2

Introducing PowerShell 7

This chapter covers the following recipes:

* Exploring new operators
* Exploring parallel processing
* Exploring performance improvements
* Using Test-Connection
* Using Select-String
* Exploring error view and Get-Error

# Introduction

In Chapter 1, you installed and configured PowerShell 7, along with VS code and a new font. In this chapter, we look at PowerShell 7 and how it differs from Windows PowerShell. The recipes in this chapter illustrate some of the important new features which come with PowerShell 7.

Now that PowerShell is cross-platform, it has a new, expanded audience, one with a background in Linux shells such as Bash. With PowerShell 7, the PowerShell team added several new operators which improved parity with other shells and made life that little bit easier for IT Pros.

With the move to open source, the PowerShell code was open to inspection by the community. Many talented developers were able to make improvements to performance and functionality. One example is how PowerShell performs iteration using ForEach and ForEach-Object. In Windows PowerShell, the ForEach syntax item and the Foreach-Object command allowed you to process collections of objects. With Windows PowerShell, each iteration through a collection was serial, which could result in long script run times. PowerShell 7 introduces an improvement in the ForEach‑Object command that enables you to run iterations in parallel. This review has led to a reduction in the overhead of using these popular language features, thus speeding up production scripts.

Another improvement is the revised Test-Connection, a command you use to test a network connection with a remote system. Test-Connection, in PowerShell 7, not only does more but is faster than with Windows PowerShell.

Error reporting in Windows PowerShell was excellent: clear and generally actionable error messages with details of exactly where the error occurred. In PowerShell 7, you now get, by default, a concise view of an error without all the extra text that was often of little value. As always, you can revert to less concise message if you choose. In the “Exploring error View and Get-Error” recipe, you see how error reporting (in Windows PowerShell) becomes better with PowerShell 7.

# Exploring New Operators

Operators are symbols or combinations of keystrokes which PowerShell recognizes and assigns some meaning. PowerShell uses the + operator to mean addition, either arithmetic addition or string addition/concatenation. Most of the PowerShell operators were defined with Windows PowerShell V1.

PowerShell 7 now implements some new operators, including:

* Pipeline chain operators: || and &&
* Null-coalescing operator: ??
* Null coalescing assignment operator: ??=
* Experimental null conditional member access operators: ?. and ?[]
* Background processing operator: &
* Ternary operator: ? <if-true> : <if-false>

You see examples of these operators in this recipe.

## Getting Ready

This recipe uses SRV1, a Windows Server 2020 host. You have installed and configured PowerShell 7 and VS Code. You run this, and all remaining recipes in this book in either a PowerShell 7 console or VS Code.

## How to do it...

1. Use PowerShell 7 to check results traditionally

Write-Output 'Something that succeeds'

if ($?) {Write-Output 'It worked'}

1. Check results with the pipeline operator &&

Write-Output 'Something that succeeds' && Write-Output 'It worked'

1. Use the pipeline chain operator  ||

Write-Output 'Something that succeeds' ||

Write-Output 'You do not see this message'

1. Define a simple function

function Install-CascadiaPLFont{

  Write-Host 'Installing Cascadia PL font...'

}

1. Use the || operator

$OldErrorAction        = $ErrorActionPreference

$ErrorActionPreference = 'SilentlyContinue'

Get-ChildItem -Path C:\FOO\CASCADIAPL.TTF ||

   Install-CascadiaPLFont

$ErrorActionPreference = $OldErrorAction

1. Create a function to test null handling

Function Test-NCO {

  if ($args -eq '42') {

    Return 'Test-NCO returned a result'

  }

}

1. Test null results traditionally

$Result1 = Test-NCO    # no parameter

if ($null -eq $Result1) {

    'Function returned no value'

} else {

    $Result1

}

$Result2 = Test-NCO 42  # using a parameter

if ($null -eq $Result2) {

    'Function returned no value'

} else {

    $Result2

}

1. Test using Null Coalescing operator ??

$Result3 =  Test-NCO

$Result3 ?? 'Function returned no value'

$Result4 =  Test-NCO 42

$Result4 ?? 'This is not output, but result is'

1. Demonstrate the Null Conditional Assignment Operator

$Result5 = Test-NCO

$Result5 ?? 'Result is null'

$Result5 ??= Test-NCO 42

$Result5

1. Run a method on a null object traditionally

$BitService.Stop()

1. Use the Null conditional operator for a method

${BitService}?.Stop()

1. Test Null property name access

$x = $null

${x}?.Propname

$x = @{Propname=42}

${x}?.Propname

1. Test array member access if a null object

$y = $null

${y}?[0]

$y = 1,2,3

${y}?[0]

1. Use the background processing operator &

Get-CimClass -ClassName Win32\_Bios &

1. Wait for the job to complete

$JobId = (Get-Job | Select-Object -Last 1).Id

Wait-Job -id $JobId

1. View the output

$Results = Receive-Job -Id $JobId

$Results | Format-Table

1. Create an object without using the ternary operator

$A = 42; $B = (42,4242) | Get-Random

$RandomTest = ($true, $false) | Get-Random

if ($A -eq $B) {

  $Property1 = $true

} else {

  $Property1 = $false

}

if ($RandomTest) {

  $Property2 = "Hello"

} else {

  $Property2 = "Goodbye"

}

[PSCustomObject]@{

  "Property1" = $Property1

  "Property2" = $Property2

}

1. Create an object using the ternary operator

[PSCustomObject]@{

    "Property1" = (($A -eq $B) ? $true : $false)

    "Property2" = (($RandomTest) ? "Hello" : "Goodbye")

}

## How it works...

In step 1, you write output, which succeeds. Then you test the value of $? to determine whether that previous step did, in fact, succeed. The output is as follows:

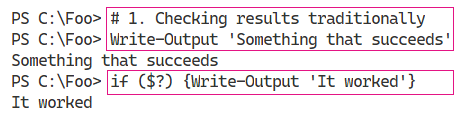


Figure 2.1: Checking results traditionally

Insert image B42024\_02\_01.png

In step 2, you use the && operator to check that a preceding command finished without an error. The output looks like this:

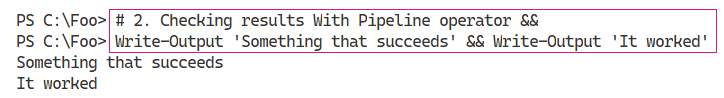


Figure 2.2: Checking results with pipeline operator

Insert image B42024\_02\_02.png

The pipeline chain operator, ||, tells PowerShell to run the commands after the operator if the preceding command fails (in effect, the opposite to &&). In step 3, you see the operator in use, with output like this:

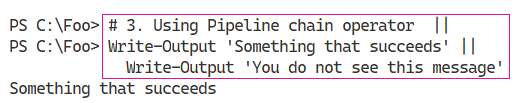


Figure 2.3: Using the pipeline chain operator

Insert image B42024\_02\_03.png

In step 4, you define a function. Defining the function produces no output. This function writes output to simulate the installation of the Cascadia Code PL font.

In step 5, you check to see if the TTF file exists, and if not, you call the Install-CascadiaPLFont function to simulate installing the font. By piping the output from Get-ChildItem to Out-Null, you avoid the actual output from Get-ChildItem, and if the file does not exist, you call Install‑CascadiaPLFont function. The output of this snippet looks like this:

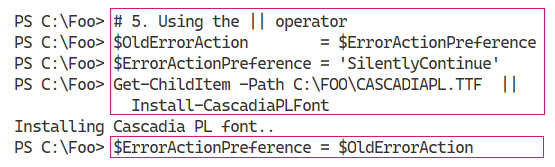


Figure 2.4: Using the || operator and installing the Cascadia font

Insert image B42024\_02\_04.png

To illustrate the handling of null results from a function, in step 6, you create a function which either returns nothing (if you call the function with no parameters) or a string value (if you call it specifying a parameter). This function illustrates how you can handle a function that returns null. This step produces no output.

In step 7, you illustrate the traditional handling of a function which returns null. You call the function, first without a parameter, which returns no result and then with a value which does return a value. You then test to see whether the function returned an actual value in each case, which looks like this:

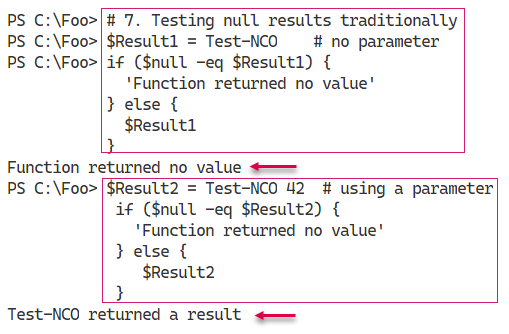


Figure 2.5: Testing null results traditionally

Insert image B42024\_02\_05.png

When you use the null coalescing operator (??) between two operands, the operator returns the value of its left-hand operand if it isn't null; otherwise, it evaluates the right-hand operand and returns the results. In step 8, you call the Test-NCO function and check whether or not the function returns a value, which looks like this:

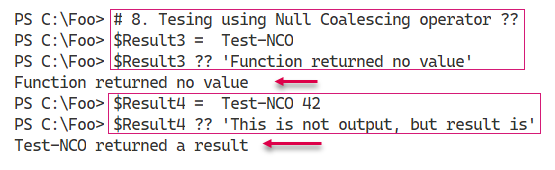


Figure 2.6: Testing using the null coalescing opereator

Insert image B42024\_02\_06.png

You use the null conditional assignment operator, ??=, to assign a value to a variable if that variable is currently null as you can see in step 9, the output from which looks like this:

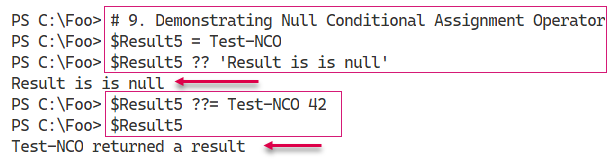


Figure 2.7: Using the null conditional assignment operator

Insert image B42024\_02\_07.png

One common issue often seen in the various PowerShell support forums arises when you attempt to invoke a method on an object which is null. You might have used an expression or a command to attempt to return a value (for example all AD users in the Marin County office) and which produces a null. In Step 10, you attempt to invoke the Stop() method on the $BitService object. Since you have not assigned a value to $BitService, you see the result (an error “You cannot call a method on a null-valued expression”). The traditional method of displaying errors looks like:

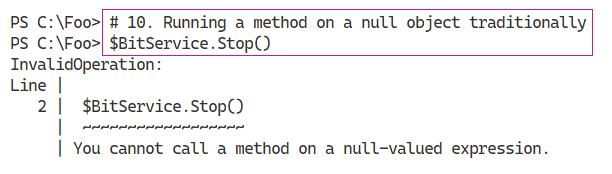


Figure 2.8: Running a method on a null object traditionally

Insert image B42024\_02\_08.png

By using the null conditional operator, you can run the Stop() method if the $BitService variable is non-null, but skip calling the method if the variable is null. In effect, what you are doing in step 11 is calling the Stop() method if the variable is non-null, and doing nothing otherwise. Because the variable does not have a value, this step does nothing (and produces no output).

When a variable is null, whether due to an error in your scripts or because a command returns a null instead of an actual value, accessing property names can also cause errors. The output of step 12 looks like this:

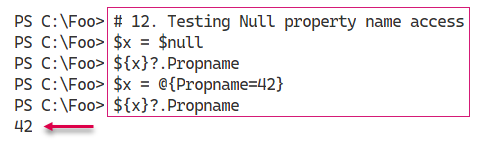


Figure 2.9: Testing null property name access

Insert image B42024\_02\_09.png

You can also encounter issues with null objects when you attempt to access an array member of an object that may or may not exist. In step 13, you attempt to access an array member of an array that does not exist, followed by one that does exist. The output from this step looks like this:

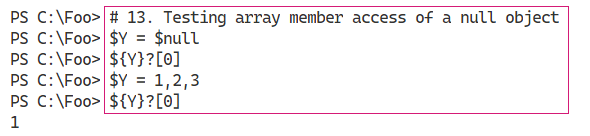


Figure 2.10: Testing array member access of a null object

Insert image B42024\_02\_10.png

In step 14, you investigate the use of the background processing operator &. The idea is that you append this character to the end of a command or script, and PowerShell runs that code in the background. The output from this step looks like this:

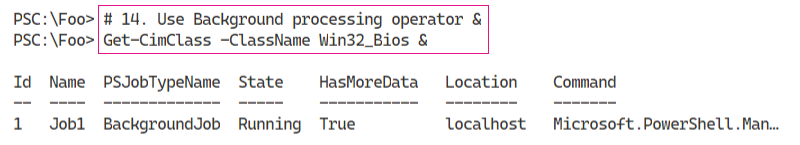


Figure 2.11: Using the background processing operator

Insert image B42024\_02\_11.png

In step 15, you wait for the job you created in step 14 to complete, which looks like this:

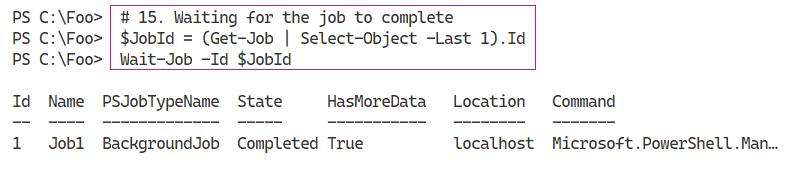


Figure 2.12: Waiting for the job to complete

Insert image B42024\_02\_12.png

After the job has completed, in step 16, you receive and display the job’s output which looks like this:

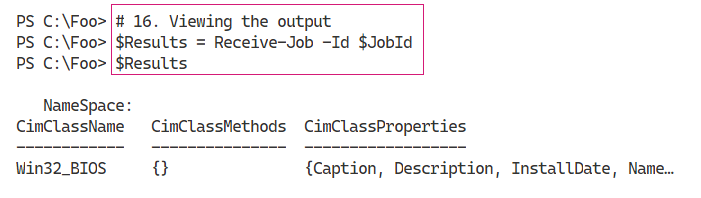


Figure 2.13: Displaying the job’s output

Insert image B42024\_02\_13.png

In step 17, you create an object using a more traditional approach. This step creates a random value for two properties. Then, you create an object using the values of these two properties. The output from this step looks like this.

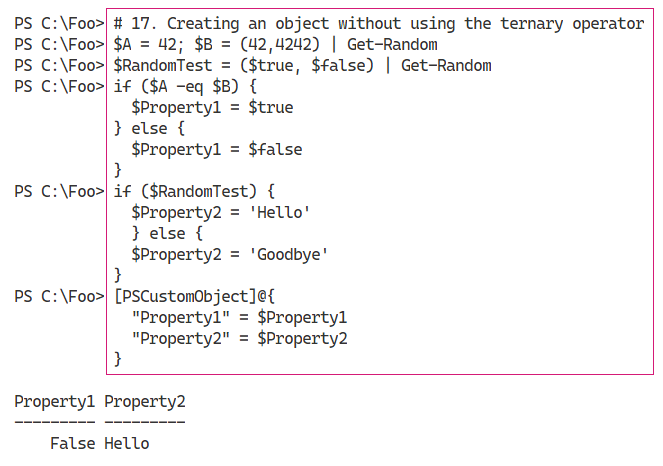


Figure 2.14: Creating an object without using the ternary operator

Insert image B42024\_02\_14.png

In step 18, you use the new PowerShell 7 ternary operator. This operator tests a condition and runs different code depending on whether the result is true or false. This is very similar to what you saw in step 17, but in a lot fewer lines of code. The output of this step looks like this:

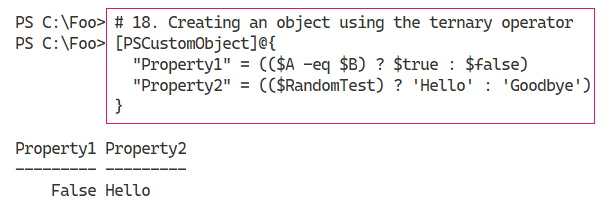


Figure 2.15: Creating an object using the ternary operator

Insert image B42024\_02\_15.png

## There's more...

In step 4, the function you create simulates the installation of a font if the font does not exist. The font file, CASCADIAPL.TTF, is a TrueType font file for the Cascadia Code Powerline font. This font is the Cascadia Code font you installed in Chapter 1, Installing and Configuring PowerShell 7.1 with the addition of symbols for Powerline. For more details on this font, see *https://www.hanselman.com/blog/PatchingTheNewCascadiaCodeToIncludePowerlineGlyphsAndOtherNerdFontsForTheWindowsTerminal.aspx*.

In step 5, you simulate the installation of the font if it does not already exist. When you check to test whether the TTF file currently exists, the default setting for $ErrorActionPreference (Continue) means you would see an error message if the file does not exist. By default, when Get‑ChildItem checks to see if the file exists, it generates an error message if the file does not exist. One approach to avoiding this error message is to set the value of $ErrorActionPreference to SilentlyContinue and, after ensuring the font file now exists, set it back to the default value. The syntax is a bit convoluted unless you are familiar with it; this may be another case where not using these operators, and using the Windows PowerShell approach instead, might make the script easier to read and understand.

In step 11 and step 12, you attempt to access a property from an object. The assumption here is that you only want to invoke the method or access a property value if the object exists and you do not care otherwise. Thus, if $BitService has a value, you call the Stop() method, otherwise the code carries on without the script generating errors. This approach is great from the command line, but in production scripts, the approach could mask other underlying issues. As with all PowerShell features, you have to use null handling with due care and attention.

With step 14, you tell PowerShell to run a command as a background job by appending the & character to the command. Using this operator is a more straightforward way to invoke the command as a job than by calling Invoke-Command and specifying the command using the -ScriptBlock or -Script parameters.

In step 15 and step 16, you use Get-Job, Wait-Job and Receive-Job to wait for the last job run and get the output. One downside to not using Start-Job to create a background is that you cannot specify a job name. That means using the technique shown in step 15 to obtain the job and job results for the job created in step 14. Using that technique is thus more useful from the command line than in a production script.

In step 17, you create an object using older Windows PowerShell syntax, whereas, in step 18, you use the ternary operator. As with other operators, use this with care – and if you use these operators in production code, make sure you document what you are doing.

In this recipe, you have seen the new operators added to PowerShell 7. Most of them provide a shortcut way to perform some operation, particularly at the command line.

# Exploring Parallel Processing with Foreach Object

There are often situations when you want to run many commands in parallel. For example, you might have a list of computer names, and for each of those computers, you want to run a script on that computer. For instance, you might wish to verify the status and resource usage of various services on each computer. In this scenario, you might use Get-Content to get an array of computer names, then use either Foreach or Foreach-Object to run the script on the computer. If there are 10 computers and the script takes 10 minutes, the total run time is over 100 minutes.

With Windows PowerShell, the only built-in methods of running scripts in parallel could be done by using background jobs or by using workflows. With background jobs, you could create a set of jobs, each of which starts some script on a single computer. In that case, PowerShell runs each job in a separate process which provides isolation between each job but is resource-intensive. The Windows PowerShell team added Workflows with Windows PowerShell V4, which also allow you to run script blocks in parallel. However, workflows are not carried forward into PowerShell 7. Like other features no longer available in PowerShell 7, you can continue to use Windows PowerShell to run workflows and gradually convert them as and when appropriate.

An alternative to background jobs is to use the ThreadJob module you can download from the PowerShell gallery. For more details on this module, see its repository page at *https://github.com/PaulHigin/PSThreadJob.*

With PowerShell 7, the PowerShell team added an option to the Foreach-Object command to allow you to run script blocks in parallel. This option simplifies running script blocks or scripts, especially long-running ones, in parallel and avoids needing third party modules or dealing with the complexity of workflows.

This recipe demonstrates running operations in parallel traditionally, using background jobs, and using Foreach-Object -Parallel.

## Getting Ready

You run this recipe on SRV1 after you have installed PowerShell 7 and optionally VS Code.

## How to do it...

1. Simulate a long-running script block

$SB1 = {

  1..3 | ForEach-Object {

    "In iteration $\_"

    Start-Sleep -Seconds 5

  }

}

Invoke-Command -ScriptBlock $SB1

1. Time the expression

Measure-Command -Expression $SB1

1. Refactor into using jobs

$SB2 = {

1..3 | ForEach-Object {

  Start-Job -ScriptBlock {param($X) "Iteration $X " ;

                          Start-Sleep -Seconds 5} -ArgumentList $\_

}

Get-Job | Wait-Job | Receive-Job -Keep

}

1. Invoke the script block

Invoke-Command -ScriptBlock $SB2

1. Remove any old jobs and time the script block

Get-Job | Remove-Job

Measure-Command -Expression $SB2

1. Define a script block using ForEach-Object -Parallel

$SB3 = {

1..3 | ForEach-Object -Parallel {

               "In iteration $\_"

               Start-Sleep -Seconds 5

         }

}

1. Execute the script block

Invoke-Command -ScriptBlock $SB3

1. Measure the script block execution time

Measure-Command -Expression $SB3

1. Create and run two short script blocks

$SB4 = {

    1..3 | ForEach-Object {

                   "In iteration $\_"

             }

}

Invoke-Command -ScriptBlock $SB4

$SB5 = {

        1..3 | ForEach-Object -Parallel {

                       "In iteration $\_"

             }

}

Invoke-Command -ScriptBlock $SB5

1. Measure execution time for both script blocks

Measure-Command -Expression $SB4

Measure-Command -Expression $SB5

## How it works...

In step 1, you create and then invoke a script block. The script block simulates how you can run several long script blocks traditionally using the Foreach-Object cmdlet, with output like this:

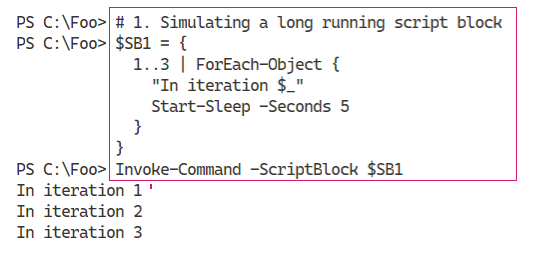


Figure 2.16: Simulating a long-running script block

Insert image B42024\_02\_16.png

In step 2, you determine how long it takes PowerShell to run this script block, with output like this:

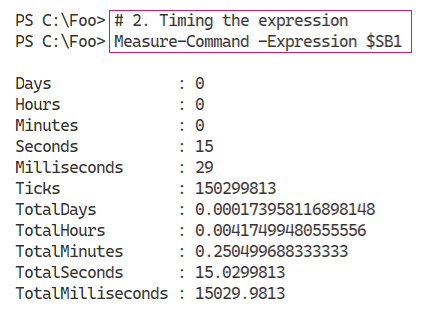


Figure 2.17: Timing the expression

Insert image B42024\_02\_17.png

In step 3, you refactor the $SB1 script block to use PowerShell background jobs. The script block runs the simulated long-running task using jobs then waits for and displays the output from each job. The concept is that instead of doing each iteration serially, all the jobs run in parallel. Defining the function creates no output.

In step 4, you invoke the script block to view the results, which looks like this:

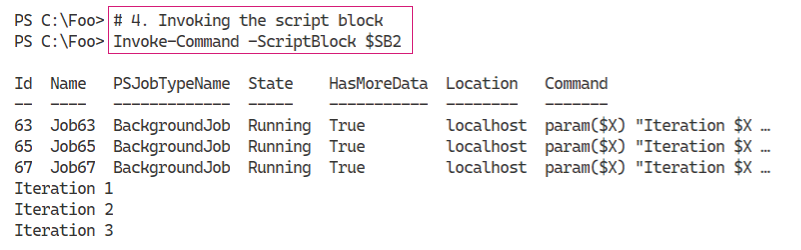


Figure 2.18: Invoking the script block

Insert image B42024\_02\_18.png

In step 5, you remove any existing jobs then re-run the updated script block. This step enabled you to determine the run time for the entire expression. The output of this step looks like this:

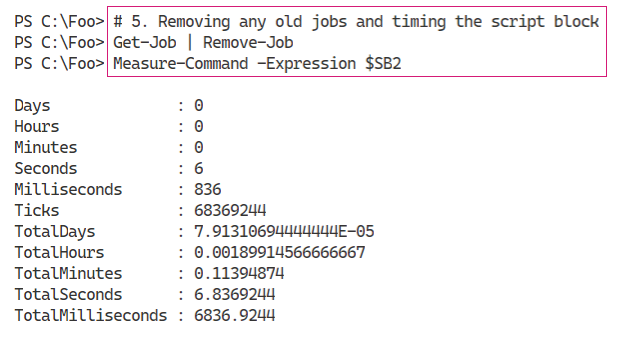


Figure 2.19: Removing any existing jobs and timing the script block

Insert image B42024\_02\_19.png

In step 6, you create another script block that uses the PowerShell 7 Foreach-Object -Parallel construct. When you define this script block, PowerShell creates no output.

In step 7, you run the script block, which looks like this,

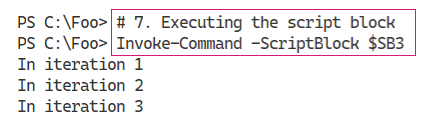


Figure 2.20: Executing the script block created in step 6

Insert image B42024\_02\_20.png

In step 8, you time the execution of the script block, making use of the Foreach-Object -Parallel feature, which looks like this:

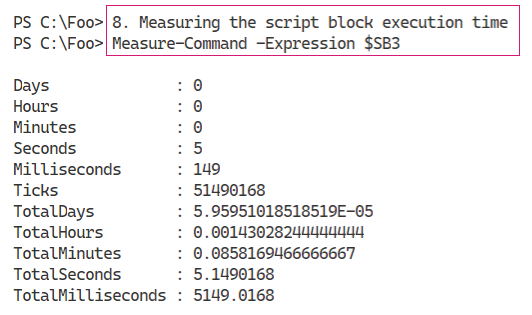


Figure 2.21: Timing the script block execution

Insert image B42024\_02\_21.png

In step 9, you define and then invoke two script blocks, which looks like this:

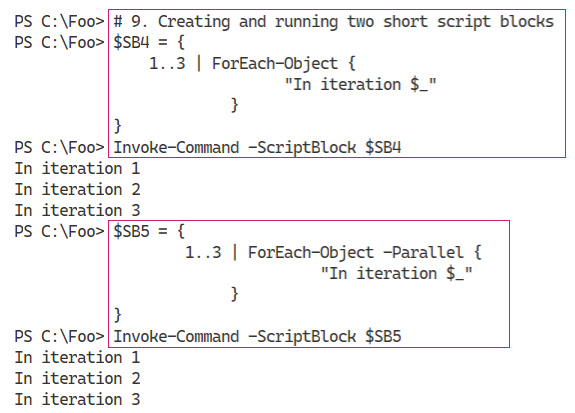


Figure 2.22: Creating and running two short script blocks

Insert image B42024\_02\_22.png

In the final step in this recipe, step 10, you measure the execution time of these two script blocks, which looks like this:

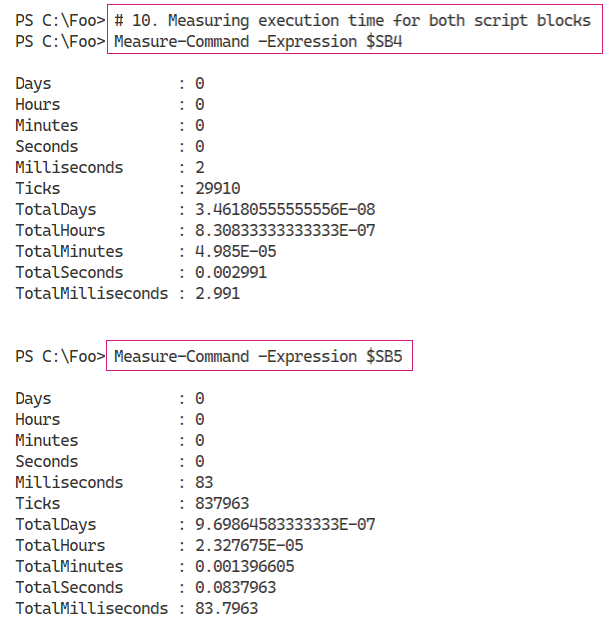


Figure 2.23: Measuring execution time of the script blocks created in step 9

Insert image B42024\_02\_23.png

## There's more...

In step 1 and step 2, you invoke a long-running task multiple times. As you can see, running these script blocks, one at a time, takes just over 15 seconds. In step 5, you see that by refactoring the long-running task into PowerShell background jobs, you reduce the run time to 6.83 seconds. Finally, in step 8, you measure the elapsed run time when you use ForEach-Object -Parallel, which is now a little over 5 seconds.

As this recipe shows, if you have independent script blocks, you can run them in parallel to reduce the overall run time. In this case, from just over 15 seconds to just over 5. And the gains would have been even higher had you run the loop more than three times. Running the loop serially 10 times would have taken over 50 seconds, compared to just over 5 for ForEach-Object -Parallel.

However, there is a default limit of five script blocks that PowerShell can run simultaneously. You can use the -ThrottleLimit parameter to allow more or less than that default. One thing to note: if you attempt to run more parallel script blocks than you have processor cores, PowerShell just uses a processor core queue. This all takes time and would end up raising the overall runtime. The good news is that PowerShell handles all this – so if you run, say, 1000 parallel script blocks on a system with 12 processor cores. PowerShell works as fast as your host computer allows.

It is also worth remembering that there is some overhead involved in ForEach-Object -Parallel. Under the covers, the command has to set up and then manage separate threads of execution. If the script block is very short, you can find that the overhead involved results in slower runtimes. In this case, the runtime went from 2.9ms to 83.7ms. The critical point here is that this construct is useful for non-trivial script blocks (or scripts) that you run in parallel. You benefit up to the number of cores you have available.

Another thing to note is that when you use the syntax ForEach-Object {script}, you are using a positional parameter (‑Process). On the other hand, when you use the -Parallel parameter, the parameter value is the script block you wish PowerShell to run in parallel.

# Improvements in ForEach and ForEach-Object

Windows PowerShell users are well-versed in the use of both the ForEach statement and the Foreach-Object cmdlet. You can use both of these methods in your scripts to process collections such as all the users in a specific Active Directory group, or the audio files in a file share. In PowerShell 7, both of these iteration methods are considerably faster.

Using either ForEach mechanism is a quick and easy way of processing a collection. One downside some IT Pros may have noticed is that the overhead of ForEach processing grows with the size of the collection. With small collection sizes, you are not likely to notice any difference. As the collection size grows, so does the overhead.

## Getting Ready

You run this recipe on SRV1 after you have installed PowerShell 7 and have created a console profile file. Run this recipe in an elevated console.

## How to do it...

1. Create a remoting connection to the local host using Windows PowerShell

New-PSSession -UseWindowsPowerShell -Name 'WPS'

1. Get remoting session

$Session = Get-PSSession -Name 'WPS'

1. Check the version of PowerShell in the remoting session

Invoke-Command -Session $Session  -ScriptBlock {$PSVersionTable}

1. Define a long running script block using ForEach-Object

$SB1 = {

  $Array  = (1..10000000)

  (Measure-Command {

    $Array | ForEach-Object {$\_}}).TotalSeconds

}

1. Run the script block locally

[gc]::Collect()

$TimeInP7 = Invoke-Command -ScriptBlock $SB1

"Foreach-Object in PowerShell 7.1: [{0:n4}] seconds" -f $TimeInP7

1. Run the script block in PowerShell 5.1

[gc]::Collect()

$TimeInWP  = Invoke-Command -ScriptBlock $SB1 -Session $Session

"ForEach-Object in Windows PowerShell 5.1: [{0:n4}] seconds" -f $TimeInWP

1. Define another long running script block using ForEach

$SB2 = {

    $Array  = (1..10000000)

    (Measure-Command {

      ForEach ($Member in $Array) {$Member}}).TotalSeconds

}

1. Run it locally in PowerShell 7

[gc]::Collect()

$TimeInP72 = Invoke-Command -ScriptBlock $SB2

"Foreach in PowerShell 7.1: [{0:n4}] seconds" -f $TimeInP72

1. Run it in Windows PowerShell 5.1

[gc]::Collect()

$TimeInWP2  = Invoke-Command -ScriptBlock $SB2 -Session $Session

"Foreach in Windows PowerShell 5.1: [{0:n4}] seconds" -f $TimeInWP2

## How it works…

In step 1, you use New-PSSession to create a remoting session using a Windows PowerShell endpoint. This step produces output like this:

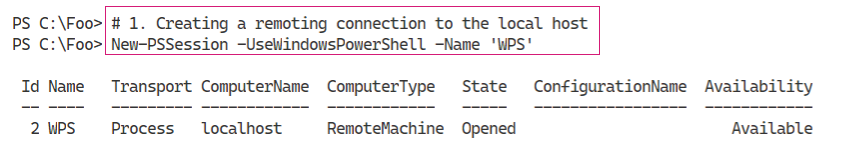


Figure 2.24: Creating a remoting connection to the local host

Insert image B42024\_02\_24.png

In step 2, you get the session object representing the session you created in the previous step – this creates no output.

In step 3, you obtain the version of PowerShell that the remoting session is using to process commands, namely Windows PowerShell 5.1. The output of this step looks like this:

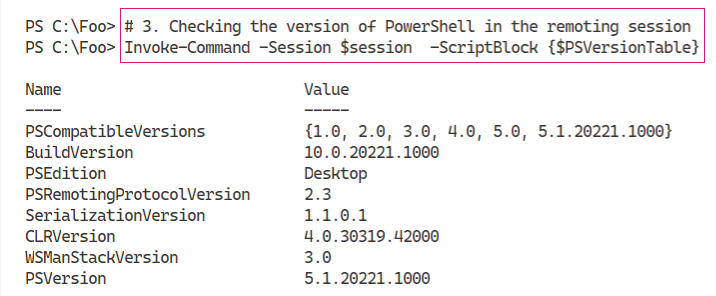


Figure 2.25: Checking the PowerShell version in the remoting session

Insert image B42024\_02\_25.png

In step 4, you create a script block, $SB1, which uses the ForEach-Object cmdlet to iterate over a large collection. This step creates no output.

You invoke the $SB1 script block in the local session, in step 5. This step runs the script block in PowerShell 7. The output from this step looks like this:

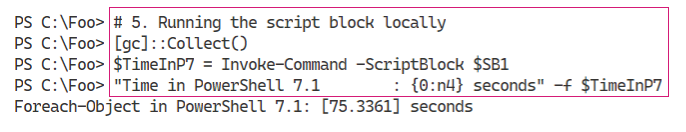


Figure 2.26: Running the script block locally

Insert image B42024\_02\_26.png

With step 6, you run the $SB1 script block in Windows PowerShell 5.1, which produces output like this:

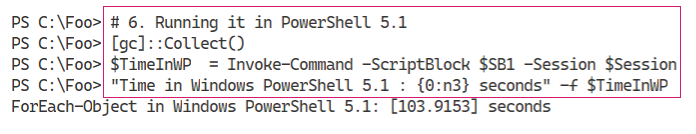


Figure 2.27: Running the script block in PowerShell 5.1

Insert image B42024\_02\_27.png

You next create a script block that makes use of the ForEach syntax item, in step 7, producing no output. You then run this second script block in PowerShell 7, in step 8, which produces output like this:

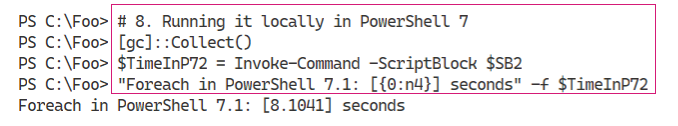


Figure 2.28: Running the script block locally in PowerShell 7

Insert image B42024\_02\_28.png

In the final step, step 9, you run $SB1 in the remoting session created earlier (i.e. in Windows PowerShell 5.1) which produces output like this:

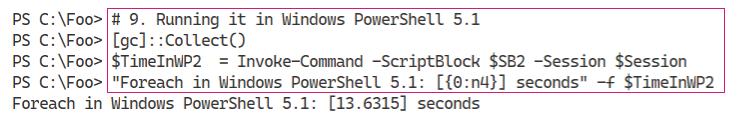


Figure 2.29: Running the script block in PowerShell 5.1

Insert image B42024\_02\_29.png

## There's more...

In step 1, you create, implicitly, a remoting session to the local host using a process transport which is much faster than the traditional remoting session using WinRM.

In step 5, step 6, step 8, and step 9, you force .NET to perform a garbage collection. These are steps you can use to minimize the performance hits of running a script block in a remote session (in Windows PowerShell) and to reduce any impact of garbage collections while you are performing the tests in this recipe.

As you can see from the outputs, running Foreach-Object is much faster in PowerShell 7, as is running Foreach in PowerShell 7. As you can see, processing large collections of objects is a lot faster in PowerShell 7.

The improvements to loop processing that you can see in the recipe, combined with the use of ForEach-Object -Parallel you saw in “Exploring Parallel Processing with Foreach -Parallel”, provide an excellent reason to switch to PowerShell 7 for most operations.

The performance of iterating through large collections is complex. You can read an excellent article on this subject at *https://powershell.one/tricks/performance/pipeline*. This article also addresses the performance of using the pipeline versus using ForEach to iterate across collections in a lot more detail.

# Improvements in Test-Connection

In Windows PowerShell, you could use the Test-Connection cmdlet as a replacement to the Win32 console command ping.exe. One advantage of using the cmdlet was that the cmdlet returns objects which you can use more easily in scripting. You can use string hacking and regular expressions to extract the same data from the output of ping.exe but that is a lot more work and results in scripts that are harder to read.

With Windows PowerShell 5.1, the Test-Connection command makes use of WMI. The command returns objects of the type System.Management.ManagementObject#root\cimv2\Win32\_PingStatus. With PowerShell 7.1, the command no longer depends on WMI and returns objects of the type Microsoft.PowerShell.Commands.TestConnectionCommand+PingStatus. As a result of this change of object type, property names returned in PowerShell 7.1 differ from the properties returned in Windows PowerShell. Scripts that made use of some properties may not work properly without adjustment, in PowerShell 7, but that should not be a common issue.

## Getting Ready

You run this recipe on SRV1 after you have installed PowerShell 7 and have created a console profile file.

## How to do it...

1. Use Test-Connection with the -Target parameter

Test-Connection -TargetName www.packt.com -Count 1

1. Use Test-Connection with an IPv4 address

Test-Connection -TargetName www.packt.com -Count 1 -IPv4

1. Use Resolve-DnsName to resolve destination address

$IPs = (Resolve-DnsName -Name Dns.Google -Type A).IPAddress

$IPs |

  Test-Connection -Count 1 -ResolveDestination

1. Resolve destination and perform a traceroute

Test-Connection -TargetName 8.8.8.8 -ResolveDestination -Traceroute |

  Where-Object Ping -eq 1

1. Use infinite ping and stop with Ctrl-C

Test-Connection -TargetName www.reskit.net -Repeat

1. Check speed of Test-Connection in PowerShell 7

Measure-Command -Expression {test-connection 8.8.8.8 -count 1}

1. Check speed of Test-Connection in Windows PowerShell

$Session = New-PSSession -UseWindowsPowerShell

Invoke-Command -Session $Session -Scriptblock {

Measure-Command -Expression {

Test-Connection -ComputerName 8.8.8.8 -Count 1}

}

## How it works…

In step 1, you test the connection between SRV1 and our publisher’s online website. The output of this command looks like this:

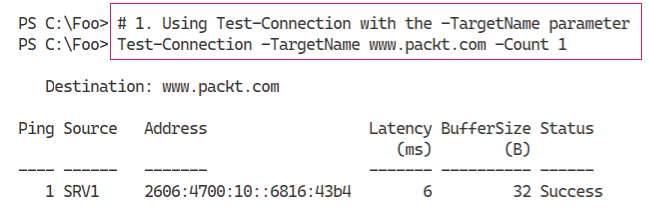


Figure 2.30: Using Test-Connection with the -TargetName parameter

Insert image B42024\_02\_30.png

If you have a computer with a working IPv6 address, Test-Connection prefers using IPv6, by default, as shown in the output from step 1. Should you want to test the IPv4 connection specifically, you can specify the -IPv4 switch explicitly, as shown in step 2:

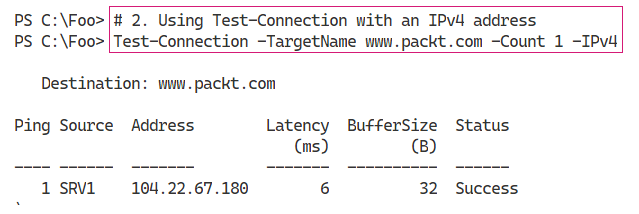


Figure 2.31: Using Test-Connection with an IPv4 address

Insert image B42024\_02\_31.png

In step 3, you use Resolve-DnsName cmdlet to determine the IPv4 address(es) for Dns.Google. This site is Google’s free DNS service which Google offers via two well known IPV4 addresses (8.8.8.8 and 8.8.4.4), which you can see in the output from this step:

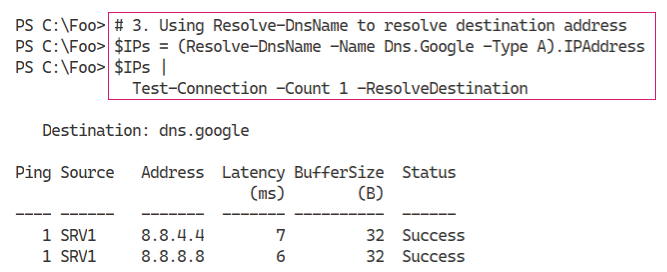


Figure 2.32: Using Resolve-DnsName to resolve destination address

Insert image B42024\_02\_32.png

The tracert.exe Win32 console application allows you to trace the route between your host system and some external host. In step 4, you use Test-Connection to trace the route between SRV1 and the computer4 at 8.8.8.8. The output of this step looks like this:

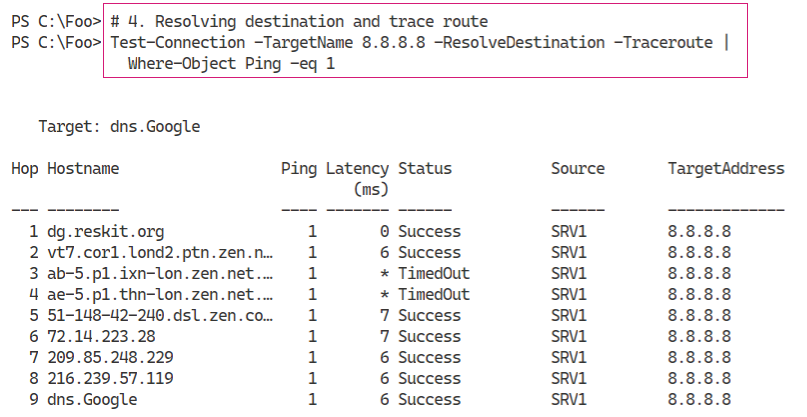


Figure 2.33: Resolving destination and trace route

Insert image B42024\_02\_33.png

With the ping.exe Win32 console application, you can specify the -t parameter to ping the target host continuously, which you can then stop by entering Ctrl-C. With Test-Connection in PowerShell 7, you can now use the -Repeat parameter to achieve the same outcome (and stop the test using Ctrl-C). You can see this in the output of step 5:

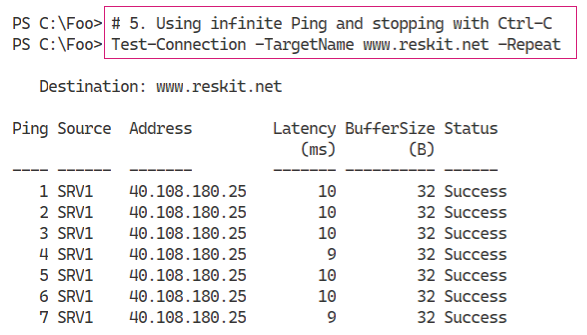


Figure 2.34: Using infinite ping and stopping with Ctrl-C

Insert image B42024\_02\_34.png

In step 6 and step 7, you compare the speed of Test-Connection in Windows PowerShell and PowerShell 7. Running the Test-Connection in PowerShell 7 looks like this:

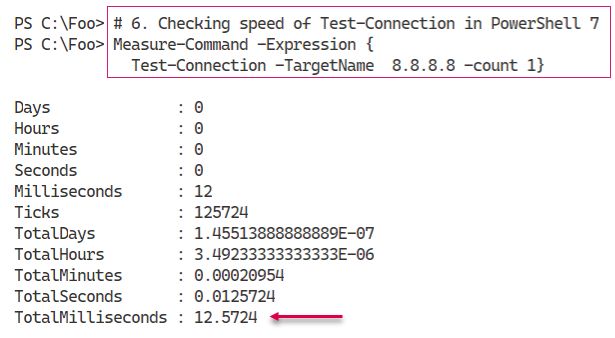


Figure 2.35: Checking the speed of Test-Connection in PowerShell 7

Insert image B42024\_02\_35.png

In Windows PowerShell 5.1, the output is similar (although slower), and looks like this:

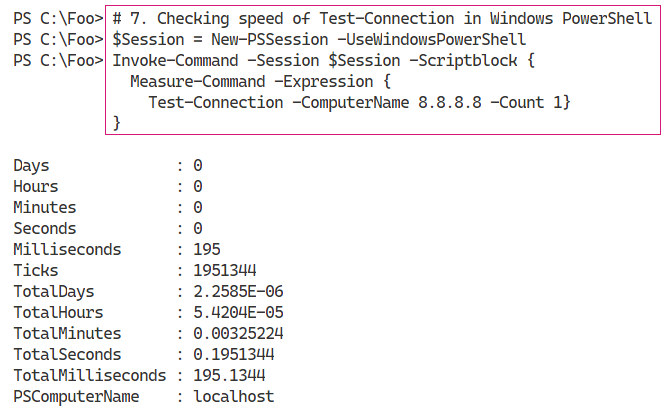


Figure 2.36: Checking the speed of Test-Connection in Windows PowerShell 5.1

Insert image B42024\_02\_36.png

## There's more...

In the output from step 1, you can see that the results are formatted differently from Windows PowerShell, with improvements to the output.

Step 2 shows how, in PowerShell 7, you can use the -IPv4 switch to use IP4 explicitly. Similarly, if you want to use IPv6 specifically, you can use the -IPv6 switch. Neither switch was available with Windows PowerShell.

In step 3, you determine the IP addresses for the host Dns.Google (8.8.8.8 and 8.8.4.4) which you then ping successfully. Note that this step both resolves the IP addresses into host names and performs the pings against each IP address. Google runs a free DNS service available to anyone on the Internet. You can find out more at https://developers.google.com/speed/public-dns.

In step 6 and step 7, you compare the speed of the Test-Connection command between Windows PowerShell 5.1 and PowerShell 7.1, As you can see in the output from these steps, the command is considerably faster in PowerShell 7.

# Using Select-String

The Select-String command, included with Windows PowerShell, has been improved in PowerShell 7. You use this command to search for strings either inside pipelined objects or within text files. This command is conceptually similar to the grep command in Linux. With PowerShell 7, the PowerShell team has added some excellent new features to this excellent command which you look at in this recipe.

## Getting Ready

You run this recipe on SRV1 after you have installed PowerShell 7 and Visual Studio Code, and you have created a console profile file.

## How to do it...

1. Get a file of text to work with

$Source       = 'https://www.gutenberg.org/files/1661/1661-0.txt'

$Destination  = 'C:\Foo\Sherlock.txt'

Start-BitsTransfer -Source $Source -Destination $Destination

1. Get book contents

$Contents = Get-Content -Path $Destination

1. Check the length of The Adventures of Sherlock Holmes

"The book is {0} lines long" -f $Contents.Length

1. Search for “Watson” in book contents

$Match1 = $Contents | Select-String -Pattern 'Watson'

"Watson is found {0} times" -f $Match1.Count

1. View first few matches

$Match1 | Select-Object -First 5

1. Search for “Dr. Watson” with a regular expression

$Contents | Select-String -Pattern 'Dr\. Watson'

1. Search for “Dr. Watson” using a simple match

$Contents | Select-String -Pattern 'Dr. Watson' -SimpleMatch

1. View output when searching from files

Get-ChildItem -Path $Destination |

  Select-String -Pattern 'Dr\. Watson'

## How it works…

In this recipe, you look at how you can use the Select-String cmdlet included with PowerShell 7. To investigate this cmdlet, you first download a text file.

In step 1, you use the Start‑BitsTransfer command to download the text for a book, *The Adventures of Sherlock Holmes* by Sir Arthur Conan Doyle, from the Project Gutenberg website. This step produces no output.

In step 2, you get the text from this book, which you store in the $Contents variable. This step produces no output. In step 3, you report the length of the book, which looks like this:

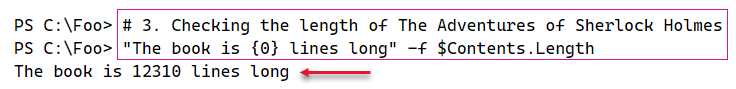


Figure 2.37: Checking the length of the book

Insert image B42024\_02\_37.png

In step 4, you search the book’s contents to find all occurrences of “Watson”, Sherlock Holmes’ faithful companion. This results in 81 occurrences, which looks like this:

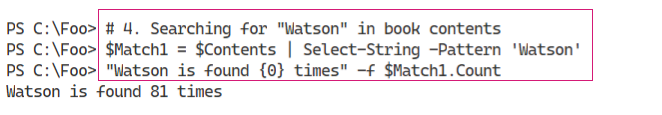


Figure 2.38: Searching for occurrences of “Watson” in the book

Insert image B42024\_02\_38.png

In step 5, you use the Select-Object cmdlet to view the first five times the command finds the term “Watson” in the book contents, which looks like this:

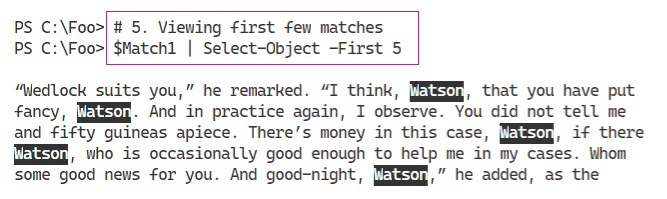


Figure 2.39: Viewing the first five “Watson” matches in the book

Insert image B42024\_02\_39.png

With Select-String, you can specify a regular expression with which to match the contents. In step 6, you specify a regular expression pattern to search for the string “Dr. Watson”. The output of this step looks like this:

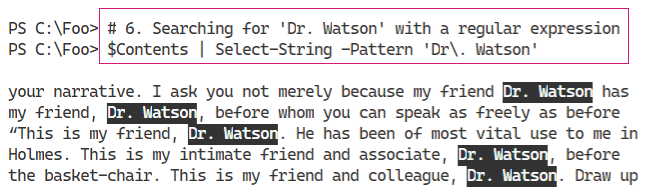


Figure 2.40: Searching for “Dr. Watson” with a regular expression pattern

Insert image B42024\_02\_40.png

As an alternative to using a regular expression to perform searching, Select-String also takes a simple match, as shown in the output of step 7:

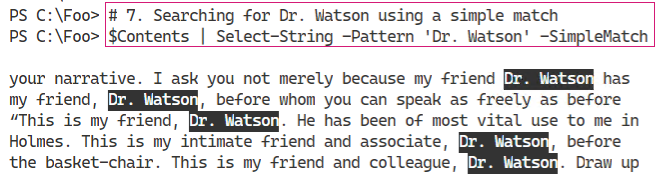


Figure 2.41: Searching for “Dr. Watson” using a simple match

Insert image B42024\_02\_41.png

In the previous steps, you have used Select-String to search for contents of a variable. Another valuable feature of Select-String is the ability to search for text in a file or even multiple files. You can see this in step 8, the output of which looks like:

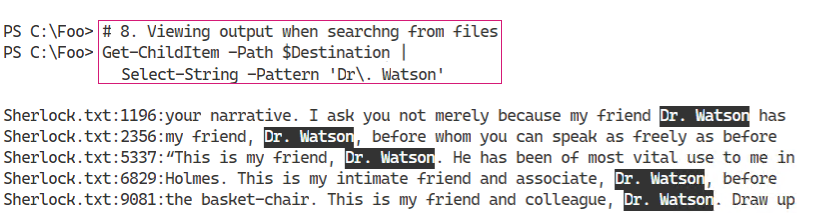


Figure 2.42: Searching for “Dr. Watson” in the Sherlock.txt file

Insert image B42024\_02\_42.png

## There's more...

In step 1 and step 2, you download a text file from Project Gutenberg, a free internet library of eBooks. This site contains a large number of free books in a variety of formats, including basic text. To find more free eBooks, visit the homepage at *https://www.gutenberg.org/* and to read more about the project, see: *https://www.gutenberg.org/about/.*

An essential improvement to Search-String in PowerShell 7 is the highlighting of the selected string, as you can see in the outputs of step 5, step 6, step 7, and step 8. From the command line, this makes viewing the output from Select-String much easier to consume. And the ability to search across multiple files, as shown in step 8, makes the Select-String cmdlet even more useful.

# Exploring Error View and Get-Error

Since the very beginning, Windows PowerShell has done a great job in displaying the results of errors: a big blob of red text on a black background that contains full details about what went wrong. It was tremendous, but many new users found it a bit off-putting – there was too much information, some of which was not very useful in most cases.

PowerShell 7 now offers a more concise view of errors which reduces the amount of text and improves the format of the output. The result is shorter and more readable output. And, on those rare occasions when it might be necessary, you can get complete error details without having to parse through $Error[0].

## Getting Ready

You run this recipe on SRV1 after you have installed PowerShell 7 and/or Visual Studio Code, and you have created a console profile file.

## How to do it...

1. Create a simple script

$SCRIPT = @'

  # divide by zero

  42/0

'@

$SCRIPTFILENAME = 'C:\Foo\ZeroDivError.ps1'

$SCRIPT | Out-File -Path $SCRIPTFILENAME

1. Run the script and see the default error view

& $SCRIPTFILENAME

1. Run the same line from the console

42/0

1. View $ErrorView variable

$ErrorView

1. View potential values of $ErrorView

$Type = $ErrorView.GetType().FullName

[System.Enum]::GetNames($Type)

1. Set $ErrorView to ‘NormalView’ and re-create the error

$ErrorView = 'NormalView'

& $SCRIPTFILENAME

1. Set $ErrorView to ‘CategoryView’ and re-create the error

$ErrorView = 'CategoryView'

& $SCRIPTFILENAME

1. Set $ErrorView to default value

$ErrorView = 'ConciseView'

## How it works...

In step 1, you create a script that contains a (deliberate) divide-by-zero error. This step creates the file but creates no other output.

In step 2, you run the script from within VS Code, and view the resulting error, which looks like this:

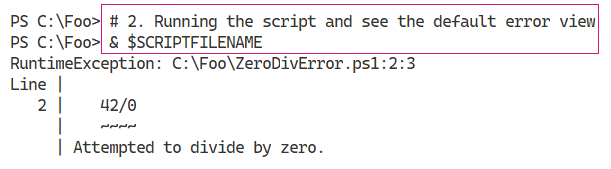


Figure 2.43: Running the script and viewing the error

Insert image B42024\_02\_43.png

In step 3, you create a divide by zero error from the command line. The output from this step looks like this:

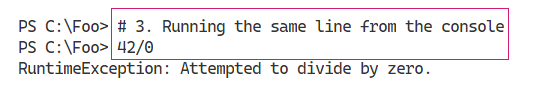


Figure 2.44: Running the same line from the console

Insert image B42024\_02\_44.png

PowerShell 7, uses the built-in variable $ErrorView to hold the name of the error view PowerShell should use to display errors. In step 4, you view the current value of this variable, which looks like this:

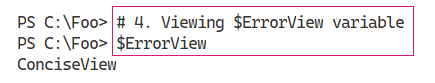


Figure 2.45: Viewing the value of the $ErrorView variable

Insert image B42024\_02\_45.png

The $ErrorView variable can take one of three values, as you can see from the output of step 5:

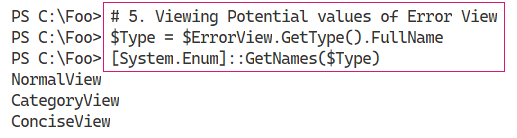


Figure 2.46: Viewing potential values of $ErrorView

Insert image B42024\_02\_46.png

In step 6, you set the value of $ErrorView to display the error using the output generated by Windows PowerShell and then re-view the error, which looks like this:

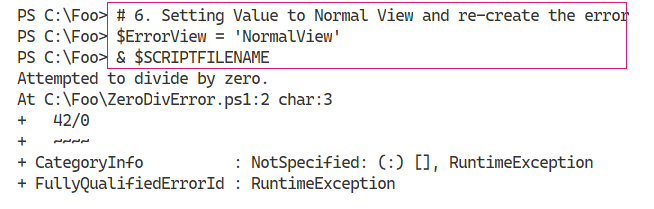


Figure 2.47: Setting $ErrorView to Normal View and recreating the error

Insert image B42024\_02\_47.png

In step 7, you set $ErrorView to display the error using Category View then re-create the error. The output from this step shows the category error view:

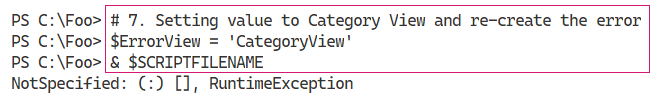


Figure 2.48: Setting $ErrorView to CategoryView and recreating the error

Insert image B42024\_02\_48.png

In step 8, you reset the value of $ErrorView to the default value. This step creates no output.

## There's more...

The concise error view you see in the output from step 2 contains all the information from the standard view which you can see in the output from step 7, except for the omission of the error category information. And if you invoke the error directly from the command line, as shown in step 3, you see only the error message, which is easier on the eyes.

In step 5, you view the error category information. In most cases, this is not particularly useful.

In step 8, you reset the value of $ErrorView. Depending on what you are doing, this step may not be needed. You can just exit the PowerShell console (or VS Code), and the next time you start PowerShell, it resets the value back to the default (ConciseView). And if you should prefer the normal or category error views, you can always set a value to $ErrorView in your profile file.

# Exploring Experimental Features

During the development of PowerShell Core, the PowerShell team have routinely added new features. Some of these new features could, at least in theory, break existing scripts and are called “experimental”. PowerShell does not, by default, enable any of these features. As shown in this recipe, you must enable them explicitly. This approach to experimental features enables you to test these new features and provide the PowerShell team with feedbacks. Should you find a feature that breaks a script for you - disable it. If you turn on (or turn off) an experimental feature, you need to restart PowerShell 7.

In general, experimental features are not intended to be used in production since the experimental features, by design, can be breaking. Also, experimental features are not officially supported. That being said, so far, these features have been very stable and reliable.

In this recipe, you look at the experimental features available in PowerShell 7.1 as released. If you are using later versions (for example. A PowerShell 7.2 preview release), you may see different experimental features. For a fuller look at PowerShell’s experimental features, see https://docs.microsoft.com/ powershell/scripting/learn/experimental-features?view=powershell-7.1

## Getting Ready

You run this recipe on SRV1 after you instal PowerShell 7 and/or Visual Studio Code, and you have created a console profile file.

## How to do it...

1. Discovering experimental features

Get-ExperimentalFeature -Name \* |

  Format-Table Name, Enabled, Description -Wrap

1. Examining command not found result with no experimental features available

Foo

1. Enabling one experimental feature as the current user

Get-ExperimentalFeature -Name \* |

  Select-Object -First 1 |

    Enable-ExperimentalFeature -Scope CurrentUser -Verbose

1. Enabling one experimental feature for all users

Get-ExperimentalFeature -Name \* |

  Select-Object -Skip 1 -First 1 |

    Enable-ExperimentalFeature -Scope AllUsers -Verbose

1. Start a new PowerShell console
2. Examining experimental features

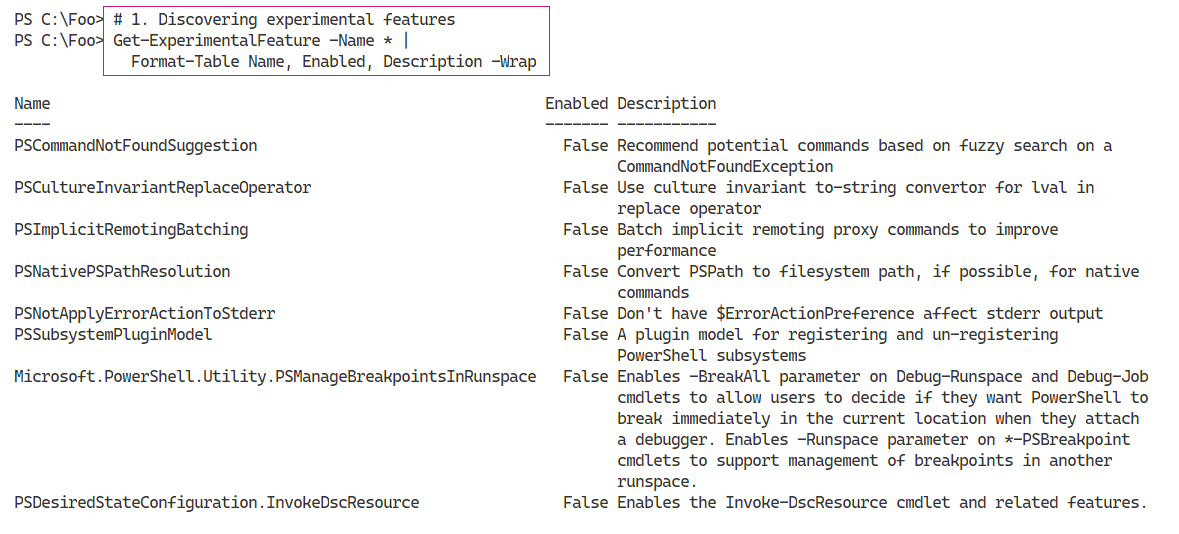
Get-ExperimentalFeature

1. Examining output from command not found suggestion feature

foo

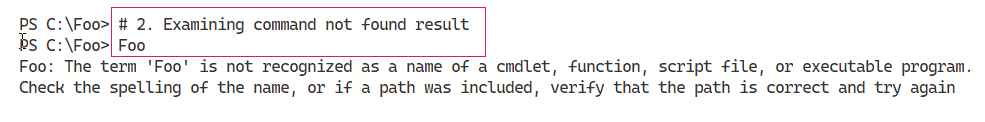
## How it works...

In step 1, you use the Get-ExperimentalFeature cmdlet to discover the available experimental features and their current state, which (by default) looks like this:



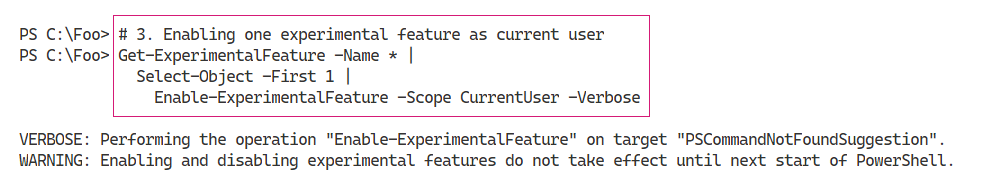
Insert image B42024\_02\_49.png

To test out an experimental feature, in step 2, you run a non-existent command, with output like this:



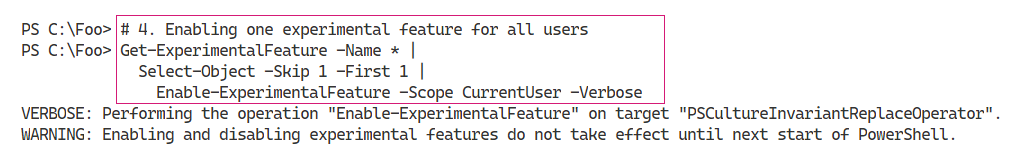
Insert image B42024\_02\_50.png

In step 3, you enable the first feature, the PSCommandNotFoundSuggestion experimental feature for the current user, which looks like this:



Insert image B42024\_02\_51.png

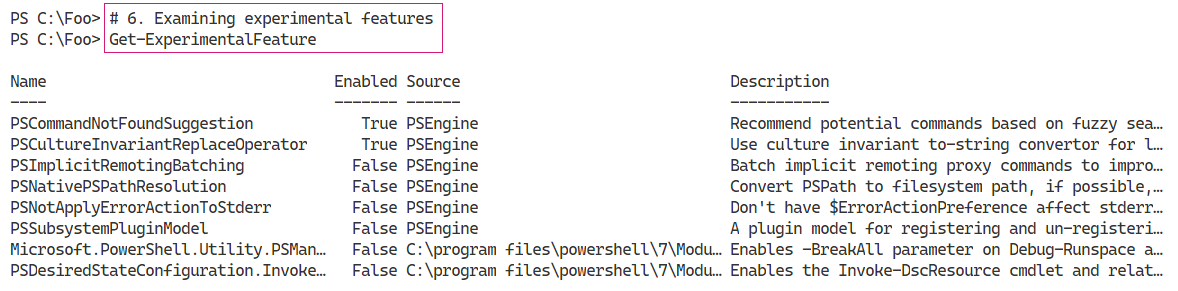
In step 4, you enable the second experimental feature, the PSCultureInvariantReplaceOperator, which looks like this:



Insert image B42024\_02\_52.png

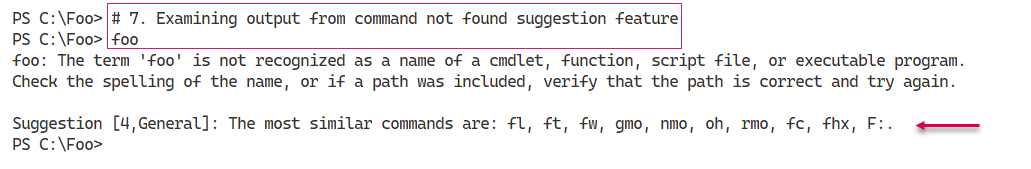
In step 5, you start a new version of PowerShell. If you are using VS Code to run this recipe, enter CTRL+SHIFT+’ to start a new terminal. If you are using the PowerShell 7 console, start a new copy of the console. This step produces no output as such.

In step 6, you examine the state of experimental features, noting that two new features are now available3, which looks like this;



Insert image B42024\_02\_53.png

In step 7, you re-run the unknown command to observe the command not found suggestions, which look like this:



Insert image B42024\_02\_54.png

## There's more...

In this recipe, you turn on two experimental features and examine one (command not found suggestions). In most cases, you should be safe to enable all of the experimental features, but it is always safer to turn them on one by one and test your scripts carefully.