes using the function difficult. What if we only want one of the things to be done? We could introduce switches to indicate which responsibilities we want to address, but this type of complexity is unnecessary. It is simpler to break the function into multiple functions, each with their own responsibility.

Don't repeat code

One of the keys to automation is that, ideally, anything that you might need to do more than once should be automated. When writing scripts, this principle can be extended to encompass code that is repeated. That is, any code that you might use more than once should be extracted into a function. The reason is that by doing this, you have isolated that operation. By isolating it, any changes to the operation need only be made in one place, rather than needing to find all of the places that we copied the code to.

In our example, we see that we are renaming databases twice, so that immediately indicates that we should have a function to handle that operation. To begin with, we don't need to actually write the function. Simply defining the function with its parameters is a good way to start. This can be done as follows:

```
function rename-Database{
Param($connection,
[string]$name,
[string]$newName)

#construct and invoke sql statement to rename the database

}
```

Since we identified the need to have a helper function to execute SQL statements, we can use that helper function in this function once it has been written. Another example of writing general purpose functions can be seen in the requirement to find the latest full backup. A first attempt at writing this might look like the following code snippet:

```
Function get-LatestFullBackup{
Param($SQLInstanceName,$DatabaseName)
   #look through the appropriate files and return the latest full
backup file
}
```

Although this does satisfy our requirement, it will only satisfy that specific requirement. A more general approach might look like the following code snippet:

```
Function get-BackupFiles{
Param($SQLInstance,
      $Database,
      [switch]$latest,
      [switch]$full)
    #get the backup files for the database in question
  #if $full was specified, use where-object to
      only include those files
  #if $latest was specified, use sort-object and
    select-object to only return the latest file
}
```

With this definition, we can possibly do other things with backup files, such as figure out how old our backups are on disk (what's the oldest full backup) or determine how much disk space is taken up by the backups of a specific database. It takes some thought to write functions with more general focus, but the time spent in designing them will pay off in the end.

Understanding the process

Once we have defined the steps in the task and written the appropriate helper functions, the next thing to do is to code each of the steps. At this point, we still might find some helper functions that need to be written. For instance, the following code snippet might be the start of the script:

```
#Find the latest full backup file
$backup=get-backupFiles –sqlinstance $source `
                        –database $DBName –latest -full
#figure out how much space is required to restore the database
```

```
$space=get-backupSize $backup

#Make sure there is enough space for the restore
If (-not (test-diskSpace -computername $dest `
-size $space)){
  Throw "$space bytes needed on $dest"
}
#Restore the database with _REFRESH suffix
Start-Restore -SQLInstance $dest -database "$DBName`_REFRESH" `
-path $backup
```

As you expand each step, you will probably realize that there is information that you need in order to execute the step. When this happens, simply insert a step to take care of it. That may involve writing more helper functions. Complex steps might be better broken down into smaller steps. One way to do this is to simply replace the step with the list of smaller steps. Another approach (which I prefer) is to write that step as a function, and include the smaller steps in it. You will need to be careful to pass all of the required information into the new function. Eventually, you will find that you have all of the steps coded and are ready to test your script!

Breaking your script into smaller sections is not a magic trick that makes it more reliable, but it does allow you to test the smaller portions (for example, the helper functions) independently of the whole process. Gaining confidence in the quality of your helper functions makes it possible to spend more time focusing on the business process, which means higher productivity. A big part of troubleshooting is knowing where to look for mistakes. Being able to eliminate large portions of your code will make a tremendous difference.

Version control

Source control, or version control, is for developers, right? Why would PowerShell scripters, who are mostly administrators of one flavor or other, need to use source control? Here are a few scenarios that will hopefully convince you.

You get a call at midnight that a mission-critical script is failing. You dial into the server and look at a thousand-line script with no indication of what the problem is. With source control you would at least be able to look at the history of the script and see if there were any changes made recently. Recent changes aren't always the culprit, but without anything else to go on, they are usually a good place to start. If you're lucky, the person who made the changes included a really good check-in comment about why the changes were made that will help you determine if it's relevant. If it sounds like it's the problem, a solution might be as simple as reverting the script to its previous version.

Let's take a look at a second scenario. The drive storing the scripts on your production server just bit the dust. You can kind of remember what the scripts on that server were doing, but do you have time to rewrite them? If you were using source control, you should be able to get the latest versions of all of the scripts.

Perhaps you have more than one administrator writing scripts. Having a central place to store scripts is a pretty straightforward result of using source control. Source control also allows more than one person to be working on a script and gives the ability to merge the changes together. This might seem somewhat far-fetched, but as more and more tasks in the Microsoft ecosystem are turning to PowerShell, the more likely this scenario is becoming. Once most admins are scripting the possibility of a collision ("Hey, I was editing that file! You just wiped out my changes!") is much greater.

With a software project, there's the idea of a project. It could be a website, a desktop application, a service, or a mobile app, but there is a definite grouping that makes sense to use when organizing a source code repository. With PowerShell, scripting generally involves building up a good selection of modules with helper functions related to the tasks you perform, and a bunch of scripts that use those modules. There's no "big thing" that stands out that will be a natural organization.

One way to organize a repository of PowerShell scripts is to have a folder of modules (which would each have its own subfolder) and a folder of scripts. This setup would mirror the `Documents\WindowsPowerShell` folder present in each Windows user profile. Another possibility is to organize the repository according to what server the scripts are going to be deployed on.

However you choose to organize your repository, the important thing is that you consistently commit code to the repository. Committing code is similar to making a backup. You know you can always go back to that point in time. Just as you wouldn't consider running a database without performing periodic backups, you should not write scripts without frequently committing those scripts. An important ingredient in a commit is a comment describing what changes were made and why. The system will take care keeping track of who made the changes, to what files, and when the changes were made.

Using version control with PowerShell

Since PowerShell runs on Windows systems, we have several good options for version control, including **Team Foundation Services** (**TFS**), Subversion, Git, and Mercurial (hg). If we consider TFS Express edition, all of these are free of charge, so there is no financial reason to keep from implementing one. Each has its own advantages and disadvantages, so choose the one that's best suited to you. Your company might already have a standardized version control system, so ask around to see if the choice has been made for you.

All four of the version control systems mentioned have popular GUI frontends and extensions to Windows Explorer, so using them can be as simple as right-clicking on files and folders and selecting the appropriate operation in a context menu. With PowerShell, though, we have a couple of other possibilities. We can use the built-in command-line interfaces for these systems or we can use PowerShell cmdlets written to interface with them.

The specific details of how to use version control (that is, which commands perform which operations) are beyond the scope of this section. The important point for you to take away is that you need to be submitting your scripts to a version control system. How that happens is mostly a matter of preference. No matter what software you decide on, whatever workflow you chose to check code, the benefits will follow from consistent use.

Unit testing

Unit testing is the logical continuation of modularizing code. Once you have broken the problem into smaller pieces or units, the next step is to test those units. To perform unit testing means that we will write tests that exercise each unit with a variety of inputs that will ensure that the code is correct. One emphasis of unit testing is that the tests need to be automatic. That is, we're not reading a list of test cases off of a piece of paper and running the code with each to verify that the results are as expected. Unit tests are code and are just as important as the code being tested (and as such should be checked into your version control system). Developers are familiar with automated unit testing, but the use of unit tests by system administrators is growing. As system administration begins to involve more and more code, the knowledge that the code we use is correct is of utmost importance.

You might hear discussion of **Test Driven Development** (**TDD**), which relies on a failing unit test in order to do any development, but for our purposes that is not necessary. Our emphasis will simply be on using unit tests to have confidence in our implementation.

Another important point is that unit tests only test the code in the function. They are not intended to test the entire environment (filesystem, network, database, and so on). End-to-end testing involving the entire infrastructure is called integration testing and is a separate subject.

Rolling your own unit tests

Let's consider a function that takes a list of computer names and a domain name as parameters and returns a list of **fully-qualified domain names** (**FQDN**). An implementation might look like the following code snippet:

```
function get-fqdn{
Param([string[]]$computerName,
      [string]$domainName)
    $output=@()
    foreach($computer in $computerName){
        $output+=$computer+'.'+$domainName
    }
    return $output
}
```

At first, we might try to think of some things to test using the following list of conditions and their respective code:

- Pass two computer names and a domain name (base test):

  ```
  get-fqdn comp1,comp2 -domain test.com #Should
    be @("comp1.test.com","comp2.test.com")
  ```

- Pass a single computer name and a domain name:

  ```
  get-fqdn comp1 -domain test.com #Should be "comp1.test.com"
  ```

- Pass a single computer name and a domain name starting with a dot:

  ```
  get-fqdn comp1 -domain .test.com #Should be
    "comp1.test.com"
  ```

- Pass a FQDN (instead of a computer name) and a domain name:

  ```
  get-fqdn comp1.domain.com -domain .test.com #Should be
    "comp1.domain.com"
  ```

It's important to understand that writing unit tests is, in a way, writing the specification for your code. For instance, the third and fourth tests are logical, but they weren't in the description of the code. By writing these tests in code, we are expressing how our function is intended to be called as well as explaining what kind of output is expected. Unit tests that are kept up to date are an important kind of documentation. In fact, functioning unit tests are often the best documentation since written documentation (in the form of documents or comment-based help, for example) can get out of sync with the code. Unit tests that pass necessarily reflect the code as written or they wouldn't pass.

To test these conditions, you would probably need some helper functions. We will introduce an open source framework to simplify this later in the chapter, but at this point we will keep it simple. The following is a simple function that tests whether two arrays are equal:

```
function test-arraysEqual{
Param([array]$a,
[array]$b)
-not (compare-Object -ReferenceObject $a -DifferenceObject $b)
}
```

This works because the `Compare-Object` cmdlet returns the difference objects for objects in the list which are different. The `-not` changes the empty list from equal arrays into a `$true` and changes the list of objects for different lists (which is logically `$true`) into `$false`. With that function, we can express the previous list of tests using the following code snippet:

```
test-arraysEqual (get-fqdn comp1,comp2 -domain test.com )
   ("comp1.test.com","comp2.test.com")

test-arraysEqual (get-fqdn comp1  -domain test.com )
   ("comp1.test.com")

test-arraysEqual (get-fqdn comp1 -domain .test.com )
   ("comp1.test.com")

test-arraysEqual (get-fqdn comp1.domain.com -domain test.com )
   ("comp1.domain.com")
```

When we run that, we get the following disappointing, but not very surprising, results:

```
True
True
False
False
```

The two tests we hadn't thought about when writing the code didn't pass. Let's make a quick adjustment to the code to fix that:

```
function get-fqdn{
Param([string[]]$computerName,
      [string]$domainName)
    $output=@()
    if($domainName.StartsWith(".")){
        $domainName=$domainName.Substring(1)
    }
    Foreach($computer in $computerName){
        if($computer.Contains('.')){
            $output+=$computerName
        } else {
            $output+=$computer+'.'+$domainName
        }
    }
    return $output
}
```

Now, we get a clean set of results:

```
True
True
True
True
```

That's the rhythm of coding with unit tests. You write code, you test it. You fix the code so the tests pass. If you think of more things to test, you add tests and repeat.

A great thing about unit tests is that if you have an implementation with all passing tests, you can change the way you implement it and you can still know if the implementation is correct (with respect to the tests). In the following example, we could remove the use of the `$output` variable to hold the results and simply output the values directly to the pipeline instead:

```
function get-fqdn{
Param([string[]]$computerName,
      [string]$domainName)
    if($domainName.StartsWith(".")){
        $domainName=$domainName.Substring(1)
    }
    foreach($computer in $computerName){
        if($computer.Contains('.')){
            $computerName
        } else {
            $computer+'.'+$domainName
        }
    }
}
```

Seeing that the unit tests all still pass tells us that this was a valid change. If we had gotten any failures, we would know we still have some work to do.

If you find a bug in your code, a good thing to do is isolate that bug in a unit test before fixing the test. When you write the test first, you are making sure you understand what went wrong. You are saying "the code should have returned this" and instead you get a negative result. That doesn't seem like it's telling you anything, but once you've corrected the code you now have a test that should pass.

Why is PowerShell testing difficult?

Consider the following function:

```
function get-dayOfWeek{
    return (get-date).DayOfWeek
}
```

While this is a simple function that is not performing anything original, and we would probably not write tests for it, it is interesting to try to imagine how we would go about trying to test it. We could certainly test it manually by running it and looking at a calendar, but the fact that the output is dependent on the current date makes it difficult. Many typical PowerShell functions have many more dependencies than this.

Thinking back to our example in the section on modularization, you'll remember we had a function that renamed databases. Writing a unit test for that would not be nearly as straightforward as the tests we showed for `get-fqdn` in the previous section. For instance, it requires that there is an SQL Server instance with a database we can connect to. It would also require that we have permissions to rename the database and that there isn't a database with the new name on that SQL instance. Similarly, most PowerShell scripts deal with external entities: filesystems, servers, active directory, exchange, and so on. It's not reasonable to assume that there is a test environment available for testing scripts. Even if there was, there is no guarantee that it is configured like the live environment that our scripts will be running in. Furthermore, the logic of our function is what we want to test, not the behavior of the external system. To isolate our tests from external dependencies like this, we will turn to the concept of mocking and the Pester testing framework.

An introduction to Pester

The Pester framework (`https://github.com/pester/Pester`) gives us a more structured way to perform our unit tests. First of all, Pester has us create a fixture for our unit testing purposes with the `new-fixture` function. A fixture is simply a folder with two scripts in it. The first script has the function we are going to test. The second is where our tests will go. The test script files must be named with an extension of `.Tests.PS1` in order for Pester to be able to find them. After copying the `get-fqdn` function into the first script, I have rewritten the tests in Pester's syntax. The vocabulary Pester uses is from the **behavior-driven development** (**BDD**) style of testing. The syntax might look strange because it doesn't follow the verb-noun format in tests. Instead, our tests are included in an `It` statement, and use the `Should` function and the `Be` assertion. Understanding exactly how the PowerShell works is interesting and I recommend reading through the `Should` function to help in this regard. For our purposes, though, it is enough to know that this is a simple and self-explanatory way to write tests, such as the following:

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path).Replace(".
Tests.", ".")
. "$here\$sut"
```

```
Describe "get-fqdn" {

    It "works with a list" {
      get-fqdn comp1,comp2 domain.com |
              Should Be 'comp1.domain.com','comp2.domain.com'
    }
    It "works with a single computername" {
      get-fqdn comp1 domain.com |
Should Be 'comp1.domain.com'
    }
    It "works with a leading dot in domainname" {
    get-fqdn comp1 .domain.com |
Should Be 'comp1.domain.com'
    }
    It "doesn't modify a specified fqdn" {
    get-fqdn comp1.domain2.com .domain.com |
Should Be 'comp1.domain2.com'
    }

}
```

Running the tests with the `Invoke-Pester` function gives us the following result which shows that all four tests passed:

```
Executing all tests in C:\Users\Mike\SkyDrive\Documents\
PowerShellTroubleshooting\ChapterDrafts\Chapter4_1stDraft\Code\get-
fqdn
Describing get-fqdn
 [+] works with a list 22ms
 [+] works with a single computername 2ms
 [+] works with a leading dot in domainname 24ms
 [+] doesn't modify a specified fqdn 2ms
Tests completed in 51ms
Passed: 4 Failed: 0
```

Mocking with Pester

One of the interesting features of the Pester framework is the ability to create mocks. In general, a mock is something that will be substituted for a dependency in a unit test. The mock is created so that it only has the behavior that we are interested in and gives predictable results. Testing the simple function we described earlier is fairly simple to accomplish with mocks. First, let's write the following function:

```
function get-dayOfWeek{
    return (get-date).DayOfWeek
}
```

Now, let's consider the following tests:

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf
  $MyInvocation.MyCommand.Path).Replace(".Tests.", ".")
. "$here\$sut"

Describe "DayOfWeek" {

    It "Returns Thursday for 6/26/2014" {
      Mock Get-Date {return [DateTime]'6/26/2014'}
            get-DayOfWeek | Should Be 'Thursday'
        }
    It "Returns Monday for 3/24/2014" {
        Mock Get-Date {return [DateTime]'3/24/2014'}
            get-DayOfWeek | Should Be 'Monday'
        }
```

Note that we used the `Mock` function to create versions of `get-date` that return specific dates. With those dates, it is trivial to check the day of the week against the return value. By using a mock, we have eliminated the dependency on the implementation of `get-date` and are free to concentrate on how the output of `get-date` is used.

A more complicated function allows us to use more of the functionality provided by Pester. The following is a function that restarts the current machine if it has been running for more than 30 days:

```
function restart-ServerAfter30Days {
  $lastBootTime=get-CIMInstance Win32_OperatingSystem |
select-object -expand LastBootUpTime
  $now=get-date
  if(($now - $lastBootTime).TotalDays -gt 30) {
    Restart-Computer -WhatIf
  }
}
```

To test this function, we first need to remove the dependency on `Get-Date` and `Get-CIMInstance` using mocks. We also need to mock `Restart-computer` because it would not be helpful for the computer running the test to restart during the test. We can use the `Assert-MockCalled` function to ensure that the `Restart-Computer` cmdlet was called (or not) in the correct situations. Finally, we have supplied a filter on the `Get-CIMInstance` mock so that it only applies if the `Win32_OperatingSystem` class name was passed as a parameter. In this case, the filter was not necessary, but it serves to illustrate the usage of filters. The following code snippet explains this:

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path).Replace(".
Tests.", ".")
. "$here\$sut"

Describe "restart-ServerAfter30Days" {
   Mock get-CIMInstance { new-object PSObject -prop @{LastBootUpTi
me=[Datetime]'2/1/2014'}}  -parameterFilter {$ClassName -eq 'Win32_
OperatingSystem'}
    Mock Restart-Computer {}
    Context "When restart date is more than 30 days old" {
        Mock get-date {return [Datetime]'4/1/2014'}
        restart-ServerAfter30Days
        It 'reboots' {
            Assert-MockCalled Restart-Computer -times 1
        }
    }
    Context "When restart date is less than 30 days old" {
        Mock get-date {return [DateTime]'2/15/2014'}
        restart-ServerAfter30Days
        It 'Does not reboot' {
            Assert-MockCalled Restart-Computer -times 0
        }
    }
}
```

Again, using `Invoke-Pester` shows us that the function passes both tests in the following code snippet:

```
Executing all tests in C:\Users\Mike\SkyDrive\Documents\
PowerShellTroubleshooting\ChapterDrafts\Chapter4_1stDraft\Code\
restartTest
Describing restart-ServerAfter30Days
    Context When restart date is more than 30 days old
      [+] reboots 4ms
    Context When restart date is less than 30 days old
      [+] Does not reboot 26ms
Tests completed in 30ms
Passed: 2 Failed: 0
```

Mocking is an important tool in the unit testing arsenal and allows us to test a number of things that would be difficult to test without them. On the other hand, as the mocking scenario gets more and more complex, there is a question about how tightly-coupled your code and the mock object are. Tightly-coupled code is a danger, as it makes changing code difficult. If you find yourself spending a lot of your time writing mocks, you might want to take a step back and re-evaluate.

Further reading

Take a look at the following references for more information:

- Naming conventions at `http://en.wikipedia.org/wiki/Naming_convention_%28programming%29`
- Dogfooding at `http://en.wikipedia.org/wiki/Eating_your_own_dog_food`
- Modularization at `http://en.wikipedia.org/wiki/Modular_programming`
- Unit testing at `http://en.wikipedia.org/wiki/Unit_testing`
- Pester at `https://github.com/pester/Pester`
- Mocking at `http://en.wikipedia.org/wiki/Mock_object`

Summary

In this chapter, we looked at several practices borrowed from traditional software development that will help your scripting look more professional. These practices include things like using naming conventions for different types of PowerShell entities, breaking scripts down into smaller units, using version control consistently, and finally unit testing and mocking.

In the next chapter, we will look at using some of the built-in features of PowerShell to proactively create more trouble-free programs. These features include error handling (`try` / `catch` vs trap), parameterization and pipeline input, parameter validation, parameter type transformation, and so on.

5 Proactive PowerShell

PowerShell includes capabilities to improve the quality of scripts that will prevent some of the problems that might be encountered in the scripting process. In this chapter, we'll cover the following topics:

- Error handling (`try`/`catch` versus `trap`)
- Parameterization and pipeline input
- Pipelines and function execution
- Parameter validation
- Parameter type transformation
- Strictmode/PSDebug
- #REQUIRES statements (version or administrator)
- CmdletBinding and common parameters

Error handling

Chapter 2, PowerShell Peculiarities introduced PowerShell's two error-handling mechanisms: the `trap` statement and the `try`, `catch`, and `finally` statements. That chapter explained how these statements function in PowerShell code. The following sections will give you some guidance on how to use them effectively and some techniques for writing error-handling code.

Error-handling guidelines

The first thing to mention is that although the `trap` statement can be effective for handling errors, its flow can be confusing, especially when considering the many ways to exit a trap statement. For this reason, it is a good idea to avoid the `trap` statement in most cases and use the `try` / `catch` / `finally` constructions instead. Since environments that only support PowerShell Version 1.0 are not very common, `try`, `catch`, and `finally` can be used almost everywhere. Also, the flow of `try` / `catch` / `finally` is much more linear, leading to less confusion about the flow of execution.

A second point is that when writing error-handling code (with either `try`/`catch`/`finally`, or `trap`), we should avoid using empty error handlers. The following code is a bad example of handling errors:

```
#this is bad!
try {
   Start-Service MSSQLSERVER -Computer CORPSQL -errorAction Stop
} catch { }
```

The reason this is a poor practice is simple. If it doesn't matter whether the code had an error, the operation couldn't have been very important. If we don't need the code to succeed, or even to know what went wrong, we probably don't need to be executing the code in the first place. The same observation can be made for using the `Ignore` value for the `ErrorAction` common parameter or the `$ErrorActionPreference` variable. Since that setting instructs PowerShell to not only refrain from writing errors to the error stream but also to not record the error in the `$Error` collection, there is no way to know whether the operation succeeded or not, and there is no way to know what went wrong.

Error-handling techniques

First of all, let's state for the record that it is important to handle errors. That might sound obvious, but it is easy to get caught up in how powerful PowerShell is and then forget that even though PowerShell has the capability to do things, the world doesn't always cooperate with our plans. For instance, consider the following line of a script:

```
$service = Get-Service MSSQLServer –computername MYSERVER
```

While the expectation might be that the statement in question will always succeed, and in ordinary circumstances it will, there are clearly some ways in which things can go wrong. Scripters with a development background usually start the error-handling process with code like this:

```
Try{
    $service = Get-Service MSSQLServer -ComputerName MYSERVER
}
catch {
    #handle the error appropriately
}
```

In this case, this approach is not successful since the `Get-Service` cmdlet doesn't throw terminating errors but emits non-terminating errors instead. Even though we're trying to handle all errors here, the `try` / `catch` statements seem to have no effect.

Using a `try` / `catch` / `finally` construct in conjunction with the `-ErrorAction STOP` parameter gives us the ability to handle non-terminating errors. The code now looks like this:

```
Try {
$service = Get-Service MSSQLServer -computername MYSERVER -ErrorAction
Stop
} catch {
   #respond appropriately to the error condition.
}
```

Knowing what kinds of errors need to be handled can be accomplished by running some sample broken scripts:

```
#test a good computer name with a bad service name
Get-Service MSSQLSERVERZZZZ -ComputerName Localhost
#test a bad computer name with a good service name
Get-Service MSSQLSERVER -ComputerName NOSUCHCOMPUTER
```

This gives a good start for trying to figure out how to respond. Now, we need to determine whether these errors are terminating errors or non-terminating errors. Like we covered in *Chapter 2*, *PowerShell Peculiarities*, non-terminating errors aren't caught by a `try` / `catch` construction, so we can do something like the following:

```
Try {
   Get-Service MSSQLSERVERZZZZ -ComputerName Localhost
} catch {
  "An exception : ($_) happened"
}
```

In this case, the code indicates that an exception (that is, a terminating error) has not occurred, so we know that this particular error is a non-terminating error. To handle non-terminating errors, we need to use the `-ErrorAction` parameter with the value `Stop`:

```
Try {
    Get-Service MSSQLSERVERZZZZ -computername Localhost -ErrorAction
Stop
} catch {
  $err=$_
  "An exception : ($err) happened"
}
```

We are now able to handle this condition. By repeating this process with each kind of bad parameter, we can figure out how to structure the error-handling code.

Investigating cmdlet error behavior

Although the PowerShell language designers have worked very hard to create a scripting environment that is very consistent and have included many language features to help out in this direction, it can be dangerous to assume that different cmdlets will respond to errors in the same way. A couple of examples should indicate the danger of assuming cmdlets behave similarly.

First, have a look at the following script, which gets references to the `Spooler` service on two computers using `Get-Service` and `Get-WMIObject`:

```
$computers='localhost','NOSUCHCOMPUTER'
Get-Service Spooler -ComputerName $computers
Get-WMIObject Win32_Service -ComputerName
  $computers -Filter "Name='Spooler'"
```

The `Get-Service` cmdlet simply returns the localhost spooler with no indication of any error except for the delay in trying to resolve the nonexistent computer. The `Get-WMIObject` cmdlet, on the other hand, emits a non-terminating error about not being able to connect to the RPC service on `NOSUCHCOMPUTER`, which is the service responsible for WMI communication.

A second example concerns operations that require administrative privileges. In a PowerShell instance that is not running with administrative privileges, the `Get-VM` cmdlet simply returns no objects even if there are VMs configured on the system. On the other hand, stop-service BITS will emit a non-terminating error if it's run as a non-administrator.

Hopefully, these examples are sufficient to convince you that actually investigating the error responses for cmdlets is a valuable exercise. Being able to respond correctly to the error conditions is an important part of troubleshooting PowerShell code.

Catch and release

One last thing to mention about error handling is that error handling in a function or script does not necessarily need to respond to every possible error. This might seem to contradict the previous sections' emphasis on investigating error modes for cmdlets that you're using, but there is an important distinction. It really only makes sense to handle the errors that are related to the process that is being performed. For instance, code that deals with reading a performance counter should probably be expected to react to only a few kinds of errors:

- Missing performance counters
- Insufficient privileges to read the performance counter

What conditions are we not considering? What about the *Out of Memory* errors? What if the OS is shutting down? What if a non-terminating error is not one of the conditions that you have a response for? In cases such as these, it might make sense to either throw the exception again (in the case of a terminating error) or rewrite the error to the error stream, as shown in the following code snippet:

```
try {
   Get-Service BLAH -ErrorAction stop
} catch {
  if($_.Exception.Message -like 'Cannot find any service*'){
     #do something about the missing service
  } else {
    throw
  }
}
```

This is not always necessary, but it does allow system-level errors to be handled at the appropriate level.

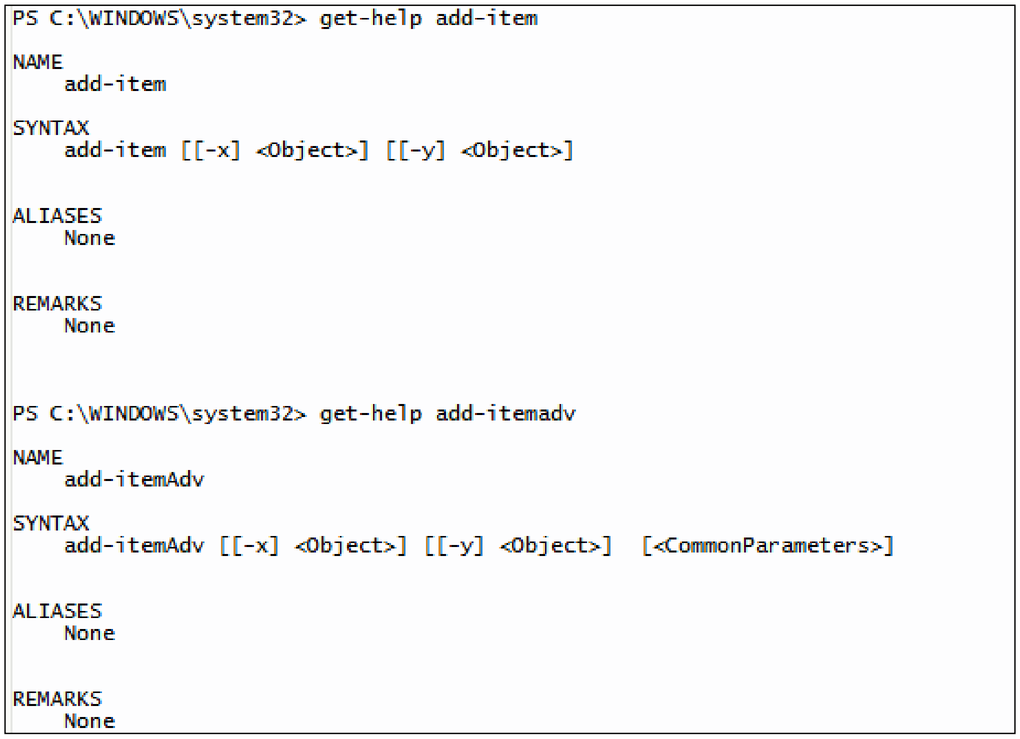
CmdletBinding()

In PowerShell Version 1.0, the only way to write cmdlets was with managed code. Starting with Version 2.0, it became possible to write advanced functions that have all of the capabilities of managed cmdlets but are written in 100 percent PowerShell. The key to writing advanced functions (also sometimes called script cmdlets) is the `CmdletBinding()` attribute, which is added to the `Param()` statement. Since the attribute is tied to the `Param()` statement, advanced functions must have a `Param()` statement even if they have no parameters. In this case, an empty `Param()` statement can be used. The following is an example of a normal function and an advanced function, which are nominally the same:

```
#this is a normal function
function add-item{
param($x,$y)
    Write-Output $x+$y
}

#the same function as an advanced function
function add-itemAdv{
[CmdletBinding()]
param($x,$y)
    Write-Output $x+$y
}
```

Common parameter support

Although the two functions seem to be the same, `Get-Help` shows that there is a difference:



```
PS C:\WINDOWS\system32> get-help add-item

NAME
    add-item

SYNTAX
    add-item [[-x] <Object>] [[-y] <Object>]


ALIASES
    None


REMARKS
    None


PS C:\WINDOWS\system32> get-help add-itemadv

NAME
    add-itemAdv

SYNTAX
    add-itemAdv [[-x] <Object>] [[-y] <Object>]  [<CommonParameters>]


ALIASES
    None


REMARKS
    None
```

One of the benefits of writing advanced functions is the support for common parameters. From the `about_CommonParameters` help topic, we can see that the basic set of common parameters that are available to every advanced function or cmdlet (in PowerShell Version 4.0) include the following:

- `-Debug`
- `-ErrorAction`
- `-ErrorVariable`
- `-OutVariable`
- `-OutBuffer`

- `-PipelineVariable`
- `-Verbose`
- `-WarningAction`
- `-WarningVariable`

Supporting these parameters means that we don't have to do anything to make them work. For instance, we can use `Write-Verbose` throughout our advanced function and the verbose output will show up if the `-Verbose` switch is specified. Similarly, we can use `Write-Error` to emit non-terminating errors and the engine will convert them to exceptions if the caller specifies `-ErrorAction Stop`.

SupportsShouldProcess

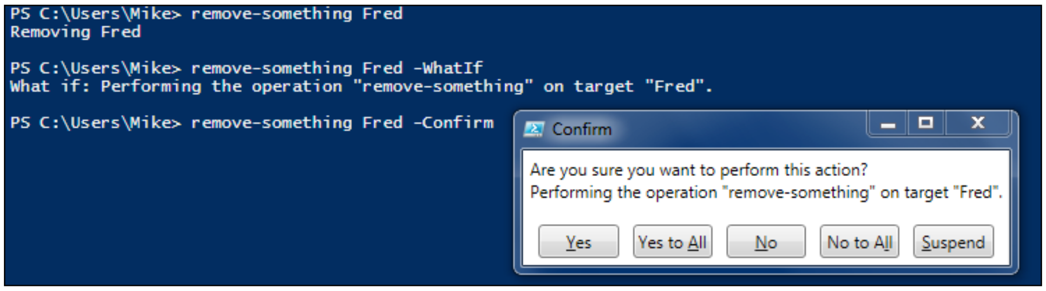
Cmdlets whose execution changes the state of the system where some risk is involved should include risk mitigation parameters, which are the following:

- `-WhatIf`
- `-Confirm`

For an advanced function to use these parameters, the `SupportsShouldProcess` parameter of the `CmdletBinding` attribute should be given a value of `$true`. Portions of code that involve the risk should be guarded with the `$PSCmdlet.ShouldProcess()` method. This method returns `$true` unless the caller specified the `-WhatIf` switch or the `-Confirm` switch followed by a negative response:

```
function remove-something{
[CmdletBinding(SupportsShouldProcess=$true)]
Param($item)

    if ($PSCmdlet.ShouldProcess($item)){
       Write-Output "Removing $item"
    }

}
```

Here is some sample output from that advanced function showing the operation of the `-Whatif` and `-Confirm` switches:



```
PS C:\Users\Mike> remove-something Fred
Removing Fred

PS C:\Users\Mike> remove-something Fred -WhatIf
What if: Performing the operation "remove-something" on target "Fred".

PS C:\Users\Mike> remove-something Fred -Confirm
```

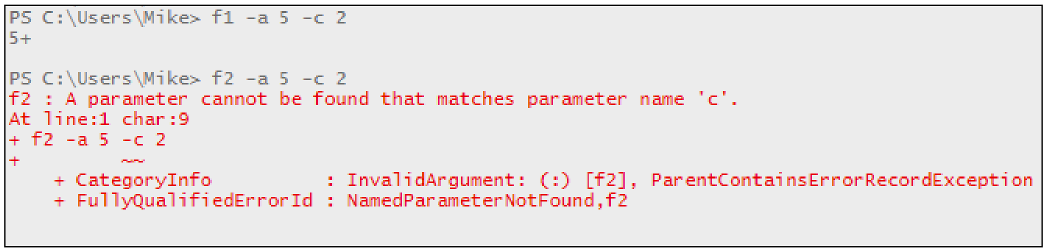
Are you sure you want to perform this action?
Performing the operation "remove-something" on target "Fred".

Parameter name validation

One important consequence of writing advanced functions (that is, using `CmdletBinding`) is that named parameters that are passed but do not exist in the function will cause the parameter binding to fail. The following code will help you understand this:

```
function f1{
Param($a,$b)
     Write-Host $a+$b
}

function f2{
[CmdletBinding()]
Param($a,$b)
     Write-Host $a+$b
}
```

Given these two functions, observe the following output:



```
PS C:\Users\Mike> f1 -a 5 -c 2
5+

PS C:\Users\Mike> f2 -a 5 -c 2
f2 : A parameter cannot be found that matches parameter name 'c'.
At line:1 char:9
+ f2 -a 5 -c 2
+         ~~
    + CategoryInfo          : InvalidArgument: (:) [f2], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : NamedParameterNotFound,f2
```

Though this is a very simple example, and it doesn't seem like a big deal, parameter name checking is an essential ingredient in writing robust scripts. The reason is that we don't generally name parameters things such as `A` and `B`. Longer parameter names such as `ConnectionString` and `ComputerName` are much easier to mistype, and without this feature, parameter name typos will go unnoticed. Also, because the parser is checking parameter names, it is possible to abbreviate parameter names and the parser will be able to determine which parameter is being referenced.

This kind of mistake bit me early in my PowerShell experience when I was testing some code. I was trying to stop a demo system and "typoed" the name of the parameter that specified which system to stop. The code in question didn't do any sanity checking to make sure that reasonable parameters were supplied. Since this was PowerShell Version 1.0, which didn't have advanced functions, the function didn't receive any `-System` parameter and ignored the demo system name that I passed with the wrong parameter name. I shut down all relevant systems in my company. Oops! This is not the kind of mistake you want to make.

Parameter value validation

In addition to validating the names of parameters, using `CmdletBinding()` allows us to provide several types of validation rules for values of parameters. Validation rules enable us to provide checks that are performed by the parameter binding engine that specify when values are appropriate or not for the function. These parameter checking attributes can be divided into two groups: requirement validation and value checking. Requirement checking indicates whether the parameter is required or if various types of empty values are allowed. These attributes include the following:

- `Mandatory`
- `AllowNull`
- `AllowEmptyString`
- `AllowEmptyCollection`
- `ValidateNotNull`
- `ValidateNotNullOrEmpty`

Value checking attributes restrict the values that are allowed for a parameter. Value checking attributes include the following:

- `ValidateCount`
- `ValidateLength`
- `ValidatePattern`
- `ValidateRange`
- `ValidateScript`
- `ValidateSet`

Parameter attribute usage is illustrated in the following script:

```
Function test-validation{
[CmdletBinding()]
Param([ValidateLength(4,10)][string]$word)
Write-Output "the word was $word"
}
```

In this script, we have used the `ValidateLength` parameter attribute to ensure that values passed in for `$word` are strings between 4 and 10 characters in length. Passing invalid values (either longer or shorter) will cause this validation rule to fail and an error will be emitted without executing the function. We can fairly easily write code to validate the parameter without attributes like this:

```
Function test-validation{
[CmdletBinding()]
Param([string]$word)
  #Don't do this!
    If($word.Length -lt 4 -or $word.Length -gt 10){
    Throw "The value for $word has an invalid length"
    }
Write-Output "the word was $word"
}
```

There are several reasons this approach should be avoided. Some of them are listed as follows:

- It involves more manual coding
- The function is actually executing, so any errors in the parameter validation might end up with inadvertent results
- The parameter checking code is separated from the parameter definition
- Custom error messages will be inconsistent between different scripters (and often even with the same scripters)
- Custom error messages will not (in general) be localized, so they will appear in the scripter's language only

Pipeline input

The ability to accept pipeline input has been included in PowerShell since Version 1.0. There are three main ways to deal with the pipeline: `$input`, filters, and (fully specified) functions.

The `$input` automatic variable exposes an enumerator (think collection) of all the values passed in on the pipeline. An example using the `$input` automatic variable might look like this:

```
Function get-pipelineinput{
    $input | Foreach-Object {Write-Host "the object was $_"}
}
```

The second option, filters, are simply functions whose bodies are executed for each pipeline element. Filters use the `$_` symbol to represent the current pipeline element. Here is an example of a filter script:

```
Filter get-reverse{
      $_.ToString().Reverse()
}
```

Because there is no way to specify what types of values were available for pipeline input using the `$input` or `$_` variables, these are not good solutions for most production scripts. With PowerShell Version 2.0 and the introduction of `CmdletBinding()`, another more powerful option became available, using parameter attributes to indicate pipeline binding. Before illustrating these parameter attributes, we need to explain that the example functions presented in *Chapter 1*, *PowerShell Primer*, did not illustrate the full form of a function definition. When working with the pipeline we need to know that there are three possible sections in a function definition: `Begin`, `Process`, and `End`. If no named section is used, the (unnamed) function body is the end section. Here is the full form of a function:

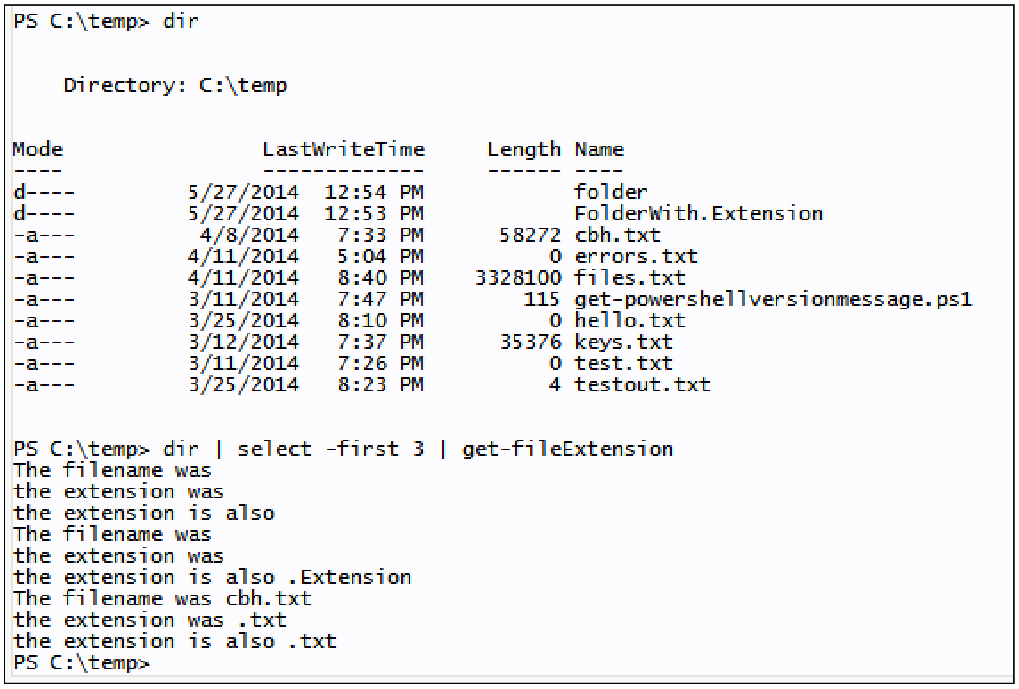
```
Function <function name>{
[CmdletBinding()]
Param(<parameters>)
Begin {
        #<executed before processing pipeline items>
      }
Process {
          #<code executed for each pipeline item>
        }
End {
      #<executed after processing pipeline items>
      }
}
```

It should be clear from the outline that the begin block is executed at the beginning of the pipeline, before any items have been processed. The process block is then executed for each item received from the pipeline that can be accessed via the `$_` variable which is also used in filters or with parameters that have been designated to bind to pipeline items.

The `ValueFromPipeline` and `ValueFromPipelineByPropertyName` parameter attributes allow a function to indicate how specific parameters bind to items on the pipeline. `ValueFromPipeline` tells the engine to attempt to bind each pipeline item to a parameter as an object. The `ValueFromPipelineByPropertyName` attribute instructs the engine to examine each item on the pipeline and use the object properties whose names match the parameter name to populate the parameter. Unlike with `ValueFromPipeline`, more than one parameter can bind to a property name with the same pipeline object. This should be made clearer with the following example:

```
Function get-fileExtension{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true)]
      [System.IO.FileInfo]$file,
      [Parameter(ValueFromPipelineByPropertyName=$true)]
      [String]$extension)
Process{
      Write-Output "The filename was $($file.Name)"
      Write-Output "the extension was $($file.Extension)"
      Write-Output "the extension is also $extension"
}
}
```

In this example, the `$file` parameter is set to bind with the `FileInfo` objects in the pipeline. Likewise, the `$extension` parameter is set to bind with properties called `Extension` on objects in the pipeline. Note that folders are represented by the `System.IO.DirectoryInfo` class, so the `$file` parameter will not bind but since those objects have the `Extension` properties the `$extension` parameter will be populated. Here is an example of calling this function:



```
PS C:\temp> dir


    Directory: C:\temp


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         5/27/2014  12:54 PM            folder
d----         5/27/2014  12:53 PM            FolderWith.Extension
-a---          4/8/2014   7:33 PM      58272 cbh.txt
-a---         4/11/2014   5:04 PM          0 errors.txt
-a---         4/11/2014   8:40 PM    3328100 files.txt
-a---         3/11/2014   7:47 PM        115 get-powershellversionmessage.ps1
-a---         3/25/2014   8:10 PM          0 hello.txt
-a---         3/12/2014   7:37 PM      35376 keys.txt
-a---         3/11/2014   7:26 PM          0 test.txt
-a---         3/25/2014   8:23 PM          4 testout.txt


PS C:\temp> dir | select -first 3 | get-fileExtension
The filename was
the extension was
the extension is also
The filename was
the extension was
the extension is also .Extension
The filename was cbh.txt
the extension was .txt
the extension is also .txt
PS C:\temp>
```

The process block of a function is repeated for each object in the pipeline. However, if the user calls the function supplying the values for the parameters on the command line, we also need to make sure that our function will handle those seamlessly. Here is a useful pattern to handle pipeline and command-line input with the same parameter:

```
function get-value{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true)]
       [string[]]$computername)
begin {
  #initialize
}
process{
   Foreach ($computer in $computername){
  #process one value from the pipeline or commandline
    }
}
end {
  #finish up
}
}
```

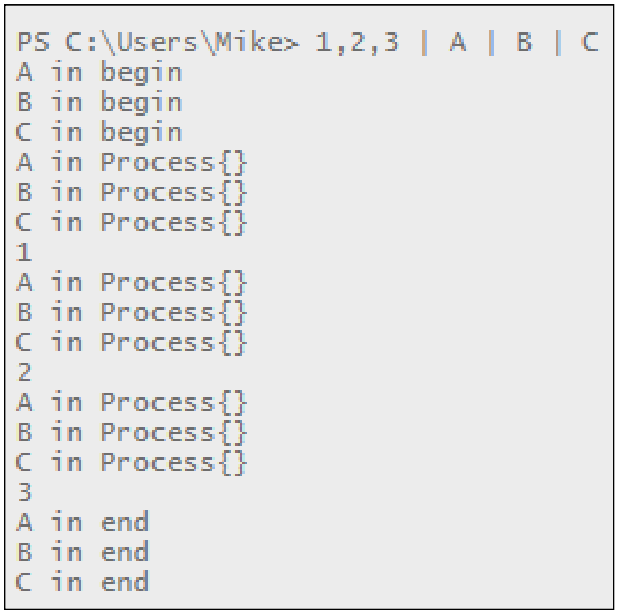
The new features of this function are making the parameter an array and adding a loop in the process block. In this code, the process block will be executed once for each pipeline item and the `foreach` loop in it will execute once per item. For command-line input, the `$computername` parameter will have all of the values supplied (rather than one at a time) and the loop in the process block will loop through them. With command-line input, the process block will only execute once.

Pipelines and function execution

Although pipelines are written sequentially with each element following the previous one, function execution in a pipeline is somewhat different. In order to illustrate this, consider the following advanced functions that all allow pipeline input:

```
function A{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true)]$x)
  begin { Write-Host "A in begin"}
  process { Write-Host "A in Process{}"
            Write-Output $x }
  end { Write-Host "A in end"}
}
function B{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true)]$y)
  begin { Write-Host "B in begin"}
  process { Write-Host "B in Process{}"
           Write-Output $y }
  end { Write-Host "B in end"}
}
function C{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true)]$z)
  begin { Write-Host "C in begin"}
  process { Write-Host "C in Process{}"
           Write-Output $z }
  end { Write-Host "C in end"}
}
```

The only thing these functions do is display a message when one of the three `Begin-Process-End` script blocks is run and output any objects that come in from the pipeline. With this in mind, examine the output from the command line `1,2,3 | A | B | C` in the following screenshot:



```
PS C:\Users\Mike> 1,2,3 | A | B | C
A in begin
B in begin
C in begin
A in Process{}
B in Process{}
C in Process{}
1
A in Process{}
B in Process{}
C in Process{}
2
A in Process{}
B in Process{}
C in Process{}
3
A in end
B in end
C in end
```

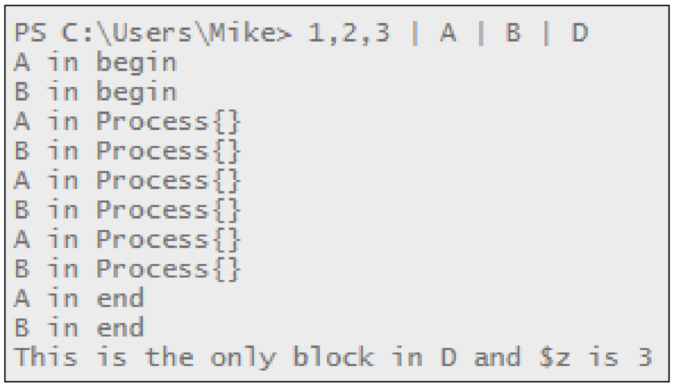
The sequence of execution should be clear:

1. The `Begin` block of each function in the pipeline is executed in sequence.
2. The `Process` block of the first function is passed the first item in the pipeline.
3. Since the first `Process` block outputs an object, it is passed to the second `Process` block, and so on.
4. After all of the pipeline objects have been processed, the `End` blocks are all called in sequence.

If we don't use `Begin-Process-End` blocks in a function, only the `End` block is called. In this case, the last value from the pipeline is still assigned to the parameter. To see this, have a look at the following simple function:

```
function D{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true)]$z)
  Write-Host "This is the only block in D and `$z is $z"
}
```

Now, observe the output in the following screenshot from `1,2,3 | A | B | D` and compare it to the previous output:



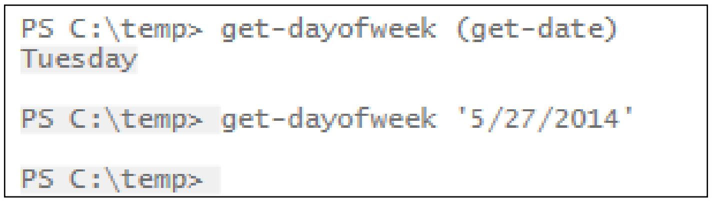
```
PS C:\Users\Mike> 1,2,3 | A | B | D
A in begin
B in begin
A in Process{}
B in Process{}
A in Process{}
B in Process{}
A in Process{}
B in Process{}
A in end
B in end
This is the only block in D and $z is 3
```

Parameter type transformation

PowerShell allows us to ignore the idea of variable types in most situations and this is a tremendous productivity boost. When you consider all of the different Common Language Runtime (CLR) types that are used in a typical script it's easy to see why not worrying about naming the types saves a lot of time. Adding in all of the anonymous types (for instance, results of a `select-object` call), the need for a very liberal typing system is obvious. One example when specifying types is useful, or even critical, is when specifying parameters to a function:

```
function get-dayofweek{
param($date)
  Write-Output $date.DayOfWeek
}
```

This function seems like it would work well, but testing it shows that it's not quite right, as shown in the following screenshot:

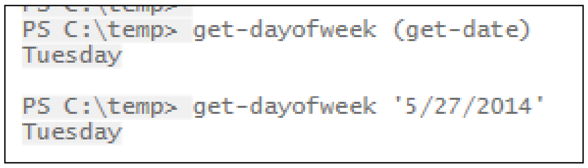


```
PS C:\temp> get-dayofweek (get-date)
Tuesday

PS C:\temp> get-dayofweek '5/27/2014'

PS C:\temp>
```

Since we didn't specify what type the parameter was, the problem was that the constant string `'5/27/2014'` was passed into the parameter as is, that is, as a string. Since the string didn't have a `DayOfWeek` property, the output is `$null`. The solution, of course, is to include the type of the parameter as part of the definition, as shown in the following script:

```
function get-dayofweek{
param([datetime]$date)
  Write-Output $date.DayOfWeek
}
```

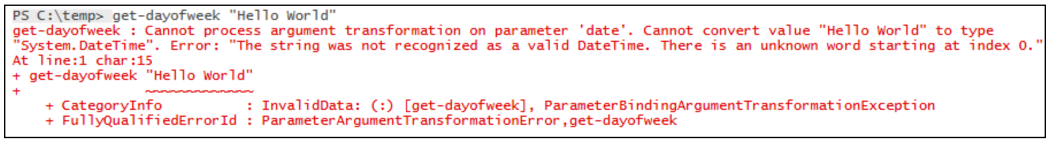
Now the function works as expected:



```
PS C:\temp> get-dayofweek (get-date)
Tuesday

PS C:\temp> get-dayofweek '5/27/2014'
Tuesday
```

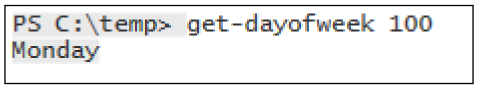
It's important to include the type of your parameter because the PowerShell engine does a remarkable job to not only make sure that the type of object that is passed is the correct type, but it also tries to coerce the value into the correct type. This is how the `'5/27/2014'` string was changed into a `DateTime` object. PowerShell uses several different methods to try to convert the supplied value to the requested type. In this instance, it found a `Parse()` static method on the `DateTime` type with a single string parameter and passed `'5/27/2014'` to it to create the value that was accepted by the function.

Passing a value that does not correspond to a legitimate object of the correct type will result in an error as the engine attempts to bind the value to the parameter:



```
PS C:\temp> get-dayofweek "Hello World"
get-dayofweek : Cannot process argument transformation on parameter 'date'. Cannot convert value "Hello World" to type
"System.DateTime". Error: "The string was not recognized as a valid DateTime. There is an unknown word starting at index 0."
At line:1 char:15
+ get-dayofweek "Hello World"
+               ~~~~~~~~~~~~~
    + CategoryInfo          : InvalidData: (:) [get-dayofweek], ParameterBindingArgumentTransformationException
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,get-dayofweek
```

PowerShell's method of automatic type conversion is almost always what is expected. On the other hand, some types have constructors, such as shown in the following screenshot, which we might not have considered:

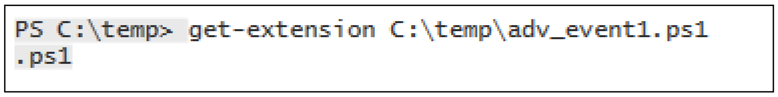


```
PS C:\temp> get-dayofweek 100
Monday
```

While this doesn't seem to make sense, PowerShell pushed through and gave an answer. It turns out that the `DateTime` type has a constructor that takes a single integer parameter corresponding to the number of ticks (since the minimum `DateTime` value). It's probably not what we wanted, but it's the price we pay for the 99 percent of the time where PowerShell is silently converting values (for example, filename strings to `FileInfo` objects):

```
function get-extension{
param([System.io.fileinfo]$file)
  Write-Output $file.extension
}
```

Because the text value on the command-line was converted to a FileInfo object, we are able to refer to its extension property:



```
PS C:\temp> get-extension C:\temp\adv_event1.ps1
.ps1
```

The point to remember is that we can control what objects are accepted for parameters by specifying the type of the parameter. We can't control what conversions PowerShell will attempt to use to give us a correctly-typed object. We might get an unexpected value if PowerShell uses an unusual transformation, but we won't have to validate the type of the object we receive.

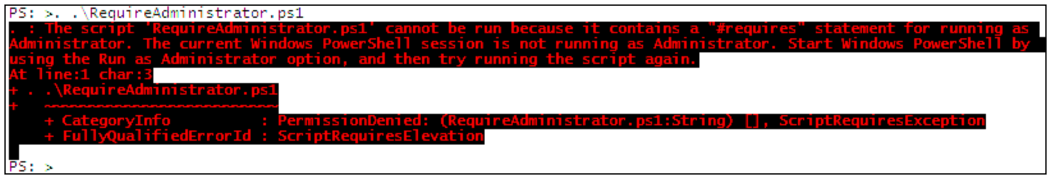
#REQUIRES statements

It's no secret that some code has prerequisites that need to be met in order for it to work. In PowerShell, certain types of requirements can be specified in scripts using a `#REQUIRES` statement. Although the `#REQUIRES` statement looks like a comment (that is, it starts with `#`), it is a statement to the engine that tells PowerShell not to run the script unless the requirements are met. As of PowerShell Version 4.0, the following options are available in a `#REQUIRES` statement:

Option	**Parameters**	**Notes**
`-Version`	N[.N] (for example, 4.0)	Required version of PowerShell engine
`-PSSnapIn`	PSSnapinName [-Version N[.N]]	Required Snapin (with optional minimum version #)
`-Modules`	ModuleName[,ModuleName] or Hashtable	Required modules to be loaded
`-ShellID`	ShellID	Required PowerShell Host (for example, Microsoft. PowerShellISE)
`-RunAsAdministrator`	none	Required session privileges

When the conditions of the `#REQUIRES` statement are not met, or cannot be met, the script will not load and will emit an error. With the following code in `RequireAdministrator.ps1` and a PowerShell session not running as administrator, observe the error message that follows:

```
#Requires -RunAsAdministrator

Write-Host "hello world"
```

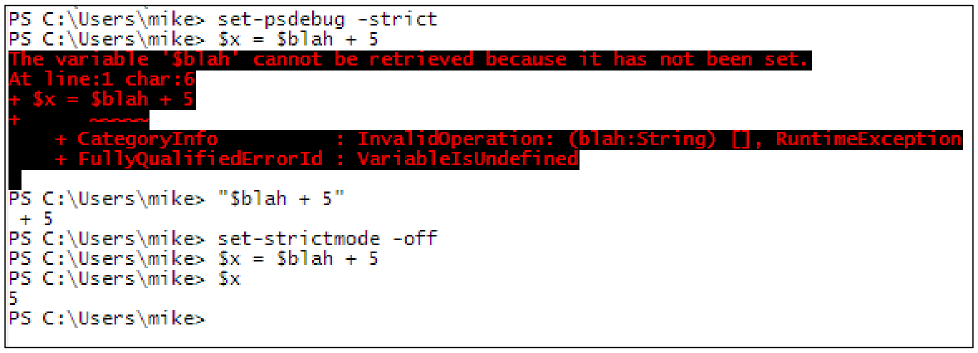


```
PS: >. .\RequireAdministrator.ps1
. : The script 'RequireAdministrator.ps1' cannot be run because it contains a "#requires" statement for running as
Administrator. The current Windows PowerShell session is not running as Administrator. Start Windows PowerShell by
using the Run as Administrator option, and then try running the script again.
At line:1 char:3
+ . .\RequireAdministrator.ps1
+   ~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : PermissionDenied: (RequireAdministrator.ps1:String) [], ScriptRequiresException
    + FullyQualifiedErrorId : ScriptRequiresElevation

PS: >
```

Set-StrictMode and Set-PSDebug -strict

In addition to PowerShell not requiring a script to specify the type of a variable, it also doesn't require any kind of declaration prior to using the variable. If the first use of a variable is to assign a value to it, this relaxed attitude doesn't cause any harm. On the other hand, reading a variable that hasn't been written to is generally not what is intended.

There are two ways to ensure that reading from an uninitialized variable will cause an error. The first, introduced in PowerShell Version 1.0, was to use the `-strict` switch on the `Set-PSDebug` cmdlet. Once this has been issued in a PowerShell session, references to uninitialized variables (except in string substitution) will produce an error. References inside strings will resolve to `$null`. This is a global switch in the engine and is reversed by issuing `Set-PSDebug` with the `-off` switch. The following screenshot explains this:



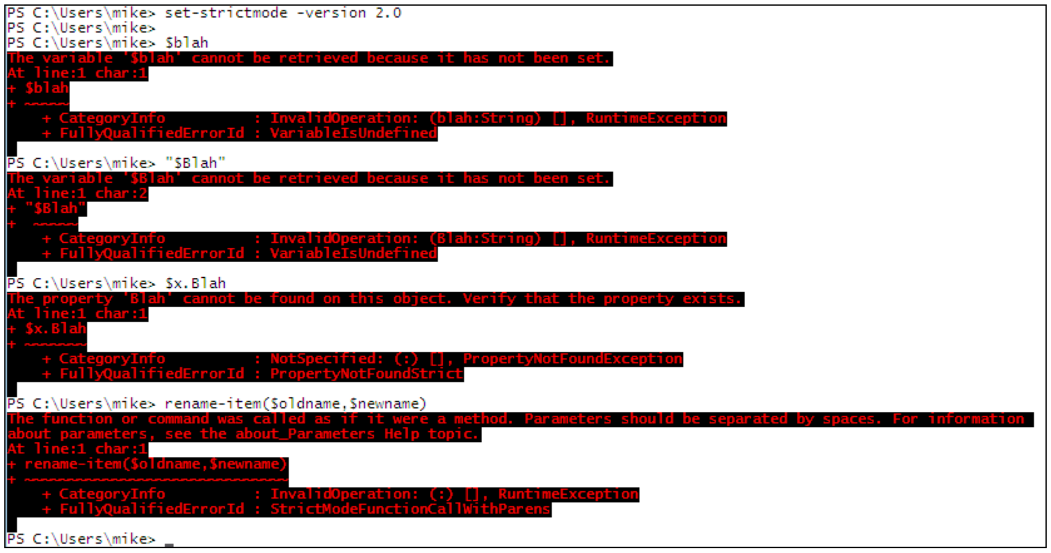
```
PS C:\Users\mike> set-psdebug -strict
PS C:\Users\mike> $x = $blah + 5
The variable '$blah' cannot be retrieved because it has not been set.
At line:1 char:6
+ $x = $blah + 5
+      ~~~~~
    + CategoryInfo          : InvalidOperation: (blah:String) [], RuntimeException
    + FullyQualifiedErrorId : VariableIsUndefined

PS C:\Users\mike> "$blah + 5"
 + 5
PS C:\Users\mike> set-strictmode -off
PS C:\Users\mike> $x = $blah + 5
PS C:\Users\mike> $x
5
PS C:\Users\mike>
```

The second method, introduced in PowerShell Version 2.0, is to use `Set-StrictMode` and use the `-Version` parameter to specify the level of strictness. `Set-Strictmode -Version 1.0` gives the same results as `Set-PSDebug -strict`. `Set-StrictMode -Version 2.0` and causes the following conditions to result in an error:

- Referencing uninitialized variables in strings
- Referencing nonexistent properties of an object

- Calling a cmdlet using parentheses (as if it were an object method)
- Referencing a variable with no name (`${}`)

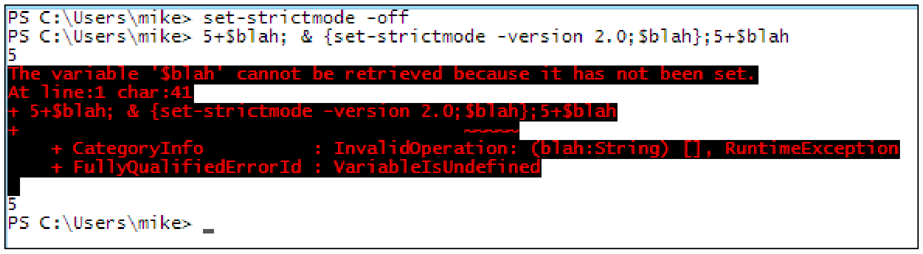
The following screenshot illustrates the strict-mode errors:



```
PS C:\Users\mike> set-strictmode -version 2.0
PS C:\Users\mike>
PS C:\Users\mike> $blah
The variable '$blah' cannot be retrieved because it has not been set.
At line:1 char:1
+ $blah
+ ~~~~~
    + CategoryInfo          : InvalidOperation: (blah:String) [], RuntimeException
    + FullyQualifiedErrorId : VariableIsUndefined

PS C:\Users\mike> "$Blah"
The variable '$Blah' cannot be retrieved because it has not been set.
At line:1 char:2
+ "$Blah"
+  ~~~~~
    + CategoryInfo          : InvalidOperation: (Blah:String) [], RuntimeException
    + FullyQualifiedErrorId : VariableIsUndefined

PS C:\Users\mike> $x.Blah
The property 'Blah' cannot be found on this object. Verify that the property exists.
At line:1 char:1
+ $x.Blah
+ ~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], PropertyNotFoundException
    + FullyQualifiedErrorId : PropertyNotFoundStrict

PS C:\Users\mike> rename-item($oldname,$newname)
The function or command was called as if it were a method. Parameters should be separated by spaces. For information
about parameters, see the about_Parameters Help topic.
At line:1 char:1
+ rename-item($oldname,$newname)
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : StrictModeFunctionCallWithParens

PS C:\Users\mike> _
```

As of PowerShell 4.0, `-Version 2.0` is the most restrictive setting available. To ensure that code uses all possible restrictions for future versions as well, the value of `Latest` is available, which currently gives the same results as `-Version 2.0`.

One difference between `Set-PSDebug` and `Set-StrictMode` is that while `Set-PSDebug` is a session-level setting, `Set-StrictMode` is scoped, that is, the setting is changed in the current scope and its children. For this reason it can be used to guard against errors in a function, module, or script without concern that it will place restrictions on the session in general and is preferred over `Set-PSDebug` unless working in PowerShell Version 1.0.



```
PS C:\Users\mike> set-strictmode -off
PS C:\Users\mike> 5+$blah; & {set-strictmode -version 2.0;$blah};5+$blah
5
The variable '$blah' cannot be retrieved because it has not been set.
At line:1 char:41
+ 5+$blah; & {set-strictmode -version 2.0;$blah};5+$blah
+                                         ~~~~~
    + CategoryInfo          : InvalidOperation: (blah:String) [], RuntimeException
    + FullyQualifiedErrorId : VariableIsUndefined

5
PS C:\Users\mike> _
```

`Set-Strictmode` is an important tool in our toolbox because the restrictions it makes are generally encountered in erroneous code. Referencing uninitialized variables usually happens when a variable is misspelled and similarly, with nonexistent property names. Calling functions or cmdlets using method syntax (parentheses and commas) is usually a mistake as it packages up the arguments as an array and passes that array to the first parameter, which is rarely intended.

Further reading

You can go to the following references for more information:

- `get-help about_try_catch_finally`
- `get-help about_functions_cmdletbinding_attribute`
- `get-help about_commonparameters`
- `get-help about_functions_advanced_methods`
- `get-help about_functions_advanced_parameters`
- `get-help about_functions`
- `get-help about_pipelines`
- Parameter Type Transformation at `http://blogs.msdn.com/b/powershell/archive/2013/06/11/understanding-powershell-s-type-conversion-magic.aspx`
- `get-help about_requires`
- `get-help set-strictmode`
- `get-help set-psdebug`

Summary

This chapter has explored some really important PowerShell features in depth. We spent a lot of time talking about advanced functions, parameters, and the pipeline. The important `Set-StrictMode` cmdlet was introduced to show how to restrict the PowerShell language slightly in ways that will help us script more carefully. Hopefully, by employing some of these practices, you will be to write more powerful, flexible scripts and avoid some common errors.

In the next chapter, we will turn our thoughts to the environment that scripts run in. We will look at several ways to determine the characteristics of the environment in order to eliminate errors that lie outside of our scripts.

6
Preparing the Scripting Environment

Writing scripts carefully and leveraging the PowerShell language is important, but deploying those scripts into an uncertain environment can cause any number of headaches. Validating that the scripting environment is configured as expected will eliminate many of the potential errors that might otherwise occur. Several methods to check the existing configuration will be presented in this chapter. Many of these techniques are also useful for routine validation after maintenance (patching, upgrading, and so on). In this chapter, we will cover the following topics:

- Validating the operating system (OS) version and 32/64-bit
- Validating the service status
- Validating disk and memory availability
- Validating network connectivity

Validating operating system properties

Details about the installed operating system on a computer can have a tremendous impact on the operation of a script. In the following sections, we will examine the `Win32_OperatingSystem` class and build a function that provides us with the data we need to support our scripts. The class has over 60 properties, but a handful of them are all we will need. We will use the **Common Information Model** (**CIM**) cmdlets to retrieve the data, though using the **Windows Management Instrumentation** (**WMI**) cmdlets will work as well. To start off, let's retrieve the (only) instance of the class using the following script:

```
$os=get-CIMInstance –class Win32_OperatingSystem
```

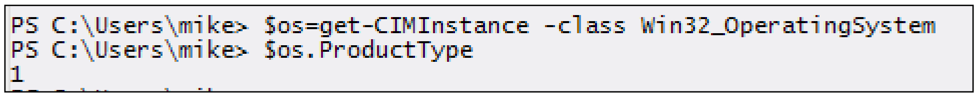
There is a lot of confusion between WMI and CIM. The CIM cmdlets introduced in PowerShell Version 3.0 seem to provide the same functionality as the WMI cmdlets that have always been present. The main difference is that the CIM cmdlets use **Web Services Management** (**WSMAN**) by default instead of using DCOM like the WMI cmdlets. Also, CIM cmdlets allow you to use a session for multiple requests, which reduces the overhead. If your environment doesn't allow CIM cmdlets to be used, you should be able to use the corresponding WMI cmdlets.

Workstation/server version

The designation of the installed operating system as a workstation or server (or domain controller) is given by the `ProductType` property, which takes one of the following values:

Value	Meaning
1	Workstation
2	Domain controller
3	Server

Here, we see a computer that is running a workstation version of Windows:



```
PS C:\Users\mike> $os=get-CIMInstance -class Win32_OperatingSystem
PS C:\Users\mike> $os.ProductType
1
```

If we store these values in an array with a dummy entry, the indices will correspond with the product values:

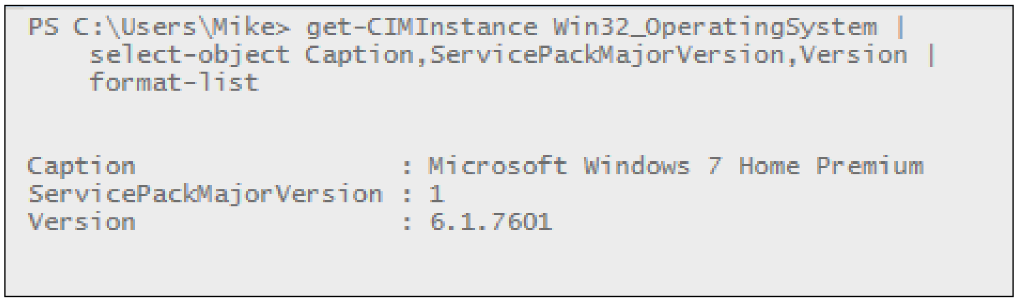
```
$ProductTypes="Unused",
               "Workstation",
               "Domain Controller",
               "Server"
```

Operating system version

We can get other information from `Win32_OperatingSystem` about the specific version of the operating system that we have installed. The properties we will use are:

- `Caption`
- `ServicePackMajorVersion`
- `Version`

In the following screenshot, you can see the values my laptop shows for these properties:

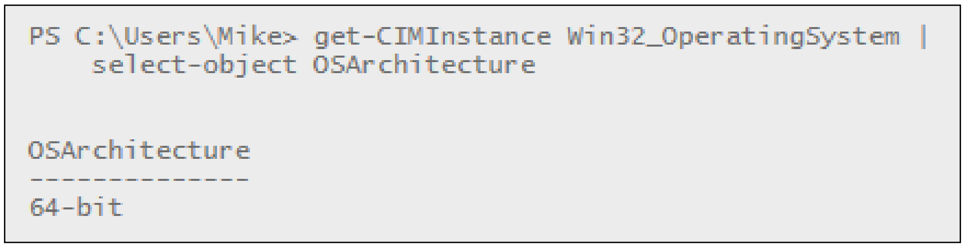


```
PS C:\Users\Mike> get-CIMInstance Win32_OperatingSystem |
    select-object Caption,ServicePackMajorVersion,Version |
    format-list


Caption                 : Microsoft Windows 7 Home Premium
ServicePackMajorVersion : 1
Version                 : 6.1.7601
```

The `Caption` property is clearly what a user will be familiar seeing, but it is going to be difficult to use that in a script. The `Version` property will be easier to use in a script, but the values aren't necessarily what would be expected by us. For instance, I would have expected Windows 7 to be Version 7. Here is a table of the values that are returned in this property and the corresponding operating system versions:

Version number	**Operating system**
5.1	Windows XP
5.2	Windows Server 2003
5.2.3	Windows Server 2003 R2
6.0	Windows Vista or Windows Server 2008
6.1	Windows 7 or Windows Server 2008 R2
6.2	Windows 8 or Windows Server 2012
6.3	Windows 8.1 or Windows Server 2012 R2

For versions 6.x we can use the `ProductType` to determine whether the OS is the desktop (workstation) or Server edition. One last piece of information that will come in handy is knowing whether the OS installation is a 32-bit or 64-bit installation. We can start with the `OSArchitecture` property, which shows my computer to be 64-bit, as shown in the following screenshot:



```
PS C:\Users\Mike> get-CIMInstance Win32_OperatingSystem |
    select-object OSArchitecture


OSArchitecture
--------------
64-bit
```

The `OSArchitecture` property does not exist in the WMI class definition delivered on Windows XP or Windows Server 2003, which exist in both 32-bit and 64-bit versions. There are other indicators of what architecture is present in hardware, like the `SystemType` property of the `Win32_ComputerSystem` class or the `Architecture` property of the `Win32_Processor` class, but those refer to the hardware rather than the installed operating system. Rather than digging through all of the possible WMI classes, a simpler approach is to simply look for the existence of a `ProgramFiles(x86)` environment variable. All 64-bit installations will include this environment variable, but 32-bit systems won't have it.

This gives us the following function to retrieve the architecture of the OS, which will work for all versions:

```
function get-OSArchitecture{
    if (test-path "env:programfiles(x86)"){
        "64-bit"
    } else {
        "32-bit"
    }
}
```

This code only works on the current machine because it uses ENV: drive. A more general solution involves accessing environment settings through the `Win32_Environment` class. Unfortunately, the `ProgramFiles(x86)` environment variable isn't available in that class as a property. Fortunately, there are some other environment variables we can use (in combination) if the `OSArchitecture` property is not present. We can use the following function to retrieve environment variables from remote machines:

```
Function Get-EnvironmentVariable{
Param(
[string]$name='%',
[string]$username='<SYSTEM>',
[parameter(ValueFromPipeline=$true,
  ValueFromPipelineByPropertyName=$true)]
[string[]]$computername='localhost')

    Process{
      $values=Get-WMIObject -class
        Win32_Environment -computerName $computerName
        -filter "Name like '$name' and UserName='$username'"
        if($name.Contains('%')){
            $values
        } else {
            $values | select-object -ExpandProperty VariableValue
        }
    }
}
```

I've written the function so that it will return the value of a single variable or the list of `Win32_Environment` objects if the name parameter contains a wildcard. This makes it simple to retrieve a single value if that's what is needed.

With the `Get-EnvironmentVariable` function, we can write the following script:

```
Function get-legacyOSArchitecture{
Param($computername)
    If((get-EnvironmentVariable -name 'PROCESSOR_ARCHITECTURE' -
computername $computername) -eq 'AMD64'){
    '64-bit'
    } else {
    If((get-EnvironmentVariable -name 'PROCESSOR_ARCHITEW6432'
      -computername $computername) -eq 'AMD64'){
         '64=bit'
        } else {
                '32-bit'
            }
        }
    }
```

If you don't have a Windows 2003 or XP system, I would recommend using the `OSArchitecture` property on `Win32_OperatingSystem` since there are several other properties that we will want to look at on that class. Since both of these operating systems are no longer supported, the need for a workaround should be short-lived.

Putting it all together

We can combine all of the research in the following sections into a flexible script which retrieves all of the operating system details that we're interested in. To start, our input will be a list of computer names. We should allow the parameter to be provided on the command line or from the pipeline (by value or from a property).

Troubleshooting tip

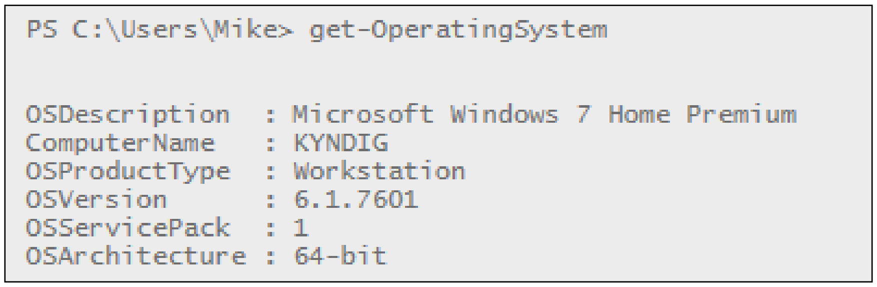
When including a `ComputerName` parameter, remember to allow the aliases of `CN` and `MachineName`. When populating the parameter from the pipeline by property name, this will enable input from Active Directory cmdlets as well as other sources.

The function looks like this:

```
function get-OperatingSystem{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true,
  ValueFromPipelineByPropertyName=$true)]
      [Alias('CN','MachineName')]
      [string[]]$computerName=$env:ComputerName)
begin {
$ProductTypes='Unused',
              'Workstation',
              'Domain Controller',
              'Server'
}
process {
  foreach($computer in $computerName){
    $output=@{ComputerName=$computer}
```

```
        $os=get-CIMInstance -computerName $computer
          -class Win32_OperatingSystem
        $output.OSDescription=$os.Caption
        $output.OSVersion=$os.Version
        $output.OSServicePack=$os.ServicePackMajorVersion
        $output.OSProductType=$ProductTypes[$os.ProductType]
        If($os | get-member OSArchitecture){
        $output.OSArchitecture=$os.OSArchitecture
              } else {
        $output.OSArchitecture=get-legacyOSArchitecture $computer
        }
        new-object PSObject -property $output
      }
    }
    }
```

As usual, we output objects from our function. Running this on my laptop yields the following results:



```
PS C:\Users\Mike> get-OperatingSystem


OSDescription   : Microsoft Windows 7 Home Premium
ComputerName    : KYNDIG
OSProductType   : Workstation
OSVersion       : 6.1.7601
OSServicePack   : 1
OSArchitecture  : 64-bit
```

Validating service status

One of the first things we learn to do with PowerShell is to inspect the services on a computer. The `Get-Service` cmdlet with its `-ComputerName` parameter makes this a simple task. For instance, if we had a list of computers that have SQL Server installed in a variable called `$servers`, we could issue the following script to get the status of the service running the default instance like this:

```
Get-service -name MSSQLSERVER -computername $server |
  Select-object -property Name,Status,MachineName
```

If our goal was simply to find out whether the service is running, this will do the trick. One piece of information that is missing from the objects output from the `Get-Service` cmdlet is the name of the account that is used to run the service, the run as account. To find that detail, we must turn to WMI. The class to use is `Win32_Service`, which contains the `StartName` property. The value of the `StartName` property is the name of the user account running the service. To find the run as user, as well as the service state for the list of computers in `$servers`, we could use the following command:

```
Get-WMIObject -class Win32_Service -filter "Name='mssqlserver'"
  -ComputerName $computer | select-object -Property
  Name,StartName,State
```

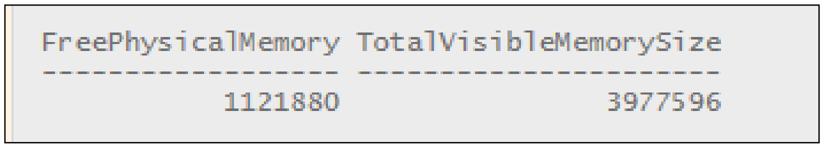
Validating disk and memory availability

Trying to run a script on a system that is out of memory or disk space is a frustrating experience. An important piece of preparation is to determine the amount of memory and disk space present as well as how much of each is unused. Retrieving these statistics with WMI is a simple matter, and with a bit of effort we can make the results more user-friendly.

We already encountered the `Win32_OperatingSystem` class in the first part of this chapter. Fortunately for us, there are two properties on this class that will tell us both the total amount and the free amount of memory.

```
Get-CIMInstance Win32_OperatingSystem |
  Select-object FreePhysicalMemory,TotalVisibleMemorySize
```

The output on my laptop is this:



```
FreePhysicalMemory TotalVisibleMemorySize
------------------ ----------------------
           1121880                3977596
```

Since we're going to probably want to see these in a different unit than kilobytes, we can use a trick recommended by Jeffery Hicks (Microsoft MVP in PowerShell) to add script properties to the class returned by `get-CIMInstance`:

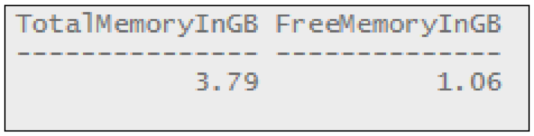
```
Update-TypeData -TypeName Microsoft.Management.Infrastructure.
CimInstance#root/cimv2/Win32_OperatingSystem `
    -MemberType ScriptProperty -MemberName TotalMemoryInGB
      -Value {[math]::Round($this.TotalVisibleMemorySize/1MB,2)}
```

```
Update-TypeData -TypeName Microsoft.Management.Infrastructure.
CimInstance#root/cimv2/Win32_OperatingSystem `
    -MemberType ScriptProperty -MemberName FreeMemoryInGB
      -Value {[math]::Round($this.FreePhysicalMemory/1MB,2)}
```

The `Update-TypeData` cmdlet uses the extended type system in PowerShell to add metadata to the class definition. The actual .NET framework class involved is `Microsoft.Management.Infrastructure.CimInstance`, but PowerShell allows us to differentiate between different types of `CIMInstance` objects by appending the full WMI path to the type name, in this case, `#root/cimv2/Win32_OperatingSystem`. This is a very useful thing to do since the data in different types of WMI instances are very different. The results of the `Update-TypeData` cmdlet only affect the current session, but they do make the display a lot nicer. In the following script, I'm using a wildcard to select these properties from the object:

```
PS C:\> Get-CIMInstance win32_operatingsystem |select *inGB
```

The following screenshot shows the result:



```
TotalMemoryInGB FreeMemoryInGB
--------------- --------------
           3.79           1.06
```

A general purpose function to get memory statistics for a set of servers is as follows:

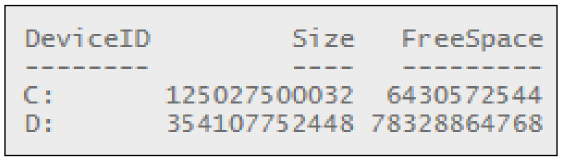
```
function get-ComputerMemory{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true,
  ValueFromPipelineByPropertyName=$true)]
      [Alias('CN','MachineName')]
      [string[]]$computerName=$env:ComputerName)
    process {
        Get-CIMInstance Win32_OperatingSystem
          -ComputerName $computerName |
            select PSComputerName,@{N='TotalMemoryInGB';
  E={[math]::Round($_.TotalVisibleMemorySize/1MB,2)}},
          @{N='FreeMemoryInGB';
    E={[math]::Round($_.FreePhysicalMemory/1MB,2)}}
    }
}
```

Here, I added the property conversions to gigabytes using expressions rather than adjusting the type data.

To find the total disk space and free disk space we need to use the `Win32_LogicalDisk` class. Fixed disks have `DriveType` of 3. The following command line is used to find the disk space:

```
PS C:\ > get-CimInstance win32_logicaldisk -filter 'DriveType=3' |
  select DeviceID,Size,FreeSpace
```

Looking at the disks on my laptop shows the following results:

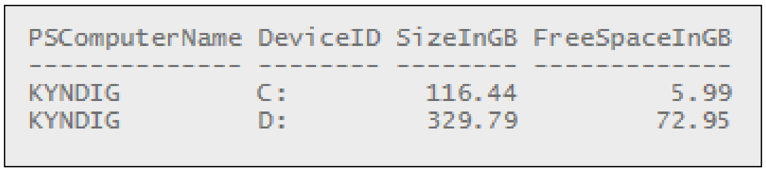


```
DeviceID         Size   FreeSpace
--------         ----   ---------
C:       125027500032  6430572544
D:       354107752448 78328864768
```

A function to find this information for a group of computers is similar to the function to find memory statistics, as follows:

```
function get-DiskSpace{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true,ValueFromPipelineByPropertyN
ame=$true)]
      [Alias('CN','MachineName')]
      [string[]]$computerName=$env:ComputerName)
    process {
        get-CIMInstance Win32_LogicalDisk -filter 'DriveType=3'
-ComputerName $computerName |
            select PSComputerName,DeviceID,@{N='SizeInGB';
              E={[math]::Round($_.Size/1GB,2)}},
                @{N='FreeSpaceInGB';E={[math]::
                  Round($_.Freespace/1GB,2)}}
    }
}
```

The results of this function will look like this:



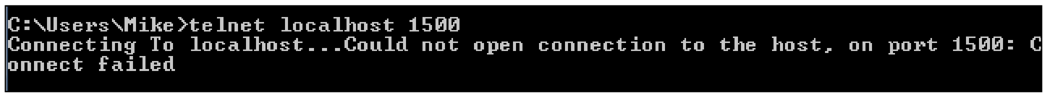
```
PSComputerName DeviceID SizeInGB FreeSpaceInGB
-------------- -------- -------- -------------
KYNDIG         C:         116.44          5.99
KYNDIG         D:         329.79         72.95
```

Validating network connectivity

In a perfect world, every computer system would be able to connect to any other computer system that it needed to. In the real world, there are often complications that arise due to firewalls and IPSec rules. Making sure that all of the needed network connectivity is in place will help to eliminate a common source of script failure.

Using telnet

You may be familiar with using telnet to test TCP connections. To do this, you need to have telnet installed (it's not installed by default on recent server editions). Once you have it installed, you simply run `telnet` hostname port, where you replace the hostname and port with the appropriate values for your test. To see whether the box you're on can connect to DBSERVER01 on port 1433 (the default SQL Server port), run `telnet DBSERVER01 1433`. Since the port being tested is not necessarily a telnet server, the output isn't always clear, but in general, when a connection is successful, the screen is cleared. A failed connection will give an error message, as shown in the following screenshot:



```
C:\Users\Mike>telnet localhost 1500
Connecting To localhost...Could not open connection to the host, on port 1500: C
onnect failed
```

For one-off checking, telnet is pretty convenient. The main issue is that you'll need to have the telnet client installed on every machine you'll be testing. Since you probably don't need telnet otherwise, this can be a problem. This solution involves logging in to each machine and testing each port individually, so it doesn't work well for larger applications.

Before proceeding further, it is probably worth mentioning that there are products designed primarily to test network connectivity, such as nmap or netcat. If you have one of these at your disposal, or if you can get it approved by your IT team, these will save you some time and effort. If you would like to implement something of this nature using PowerShell, however, this section is for you.

Using Test-NetConnection

An alternative to telnet was introduced in Windows 8.1 and Server 2012 R2 with the `Test-NetConnection` cmdlet. On these versions of the operating system, the `Test-NetConnection` cmdlet can be used to test TCP communication on a port similar to how the telnet client does. The earlier test would be done with `Test-Netconnection`, as shown in the following script:

```
Test-NetConnection -computerName DBSERVER01 -port 1433
```

This will return a detailed report about the availability of the connection. To get a true/false result, append `–InformationLevel Quiet`. This is another convenient solution, but it does require a specific operating system to be useful. Also, this requires logging in to each machine in the environment, which might not be very practical.

Writing Test-NetConnection in downstream versions

Since we can't depend on `Test-NetConnection` being available everywhere, we can write our own using .Net classes. The crucial class we will use is the `[System.Net.Sockets.TCPClient]` class. Here is a somewhat flexible implementation:

```
function test-TCPConnection{
[CmdletBinding()]
Param([Parameter(ValueFromPipelineByPropertyName=$true)]
      [string]$computerName,
      [Parameter(ValueFromPipelineByPropertyName=$true)]
      [int]$port,
      [switch]$quiet)
process{
      try {
          $client=New-Object System.Net.Sockets.TCPClient
          $client.Connect($computerName,$port)
      $result=$true
      } catch {
        $result=$false
      } finally {
        $client.Close()
      }
      if($quiet){
  $result
      } else {
        New-Object PSObject -property @{
ComputerName=$ComputerName;
      Port=$port;
      Connected=$result}
      }
}
}
```

There are several ways to use this function. The first and most straightforward way is to simply test a single connection via the `ComputerName` and `port` parameters, as shown in the following script:

```
Test-TCPConnection -computername DBServer01 -port 1433
```

This will return an object with the computer name, port, and the protocol (TCP) as well as a Boolean property called `connected` that indicates the success of the connection. If you just need a Boolean value for a test, like in an `if` statement, you can supply the `–quiet` switch:

```
Test-TCPConnection -computername DBServer01 -port 1433 -quiet
```

By including the `ValueFromPipelineByPropertyName` parameter attribute on both parameters, we can also easily consume input from a **comma-separated value** (**CSV**) file. For instance, if we have a CSV file called `Network.csv` with the following contents:

```
ComputerName,Port
DBServer01,1433
SMTPServer01,25
FTPServer01,22
```

We can then execute the following script:

```
Import-csv Networkcsv | test-TCPConnection
```

This will output a list of objects reporting the details of our connectivity. Note that we did not include the `ValueFromPipeline` attribute for either of the parameters because only one parameter at a time can receive pipeline input from an object, rather than from a property, and we really need the computer name / port pair for the parameter to make sense.

Testing UDP and ICMP connectivity

Network connectivity is not limited to TCP though. An almost identical function can be written to test for UDP connectivity, as follows:

```
function test-UDPConnection{
[CmdletBinding()]
Param([Parameter(ValueFromPipelineByPropertyName=$true)]
      [string]$computerName,
      [Parameter(ValueFromPipelineByPropertyName=$true)]
      [int]$port,
      [switch]$quiet)
process{
      try {
          $client=New-Object System.Net.Sockets.UDPClient
         $client.Connect($computerName,$port)
         $result=$true
      } catch {
      $result=$false
      } finally {
      $client.Close()
      }
      if($quiet){
      $result
      } else {
        new-object PSObject -property @{ComputerName=$ComputerName;
          Port=$port;
          Connected=$result}
      }
}
}
```

A final kind of connectivity is ICMP or ping. There is a built-in cmdlet called `Test-Connection` introduced in PowerShell Version 2.0. We can write a wrapper function so the signatures of our connectivity-testing functions will match. Note that ICMP is not port-based, so the port parameter will be ignored. The following code snippet explains this:

```
function test-ICMPConnection{
[CmdletBinding()]
Param([Parameter(ValueFromPipelineByPropertyName=$true)]
      [string]$computerName,
      [Parameter(ValueFromPipelineByPropertyName=$true)]
      [switch]$quiet)
process{
      try {
      $result=test-connection -computerName $computerName -quiet
      } catch {
      $result=$false
      }
      if($quiet){
      $result
      } else {
        new-object PSObject -property @{ComputerName=$ComputerName;
          Port=-1;   #ICMP doesn't use a port
          Protocol='ICMP';
          Connected=$result}
      }
}
}
```

Having function signatures that match will be helpful when we need to be able to test different connectivity types with the same code in a later section.

Validating connectivity prior to implementation

The previous functions work fine if the software being connected to has already been implemented. But how can you check whether a computer can connect to an SQL Server, for example, if SQL Server hasn't been installed yet? To handle this scenario, we can write a script that listens on the appropriate port and run that on the server we're trying to connect to. Since we don't want to accidentally leave a process listening on a port that a piece of software will eventually try to use, we can build in a timeout so that the listener will stop after a preset time interval. The following code snippet demonstrates this:

```
function Start-TCPListener{
param([System.Net.IPAddress]$IPAddress=[System.Net.IPAddress]
  '0.0.0.0',[int]$port,[int]$timeout=10)
    try {

        $listener = new-object
          System.Net.Sockets.TcpListener($IPAddress, $port)
        $listeningTimeout=(get-date).AddSeconds($timeout)
        $listener.Start()
        while ((get-date) -lt $listeningTimeout
          -and !$listener.Pending() ){
```

```
                    start-sleep -Milliseconds 50
            }
            if ($listener.Pending()){
              $client = $listener.AcceptTcpClient()
              Write-Output "Connected from
                $($client.Client.RemoteEndPoint)"
            } else {
              Write-Output "listener timed out after $timeout seconds"
            }
            $listener.Stop()
        } catch {
          write-Error "unable to start listener : $($_.Message)"
        }
    }
```

Here, we're not interested in pipeline input because we're only going to start a single listener at a time that will block until it receives a request or the time-out is reached. By doing this, we can simulate the connectivity of the software that is going to be installed. To use this function in conjunction with the `test-TCPConnection` function from the previous section, follow these steps:

1. Open a PowerShell session on the listening computer
2. Open a PowerShell session on the calling computer
3. Run the `Start-TCPListener` function on the listening computer
4. Run the `Test-TCPConnection` function on the calling computer

With PowerShell remoting, you can open the sessions in remote PowerShell tabs in ISE and you do not have to use remote desktop to make this work.

Putting it all together

The previous sections showed how to check ports in a one-at-a-time way, and to check a number of remote connections from a single box as well as starting a TCP listener in case a piece of software hasn't been installed yet. Unfortunately, in anything but a simple system, the process of checking every kind of connectivity between every computer is going to be a very time-consuming process. Also, due to the number of combinations, it is going to be fraught with human error. To overcome these challenges, we can turn to PowerShell remoting and automate much of the process.

An assumption we're going to make is that you have a machine that can reliably connect to all of the computers in the environment using PowerShell remoting. With that in place, let's expand the input file we used for our `TCPListener` function to include a source computer, a protocol, and a flag to say whether we need to start a listener. Our sample input file might look like this:

```
Source,Destination,Port,Protocol,StartListener
Web01,DBServer01,1433,TCP,Y
Web02,DBServer02,1433,TCP,Y
Web01,SMTPServer01,25,TCP,N
Web01,NTPServer01,123,UDP,N
```

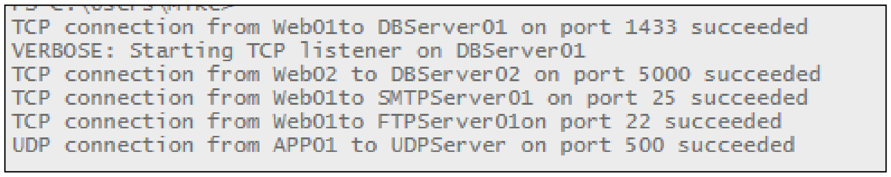
The function to test the ports would look something like this:

```
function Test-EnvironmentConnectivity{
Param([string]$path,[PSCredential]$Cred)

    $tests=import-csv $path
    if($Cred){
        $CredParam=@{Credential=$Cred}
    } else {
        $CredParam=@{}
    }
    $listeners=@{TCP=${function:start-tcpListener}}
    $testFunctions=@{TCP=${function:test-tcpConnection};
                    UDP=${function:test-udpConnection};
                    ICMP=${function:test-icmpConnection}}
    foreach($test in $tests){
        if($test.Startlistener -eq 'Y'){
            if(($listeners.ContainsKey($test.Protocol))){
                write-verbose "Starting $($test.Protocol)
                  listener on $($test.Destination)"
                $listenerjob=invoke-command -ScriptBlock
                  $listeners[$test.Protocol] -argumentList
                  $test.Port -ComputerName $test.Destination
                  @CredParam -AsJob
            }
        }
        try {
            $result=invoke-command -ScriptBlock
              $testFunctions[$test.Protocol] -ComputerName
              $test.Source -ArgumentList
              $test.Destination,$test.Port @CredParam
            if($result){
              write-output ("{0} connection from {1} to {2} on
                port {3} succeeded" -f
                $test.Protocol,$test.Source,
                $test.Destination,$test.Port)
            } else {
              write-output ("{0} connection from {1} to {2} on
                port {3} failed" -f
                $test.Protocol,$test.Source,
                $test.Destination,$test.Port)
          }
        } catch {
          write-output  ("{0} connection from {1} to {2}
            on port {3} failed" -f
            $test.Protocol,$test.Source,
            $test.Destination,$test.Port)
        } finally {
          if($test.StartListener -eq 'Y'){
            remove-job -job $listenerJob -Force
          }
        }
    }
}
```

Here's a sample input file with representative output:

```
Source,Destination,Port,Protocol,StartListener
Web01,DBServer01,1433,TCP,Y
Web02,DBServer02,1433,TCP,Y
Web01,SMTPServer01,25,TCP,N
Web01,FTPServer01,22,TCP,N
APP01,UDPServer,500,UDP,N
```

The following screenshot shows the corresponding output:



```
TCP connection from Web01to DBServer01 on port 1433 succeeded
VERBOSE: Starting TCP listener on DBServer01
TCP connection from Web02 to DBServer02 on port 5000 succeeded
TCP connection from Web01to SMTPServer01 on port 25 succeeded
TCP connection from Web01to FTPServer01on port 22 succeeded
UDP connection from APP01 to UDPServer on port 500 succeeded
```

When running this function, you will probably want to inform your network security group about your activities. Otherwise, they might think you're performing a port-scanning attack on the network.

Further reading

The following references will help you get more information on the topics covered in this chapter:

- Enabling telnet at `http://social.technet.microsoft.com/wiki/contents/articles/910.windows-7-enabling-telnet-client.aspx`
- `get-help get-CIMInstance`
- `get-help about_WMI`
- `get-help about_WMI_Cmdlets`
- `get-help get-WMIObject`
- `get-help get-service`
- `get-help Update-TypeData`
- `get-help about_Types.ps1xml`
- `NetCat and NMAP- http://nmap.org/`
- `get-help Test-NetConnection`
- `get-help New-Object`

Summary

This chapter focused on the environment that the script runs in, namely the operating system installed on the servers, the hardware of the servers, and the network connectivity between the servers. Making sure that all of these are accounted for will make your PowerShell troubleshooting much easier, as an unexpected result in one of these areas might make your script fail.

The next chapter will cover traditional troubleshooting techniques, including debugging scripts both at the command line and in the ISE, using risk-mitigation parameters, and other techniques that are unique to PowerShell.

7 Reactive Practices – Traditional Debugging

So far, we have focused on making the code easier to troubleshoot. This chapter will introduce the techniques used to troubleshoot the code while it's running. We will see that the investment we've put into the proper design and implementation of our code will make the job of troubleshooting much easier. The specific techniques that we will cover in this chapter are as follows:

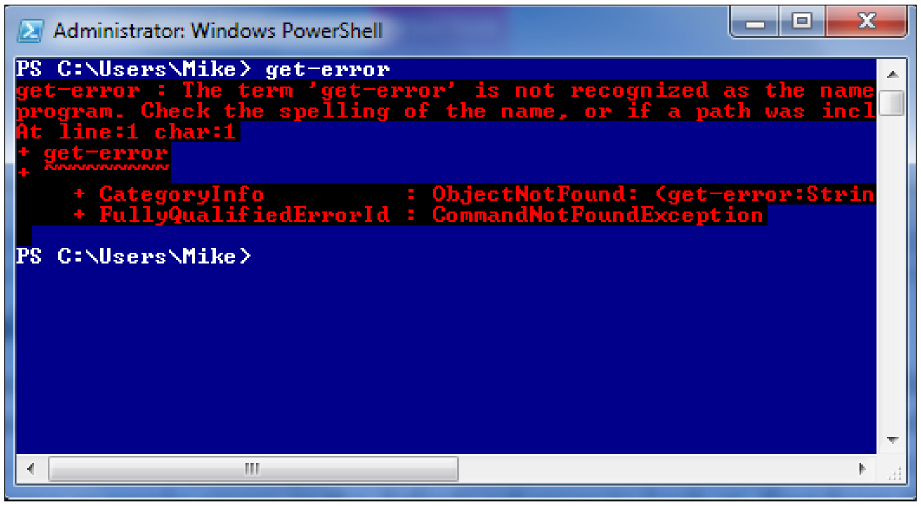
- Reading error messages
- Using `Set-PSDebug`
- Debugging in the ISE (or other integrated environment)
- Debugging in the console
- Event logs
- The PSDiagnostics module
- Using `–confirm` and `–whatif`
- Reducing input size
- Using `Tee-Object`
- Replacing the `foreach` loop with the `foreach-object` cmdlet

Reading error messages

This section shouldn't be necessary for people who are serious about writing scripts, but in my experience it involves one of the simplest techniques of troubleshooting and, unfortunately, one of the techniques that is often overlooked. We have talked about how to handle errors using `try` / `catch` / `finally`, and about the difference between terminating and non-terminating errors, but we haven't spent any time talking about error messages themselves. The simple practice of carefully reading the error messages that occur can help to pinpoint not only the problem, but also where the problem occurred in the code. While that information isn't unique to error messages in PowerShell, I have seen countless occasions where it is overlooked.

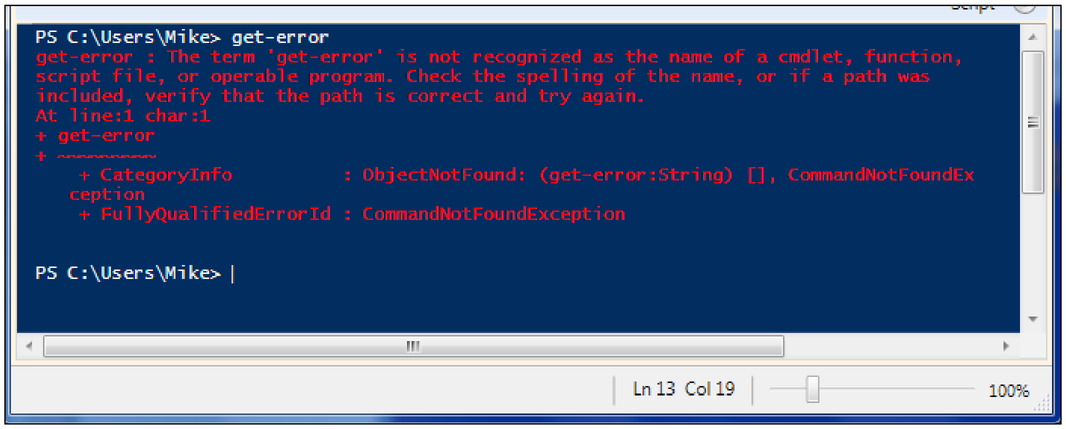
The color problem

My personal opinion is that the default color scheme in the console and the ISE is part of the reason. Due to the default color scheme, errors in PowerShell look somewhat jarring and cause me, at least, to try to skip over them. If you're not familiar with this phenomenon, the following screenshot shows some example errors in the default schemes, first from the console:



```
PS C:\Users\Mike> get-error
get-error : The term 'get-error' is not recognized as the name
program. Check the spelling of the name, or if a path was incl
At line:1 char:1
+ get-error
+ ~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (get-error:Strin
    + FullyQualifiedErrorId : CommandNotFoundException

PS C:\Users\Mike>
```

The error-display interface in the ISE isn't very good in my opinion, as is shown in the following screenshot:



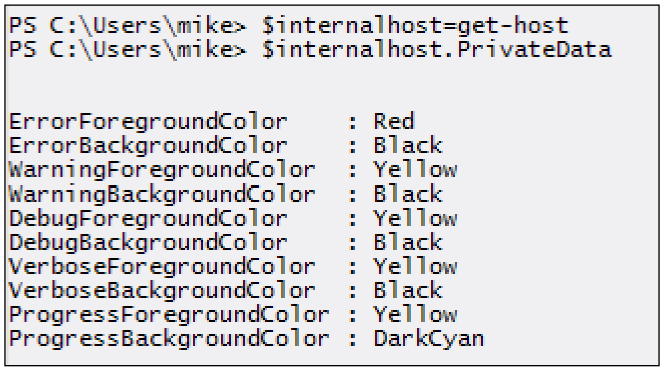
```
PS C:\Users\Mike> get-error
get-error : The term 'get-error' is not recognized as the name of a cmdlet, function,
script file, or operable program. Check the spelling of the name, or if a path was
included, verify that the path is correct and try again.
At line:1 char:1
+ get-error
+ ~~~~~~~~~
 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


PS C:\Users\Mike> |
```

Ln 13 Col 19 100%

The red text on the dark blue background is hard to read (for me) and I know I have often received bug reports of some red text rather than a useful cut-and-paste of the error message. As a color-blind person, I don't tend to give advice on changing the colors in a program. In this case, though, I always recommend that the foreground and background color for error messages be changed.

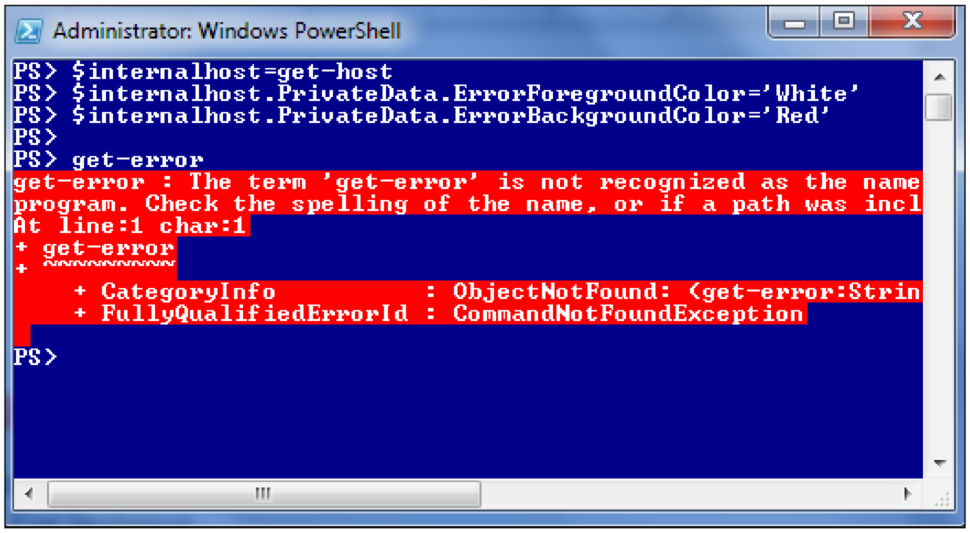
Changing console colors

The configuration of the PowerShell console colors is done through the `Get-Host` cmdlet. This cmdlet returns an `InternalHost` object that has a property called `PrivateData`. The `PrivateData` property, in turn, has properties that describe the foreground and background colors used to display different items in the console. The list of colors is shown in the following screenshot:



```
PS C:\Users\mike> $internalhost=get-host
PS C:\Users\mike> $internalhost.PrivateData


ErrorForegroundColor    : Red
ErrorBackgroundColor    : Black
WarningForegroundColor  : Yellow
WarningBackgroundColor  : Black
DebugForegroundColor    : Yellow
DebugBackgroundColor    : Black
VerboseForegroundColor  : Yellow
VerboseBackgroundColor  : Black
ProgressForegroundColor : Yellow
ProgressBackgroundColor : DarkCyan
```

Note the red foreground and black background for errors. To change these, simply set these properties to a more reasonable value. For instance, the following screenshot illustrates a red background and white foreground error display:

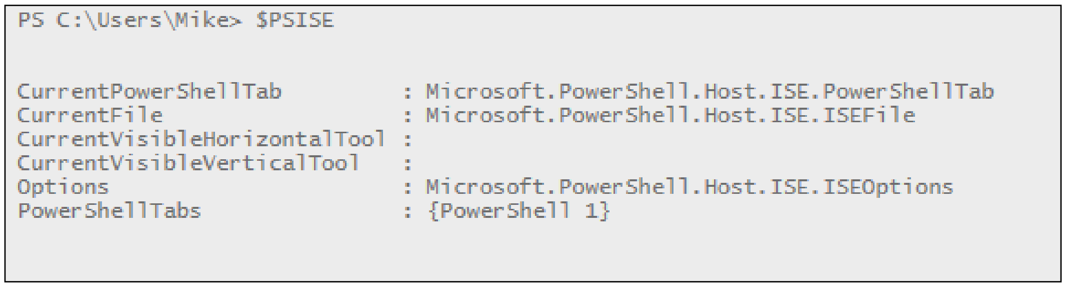


```
PS> $internalhost=get-host
PS> $internalhost.PrivateData.ErrorForegroundColor='White'
PS> $internalhost.PrivateData.ErrorBackgroundColor='Red'
PS>
PS> get-error
get-error : The term 'get-error' is not recognized as the name
program. Check the spelling of the name, or if a path was incl
At line:1 char:1
+ get-error
+ ~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (get-error:Strin
    + FullyQualifiedErrorId : CommandNotFoundException

PS>
```

Now, the error messages in the console still show red, but they are much more readable.

Changing ISE colors

The process of changing the colors in the ISE is similar to changing the console colors, but it uses different objects. The ISE exposes its object model through the `$PsISE` variable, which lets us work with several different components in the ISE, as shown in the following screenshot:

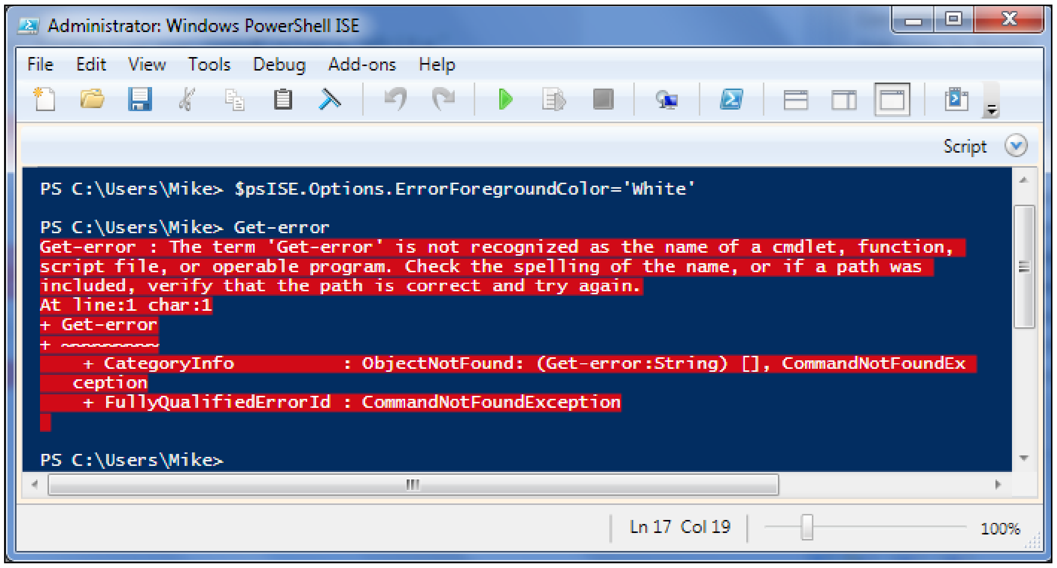


```
PS C:\Users\Mike> $PSISE


CurrentPowerShellTab          : Microsoft.PowerShell.Host.ISE.PowerShellTab
CurrentFile                   : Microsoft.PowerShell.Host.ISE.ISEFile
CurrentVisibleHorizontalTool  :
CurrentVisibleVerticalTool    :
Options                       : Microsoft.PowerShell.Host.ISE.ISEOptions
PowerShellTabs                : {PowerShell 1}
```

For our purposes, we will use the `Options` property, which again shows a lot of potential places to customize the ISE:

```
PS C:\Users\Mike> $psISE.Options


SelectedScriptPaneState                     : Top
ShowDefaultSnippets                         : True
ShowToolBar                                 : True
ShowOutlining                               : True
ShowLineNumbers                             : True
TokenColors                                 : {[Attribute, #FF00BFFF], [Command, #FF0000FF], [Commar
ConsoleTokenColors                          : {[Attribute, #FF00BFFF], [Command, #FF0000FF], [Commar
XmlTokenColors                              : {[Comment, #FF006400], [CommentDelimiter, #FF008000],
DefaultOptions                              : Microsoft.PowerShell.Host.ISE.ISEOptions
FontSize                                    : 9
Zoom                                        : 100
FontName                                    : Lucida Console
ErrorForegroundColor                        : #FFE50000
ErrorBackgroundColor                        : #00FFFFFF
WarningForegroundColor                      : #FFB26200
WarningBackgroundColor                      : #00FFFFFF
VerboseForegroundColor                      : #FF007F7F
VerboseBackgroundColor                      : #00FFFFFF
DebugForegroundColor                        : #FF007F7F
DebugBackgroundColor                        : #00FFFFFF
ConsolePaneBackgroundColor                  : #FFEAEAEA
ConsolePaneTextBackgroundColor              : #FFEAEAEA
ConsolePaneForegroundColor                  : #FF626262
ScriptPaneBackgroundColor                   : #FFFFFFFF
ScriptPaneForegroundColor                   : #FF000000
ShowWarningForDuplicateFiles                : True
ShowWarningBeforeSavingOnRun                : False
```

The `ErrorForegroundColor` and `ErrorBackgroundColor` properties are clearly the properties that control the appearance of errors, so let's set them first and see what effect they have:

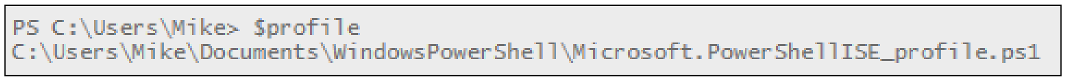


```
PS C:\Users\Mike> $psISE.Options.ErrorForegroundColor='White'

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


PS C:\Users\Mike>
```

Remember that these changes will need to be made in each session. One way to make this happen all the time is to place them in a *profile*.

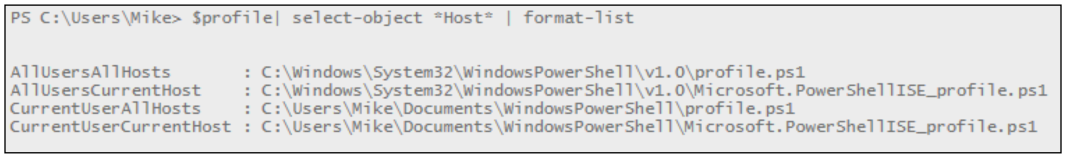
PowerShell profiles

Profiles are scripts that are run when a PowerShell session is started. Each PowerShell host has four possible profiles. The value of the `$profile` variable shows one of these. Note that just because `$profile` has a value, it doesn't mean that the file exists. The following screenshot shows the profile:



```
PS C:\Users\Mike> $profile
C:\Users\Mike\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1
```

To find the other three locations, we need to look at properties that have been added to the `$profile` variable. Since the property names all include the word `Host`, they are easy to isolate, as shown in the following screenshot:



```
PS C:\Users\Mike> $profile| select-object *Host* | format-list


AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost    : C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1
CurrentUserAllHosts    : C:\Users\Mike\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost : C:\Users\Mike\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1
```

The two `AllHosts` values point to scripts that will run no matter what host is running. The `CurrentHost` values point to profile scripts that are specific to the current host. Note that you can see `PowerShellISE` in the path of the output, so these will only run in the ISE. Since the code to change colors is different between the console and the ISE, either of the `CurrentHost` profiles would be an appropriate choice. Depending on whether you wanted the customizations to be present for all users, or just for yourself, would determine which of the two `CurrentHost` profiles you used.

Error message content

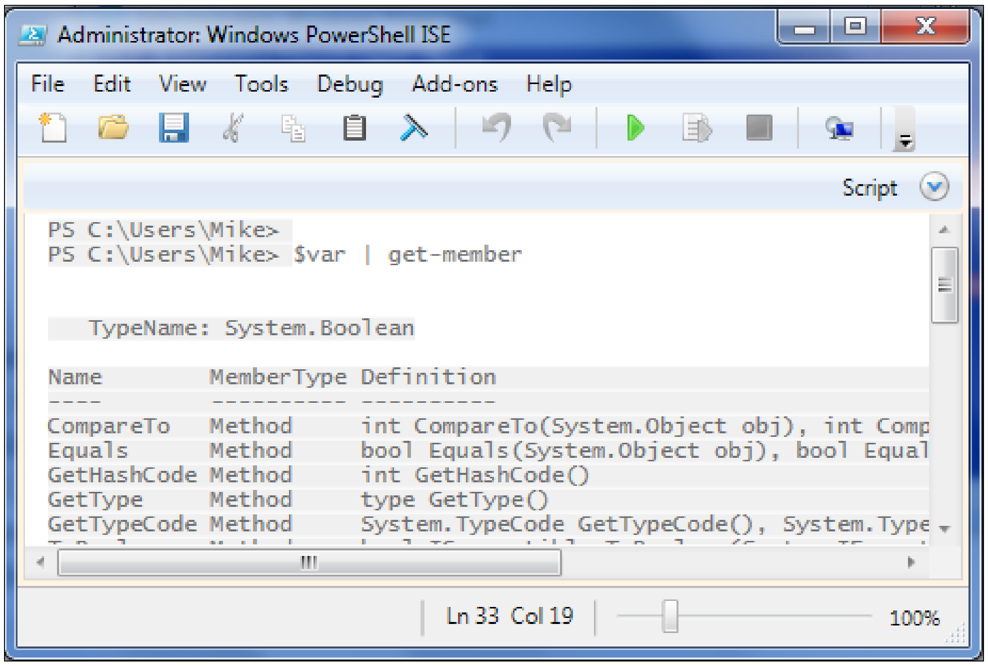
The content of the error messages is not particularly surprising. It contains the following items:

- The command (cmdlet, function, or script) where the error occurs
- The text of the error message
- The location of the error (line and column)
- The source code of the line where the error occurs (underlined to show the error)
- The category and full type name of the error

Although all of this information is expected, it is completely wasted if you don't read it. Reading it doesn't help, either, unless you look at the code where the error is and the error message itself to try to determine what is causing the problem. For instance, a common error message is that a property being referenced doesn't exist on the object in question. There are several reasons that might lead to that particular error, and some are listed as follows:

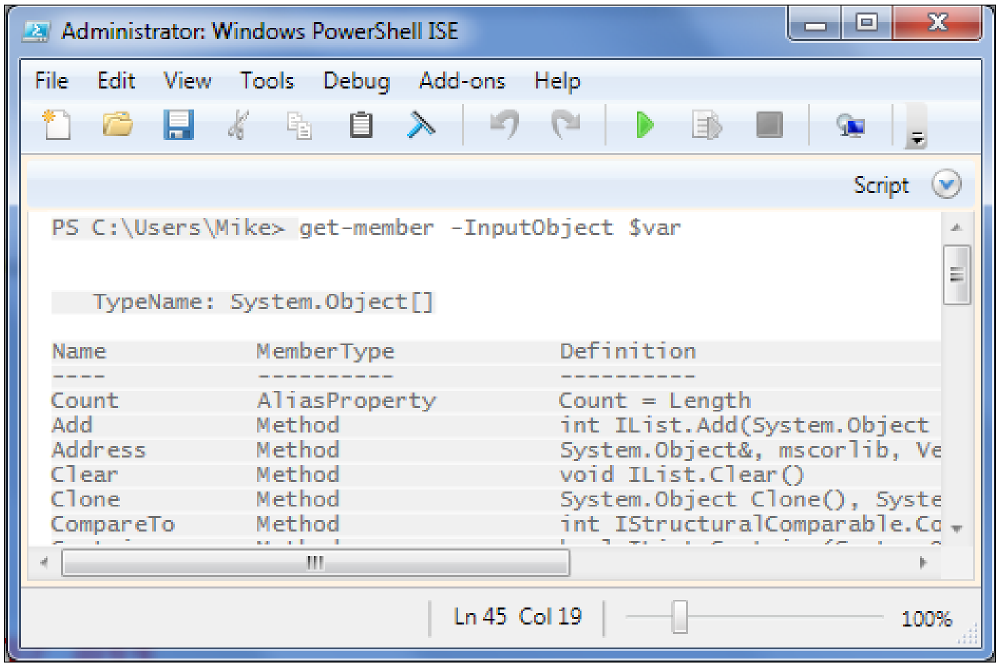
- Misspelling the name of the property
- Misspelling the name of the variable that has the property
- Using the wrong variable
- The variable isn't the expected type

Spending the time to analyze the error, and probably eliminating most of them, is definitely worth the effort. In this case, the first three should be simple to validate (for example, check the spelling and match the variable name). The fourth is a little trickier to determine. If you recall from *Chapter 1*, *PowerShell Primer*, the `Get-Member` cmdlet outputs the types of the objects that are piped to it, so it is often the first cmdlet that we turn to. Our attempt would look something like what is shown in the following screenshot:


Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script
Ln 33 Col 19
100%

```
PS C:\Users\Mike>
PS C:\Users\Mike> $var | get-member


   TypeName: System.Boolean

Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     int CompareTo(System.Object obj), int Comp
Equals      Method     bool Equals(System.Object obj), bool Equal
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode(), System.Type
```

With this information in hand, we proceed with the information that our variable holds a Boolean value. The problem is that we're misusing `Get-Member` here. The purpose of `Get-Member` is to list the members of the distinct types of objects that it receives from the pipeline or the `-InputObject` parameter. By using the pipeline we have obscured the value of the variable. Using the `-InputObject` parameter gives us a different answer altogether, as shown in the following screenshot:

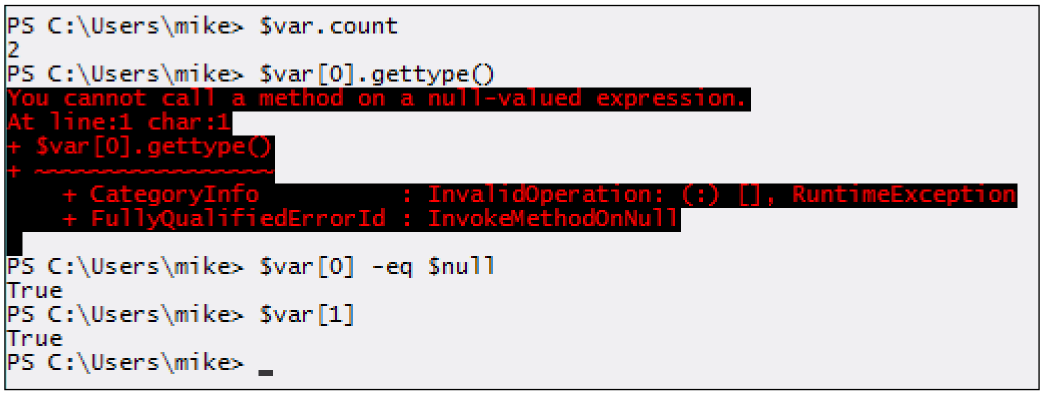

Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script
Ln 45 Col 19
100%

```
PS C:\Users\Mike> get-member -InputObject $var


   TypeName: System.Object[]

Name           MemberType          Definition
----           ----------          ----------
Count          AliasProperty       Count = Length
Add            Method              int IList.Add(System.Object
Address        Method              System.Object&, mscorlib, Ve
Clear          Method              void IList.Clear()
Clone          Method              System.Object Clone(), Syste
CompareTo      Method              int IStructuralComparable.Co
```

Here, we see that the variable actually contains an array of objects. We can see this as well by calling the `GetType()` method of the variable:

```
PS C:\Users\mike> $var.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

To see the individual objects, we can check the count and index the items separately. What we will find is that `$var` is an array of a `$null` and `$true` value, as shown in the following screenshot:



```
PS C:\Users\mike> $var.count
2
PS C:\Users\mike> $var[0].gettype()
You cannot call a method on a null-valued expression.
At line:1 char:1
+ $var[0].gettype()
+ ~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvokeMethodOnNull

PS C:\Users\mike> $var[0] -eq $null
True
PS C:\Users\mike> $var[1]
True
PS C:\Users\mike> _
```

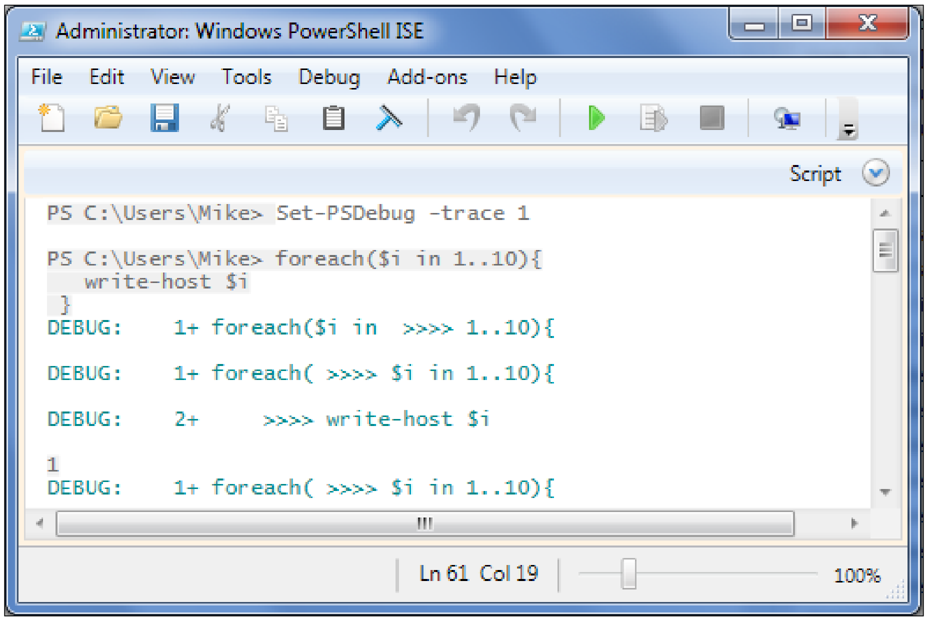
The reason this is important is that when you are troubleshooting, knowing the types and values of variables is of the utmost importance. Using the pipeline to provide the input to `Get-Member` hid the value because the pipeline unrolls arrays and `Get-Member` saw a `$null` and a Boolean value. There's no object corresponding to `$null`, so it wasn't represented in the output. Given that it's possible to accidentally output more objects from a function than intended, this kind of investigation is something that you will probably be doing a lot, especially when the errors don't seem to make sense.

Using Set-PSDebug

We already met the `Set-PSDebug` cmdlet in *Chapter 5, Proactive PowerShell*, where we learned that the `-Strict` switch can be used to ensure that references to variables that haven't been assigned will cause an error. In the context of debugging, the `Set-PSDebug` cmdlet gives a very simple debugging experience at the command line using the `-Trace` parameter and the `-Step` switch. Let's use a simple script to illustrate this:

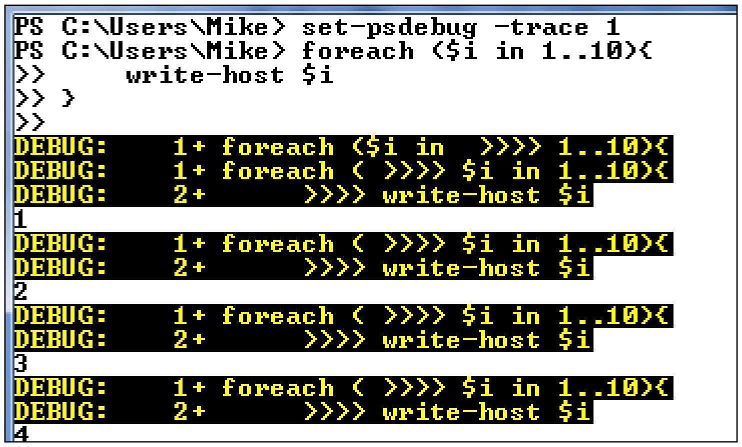
```
foreach ($i in 1..10){
    write-host $i
}
```

While there should be no confusion over what this script will do when executed, watch what happens when we run `Set-PSDebug -Trace 1` and then run the script:



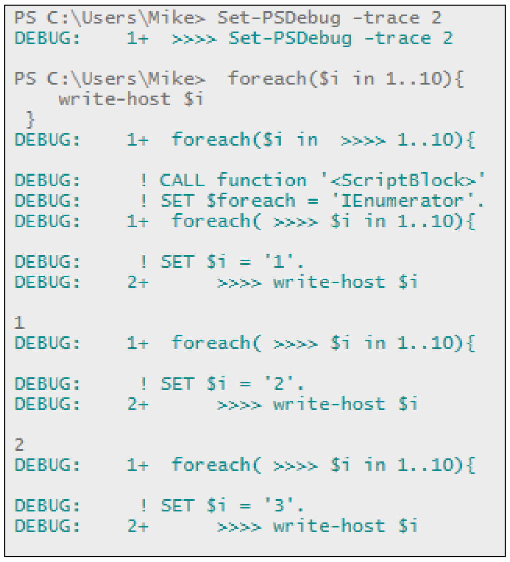
```
PS C:\Users\Mike> Set-PSDebug -trace 1

PS C:\Users\Mike> foreach($i in 1..10){
   write-host $i
 }
DEBUG:     1+ foreach($i in  >>>> 1..10){

DEBUG:     1+ foreach( >>>> $i in 1..10){

DEBUG:     2+      >>>> write-host $i

1
DEBUG:     1+ foreach( >>>> $i in 1..10){
```

I've truncated the output, but it should be clear that the PowerShell engine is outputting debug messages (like we could with `Write-Debug`) for each line that is executed. The output shown is from the ISE, but the cmdlet works in the console as well, although the formatting is slightly different:



```
PS C:\Users\Mike> set-psdebug -trace 1
PS C:\Users\Mike> foreach ($i in 1..10){
>>      write-host $i
>> }
>>
DEBUG:     1+ foreach ($i in   >>>> 1..10){
DEBUG:     1+ foreach ( >>>> $i in 1..10){
DEBUG:     2+         >>>> write-host $i
1
DEBUG:     1+ foreach ( >>>> $i in 1..10){
DEBUG:     2+         >>>> write-host $i
2
DEBUG:     1+ foreach ( >>>> $i in 1..10){
DEBUG:     2+         >>>> write-host $i
3
DEBUG:     1+ foreach ( >>>> $i in 1..10){
DEBUG:     2+         >>>> write-host $i
4
```

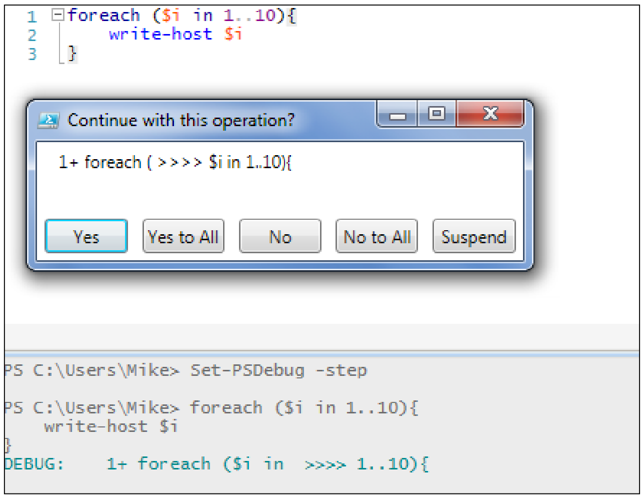
Changing the value of the trace parameter to `2` gives a more detailed result, as shown in the following screenshot:



```
PS C:\Users\Mike> Set-PSDebug -trace 2
DEBUG:    1+  >>>> Set-PSDebug -trace 2

PS C:\Users\Mike>  foreach($i in 1..10){
    write-host $i
 }
DEBUG:    1+  foreach($i in  >>>> 1..10){

DEBUG:     ! CALL function '<ScriptBlock>'
DEBUG:     ! SET $foreach = 'IEnumerator'.
DEBUG:    1+  foreach( >>>> $i in 1..10){

DEBUG:     ! SET $i = '1'.
DEBUG:    2+      >>>> write-host $i

1
DEBUG:    1+  foreach( >>>> $i in 1..10){

DEBUG:     ! SET $i = '2'.
DEBUG:    2+      >>>> write-host $i

2
DEBUG:    1+  foreach( >>>> $i in 1..10){

DEBUG:     ! SET $i = '3'.
DEBUG:    2+      >>>> write-host $i
```

Now we see the (implied) assignment to the `$i` loop variable, as well as some bookkeeping done by PowerShell with the `$foreach` variable. Function calls, and calls to scripts, would also be called out by this trace level.

The final option with `Set-PSDebug` is the `-Step` switch. This switch causes the execution of code to be interrupted. Here, I've run the code with the `-Step` switch and clicked on the **Yes** button one time:



```
foreach ($i in 1..10){
    write-host $i
}
```

Continue with this operation?

1+ foreach (>>>> $i in 1..10){

```
PS C:\Users\Mike> Set-PSDebug -step

PS C:\Users\Mike> foreach ($i in 1..10){
    write-host $i
}
DEBUG:    1+ foreach ($i in  >>>> 1..10){
```

An interesting option with `-Step` is the ability to suspend. By suspending, a nested shell is started. Nested shells are essentially a new PowerShell instance running inside the current instance. You have access to all of the variables and functions from the outer shell and can investigate or modify the state of the system including changing variable values, importing modules, or whatever you wish. Once you are done, the `exit` keyword causes the nested shell to end and you are back at the `Set-PSDebug` prompt at the same line of code that you suspended. Once the `-Trace` parameter or `-Step` switch have been used, the `-Off` switch causes the `Set-PSDebug` cmdlet to stop outputting `Debug` statements and prompting at each line.

The `Set-PSDebug` cmdlet with the `-Trace` parameter and `-Step` switch applies to whatever code is running so you can't use them to set breakpoints. On the other hand, there is no reason that the `Set-PSDebug` cmdlet can't be included in a script. Setting the trace level to 1 or 2 before a critical section of code, and turning it off with the `-Off` switch, would allow you to get this level of debugging information for that section of code without needing to manually step through all of the code up until that point.

Debugging in the console

The `Set-PSDebug` cmdlet gives a good amount of detail, but getting to a particular point in the code requires you to add `Set-PSDebug` statements in the code (which you might not be able to do) or step through all of the code up until that point. Fortunately, there is another set of cmdlets that allows interactive debugging, the `PSBreakPoint` cmdlets:

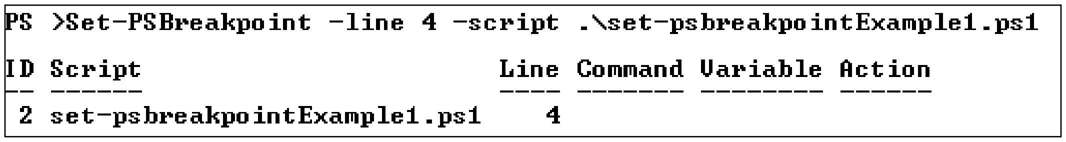
- `Set-PSBreakPoint`
- `Remove-PSBreakPoint`
- `Get-PSBreakPoint`
- `Enable-PSBreakPoint`
- `Disable-PSBreakPoint`

With `Set-PSBreakpoint`, it is easy to create a breakpoint to cause the execution to be suspended when a certain line is reached using the `-Line` parameter. One caveat is that the parameter sets including the `-Line` parameter, also include a mandatory script parameter that refers to a file on disk. This isn't much of a barrier in the console since we would generally be working with a script file.

Here's another sample script:

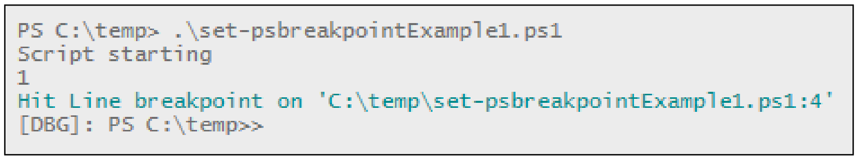
```
Write-Host "Script starting"
Foreach ($i in 1..10){
    Write-Host $i
    Write-Host "Inside the loop"
}
```

With that script in `.\`, issuing `Set-PSBreakPoint -Line 4 -Script .\Set-PSBreakPointExample1.ps1` will cause the execution to stop if line 4 is reached. Note that the `Set-PSBreakPoint` cmdlet outputs an object that we could store if we needed to refer to it later, as shown in the following screenshot:



```
PS >Set-PSBreakpoint -line 4 -script .\set-psbreakpointExample1.ps1

ID Script                           Line Command Variable Action
-- ------                           ---- ------- -------- ------
 2 set-psbreakpointExample1.ps1        4
```

When we execute the code, it stops at the specified line, as shown in the following screenshot:



```
PS C:\temp> .\set-psbreakpointExample1.ps1
Script starting
1
Hit Line breakpoint on 'C:\temp\set-psbreakpointExample1.ps1:4'
[DBG]: PS C:\temp>>
```

Then, when a breakpoint is encountered, a nested shell is created exactly like what is used with `Set-PSDebug -Step` when the `Suspend` option is chosen. If we use the `Get-PSBreakPoint` cmdlet to see what breakpoints exist, and use `Format-List *` to show us all of the properties, we can see that there's a `HitCount` property on the breakpoint object that tells us how many times the breakpoint has been hit. The prompt also changes to indicate that you are now in debug mode. There are several commands you can enter at the prompt to control the debugging, as shown in the following table:

Command	**Action**
`S,stepInto`	Step to the next statement
`V,stepOver`	Step to the next statement in this scope (that is, don't step into function calls)
`O,stepOut`	Step out of the current function or script
`C,Continue`	Continue executing the script or function

Command	Action
`K,Get-PSCallstack`	Display the current call stack
`L,List`	Show the source code for the current script or function
`Q,quit`	Exit the debugger
`<enter>`	Repeat the last command
`?,h`	Display a list of possible commands

Troubleshooting tip

If you have customized your prompt function (that is, redefined it), you won't get an indication that you're in debug mode in the prompt. You will still get a message indicating that you're in debug mode.

There is a `-Column` parameter that can be used in conjunction with the `-Line` parameter to indicate that the breakpoint is only active if the execution hits the code in a particular column. This can be useful if code includes long pipelines.

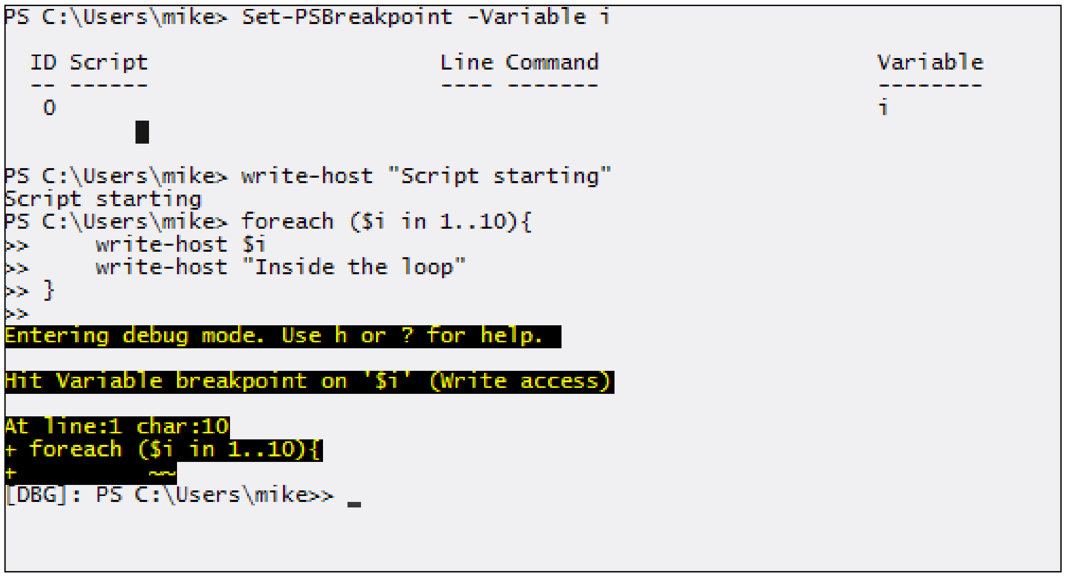
Troubleshooting Tip

If your code is formatted so that each segment of a pipeline is on a separate code line, the `-Column` parameter is not needed.

A second way to use `Set-PSBreakpoint` is to cause the execution to be stopped when a variable is accessed. Since the focus with a variable breakpoint is not on a line of code, a script file is not required. The `-variable` parameter takes a list of variable names (without `$`) to be watched. The `-Mode` parameter allows us to specify what kind of variable activity triggers the breakpoint. The possible values are `Read`, `Write` (the default), and `ReadWrite`. With this example code, let's see what a variable breakpoint shows us:

```
write-host "Script starting"
foreach ($i in 1..10){
    write-host $i
    write-host "Inside the loop"
}
```

The execution is suspended as soon as the variable is referenced, as shown in the following screenshot:

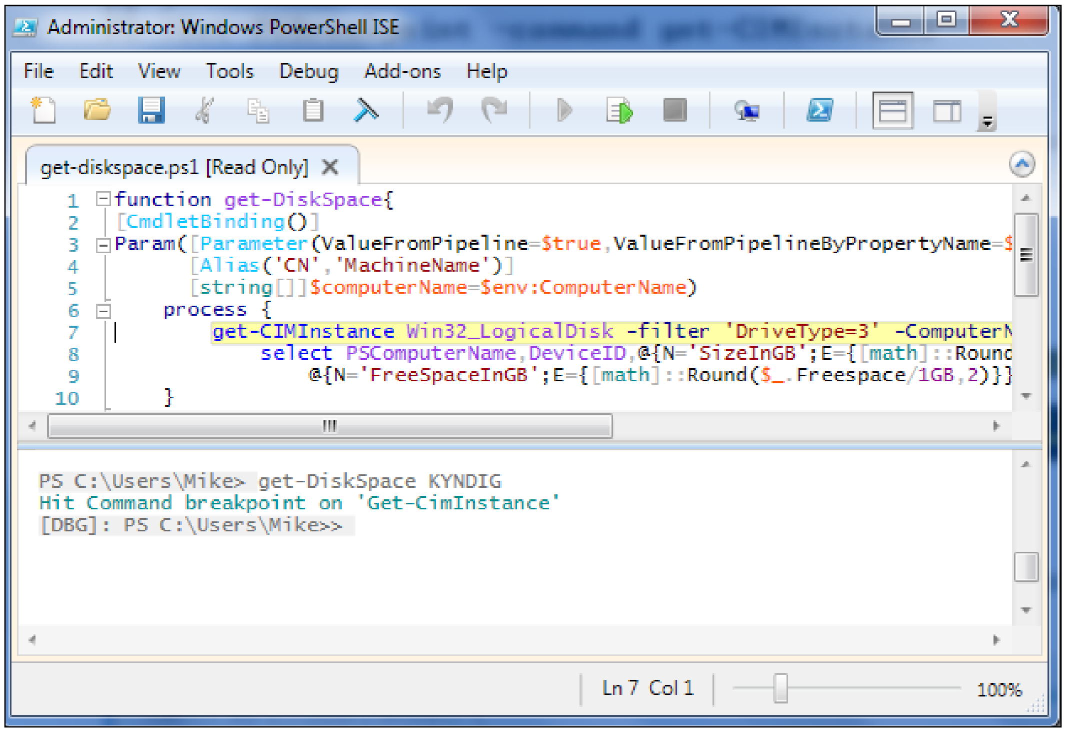


```
PS C:\Users\mike> Set-PSBreakpoint -Variable i

  ID Script                          Line Command                            Variable
  -- ------                          ---- -------                            --------
   0                                                                         i


PS C:\Users\mike> write-host "Script starting"
Script starting
PS C:\Users\mike> foreach ($i in 1..10){
>>     write-host $i
>>     write-host "Inside the loop"
>> }
>>
Entering debug mode. Use h or ? for help.

Hit Variable breakpoint on '$i' (Write access)

At line:1 char:10
+ foreach ($i in 1..10){
+          ~~
[DBG]: PS C:\Users\mike>> _
```

Setting a breakpoint on a variable can be very useful in troubleshooting, especially if a variable ends up with an unexpected value. A final way to set a breakpoint instructs the engine to stop when a particular command (function or cmdlet) is executed. This could be useful if you wanted to know where a built-in or binary cmdlet was being called. Since you wouldn't have the source code, a line-level breakpoint would not be possible. For example, if you knew that `Get-CIMInstance` was referenced at several points in a script, but didn't know which one was being called, you could issue the following command:

```
Set-PSBreakPoint -command get-CIMInstance
```

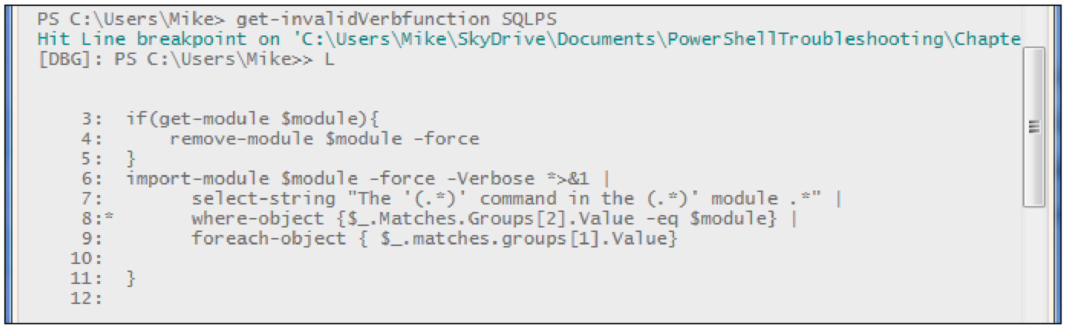
Then, when the script runs, anytime the `get-CIMInstance` cmdlet is invoked the script execution will stop at that point. Here are the results of that breakpoint using the `Get-DiskSpace` function we wrote in *Chapter 6*, *Preparing the Scripting Environment*:


Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
get-diskspace.ps1 [Read Only]
Ln 7 Col 1
100%

```
function get-DiskSpace{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true,ValueFromPipelineByPropertyName=$
      [Alias('CN','MachineName')]
      [string[]]$computerName=$env:ComputerName)
    process {
        get-CIMInstance Win32_LogicalDisk -filter 'DriveType=3' -ComputerN
            select PSComputerName,DeviceID,@{N='SizeInGB';E={[math]::Round
                @{N='FreeSpaceInGB';E={[math]::Round($_.Freespace/1GB,2)}}
    }
```

```
PS C:\Users\Mike> get-DiskSpace KYNDIG
Hit Command breakpoint on 'Get-CimInstance'
[DBG]: PS C:\Users\Mike>>
```

At this point, we could use the debug mode options to display the source code around where the breakpoint was set using the `List` or `L` command:



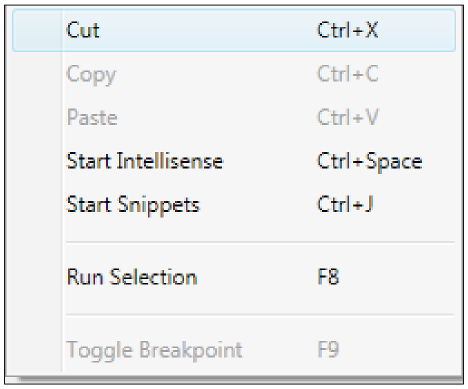
```
PS C:\Users\Mike> get-invalidVerbfunction SQLPS
Hit Line breakpoint on 'C:\Users\Mike\SkyDrive\Documents\PowerShellTroubleshooting\Chapte
[DBG]: PS C:\Users\Mike>> L


    3:  if(get-module $module){
    4:      remove-module $module -force
    5:  }
    6:  import-module $module -force -Verbose *>&1 |
    7:        select-string "The '(.*)' command in the (.*)' module .*" |
    8:*       where-object {$_.Matches.Groups[2].Value -eq $module} |
    9:        foreach-object { $_.matches.groups[1].Value}
   10:
   11:  }
   12:
```

To summarize, there are four different types of breakpoints that can be set using the `Set-PSBreakPoint` cmdlet:

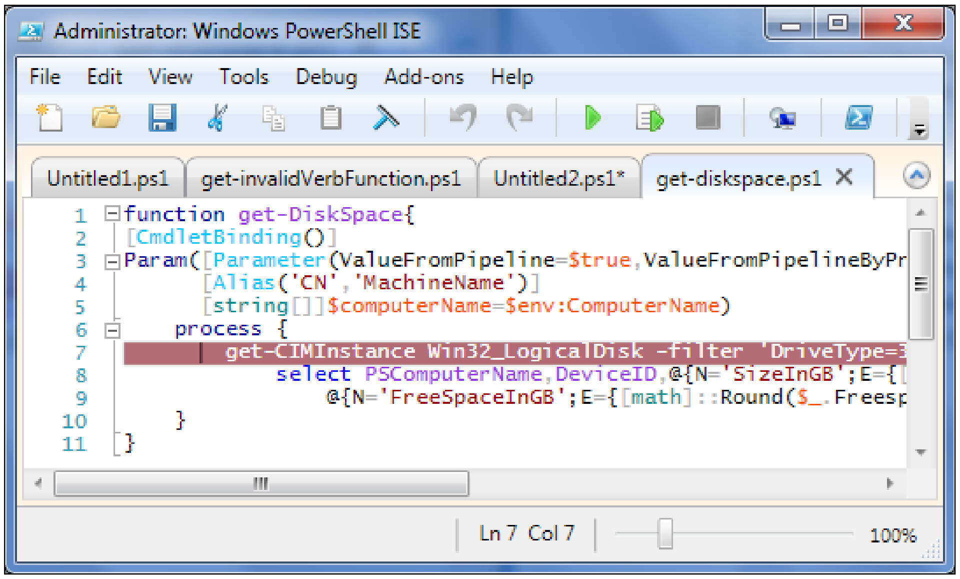
- Line breakpoints (with the `-line` parameter)
- Column breakpoints (with the `-line` and `-column` parameters)
- Variable breakpoints (with the `-variable` parameter)
- Command breakpoints (with the `-command` parameter)

Debugging in the ISE

We've already discussed the use of `Set-PSDebug` in the ISE, so we know that we can use the `-Trace` and `-Step` parameters to get extra output and control options as the script runs in the ISE. As simple as using `Set-PSDebug` is, debugging in the ISE using the GUI is probably used more often than in the console. This is due to the simple point-and-click operation of the ISE. As in most development environments, a breakpoint can be set on a line of code by right-clicking on a line and selecting **Toggle Breakpoint** from the context menu:

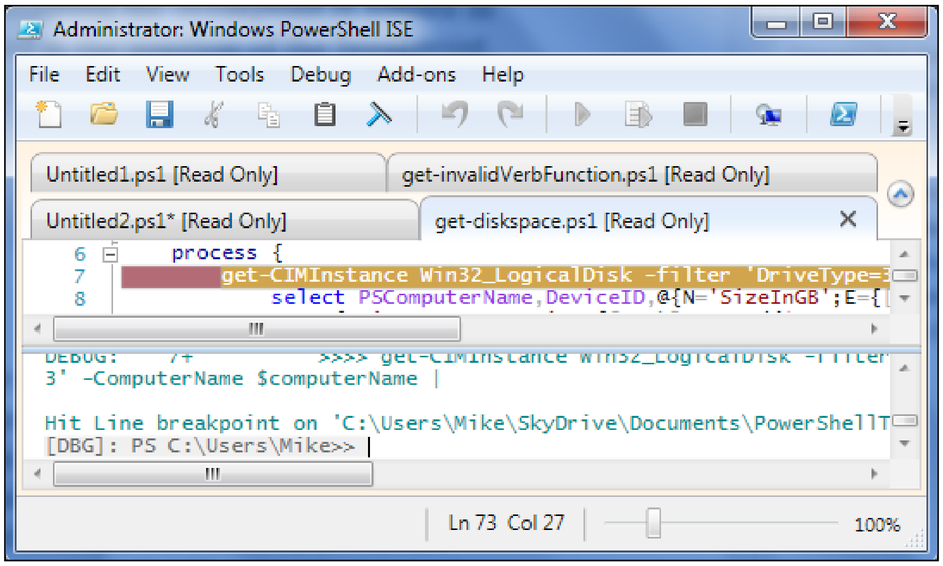


When a breakpoint has been set, the line will be highlighted in red, as shown in the following screenshot:


Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Untitled1.ps1
get-invalidVerbFunction.ps1
Untitled2.ps1*
get-diskspace.ps1
Ln 7 Col 7
100%

```
function get-DiskSpace{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true,ValueFromPipelineByPr
       [Alias('CN','MachineName')]
       [string[]]$computerName=$env:ComputerName)
    process {
        get-CIMInstance Win32_LogicalDisk -filter 'DriveType=3
            select PSComputerName,DeviceID,@{N='SizeInGB';E={[
                @{N='FreeSpaceInGB';E={[math]::Round($_.Freesp
    }
}
```

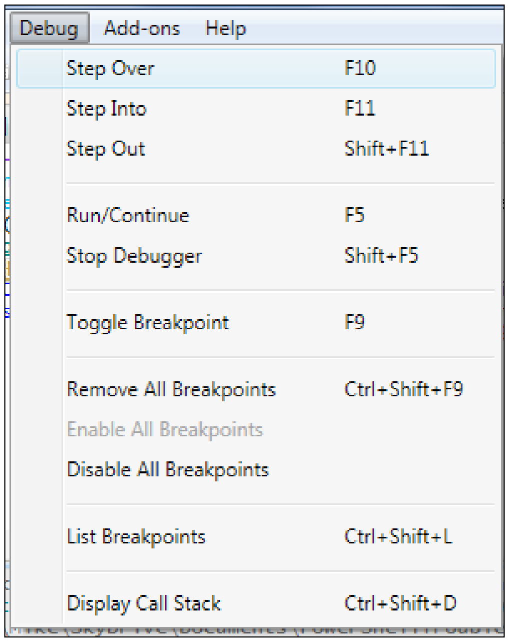
When a breakpoint is reached in the ISE, the script containing the breakpoint is loaded in the ISE if it isn't already loaded, and the cursor is placed at the position of the breakpoint. The current line is highlighted yellow, as shown in the following screenshot:


Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Untitled1.ps1 [Read Only]
get-invalidVerbFunction.ps1 [Read Only]
Untitled2.ps1* [Read Only]
get-diskspace.ps1 [Read Only]
Ln 73 Col 27
100%

```
    process {
        get-CIMInstance Win32_LogicalDisk -filter 'DriveType=3
            select PSComputerName,DeviceID,@{N='SizeInGB';E={[
```

```
DEBUG:    7+          >>>> get-CIMInstance Win32_LogicalDisk -filter
3' -ComputerName $computerName |

Hit Line breakpoint on 'C:\Users\Mike\SkyDrive\Documents\PowerShellT
[DBG]: PS C:\Users\Mike>> 
```

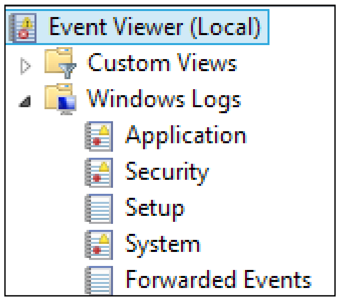
Once a breakpoint has been reached, the ISE has some additional features to help guide the debugging session. First, the **Step Into** (*F11*) command in the **Debug** menu will execute the current statement. If the current statement is a function call, execution will step into the function. Using the **Step Over** (*F10*) command, the next statement is also executed, but if it is a function call it will be considered a single statement and will not step into the function. Finally, the **Run/Continue** (*F5*) command will cause the execution to continue from the current line without breaking (until a breakpoint is reached, of course), and the **Stop Debugger** (*Shift* + *F5*) command will halt the execution of the current script. The following screenshot shows the list of features in the **Debug** menu:



Event logs

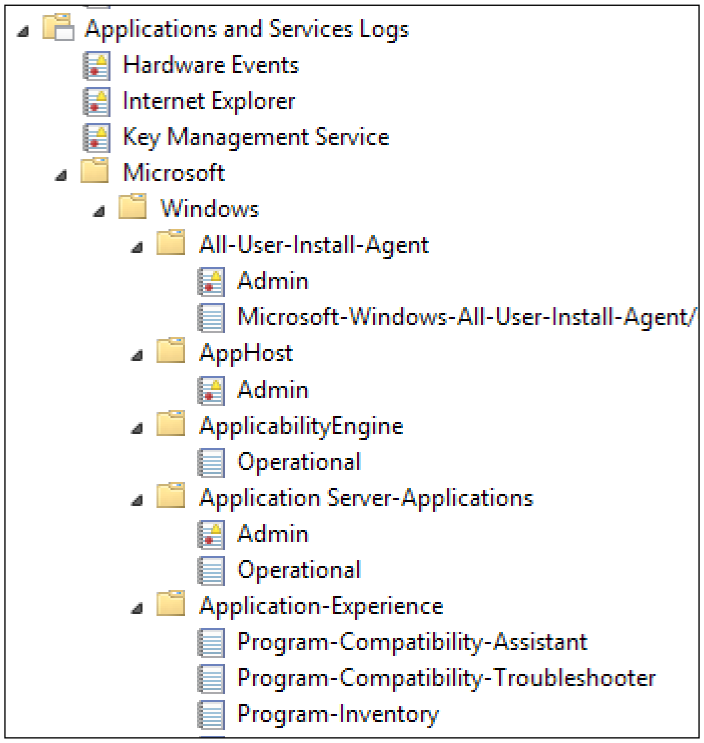
If you've ever spent time troubleshooting a Windows system, you have probably dealt with event logs. Windows writes the details of several kinds of activities into two different kinds of logs. The first is called *classic* because this type of log has been present since the early days of Windows. The classic logs called **Application**, **Security**, **Setup**, and **System** are found on all systems. There can also be a classic log called **Forwarded Events** if you have subscribed to events from a remote computer. There may be other classic logs present on your system depending on what software, roles, and features you have installed.

The following screenshot shows the features of the classic log:



Event Viewer (Local)
Custom Views
Windows Logs
Application
Security
Setup
System
Forwarded Events

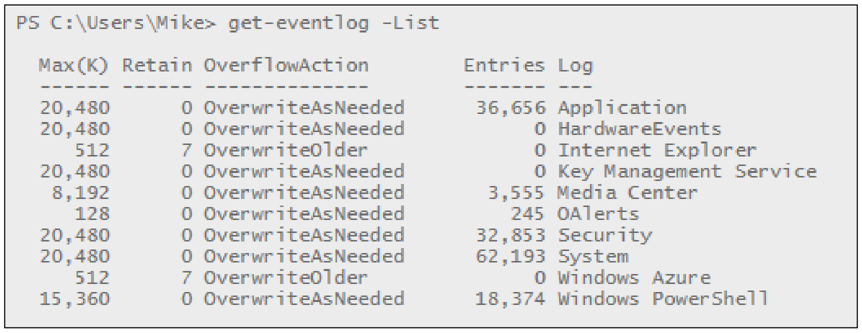
The newer type of event logs are an XML-based system introduced in Windows Vista and have the (not very helpful) name of Windows Event Log technology. We will call them WEL for short since Windows Event Log sounds like it could refer to either type of log. These WEL event logs are listed in a section called **Applications and Services Logs**. Each WEL log can have a subtype of a log called **Admin**, **Operational**, **Analytic**, and **Debug**. Here is a view of **Event Viewer** displaying a few of the hundreds of WEL logs on a computer:



Applications and Services Logs
Hardware Events
Internet Explorer
Key Management Service
Microsoft
Windows
All-User-Install-Agent
Admin
Microsoft-Windows-All-User-Install-Agent/
AppHost
Admin
ApplicabilityEngine
Operational
Application Server-Applications
Admin
Operational
Application-Experience
Program-Compatibility-Assistant
Program-Compatibility-Troubleshooter
Program-Inventory

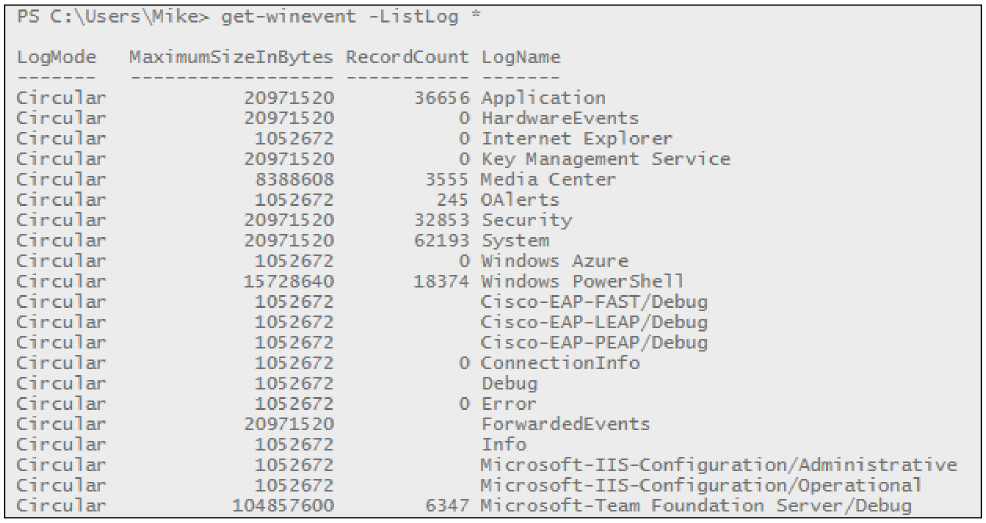
Listing event logs

PowerShell Version 1.0 only included cmdlet support for classic event logs. The `Get-EventLog` cmdlet includes a `–List` switch that causes it to list the classic event logs present on a computer. There is also a `–ComputerName` parameter that allows the cmdlet to target a list of computers. The output of `Get-EventLog -list` will look similar to this:



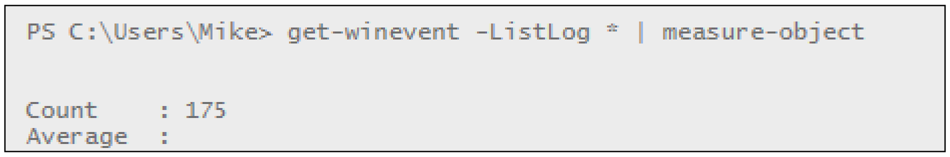
```
PS C:\Users\Mike> get-eventlog -List

  Max(K) Retain OverflowAction        Entries Log
  ------ ------ --------------        ------- ---
  20,480      0 OverwriteAsNeeded      36,656 Application
  20,480      0 OverwriteAsNeeded           0 HardwareEvents
     512      7 OverwriteOlder              0 Internet Explorer
  20,480      0 OverwriteAsNeeded           0 Key Management Service
   8,192      0 OverwriteAsNeeded       3,555 Media Center
     128      0 OverwriteAsNeeded         245 OAlerts
  20,480      0 OverwriteAsNeeded      32,853 Security
  20,480      0 OverwriteAsNeeded      62,193 System
     512      7 OverwriteOlder              0 Windows Azure
  15,360      0 OverwriteAsNeeded      18,374 Windows PowerShell
```

PowerShell Version 2.0 introduced the `Get-WinEvent` cmdlet, which is able to access both classic and WEL logs. It is also capable of reading the files generated by **Event Tracing for Windows** (**ETW**) which can, for instance, be recorded with the Performance Monitor application. The parameter to get the list of logs using `Get-WinEvent` is called `–ListLog`, and unlike with `Get-EventLog`, `–ListLog` is not a switch. The `–ListLog` takes an array of strings, so in order to get a complete list of logs on a system with `Get-WinEvent`, you would issue the `Get-WinEvent -listlog *` command. On my system, the output starts:



```
PS C:\Users\Mike> get-winevent -ListLog *

LogMode   MaximumSizeInBytes RecordCount LogName
-------   ------------------ ----------- -------
Circular            20971520       36656 Application
Circular            20971520           0 HardwareEvents
Circular             1052672           0 Internet Explorer
Circular            20971520           0 Key Management Service
Circular             8388608        3555 Media Center
Circular             1052672         245 OAlerts
Circular            20971520       32853 Security
Circular            20971520       62193 System
Circular             1052672           0 Windows Azure
Circular            15728640       18374 Windows PowerShell
Circular             1052672             Cisco-EAP-FAST/Debug
Circular             1052672             Cisco-EAP-LEAP/Debug
Circular             1052672             Cisco-EAP-PEAP/Debug
Circular             1052672           0 ConnectionInfo
Circular             1052672             Debug
Circular             1052672           0 Error
Circular            20971520             ForwardedEvents
Circular             1052672             Info
Circular             1052672             Microsoft-IIS-Configuration/Administrative
Circular             1052672             Microsoft-IIS-Configuration/Operational
Circular           104857600        6347 Microsoft-Team Foundation Server/Debug
```

I have truncated the output because it runs for several pages. To see just how many logs are shown, we can use the `Measure-Object` cmdlet, as shown in the following screenshot:



```
PS C:\Users\Mike> get-winevent -ListLog * | measure-object


Count    : 175
Average  :
```

You might also see errors corresponding to logs that don't exist on your system. I get an error on this machine about the `Microsoft-Windows-DxpTaskRingtone/Analytic` log, which does not have a valid path set. For what it's worth, I get an error trying to view the log using **Event Viewer** as well.

Reading event logs

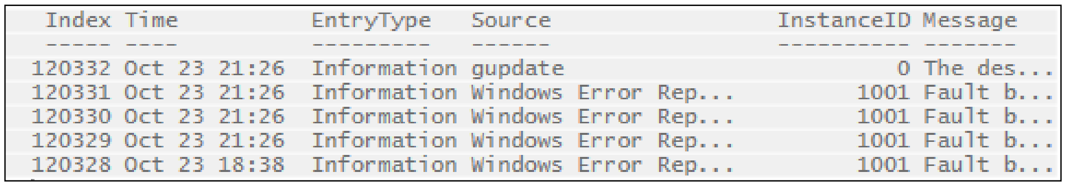
Although both `Get-EventLog` and `Get-WinEvent` allow you to read event logs, `Get-EventLog` can only work with the classic logs. To get entries from a classic log using `Get-EventLog`, you will use the `–LogName` parameter to specify which log to read and you will generally include one or more parameters to narrow down which log messages are required. Some of the more useful filtering parameters for `Get-EventLog` are:

Parameter	**Meaning**
`-Index`	The numerical index of the log entry
`-EntryType`	Includes `Error`, `Information`, `FailureAudit`, `SuccessAudit`, and `Warning`
`-Message`	Filter by the contents of the message (allows wildcards)
`-Source`	Include messages written to the log by certain sources
`-Before` and `–After`	Filter based on the time the message was written
`-Newest`	Only retrieve the latest log entries
`-UserName`	Include messages associated with certain usernames

For instance, to read the most recent five entries in the **Application** event log, you would use the following command:

```
Get-EventLog -LogName Application -newest 5
```

The results are as expected:



```
  Index Time          EntryType   Source                 InstanceID Message
  ----- ----          ---------   ------                 ---------- -------
 120332 Oct 23 21:26  Information gupdate                         0 The des...
 120331 Oct 23 21:26  Information Windows Error Rep...         1001 Fault b...
 120330 Oct 23 21:26  Information Windows Error Rep...         1001 Fault b...
 120329 Oct 23 21:26  Information Windows Error Rep...         1001 Fault b...
 120328 Oct 23 18:38  Information Windows Error Rep...         1001 Fault b...
```

Reading event logs with `Get-WinEvent` uses a `-LogName` parameter to indicate which log we want to look at, but now we have the option of listing more than one event log and are allowed to use wildcards. Given the large number of logs accessible by using `Get-WinEvent`, that flexibility is important. The filtering parameters for `Get-WinEvent` are given as follows:

Parameter	**Purpose**
`-FilterXPath`	Uses an XPath expression to filter the log entries
`-FilterHashTable`	Uses a hashtable of keys and values to filter the log entries
`-FilterXML`	Uses a structured XML query to filter the log entries
`-MaxEvents`	Limits the number of entries returned
`-Oldest (switch)`	Forces retrieval of oldest entries first instead of the default (newest first)

The first thing to note is that there aren't parameters to filter by specific properties of the entries. One reason for this is the large number of properties exposed by the new log format. The `-FilterXPath`, `-FilterXML`, and `-FilterHashTable` parameters allow us to filter by multiple properties at the same time. The second thing to note is that the `-LogName` parameter is not in the same parameter set as any of these three filtering parameters, so the name of the log will have to be specified in in the hashtable, XPath, or XML query.

For instance, finding the most recent five events in the **System** event log with `EventID` of `6013` (uptime messages) could be achieved with the `-FilterHashTable` parameter using the following command:

```
Get-WinEvent -FilterHashtable @{LogName='System';Id=6013} -MaxEvents 5
```

It is important to realize that there have been some terminology changes between some of the parameter names. For instance, with `Get-EventLog` you can only limit the number of events with the `–Newest` parameter. With `Get-WinEvent`, you can use `–MaxEvents` to find the most recent events, or add the `–Oldest` switch to find the earliest events. Also, some of the parameters for `Get-EventLog` have corresponding entries in the `–FilterHashtable` parameter for `Get-WinEvent` with different names. The following table lists such parameters:

Get-EventLog parameter	**Get-WinEvent Hashtable entry**
`–Source`	`Provider`
`–EventID`	`InstanceID`

It should be clear that there is quite a difference in using the `Get-EventLog` and `Get-WinEvent` cmdlets. Deciding which to use in a given situation can be tricky. Since `Get-EventLog` only works with classic logs, that might be the deciding factor if you need to work with a log that's in the newer WEL format. On the other hand, if you need to work with a classic log, you might find the filtering options provided by `Get-EventLog` to be more convenient. Remembering the filter left principle from *Chapter 3*, *PowerShell Practices*, the greater range of filtering parameters for `Get-WinEvent` will probably tip the balance if you are querying event logs on multiple machines or reading very large event logs. Whatever your situation, try to do some experiments to see which cmdlet gives you the best balance of performance and ease of use. Only you can decide what the correct choice is.

Writing to event logs

Writing to classic event logs is accomplished via the `Write-Eventlog` cmdlet. Each entry in a classic log has an associated source that is a required item when trying to write an entry. A very simple example using an existing log and source is as follows:

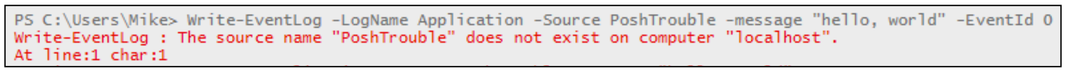
```
Write-EventLog -LogName Application -Source msiInstaller
  -message "hello, world" -EventId 0
```

Here, we've used the `msiInstaller` source and a dummy value of `0` for `EventID`. Event IDs correspond to resources in a `.dll` file, so we have some options for how to proceed:

- Simply use 0 (or some other constant) for all of our messages
- Create a convention that maps values to messages
- Create our own `.dll` file to contain message resources

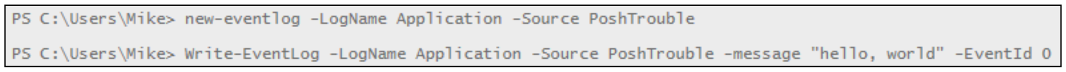
Since the first two will both show errors in **Event Viewer** as there's not a resource for all of the event IDs we would be using, and the third option is beyond the scope of this book, I will choose to use 0 for all of my entries.

Another issue with this sample is that we used the value of `msiInstaller` even though we are clearly not an installer. We really should use a different source to distinguish our events from any others. Unfortunately, an attempt to do this fails, as shown in the following screenshot:



```
PS C:\Users\Mike> Write-EventLog -LogName Application -Source PoshTrouble -message "hello, world" -EventId 0
Write-EventLog : The source name "PoshTrouble" does not exist on computer "localhost".
At line:1 char:1
```

To overcome this error we need to use the `New-EventLog` cmdlet. The obvious use case for `New-EventLog` is to create a completely new classic event log. It can also be used to add a new source (or sources) to an existing event log, as shown in the following screenshot:



```
PS C:\Users\Mike> new-eventlog -LogName Application -Source PoshTrouble

PS C:\Users\Mike> Write-EventLog -LogName Application -Source PoshTrouble -message "hello, world" -EventId 0
```

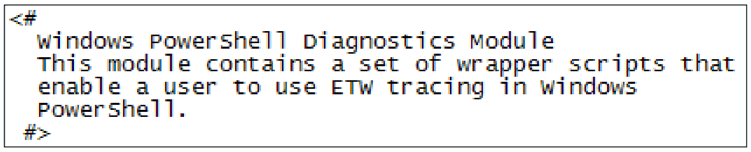
Once we've added the new source to the existing event log, the `Write-EventLog` cmdlet is able to use that source. Creating a custom event log with `New-EventLog` uses the same syntax as we used to create the new source. Also, we can add more than one source at the same time, which is convenient, as shown in the following code snippet:

```
New-EventLog -LogName PowerShelTroubleShooting -Source
  Script,Text,Chapters,Images
```

Note that the event log won't show up in the results of `Get-EventLog`, `Get-WinEvent`, or in **Event Viewer** until an entry has been written to the log.

The PSDiagnostics module

PowerShell Version 2.0 is shipped with a new module called PSDiagnostics, which has some interesting capabilities. One drawback to the PSDiagnostics module is that it contains no documentation of any kind except for a single comment at the beginning of the `.psm1` file, as shown in the following screenshot:



```
<#
  Windows PowerShell Diagnostics Module
  This module contains a set of wrapper scripts that
  enable a user to use ETW tracing in Windows
  PowerShell.
 #>
```

ETW provides, among other things, the capability to trace system and application activity and write the triggered events to a logfile. The simplest functions in the module are the `Enable-PSTrace` and `Disable-PSTrace` functions that turn the **Analytic** and **Debug** logs for the Microsoft/Windows/PowerShell log on (or off). Since the functions eventually call the command-line `wevutil.exe` application that requires input, you will want to use the `-Force` switch if you are running the ISE since it doesn't handle console input well. Enabling these logs will clear them if they already exist, so make sure that's what you want to do.

There are similar functions that start and stop a tracing session involving WS-Man (`Enable-WSManTrace` and `Disable-WSManTrace`) or that involve both WS-Man and PowerShell (`Enable-PSWSManCombinedTrace` and `Disable-PSWSManCombinedTrace`). Note that the verbs are inconsistent: `Enable-PSTrace` simply enables the logfiles, it does not start a tracing session, but the other two enabling cmdlets start tracing sessions.

WS-Man is the protocol implemented by WinRM and is the basis for the CIM cmdlets. PowerShell remoting is also based on WinRM, so troubleshooting issues involving CIM and remoting will definitely benefit from using the combined trace that is generated by `Enable-PSWSManCombinedTrace`. A thorough walk-through of this functionality can be found at the following links:

- `http://windowsitpro.com/blog/troubleshooting-winrm-and-powershell-remoting-part-1`
- `http://windowsitpro.com/blog/tools-troubleshooting-powershell-remoting-and-winrm-part-2`

Using –confirm and –whatif

We've already covered what the`-confirm` and `-whatif` risk mitigation parameters do. We've also seen how it's not difficult to include support for these parameters in your functions. It is now time to think about how to use them in a troubleshooting session. We will focus on `-whatif`, but the discussion applies to `-confirm` in exactly the same way. Obviously, if you are trying to troubleshoot a single function that includes support for `-whatif`, you can add `-whatif` to be sure that any system changes that the function would have made are skipped, while finding out what those would have been. This ability is incredibly beneficial to testing because we don't need to worry about the negative effects of running the code.

The way that the risk mitigation parameters (as well as the output preference parameters) work is something like the following. If you specify `-whatif` in a function call, as the function enters scope, the `$whatifpreference` variable is set to `$true`. The `$PSCmdlet.ShouldProcess` function uses the value of this variable to know whether it should output the string representing the task that would have been performed and return `$false` (skipping the task) or if it should return `$true`.

Besides calling a function and specifying the `-whatif` switch, there are a couple of other ways to activate the functionality. First, you can explicitly set the `$whatifpreference` variable to `$true` and reset it to its original state after the end of the code you want to be able to avoid running. The following code snippet is an example of `-whatif`:

```
function update-MyProcess{
[CmdletBinding(SupportsShouldProcess=$true)]
Param()

    # do something "safe"
    # do something "safe"
    write-host "whatif=$whatifpreference"
    $whatifpreference=$true
    write-host "whatif=$whatifpreference"
    # do something I don't want to do at this time
    $whatifpreference=$PSBoundParameters.ContainsKey('Whatif')
    write-host "whatif=$whatifpreference"
}
```

I used the `$PSBoundParameters` hashtable to check whether `-whatif` was specified on the command line in order to set the value back. Another approach would be to store the original value in a variable.

A final way to force the `whatif` functionality is to use the `$PSDefaultParameterValues` hashtable to set the `whatif` parameter to `$true` for the functions where you want `-whatif` to always be applied. For instance, if you want to be sure never to run the `restart-computer` cmdlet, you could run this command:

`$PSDefaultParameterValues['restart-computer:whatif']=$True`

The key to the `$PSDefaultParameterValues` hashtable is the name of the cmdlet (wildcards are allowed) followed by a colon (`:`) and the name of the parameter. The value associated with the key will be supplied for the parameter for that function or set of functions if no value is explicitly passed. So it would not keep the `Restart-Computer -whatif:$false` command from executing, but that kind of thing is not likely to happen.

Reducing input set

It's often easy when debugging, or trying to understand what has happened in a script, to be overwhelmed by the sheer volume of output. One simple strategy to help troubleshoot a script is to reduce the number of objects or input being considered in the script in order to be able to focus more clearly on what is being done.

The first way I do this is to only consider a data point or set of points that I understand really well. So instead of having the script pull in all of the servers from a CSV file, I might use a smaller CSV file or use a different parameter to have it look at a specific server. For instance, consider the following script:

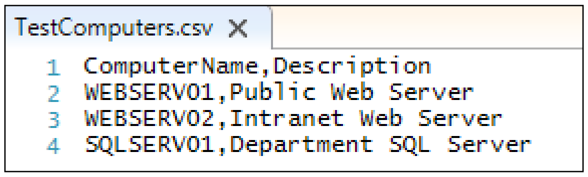
```
function generate-bigreport{
[CmdletBinding()]
Param([Parameter(ValueFromPipeline=$true,
  ValueFromPipelineByPropertyName=$true)]
      [Alias('CN','MachineName')]
      [string[]]$computerName=$env:ComputerName)
    #do lots of interesting and complicated stuff here
}
```

Instead of piping in all of the computers in my active directory, I would run the function with a constant input using a known computer name. For instance, consider the following script:

```
'TESTSRV01' | generate-bigreport
```

By doing this, I have accomplished several things. First of all, I should know exactly what the output will look like. If I don't, data points for a single computer might not be very hard to validate manually. Secondly, the execution of the script will undoubtedly be much quicker than if I had run it against hundreds of computers. Thirdly, if I do need to step through the code in the debugger, I won't have to trace through several iterations to get to the end of the function.

If the issue I'm troubleshooting in a script involves the interaction between data points, you can try using pairs or other sets of known computer names (or whatever kind of input is required). Using a CSV file for these types of input might be natural. For example, the following screenshot shows the contents of `TestComputers.csv`:



```
ComputerName,Description
WEBSERV01,Public Web Server
WEBSERV02,Intranet Web Server
SQLSERV01,Department SQL Server
```

With that, we can run the function in the preceding screenshot using the following script:

```
Import-csv .\TestComputers.csv | generate-bigreport
```

If, you are thinking that this is just common sense, remember that some administrators have never done any kind of programming. While this kind of thought process comes naturally to some people, it is generally learned as part of training in programming. Administrators aren't always wired the same way as developers (which is a good thing) and might need a nudge in areas where developers wouldn't necessarily.

If the function gathers its input internally, it might be possible to make slight changes (with the original source checked into source control, right?) in order to shorten the process. As an example, if we had a function called `get-computer` that returned all of the computers we managed, the code in a function might look like this:

```
function generate-bigreport{
  [Array]$computers=get-computer
  #do lots of interesting and complicated stuff here
}
```

Changing the assignment statement to be a pipeline with some filtering can have the same effect as limiting the input. The following script shows some possible options:

```
[Array]$computers=get-computer | select -first 1   #or 2
[Array]$computers=get-computer | where ComputerName -in
  'TESTSERV01','TESTSERV02'
```

Using `select-object -first n` on the output of a function is useful at times as well, when the output is lengthy. In this case, we would need to include enough output to be able to tell whether the function seems to be working properly, but not so much that it would not be practical to check each output object.

Using Tee-Object to see intermediate results

Debugging long pipelines can be tricky. While it is possible to split a pipeline over several source lines in order to set a breakpoint on a particular segment of a pipeline, it is often the cumulative results of the pipeline that are important. In cases where the end result is not what is expected, it can be helpful to be able to see what the intermediate results are in the pipeline. For instance, consider the following (incorrect) code to find the largest items in a folder structure:

```
#find largest 5 items in the directory tree
dir -recurse |
  sort-object Length |
  select-object -last 5
```

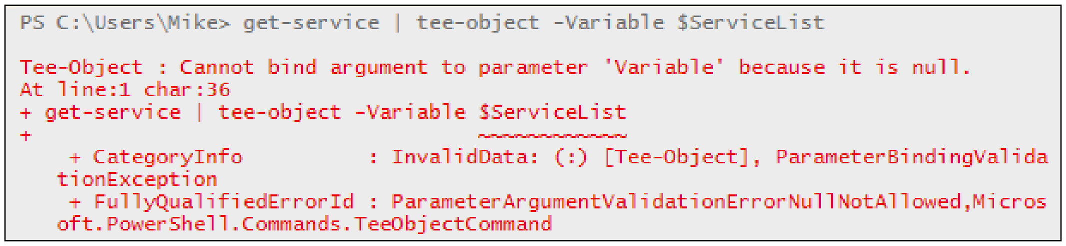
Since the output is incorrect, we can insert `Tee-Object` commands into the pipeline to save the intermediate results (after `dir` and after `sort`) into variables or files for our convenience. First, let's look at how to get the results into files using the following code:

```
#find largest 5 items in the directory tree
dir -recurse |
  tee-object -FilePath c:\temp\files.txt |
  sort-object Length |
  tee-object -FilePath c:\temp\sortedFiles.txt |
  select-object -last 5
```

The `Tee-Object` cmdlet with the `–FilePath` parameter works in a similar way to `Out-File`, in that it takes all of the objects coming in from the pipeline and outputs them to a text file. However, unlike `Out-File`, it also writes those same objects back to the pipeline for the next pipeline element to process. Saving the output to variables is easy as well, with one step that's important to remember. The following code shows the use of the `Tee-Object` cmdlet:

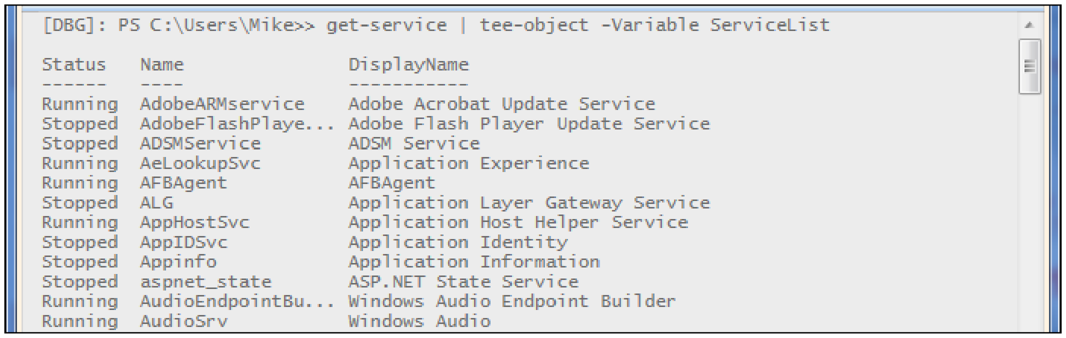
```
#find largest 5 items in the directory tree
dir -recurse |
  tee-object –Variable Files |
  sort-object Length |
  tee-object –Variable SortedFiles |
  select-object -last 5
```

The thing to remember when specifying variables in `Tee-Object` is that you only want the name of the variable (that is, without the dollar sign (`$`)). If you include the dollar sign, the objects will be placed in a variable whose name is the value of the variable. If those variables have not been set, the `Tee-Object` cmdlet will fail because the `-Variable` parameter is `$null`. This common error is illustrated in the following screenshot:



```
PS C:\Users\Mike> get-service | tee-object -Variable $ServiceList

Tee-Object : Cannot bind argument to parameter 'Variable' because it is null.
At line:1 char:36
+ get-service | tee-object -Variable $ServiceList
+                                    ~~~~~~~~~~~~
    + CategoryInfo          : InvalidData: (:) [Tee-Object], ParameterBindingValida
   tionException
    + FullyQualifiedErrorId : ParameterArgumentValidationErrorNullNotAllowed,Micros
   oft.PowerShell.Commands.TeeObjectCommand
```

The correct `-Variable` parameter would be `ServiceList` (without the `$`), as shown in the following screenshot:



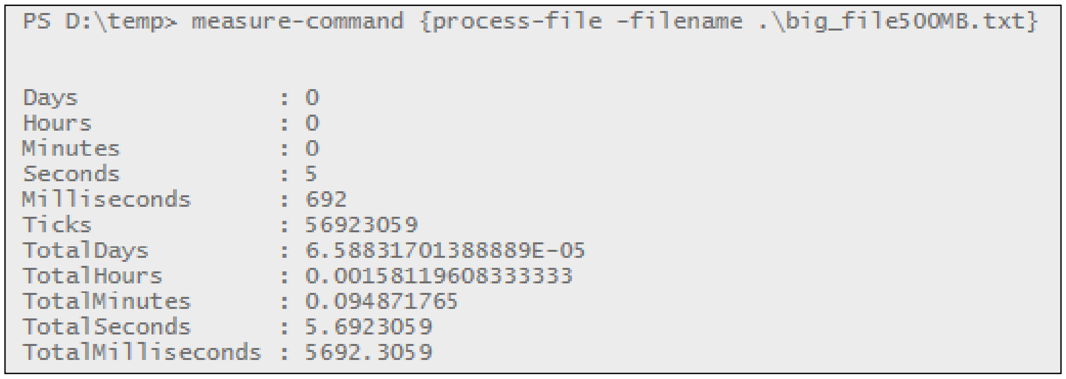
```
[DBG]: PS C:\Users\Mike>> get-service | tee-object -Variable ServiceList

Status   Name               DisplayName
------   ----               -----------
Running  AdobeARMservice    Adobe Acrobat Update Service
Stopped  AdobeFlashPlaye... Adobe Flash Player Update Service
Stopped  ADSMService        ADSM Service
Running  AeLookupSvc        Application Experience
Running  AFBAgent           AFBAgent
Stopped  ALG                Application Layer Gateway Service
Running  AppHostSvc         Application Host Helper Service
Stopped  AppIDSvc           Application Identity
Stopped  Appinfo            Application Information
Stopped  aspnet_state       ASP.NET State Service
Running  AudioEndpointBu... Windows Audio Endpoint Builder
Running  AudioSrv           Windows Audio
```

Replacing the foreach loop with the foreach-object cmdlet

When you write a function to process a file, a typical approach might look like this:

```
function process-file{
param($filename)

    $contents=get-content $filename
    foreach($line in $contents){
        # do something interesting
    }
}
```

This pattern works well for small files, but for really large files this kind of processing will perform very badly and possibly crash with an out of memory exception. For instance, running this function against a 500 MB text file on my laptop took over five seconds despite the fact that the loop doesn't actually do anything. To determine the time it takes to run, we can use the `measure-command` cmdlet, as shown in the following screenshot:


```
PS D:\temp> measure-command {process-file -filename .\big_file500MB.txt}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 5
Milliseconds      : 692
Ticks             : 56923059
TotalDays         : 6.58831701388889E-05
TotalHours        : 0.00158119608333333
TotalMinutes      : 0.094871765
TotalSeconds      : 5.6923059
TotalMilliseconds : 5692.3059
```

Note that the result is a `Timespan` object and the `TotalSeconds` object has the value we are looking for. You might not have any large files handy, so I wrote the following quick function to create large text files that are approximately the size you ask for:

```
function new-bigfile{
param([string]$path,
      [int]$sizeInMB)
    if(test-path $path){
      remove-item $path
    }
    new-item -ItemType File -Path $path | out-null
    $line='A'*78
    $page="$line`r`n"*1280000
    1..($sizeInMB/100) | foreach {$page | out-file $path -Append
      -Encoding ascii}
}
```

The code works by creating a large string using string multiplication, which can be handy in situations like this. It then writes the string to the file the appropriate number of times that are necessary. The files come out pretty close to the requested size if the size is over 100 MB, but they are not exact. Fortunately, we aren't really concerned about the exact size, but rather just that the files are very large.

A better approach would be to utilize the streaming functionality of the pipeline and use the `ForEach-Object` cmdlet instead of reading the contents into a variable. Since objects are output from `Get-Content` as they are being read, processing them one at a time allows us to process the file without ever reading it all into memory at one time. An example that is similar to the previous code is this:

```
function process-file2{
param($filename)
    get-content $filename | foreach-object{
        $line=$_
        # do something interesting
    }
}
```

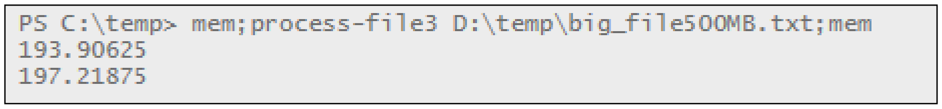
Note that since we're using the `ForEach-Object` cmdlet instead of the `foreach` loop we have to use the `$_` automatic variable to refer to the current object. By assigning that immediately to a variable, we can use exactly the same code as we would have in the `foreach` loop example (in place of the `#do something interesting` comment). In PowerShell Version 4.0, we could use the `–PipelineVariable` common parameter to simplify this code. As with all parameters where you supply the name of a variable, you don't use the dollar sign:

```
function process-file3{
param($filename)
    get-content $filename -PipelineVariable line | foreach-object{
        # do something interesting
    }
}
```

With either of these constructions, I have been able to process files of any length without any noticeable memory usage. One way to measure memory usage (without simply watching the process monitor) is to use the `Get-Process` cmdlet to find the current process and report on the `WorkingSet64` property. It is important to use the 64-bit version rather than the `WorkingSet` property or its alias: WS. A function to get the current shell's memory usage looks like this:

```
function get-shellmemory{
   (get-process -id $pid| select -expand WorkingSet64)/1MB
}
new-alias mem get-shellmemory
```

I've included an alias (`mem`) for this function to make it quicker to call on the command line. I try to avoid using aliases in scripts as a practice because they can make code harder to understand, but for command line use, aliases really are a time-saver. Here's an example of using `get-shellmemory` via its alias, `mem`:



```
PS C:\temp> mem;process-file3 D:\temp\big_file500MB.txt;mem
193.90625
197.21875
```

This shows that although the function processed a 500 MB file, it only used a little over 3 MB of memory in doing so. Combining the function to determine memory usage with `measure-command` gives us a general purpose function to measure time and memory usage:

```
function get-performance{
param([scriptblock]$block);
    $pre_mem=get-shellmemory
    $elapsedTime=measure-command -Expression $block
    $post_mem=get-shellmemory
    write-output "the process took $($elapsedTime.TotalSeconds)
seconds"
    write-output "the process used $($post_mem - $pre_mem) megabytes
of memory"
}
new-alias perf get-performance
```

One thing to note about measuring memory this way is that since the PowerShell host is a .NET process that is garbage-collected, it is possible that a garbage-collection operation has occurred during the time the process is running. If that happens, the process may end up using less memory than it was when it started. Because of this, memory usage statistics are only guidelines, not absolute indicators. Adding an explicit call to the garbage collector to tell it to collect will make it less likely that the memory readings will be unusual, but the situation is in the hands of the .NET framework, not ours.

You will find that the memory used by a particular function will vary quite a bit, but the general performance characteristics are the important thing. In this section, we're concerned about whether the memory usage grows proportionally with the size of the input file. Using the first version of the code that used the `foreach` loop, the memory use did grow with the size of the input file, which limits the usefulness of that technique.

For reference, a summary of the performance on my computer using the `foreach` loop and the `ForEach-Object` cmdlet is given in the following table:

Input size	Loop time	Loop memory	Cmdlet time	Cmdlet memory
100 MB	1.1s	158 MB	1.5s	1.5 MB
500 MB	6.1s	979 MB	8.7s	12.9 MB
1 GB	38.5s	1987 MB	16.7s	7.4 MB
2 GB	Failed		51.2s	8.6 MB
4 GB	Failed		132s	12.7 MB

While these specific numbers are highly dependent on the specific hardware and software configuration on my computer, the takeaway is that by using the `ForEach-Object` cmdlet you can avoid the high memory usage that is involved in reading large files into memory.

Although the discussion here has been around the `get-content` cmdlet, the same is true about any cmdlet that returns objects in a streaming fashion. For example, `Import-CSV` can have exactly the same performance characteristics as `Get-Content`. The following code is a typical approach to reading CSV files, which works very well for small files:

```
function process-CSVfile{
param($filename)
    $objects=import-CSV $filename
    foreach($object in $objects){
        # do something interesting
    }
}
```

To see the performance, we will need some large CSV files to work with. Here's a simple function that creates CSV files with approximately the right size that will be appropriate to test. Note that the multipliers used in the function were determined using trial and error, but they give a reasonable 10-column CSV file that is close to the requested size:

```
function new-bigCSVfile{
param([string]$path,
      [int]$sizeInMB)
    if(test-path $path){
      remove-item $path
    }
    new-item -ItemType File -Path $path | out-null
    $header="Column1"
```

```
        2..10 | foreach {$header+=",Column$_"}
        $header+="`r`n"
        $header | out-file $path -encoding Ascii
        $page=$header*12500

        1..($sizeInMB) | foreach {$page | out-file $path -
          Append -Encoding ascii}
    }
```

Rewriting the `process-CSVfile` function to use the streaming property of the pipeline looks similar to the rewritten `get-content` example, as follows:

```
    function process-CSVfile2{
    param($filename)
        import-CSV $filename |
            foreach-object -pipelinevariable object{
            # do something interesting
            }
    }
```

Now that we have the `Get-Performance` function, we can easily construct a table of results for the two implementations:

Input size	Loop time	Loop memory	Cmdlet time	Cmdlet memory
10 MB	9.4s	278 MB	20.9s	4.1 MB
50 MB	62.4s	1335 MB	116.4s	10.3 MB
100 MB	165.5s	2529 MB	361.0s	21.5 MB
200 MB	Failed		761.8s	25.8 MB

It's clear to see that trying to load the entire file into memory is not a scalable operation. In this case, the memory usage is even higher and the times much slower than with `get-content`. It would be simple to construct poorly executing examples with cmdlets such as `Get-EventLog` and `Get-WinEvent`, and replacing the `foreach` loop with the `ForEach-Object` cmdlet will have the same kinds of results in these as well. Having tools like the `Get-Performance` and `Get-ShellMemory` functions can be a great help to diagnosing memory scaling problems like this. Another thing to note is that using the pipeline is slower than using the loop, so if you know that the input file sizes are small the loop might be a better choice.

Further reading

For more information on the topics covered in this chapter, you can go through the following references:

- `get-help get-host`
- `get-help about_profiles`
- `get-help get-member`
- `get-help set-psdebug`
- `get-help about_prompts`
- `get-help get-eventlog`
- `get-help get-winevent`
- `get-help measure-object`
- `get-help write-eventlog`
- `get-help new-eventlog`
- `get-help about_commonparameters`
- `get-help about_preference_variables`
- `get-help about_functions_advanced`
- `get-help about_parameters_default_values`
- `get-help import-csv`
- `get-help select-object`
- `get-help tee-object`
- `get-help measure-command`
- `get-help foreach-object`
- `get-help about_foreach`
- `get-help get-content`
- `get-help new-alias`

Summary

This chapter has focused on things that you can do when running the program, including traditional debugging in both the ISE and the console. We spent some time looking at error messages and how to make them more readable in order to help us take the time to look carefully at all of the information contained in them. We also looked at a couple of issues involving large input sets, namely, how to more easily debug a process by reducing the input and also how to alleviate memory issues with large file input by effectively using the pipeline.

In the next chapter, we will explore some things to look out for in your code that might indicate an opportunity to clean the code up in some way. Some of these red flags, known as *code smells*, are common among programming languages, but some are very specific to PowerShell code.

8
PowerShell Code Smells

No code is ever perfect, and we must realize that throughout the act of writing code we will be making choices continually. As we mature as developers or scripters, we become more confident in our choices and will tend to make decisions that are more well-informed and lead to better, more stable solutions. As we look back on scripts that we wrote in the past, we will almost certainly find opportunities for improvement. Depending on the source, we might also find the same kinds of issues in code that we obtain from other sources such as the Internet. In this chapter, we will consider some of the things to look for in code that might indicate some improvements may be applicable. In particular, we will discuss the following:

- What are code smells?
- Language-agnostic code smells
- Why are there PowerShell-specific code smells?
- Missing `Param()` statements
- Homegrown common parameters
- Unapproved verbs
- Accumulating output objects
- Sequences of assignment statements
- Using untyped or `[object]` parameters
- Static analysis tools (ScriptCop and Script Analyzer)

Code smells

A **code smell** is a feature of code that indicates something may need to be rewritten. Just as a smell in the refrigerator or pantry gives us a clue that something may have gone bad, a code smell tells us that the code may not be written as well as it perhaps should have been. Code smells are not errors as such, in that they don't mean the code functions incorrectly. Instead, they are indicators that the code might not be as flexible or easy to maintain as it could be, or even that it doesn't take advantage of language features that would make the code easier to understand and perform better. As I mentioned in the introduction to the chapter, all coding involves choices, and the choice to address a code smell is not always cut-and-dry. It may be that the code in question has needed no maintenance and is used infrequently, resulting in a questionable return on investment for fixing the code. It may be that the design of the code was specifically non-standard for some reason, whether that was a business requirement or other requirement.

Another source of code smells is the continual progress of language releases. For example, programs that were written in C# 1.0 wouldn't have been able to take advantage of features added to the language in later versions. Similarly, scripts written in PowerShell 1.0 may have included code to implement features added to the PowerShell language in more recent versions. When we view or review old code, we might find that they "smell old."

Code smells, best practices, antipatterns, and technical debt

Three other code-related observations are best practices, antipatterns, and technical debt. A best practice is an industry standard, recognized pattern that has been shown to be the best way to accomplish something in code. Examples of a best practice are using meaningful variable names and making code modular. An antipattern is a common way of performing an operation or structuring code that is likely to cause problems, especially where there is a best practice or set of best practices that are applicable. A typical antipattern is to construct an SQL statement which includes user input using string concatenation rather than using parameterized SQL statements. Technical debt is a term that describes the effort needed to change an implementation into what would be considered a proper, complete implementation. Technical debt can include outdated frameworks or language versions, poorly documented code, tightly-coupled modules, and a number of other things. Code smells, best practices, and antipatterns are clearly similar and their meanings overlap quite a bit. Debt is a good metaphor because the longer the debt is unaddressed, the larger the debt becomes, similar to how interest accumulates. The following list introduces the idea of each observation:

- Best practices are how the industry says it should be done

- Antipatterns are how it should not be done
- Code smells say that it might be wrong
- Technical debt is how hard it will be to fix

Language-agnostic code smells

Some things look wrong no matter what language they're written in. For instance, if you have 200 lines of code in a block, it's always going to be a sign that some effort might pay off in terms of rewriting it using smaller, more modular code. Also, if you see several blocks of code that are identical, or even very similar, it will probably be worthwhile from a maintenance standpoint to factor that code out into a function. Some other code smells that are very common are:

- A very large parameter list
- Overly short identifiers
- Extremely long identifiers
- Deeply-nested loops or conditionals
- Multiple-personality functions (that is, functions that do more than one thing)

One thing that is worthwhile to point out is that these are all somewhat subjective. For instance, what is a large parameter list? Some PowerShell cmdlets have dozens of parameters. Do they smell? Everyone will have their own perception of when that line has been crossed. Even when a usage isn't excessive, the fact that the question is asked may lead to useful refactoring or rewriting.

PowerShell-specific code smells

As you have learned throughout this book, PowerShell is an interesting creation. It is a powerful scripting language that can be used to write complex solutions as well as short and quick scripts. The language design is also somewhat unique because of the PowerShell pipeline, which is a central feature. Finally, the scope of the PowerShell language has grown tremendously over the course of the last seven or eight years, so code that was written early on will probably look primitive in light of the latest version. Here some of the large changes to the language and environment that have occurred in the various PowerShell versions:

- Introduced in Version 1.0:
    - Functions, filters, scripts, and pipeline

- Introduced in Version 2.0:
    - Modules
    - PowerShell remoting
    - PowerShell ISE
    - Advanced functions
    - Background jobs
    - Eventing
- Introduced in Version 3.0:
    - Workflows
    - Scheduled jobs
    - CIM cmdlets
    - Updateable help
    - Simplified `Where-Object` syntax
- Introduced in Version 4.0:
    - Desired state configuration
    - The `PipelineVariable` parameter
    - The `$PSItem` automatic variable
    - The `.Where()` and `.ForEach()` methods

Missing Param() statements

Parameters to functions in PowerShell can be defined in two ways: either as a list following the name of the function or in a `Param()` statement. In Version 1.0 of PowerShell, there was no compelling reason to use a `Param()` statement in a function, although `Param()` statements were required to define parameters to scripts and scriptblocks. Since most programming languages use a parameter list following the function name, it was natural at that time to skip the use of `Param()` statements in functions. With the introduction of advanced functions in PowerShell Version 2.0, there was suddenly a reason to use a `Param()` statement, as the `CmdletBinding()` attribute binds to the `Param()` statement. That is, in order to include the `CmdletBinding()` attribute, you needed to include a `Param()` statement.

It is not necessary that all functions be advanced functions, but the effort required to make the change is trivial compared to the advantages. Common parameters, parameter checking, validation, and risk mitigation support are straightforward implementations given an advanced function. The only change required to make an advanced function is to replace the following code snippet:

```
function Get-Something($thing){
#get the thing
}
```

It should be replaced with:

```
function Get-Something{
[CmdletBinding()]
Param($thing)
#get the thing
}
```

When I see functions like the first code snippet, missing a `Param()` statement, my first thought is that the code was written back in the days of PowerShell Version 1.0. The process of moving the list of parameters into a `Param()` statement and (usually) adding the `CmdletBinding()` attribute is straightforward and almost always correct. Be aware that if you are using the `$args` automatic variable, the code will need to be rewritten to be an advanced function, as `$args` lists the undeclared parameters passed in and advanced functions do not allow undeclared parameters. An option to replace `$args` would be to use the `ValueFromRemainingArguments` parameter. Consider a case where the original function looked like this:

```
function Get-Args{
$args
}
```

The rewritten function could look like this:

```
function Get-Args2{
[CmdletBinding()]
Param([Parameter(ValueFromRemainingArguments=$true)]
      $myargs)
$myargs
}
```

Homegrown common parameters

Also in the time of PowerShell Version 1.0, the only way to get native support for common parameters (for example, `-Verbose` and `-ErrorAction`) was to write a cmdlet using managed code. Since scripters, in general, are not C# or VB.NET programmers, there was a tendency at that time to manually implement the common parameters. For instance, it wasn't uncommon to find code like this:

```
function Get-Stuff{
Param($stuffID,[switch]$help)

    if($help){
      write-host "Usage: get-stuff [-stuffID] ID"
      write-host "Retrieves a list of stuff which matches"
      write-host "the given stuffID"
      return
    }
    #get the stuff
}
```

This was not an unapproved method in fact. Here is a blog post from Jeffery Snover advocating implementing the `-whatif`, `-Confirm`, and `-Verbose` parameters in script:

`http://blogs.msdn.com/b/powershell/archive/2007/02/25/supporting-whatif-confirm-verbose-in-scripts.aspx#10555359`

The post even contains a note explaining how important this method is:

<This is a super-important issue so you should definitely start using this in your scripts that you share with others (that have side effects on the system).
Please try it out and blog about it to others so that it becomes a community norm.
Thanks-jps>

Mr. Snover made a very good point, which was crucial during the early days of PowerShell, about the goal of script cmdlets being as powerful as cmdlets written in managed code. He also did a great job of illustrating how to use `[ref]` parameters in PowerShell. I certainly won't contradict the creator of PowerShell, whereas this was clearly a promoted practice in Version 1.0 of PowerShell, it has become a code smell starting with Version 2.0.

With the language changes introduced in PowerShell 2.0, we know that we can use `[CmdletBinding()]` by itself to allow access to the common parameters. We have also seen that adding `SupportsShouldProcess=$true` to the `CmdletBinding()` attribute gives us the additional risk mitigation parameters of `-Whatif` and `-Confirm`. Finally, we know that help in a function should be created using comment-based help.

You might ask yourself what the risk of implementing your own parameters is or what problems might arise from this practice. There are a few good principles that should be mentioned:

- Code that you can omit is code that you don't have to troubleshoot
- Code written by the PowerShell team is documented and maintained by Microsoft
- Delivered code is always available on a given machine

With these principles in mind, it should be clear that you shouldn't try to implement features of PowerShell yourself. In instances where a feature you have implemented is added to the PowerShell language, you should consider refactoring to use the delivered feature instead.

Unapproved verbs

In *Chapter 1*, *PowerShell Primer*, and *Chapter 4*, *PowerShell Professionalism*, we discussed the approved list of verbs and how the module system will issue a warning when a module exports a function with an unapproved verb. We saw that there was a switch called `-DisableNameChecking` that suppressed the warning. Not seeing the warning is acceptable if we don't have access to the source code (as is the case with the SQLPS module discussed in *Chapter 4*, *PowerShell Professionalism*), but in our own code we are able to be more careful. In situations where we are creating a new module, consideration can be made for what functions are performed. By consulting the list of approved verbs provided by `Get-Verb`, we should have no issues creating an appropriately named set of functions.

The instance where we are bound to run into an issue, as is the case with most of the code smells we are discussing, is where we find code that is already in use and that violates this principle. Given a well-organized codebase, it is possible that we could simply rename the offending functions throughout every script in which they appear. In reality, however, there is a risk involved in this kind of operation. What if there was a file that was missed, or if a server was offline for maintenance that contained scripts that use the functions? In these situations, a different approach is usually preferable.

The solution to this problem is to rename the function with a proper verb in the module and create an alias for the function that has the original name. For instance, this code uses the unapproved verb "`perform`":

```
function perform-operation{
    #legacy code
}
```

The replacement for this function would look something like this:

```
function invoke-operation{
    #legacy code
}
new-alias -name perform-operation -value invoke-operation
Export-ModuleMember -Alias perform-operation
```

The function now uses the approved verb "`invoke`", and we have created an alias with the original name, `perform-operation` so that existing code will be able to access the function. It is crucial that an `Export-ModuleMember` function call be added to the module to export the alias, as aliases are not exported from a module by default. Assuming that this was the only improperly named function in the module, the module will now import with no warnings.

Accumulating output objects

In Chapter 7, *Reactive Practices – Traditional Debugging*, we spent a lot of time discussing replacing the `foreach` keyword (loop) with the `ForEach-Object` cmdlet. In that section, we were concerned with reading a large amount of data from a file into memory rather than processing it one line at a time. In this section, we will consider the converse, that is, functions that accumulate the objects they are going to output in a container such as an array.

The pipeline is a very powerful and unique feature of the PowerShell language. Executing the cmdlets in the pipeline at the same time as input is available to them, instead of running them sequentially, allows PowerShell scripts to deal with huge amounts of data without incurring a heavy memory footprint. However, we need to be careful if we are going to take advantage of this. Consider the following somewhat contrived function to retrieve the list of services running on a list of computers:

```
function get-RunningServices{
Param([string[]]$computerName)

    $RunningServices=@()
    foreach($computer in $computername){
        $RunningServices+=
            [array](get-service -ComputerName $computer |
                    where Status -eq 'Running')
    }
    return $RunningServices
}
```

While a list of services is probably not going to become a memory issue unless we are processing a list of thousands of computers, it is clear that there is no need to hold these objects in memory. Another consideration is that as written, no objects will be available downstream in the pipeline until all of the computers have been processed. If the pipeline contains `select-object -first 5`, for instance, the code will waste a lot of time retrieving services from every computer rather than stopping after the first five service objects have been emitted. Rewriting the function to take advantage of the pipeline could look like this:

```
function get-RunningServices2{
Param([string[]]$computerName)

    foreach($computer in $computername){
            get-service -ComputerName $computer |
                    where Status -eq 'Running'
    }
}
```

It is instructive to note that the rewriting here only involved removing code. As mentioned in the *Homegrown common parameters* section, code that you can omit is code that you don't have to troubleshoot. So in addition to performing much faster and working with downstream cmdlets, it should require less troubleshooting.

Sequences of assignment statements

As I have mentioned several times, scripting in PowerShell is somewhat different than programming in other languages. A programmer who isn't comfortable with the pipeline might easily fall into the trap of writing code in a style that matches the imperative programming language that he is most familiar with. For instance, filtering and sorting a list of files in a directory could easily be seen as a sequence of operations, as follows:

1. Get the list of files.
2. Filter the list of files to match the given criteria.
3. Sort the remaining list of files.

This thought process could lead to a PowerShell script that looks like this:

```
function get-sortedFilteredList{
Param($folder,$extension)

    $files = dir $folder
    $matchingFiles = where-object -InputObject $files -
      FilterScript {$_.Extension -eq $extension}
```

```
        $sortedMatches = sort-object -InputObject $matchingFiles -
          Property Name

        return $sortedMatches
    }
```

While this is a correct solution to the problem in some senses, it is certainly not how the problem would be solved in idiomatic PowerShell. A more typical solution would look like this:

```
function get-sortedFilteredList2{
Param($folder,$extension)
    dir $folder |
      where-object {$_.Extension -eq $extension} |
      sort-object -Property Name
}
```

The PowerShell-style solution is not only shorter, but also contains no local variables (which is worth noting). Because it doesn't contain any variables, it is much less likely that there will be a problem with a typo of a variable name or a type mismatch in an assignment statement. Again, by removing code we have less to worry about.

There are times when a pipeline needs to be broken up for readability purposes, and it's also possible that in a complex pipeline, it might be preferable to have discrete steps in order to simplify debugging. In general though, using pipelines will allow you to code more quickly and with fewer errors.

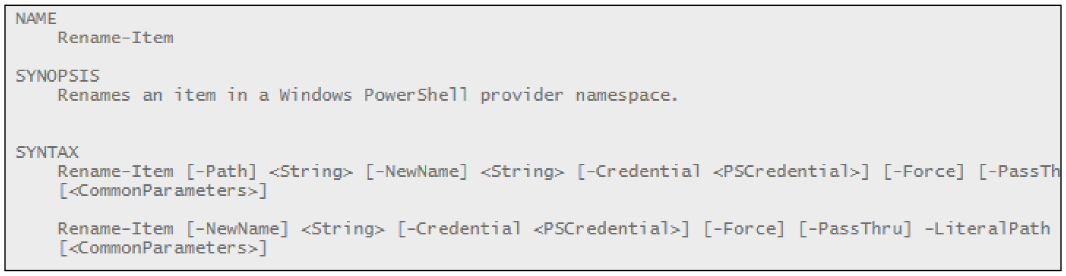
Using untyped or [object] parameters

When writing a function with flexibility in mind, we might be tempted to omit a type on a parameter in order to allow the user to supply different kinds of objects as arguments. PowerShell definitely allows for this, and in PowerShell Version 1.0 this was a common practice. With PowerShell Version 2.0, and with the introduction of advanced functions and parameter sets, we have a better option.

Recall that PowerShell doesn't allow the overloading of functions, where multiple function definitions exist with distinct signatures. If it did, some built-in cmdlets would have over a dozen different definitions. Instead, the concept of parameter sets, or mutually exclusive sets of parameters, is provided. Each parameter set corresponds to a usage pattern of the function or cmdlet. For instance, the help for `Rename-Item` shows two parameter sets:

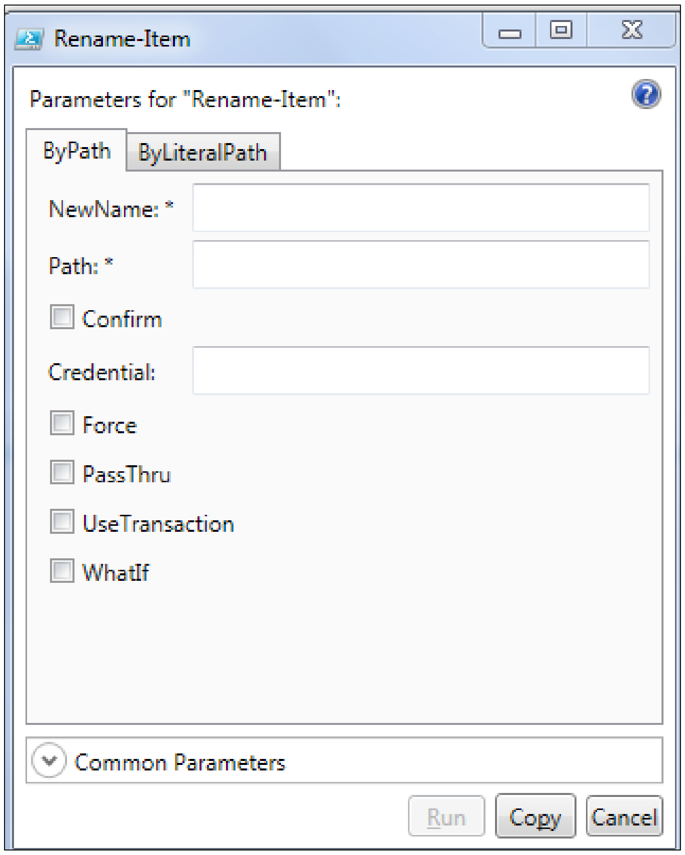
- Using a standard path
- Using a literal path

The following screenshot shows the name, synopsis, and syntax of `Rename-Item`:



```
NAME
    Rename-Item

SYNOPSIS
    Renames an item in a Windows PowerShell provider namespace.


SYNTAX
    Rename-Item [-Path] <String> [-NewName] <String> [-Credential <PSCredential>] [-Force] [-PassTh
    [<CommonParameters>]

    Rename-Item [-NewName] <String> [-Credential <PSCredential>] [-Force] [-PassThru] -LiteralPath
    [<CommonParameters>]
```

Trying to determine which parameters are unique to each parameter set can be very frustrating. Looking at the command in the `show-command` window in the ISE gives a graphical view of the two parameter sets, each represented in a tab. I find this to be a good way of trying to figure out what the parameter sets mean. The following screenshot shows the two parameter sets of `Rename-Item`:



Rename-Item

Parameters for "Rename-Item":

ByPath | ByLiteralPath

NewName: *

Path: *

Confirm

Credential:

Force

PassThru

UseTransaction

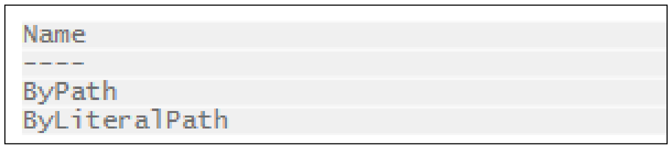
WhatIf

Common Parameters

We can also find the parameter sets programmatically by inspecting the command metadata, as shown in the following script:

```
get-command rename-item|
    select-object -expand ParameterSets |
    select-object Name
```

As expected, the results match the labels on the tabs in the `show-command` example, as shown in the following screenshot:



```
Name
----
ByPath
ByLiteralPath
```

Using parameter sets gives the user an idea of how the function is intended to be used. In this case, the `-Path` and `-LiteralPath` parameters are supplying the same information (a location), but in slightly different ways.

Compare this to the following parameter declarations from a function in an open source project. I'm not picking on the developer because it's a project that I've been associated with for a long time.

```
function Get-SqlDatabase
{
    param(
    [Parameter(Position=0, Mandatory=$true)] $sqlserver,
    [Parameter(Position=1, Mandatory=$false)] [string]$dbname,
    [Parameter(Position=2, Mandatory=$false)] [switch]$force
    )
```

Because the `$sqlserver` parameter is declared as mandatory, you have to pass it in. Unfortunately, there is no guidance on what type of object is expected to be supplied, as is the case with the `$dbname` and `$force` parameters. Inspecting the body of the function gives us some insight into how the `$sqlserver` parameter is used:

```
    switch ($sqlserver.GetType().Name)
    {
        'String' { $server = Get-SqlServer $sqlserver }
        'Server' { $server = $sqlserver }
        default { throw 'Get-SqlDatabase:Param `$sqlserver must be a
String or Server object.' }
    }
```

From this we can see that we're expected to supply either a string or a `Server` object, and that anything else will cause an exception to be thrown. We're missing a few good opportunities here by using parameter sets and specifically-typed parameters:

- We don't get any guidance when we use `Get-Help` on what type of objects to pass
- We can't tell from the parameter list that there are two different ways to call the function
- We bypass PowerShell's extensive type-conversion facilities
- The error message is not a standard error message and is not localized

In this case, the server class (`[Microsoft.SqlServer.Management.Smo.Server]`) contains a constructor that takes a single string argument naming the server, so one option we would have is to simply use the default parameter set (that is, not specify any parameter sets) and specify that the parameter is of this specific type. If a string was passed, the engine would instantiate a `Server` object using the string and the function would have had a correctly-typed object. However, this code is using a separate function (`get-sqlserver`) to convert from a string to a `Server` object, so we will need to use parameter sets to reproduce the functionality. We need to create a `$sqlserverName` parameter and assign the `$sqlserver` and `$sqlserverName` parameters to different parameter sets. Then, in the function body, we check `$PSCmdlet.ParameterSetName` to determine whether a server name was passed rather than a server object and call the conversion function, as shown in the following code snippet:

```
function Get-SqlDatabase
{
    param(
    [Parameter(Position=0, Mandatory=$true,ParameterSet='Server')]
      $sqlserver,
    [Parameter(Position=0,
      Mandatory=$true,ParameterSet='ServerName')] $sqlserverName,
    [Parameter(Position=1,
      Mandatory=$false,ParameterSet='ServerName')]
        [string]$dbname,
    [Parameter(Position=2, Mandatory=$false)] [switch]$force
    )

if ($PsCmdlet.ParameterSetName -eq 'ServerName'){
  $sqlserver=get-sqlserver $sqlserverName
}
```

With that change, we have changed the function so that we can leverage PowerShell's parameter handling capabilities to give type guidance, type checking, and documentation in the form of a more useful help display.

Using parameter sets is a powerful technique that gives a lot of control over the way parameters are passed into a function. Giving up this control by using `[object]` or untyped parameters inevitably leads to more code and frustrated users.

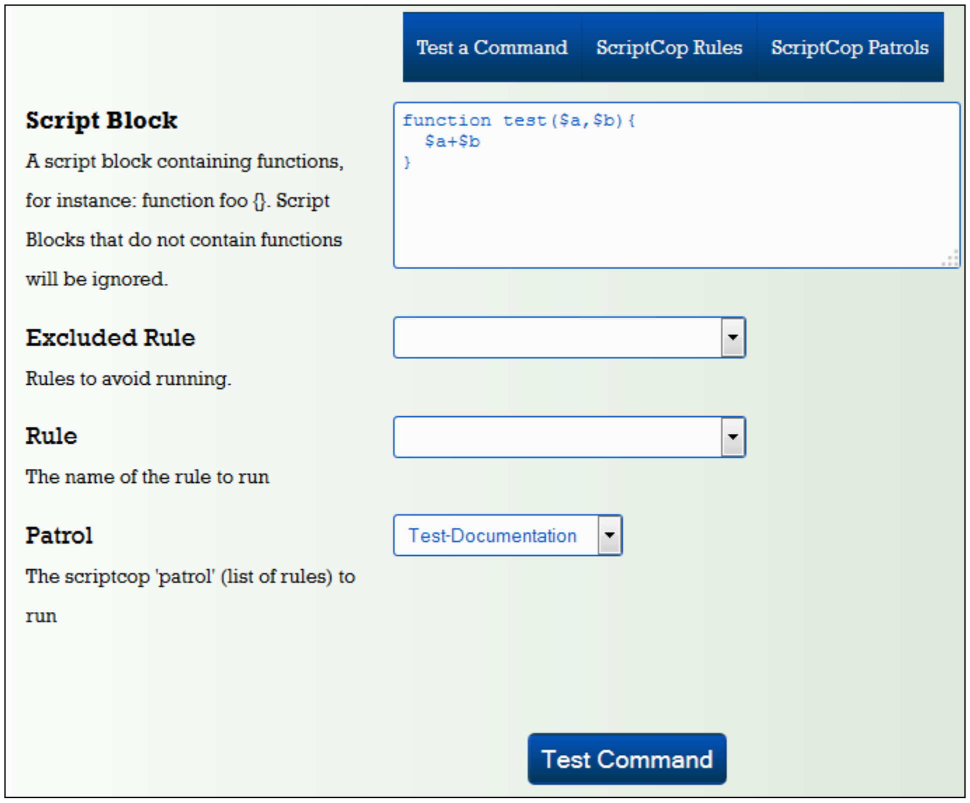
Static analysis tools – ScriptCop and Script Analyzer

This chapter has focused on things that we can see when we look at code in a code review situation. Such observations are necessarily somewhat subjective and open to disagreement. Another approach to the problem of analyzing code is to use a static analysis tool. Static analysis tools read the source code and apply a set of rules to determine places that the rules are broken. A report from such a tool gives very quick input into the quality of the code.

Some languages have a long history of static analysis tools. The C language, for example, has lint. If you're using C#, you have probably heard of StyleCop and FxCop. PowerShell has two static analysis tools: ScriptCop and Script Analyzer.

ScriptCop

ScriptCop is a tool created by Start-Automating.com in 2011 and can be found at `http://scriptcop.start-automating.com`. It can be used to test a function or module either online, through a web form, or through a cmdlet interface. ScriptCop defines a number of rules that can be tested as well as groups of rules, which it calls *patrols*. The web form is simple to use: copy the code into the form, select the rules or patrols to execute, and optionally a rule to exclude, and click on the **Test Command** button, as shown in the following screenshot:



Script Block

A script block containing functions, for instance: function foo {}. Script Blocks that do not contain functions will be ignored.

```
function test($a,$b){
  $a+$b
}
```

Excluded Rule

Rules to avoid running.

Rule

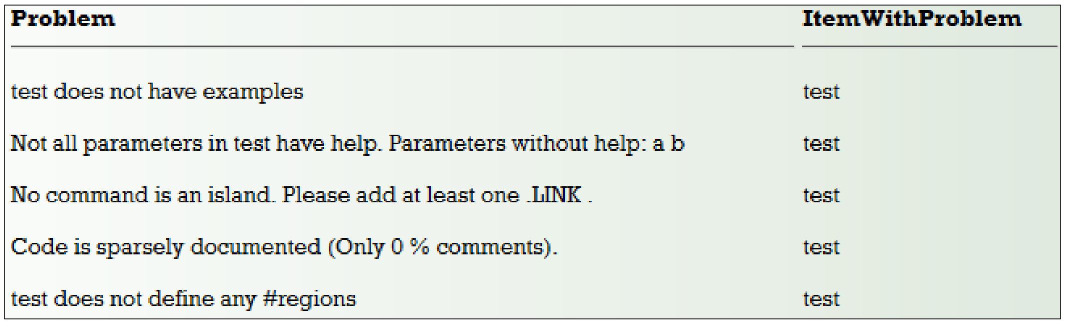
The name of the rule to run

Patrol

The scriptcop 'patrol' (list of rules) to run

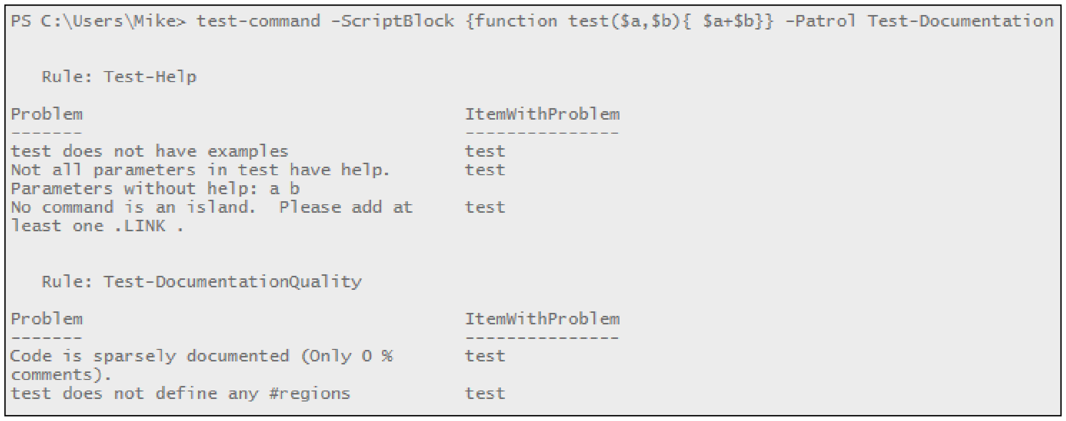
Test-Documentation

Here, I am passing in a simple function and running the `Test-Documentation` patrol against it. As expected, this example turns up a number of problems:



Problem	ItemWithProblem
test does not have examples	test
Not all parameters in test have help. Parameters without help: a b	test
No command is an island. Please add at least one .LINK .	test
Code is sparsely documented (Only 0 % comments).	test
test does not define any #regions	test

The cmdlet interface is through a module called ScriptCop that defines the rules as functions and also exports a function called `Test-Command`. We can easily use this function to reproduce the result we got on the website, as shown in the following screenshot:



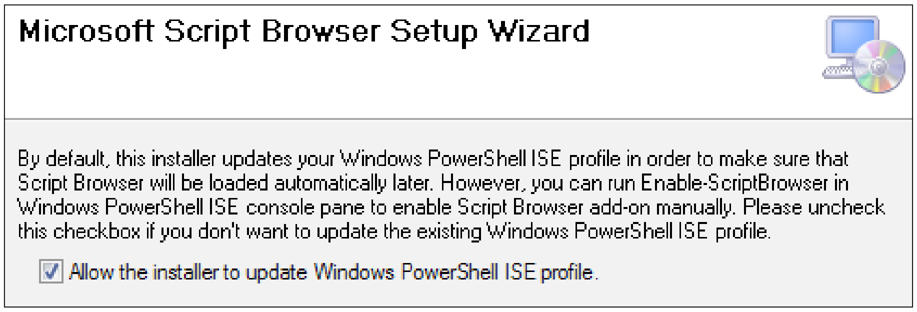
```
PS C:\Users\Mike> test-command -ScriptBlock {function test($a,$b){ $a+$b}} -Patrol Test-Documentation


   Rule: Test-Help

Problem                                      ItemWithProblem
-------                                      ---------------
test does not have examples                  test
Not all parameters in test have help.        test
Parameters without help: a b
No command is an island.  Please add at      test
least one .LINK .


   Rule: Test-DocumentationQuality

Problem                                      ItemWithProblem
-------                                      ---------------
Code is sparsely documented (Only 0 %        test
comments).
test does not define any #regions            test
```

ScriptCop defines a number of rules (currently 14 for functions and 3 for modules) and includes source code for them. The rules are advanced functions that inspect the command metadata and a list of tokens provided by the PowerShell parser.

There are a number of really strong points for ScriptCop. First, the large number of rules means that the analysis is more thorough. Second, the fact that rules and patrols are written in PowerShell means that you can write your own, or you can customize the delivered rules and patrols to match your preferences. Third, ScriptCop has a PowerShell interface, so it could be applied automatically to a library of functions and modules.

Script Analyzer

Script Analyzer is a recent arrival in the PowerShell world, showing up on TechNet in May 2014 along with the Script Browser. Script Analyzer is a PowerShell ISE add-on, that is, a graphical pane that can be installed into ISE and can interact with the ISE. The download link for the Script Browser and Script Analyzer is `http://www.microsoft.com/en-us/download/details.aspx?id=42525`. The download is an installer, which includes a note about execution policies as well as this interesting page:

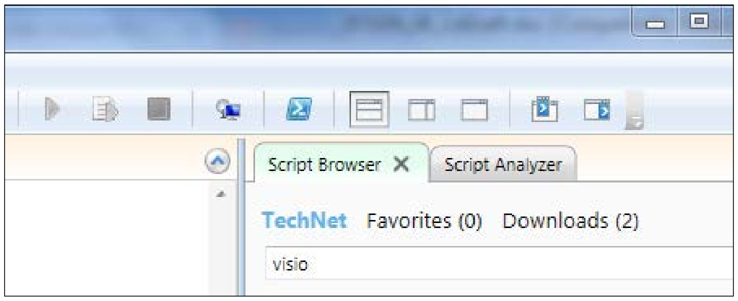


Microsoft Script Browser Setup Wizard

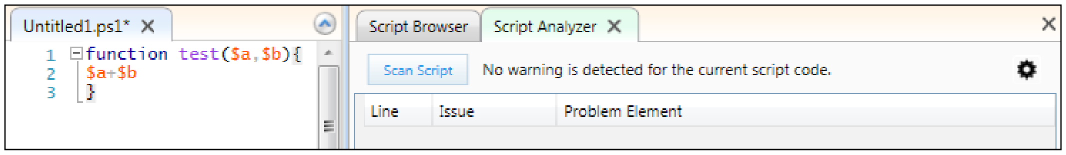
By default, this installer updates your Windows PowerShell ISE profile in order to make sure that Script Browser will be loaded automatically later. However, you can run Enable-ScriptBrowser in Windows PowerShell ISE console pane to enable Script Browser add-on manually. Please uncheck this checkbox if you don't want to update the existing Windows PowerShell ISE profile.

☑ Allow the installer to update Windows PowerShell ISE profile.

After the installation is complete, starting the ISE shows two new add-ons, as shown in the following screenshot:



The Script Browser gives an interesting searchable interface to the TechNet PowerShell gallery, but we're interested in the **Script Analyzer** tab. It features a button labeled **Scan Script**, a gear button for options, and a grid for results. Typing the same function we used in the ScriptCop section into the ISE and pressing the scan button, unfortunately, doesn't give us any results, as shown in the following screenshot:

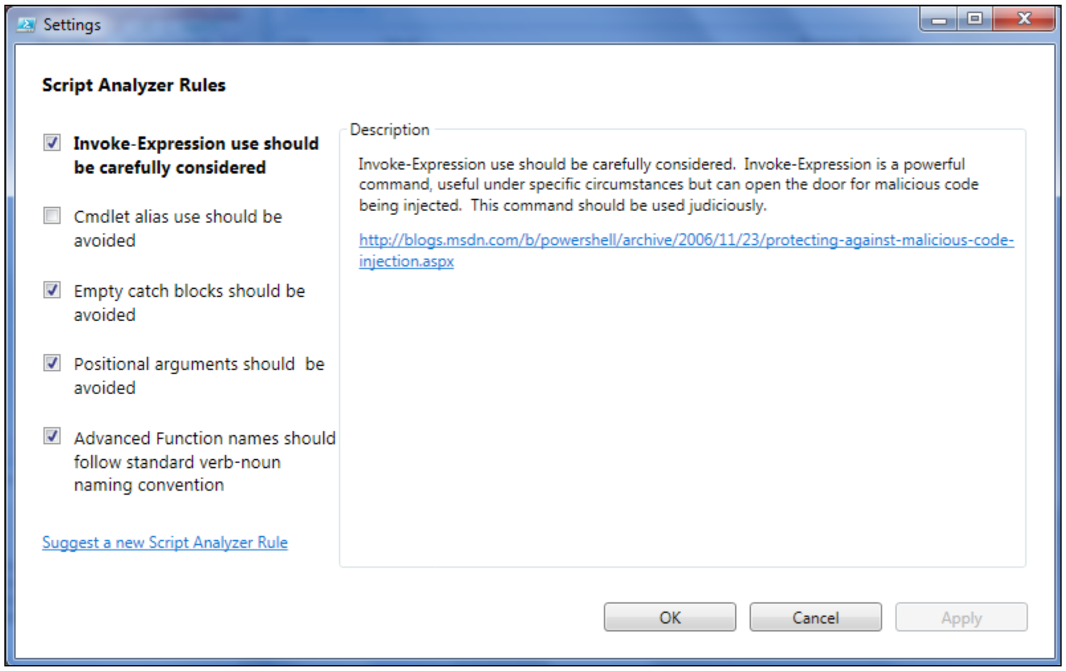


```
function test($a,$b){
$a+$b
}
```

No warning is detected for the current script code.

Line	Issue	Problem Element

Clicking on the gear to see the options shows us the reason for this, namely that there aren't many rules implemented by the analyzer at this point.



Script Analyzer Rules

- [✓] **Invoke-Expression use should be carefully considered**
- [ ] Cmdlet alias use should be avoided
- [✓] Empty catch blocks should be avoided
- [✓] Positional arguments should be avoided
- [✓] Advanced Function names should follow standard verb-noun naming convention

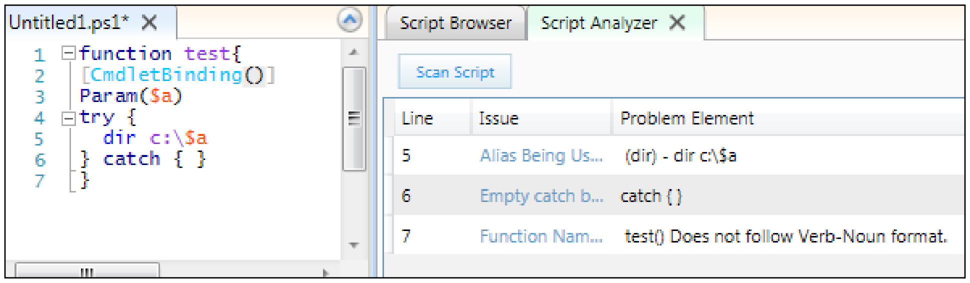
Suggest a new Script Analyzer Rule

Description

Invoke-Expression use should be carefully considered. Invoke-Expression is a powerful command, useful under specific circumstances but can open the door for malicious code being injected. This command should be used judiciously.

http://blogs.msdn.com/b/powershell/archive/2006/11/23/protecting-against-malicious-code-injection.aspx

Looking through the rules, we can create a more complex example function that triggers some of the rules, as shown in the following screenshot:

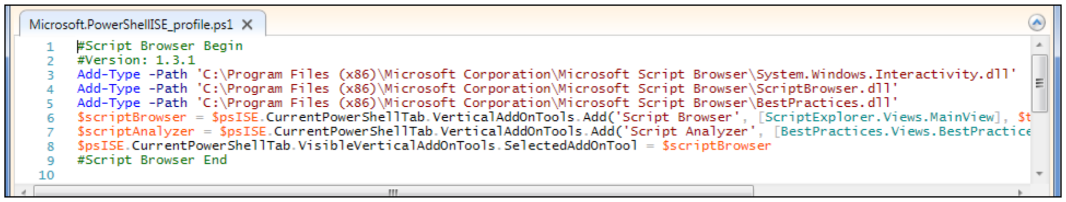


```
function test{
[CmdletBinding()]
Param($a)
try {
    dir c:\$a
} catch { }
}
```

Line	Issue	Problem Element
5	Alias Being Us...	(dir) - dir c:\$a
6	Empty catch b...	catch { }
7	Function Nam...	test() Does not follow Verb-Noun format.

The tool caught three of the rules, but apparently missed the positional argument in line 5. It is good to note that on the options page there is a link to suggest new rules. The forum that the link points to already has 14 suggestions for rules, so there's hope that the script analyzer will mature into a more detailed tool. The same forum can be used to report bugs, as I have for the positional parameter being missed in the script as illustrated in the preceding screenshot.

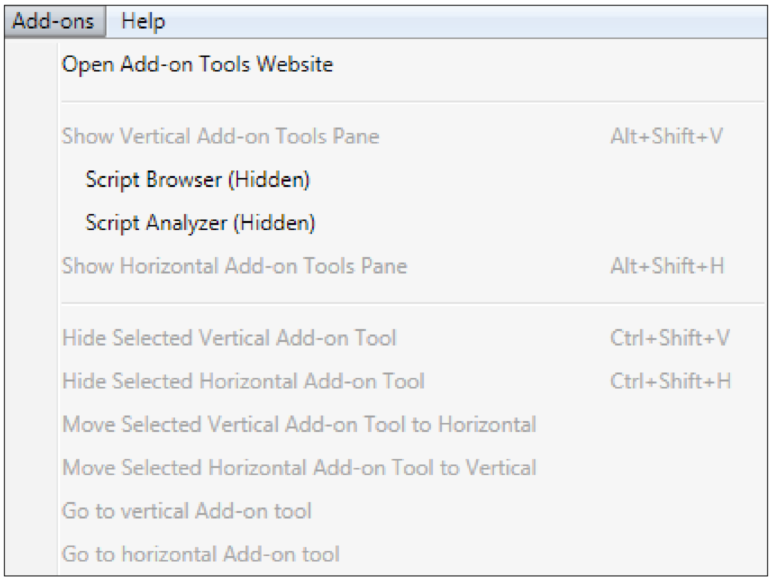
The installation procedure asked about modifying the ISE profile, so let's take a look at those changes. First, the ISE profile is called `Microsoft.PowerShellISE_profile.ps1` and is found in the `WindowsPowerShell` folder under the `MyDocuments` folder. Looking in that file, we find the following lines:



Microsoft.PowerShellISE_profile.ps1

```
#Script Browser Begin
#Version: 1.3.1
Add-Type -Path 'C:\Program Files (x86)\Microsoft Corporation\Microsoft Script Browser\System.Windows.Interactivity.dll'
Add-Type -Path 'C:\Program Files (x86)\Microsoft Corporation\Microsoft Script Browser\ScriptBrowser.dll'
Add-Type -Path 'C:\Program Files (x86)\Microsoft Corporation\Microsoft Script Browser\BestPractices.dll'
$scriptBrowser = $psISE.CurrentPowerShellTab.VerticalAddOnTools.Add('Script Browser', [ScriptExplorer.Views.MainView], $t
$scriptAnalyzer = $psISE.CurrentPowerShellTab.VerticalAddOnTools.Add('Script Analyzer', [BestPractices.Views.BestPractice
$psISE.CurrentPowerShellTab.VisibleVerticalAddOnTools.SelectedAddOnTool = $scriptBrowser
#Script Browser End

```

The code simply pulls in the `.dll` files associated with the add-on and then uses the `$psISE` variable to add the add-ons to the interface and make them visible. Once they are installed, we can also turn them on and off with the **Add-Ons** menu, as shown in the following screenshot:


Add-ons
Help
Open Add-on Tools Website
Show Vertical Add-on Tools Pane Alt+Shift+V
Script Browser (Hidden)
Script Analyzer (Hidden)
Show Horizontal Add-on Tools Pane Alt+Shift+H
Hide Selected Vertical Add-on Tool Ctrl+Shift+V
Hide Selected Horizontal Add-on Tool Ctrl+Shift+H
Move Selected Vertical Add-on Tool to Horizontal
Move Selected Horizontal Add-on Tool to Vertical
Go to vertical Add-on tool
Go to horizontal Add-on tool

Further reading

For more information on the topics covered in this chapter, take a look at the following references:

- Code smells at `http://en.wikipedia.org/wiki/Code_smell`
- Antipatterns at `http://en.wikipedia.org/wiki/Anti-pattern`

- Best practices at `http://en.wikipedia.org/wiki/Best_practice`
- Technical debt at `http://en.wikipedia.org/wiki/Technical_debt`
- `get-help about_parameters`
- `get-help out-gridview`
- `get-help about_functions_cmdletbindingattribute`
- `get-help get-verb`
- `get-help export-modulemember`
- `get-help about_pipelines`
- `get-help about_functions_advanced_parameters`
- ScriptCop at `http://scriptcop.start-automating.com`
- Script Browser and Script Analyzer at `http://www.microsoft.com/en-us/download/details.aspx?id=42525`

Summary

In this chapter, we examined the concept of code smells, which are related to best practices and antipatterns. We looked at several code smells that are found in most programming languages and then looked in-depth at code smells that are specific to PowerShell scripts due to the unique nature of PowerShell. Finally, we examined the state of static script analysis in PowerShell, reviewing the functionality of ScriptCop and the Script Analyzer add-on.

Index

Symbols

-confirm
  using 154, 155
–ErrorAction parameter
  Continue 41
  Stop 41
  Inquire 41
  Ignore 41
  SilentlyContinue 41
  Suspend 41
[object] parameter
  using 176-179
#REQUIRES statements
  -Modules option 104
  -PSSnapIn option 104
  -RunAsAdministrator option 104
  -ShellID option 104
  -Version option 104
-whatif
  using 154, 155

A

antipatterns
  about 168
  URL 185
assignment statements
  sequencing 175, 176

B

behavior-driven development (BDD) 78
best practices
  about 168
  URL 186

C

caption property 111
catch and release 89
catch statement 39
CmdletBinding()
  about 90
  parameter support 91, 92
cmdlet, naming convention 64, 65
cmdlets
  about 8, 9
  error behavior 88
  Get-Command cmdlet 10
  Get-Help cmdlet 11, 12
  Get-Member cmdlet 13
  verb list, URL 23
code 69, 70
code smells
  about 168
  language-agnostic code smells 169
  PowerShell-specific code smells 169, 170
  URL 185
color, error message
  issue 130, 131
comment-based help 49-53
Common Information Model (CIM) 109
console
  debugging in 140-145
console colors, error message
  changing 131, 132
content, error message 134-137

D

debugging
  in console 140-145
  in ISE 145-147

DebugPreference variable 56
disk
availability, validating 116-118
dogfooding
URL 82
downstream versions
Test-NetConnection, writing in 120, 121

E

ErrorActionPreference variable 56
error handling
about 85
catch and release 89
cmdlet, error behavior 88
catch statement 39
finally statement 39
guidelines 86
non-terminating errors 40-42
techniques 86, 87
terminating error 40
trap statement 37, 38
try statement 39
error messages
color, issue 130, 131
console colors, changing 131, 132
content 134-137
ISE colors, changing 132-134
PowerShell profiles 134
reading 130
event logs
about 147, 148
listing 149, 150
reading 150-152
writing to 152, 153
Event Tracing for Windows (ETW) 149

F

filter left 45, 46
finally statement 39
foreach loop
replacing, with foreach-object cmdlet 159-164
format right 47-49
formatting codes
URL 42
fully-qualified domain names (FQDN) 74
function naming 64, 65
function output 29-32
functions 13, 14

G

Get-Command cmdlet 10
Get-EventLog, filtering parameters
-Before and -After 150
-EntryType 150
-Index 150
-Message 150
-Newest 150
-Source 150
-UserName 150
Get-Help cmdlet 11, 12
Get-Member cmdlet 13
Get-Service cmdlet 88
Get-Verb cmdlet 8
Get-WinEvent, filtering parameters
-FilterHashTable 151
-FilterXML 151
-FilterXPath 151
-MaxEvents 151
-Oldest (switch) 151
Get-WMIObject cmdlet 88
grave accent 26

H

helper functions 68
here-strings 26
host. *See* **Write-Host**

I

ICMP connectivity
testing 122, 123
input set
reducing 156, 157
Integrated Scripting Environment (ISE)
about 26, 59
debugging in 145-147
intermediate results
viewing, Tee-Object used 158, 159

ISE colors, error message
changing 132-134

L

language-agnostic code smells 169

M

Measure-Command cmdlet 46
memory
availability, validating 116-118
mocking
URL 82
modularization
about 67
process 70, 71
process, breaking into subtasks 67, 68
single responsibility principle 69
URL 82
module naming 66
modules 21, 22
Monad Manifesto
URL 22

N

naming conventions
about 63
cmdlet 64, 65
function naming 64, 65
module naming 66
parameter naming 65, 66
URL 82
variable naming 67
network connectivity
validating 119
working 124
network connectivity, validating
ICMP connectivity, testing 122, 123
implementation, prior to 123, 124
telnet used 119
Test-NetConnection used 120
UDP connectivity, testing 122, 123
non-terminating error 40-42

O

operating system properties
validating 109, 110
version 111-113
working 114
workstation/server version 110
operating system version 111-113
output objects
accumulating 174, 175

P

parameter naming 65, 66
parameters
about 172, 173
blog post, URL 172
name, validating 93, 94
value, validating 94, 95
parameter type transformation
about 102-104
URL 107
Param() statements
about 90
missing 170, 171
Pester
about 78, 79
mocking with 80-82
URL 82
pipelines
about 15-19
and function execution 99, 101
input 96-99
processing 32-36
PowerShell. *See also* **Windows PowerShell**
PowerShell
error handling 37
pipeline, processing 32-36
strings 25, 26
string substitution 26-28
testing 77, 78
version control, using with 73
PowerShell profiles, error message 134

PowerShell-specific code smells 169, 170
process
about 70, 71
breaking, into subtasks 67, 68
ProgressPreference variable 56
PSDiagnostics module 153, 154
PSDrives 55
PSProviders 55

S

Script Analyzer
about 182-185
URL 182, 186
Script Browser
URL 182, 186
scriptblock 20
ScriptCop
about 180-182
URL 180, 186
script cmdlets 90
scripts 14, 15
service status
validating 115
Set-PSDebug
about 105, 106
using 137-140
Set-StrictMode 105-107
single responsibility principle 69
sorting
about 19, 20
directory 18
SQLPS 68
static analysis tools
about 180
Script Analyzer 182-185
ScriptCop 180-182
strings, PowerShell
about 25, 26
double-quoted strings, special characters 29
escaping in 29
substitution 26-28
subtasks
process, breaking into 67, 68
SupportsShouldProcess
about 92
parameter name validation 93, 94

T

Team Foundation Services (TFS) 73
technical debt
about 168
URL 186
Tee-Object
used, for viewing intermediate results 158, 159
telnet
URL 127
used, for validating network connectivity 119
terminating error 37-40
Test Driven Development (TDD) 73
Test-NetConnection
used, for validating network connectivity 120
writing, in downstream versions 120, 121
trap statement 37
try statement 39

U

UDP connectivity
testing 122, 123
unit testing
about 73, 74
rolling 74-77
URL 82
untyped parameter
using 176-179

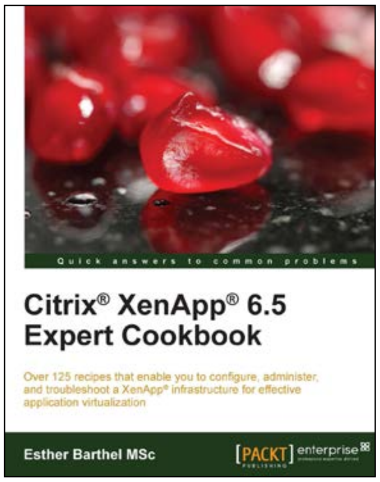
V

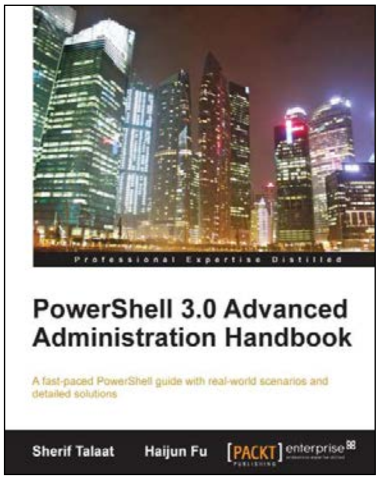
variable naming 67
variables 20, 21
VerbosePreference variable 56
verbs
unapproved 173, 174
URL 9
version control
about 71, 72
using, with PowerShell 73

W

WarningPreference variable 56

Web Services Management (WSMAN) 110
Windows Management Instrumentation (WMI) 109
Windows PowerShell
about 7, 8
cmdlets 8
functions 13, 14
modules 21, 22
pipelines 15-17
scripts 14, 15
variables 20, 21
workstation/server version, operating system properties 110
Write-* cmdlets 53-57
Write-Debug 58
Write-Error 58
Write-Host 53
Write-Output 54, 57
Write-Progress 59
Write-Verbose 57
Write-Warning 58


Quick answers to common problems
Citrix® XenApp® 6.5 Expert Cookbook
Over 125 recipes that enable you to configure, administer, and troubleshoot a XenApp® infrastructure for effective application virtualization
Esther Barthel MSc
[PACKT] enterprise


Professional Expertise Distilled
PowerShell 3.0 Advanced Administration Handbook
A fast-paced PowerShell guide with real-world scenarios and detailed solutions
Sherif Talaat
Haijun Fu
[PACKT] enterprise