O'REILLY®

4th Edition
Covers Open Source PowerShell Core & Windows PowerShell

Powershell Cookbook

Your Complete Guide to Scripting the Ubiquitous Object-Based Shell

Early Release
RAW & UNEDITED

Lee Holmes

PowerShell Cookbook

FOURTH EDITION

Your Complete Guide to Scripting the Ubiquitous Object-Based Shell

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Lee Holmes

PowerShell Cookbook

by Lee Holmes

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Acquisitions Editor: Suzanne McQuade
- Development Editor: Angela Rufino
- Production Editor: Kate Galloway
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- June 2021: Fourth Edition

Revision History for the Early Release

- 2021-01-21: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098101602 for release details.

the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10154-1

Chapter 1. The PowerShell Interactive Shell

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please visit *https://www.powershellcookbook.com/4th_ed_techreview*. You can also reach out to the author at powershellcookbook@leeholmes.com.

1.0 Introduction

Above all else, the design of PowerShell places priority on its use as an efficient and powerful interactive shell. Even its scripting language plays a critical role in this effort, as it too heavily favors interactive use.

What surprises most people when they first launch PowerShell is its similarity to the command prompt that has long existed as part of Windows. Familiar tools continue to run. Familiar commands continue to run. Even familiar hotkeys are the same. Supporting this familiar user interface, though, is a powerful engine that lets you accomplish once cumbersome administrative and scripting tasks with ease.

This chapter introduces PowerShell from the perspective of its interactive shell.

1.1 Install PowerShell Core

Problem

You want to install the most recent version of PowerShell on your Windows,

You want to install the most recent version of PowerShell on your Windows, Mac, or Linux system.

Solution

Visit *https://microsoft.com/PowerShell* to find the installation instructions for the operating system and platform you want to install on. For the most common:

Windows

Install PowerShell from Microsoft through the the Microsoft Store application in the Start Menu. Then, install Windows Terminal from Microsoft through the Microsoft Store application in the Start Menu.

Mac

Install PowerShell from Homebrew:

```
brew install --cask powershell
```

Linux

Installation instructions vary per Linux distribution, but the most common distribution among PowerShell Core users is Ubuntu:

```
# Update the list of packages
sudo apt-get update

# Install pre-requisite packages.
sudo apt-get install -y wget apt-transport-https software-properties-
common

# Download the Microsoft repository GPG keys
wget -q https://packages.microsoft.com/config/ubuntu/20.04/packages-
microsoft-prod.deb

# Register the Microsoft repository GPG keys
sudo dpkg -i packages-microsoft-prod.deb

# Update the list of packages after we added packages.microsoft.com
sudo apt-get update

# Install PowerShell
sudo apt-get install -y powershell
```

Discussion

PowerShell has already led a long and exciting life. For the first fifteen years of its existence, it was known as “Windows PowerShell”: a fantastic object-based management shell and platform that made it easy and fun for administrators, developers, and power users to get their jobs done.

In its earliest stages, this support came as part of the “Windows Management Framework”: a standalone download that provided this much needed functionality on Windows. Windows PowerShell eventually became part of Windows itself, and has been a core part of the operating system since Windows 7.

In 2016, PowerShell made a tremendous shift by announcing that it would ship PowerShell on multiple operating system platforms — and by the way — open source the entire project at the same time! Windows PowerShell got a new name with its new future: simply *PowerShell*. This major change opened the doors for vastly quicker innovation, community participation, and availability through avenues that previously would never have been possible. While the classic Windows PowerShell is still included in the operating system by default, it no longer receives updates and should be avoided.

Installing and Running PowerShell on Windows

As mentioned in the Solution, the best way to get PowerShell is to install it through the Windows Store. This makes it easy to install and easy to update. Once you’ve installed it, you can find PowerShell in the Start Menu like you would any other application.

> **NOTE**
>
> If you want to install a system-wide version of PowerShell for automation and other adminstration tasks, you will likely prefer the MSI-based installation mechanism. For more information, see *https://microsoft.com/PowerShell*.

While you’re installing PowerShell from the Windows Store, now is a good time to install the Windows Terminal application from the Windows Store. The traditional console interface (the window that PowerShell runs inside of)

included in Windows has so many tools and applications depending on its exact quirks that it is nearly impossible to meaningfully change. It has fallen woefully behind on what you would expect of a terminal console interface, so the Windows Terminal application from the Windows Store is the solution. Like PowerShell — it is open source, a focus of rapid innovation, and a vast improvement to what ships in Windows by default.

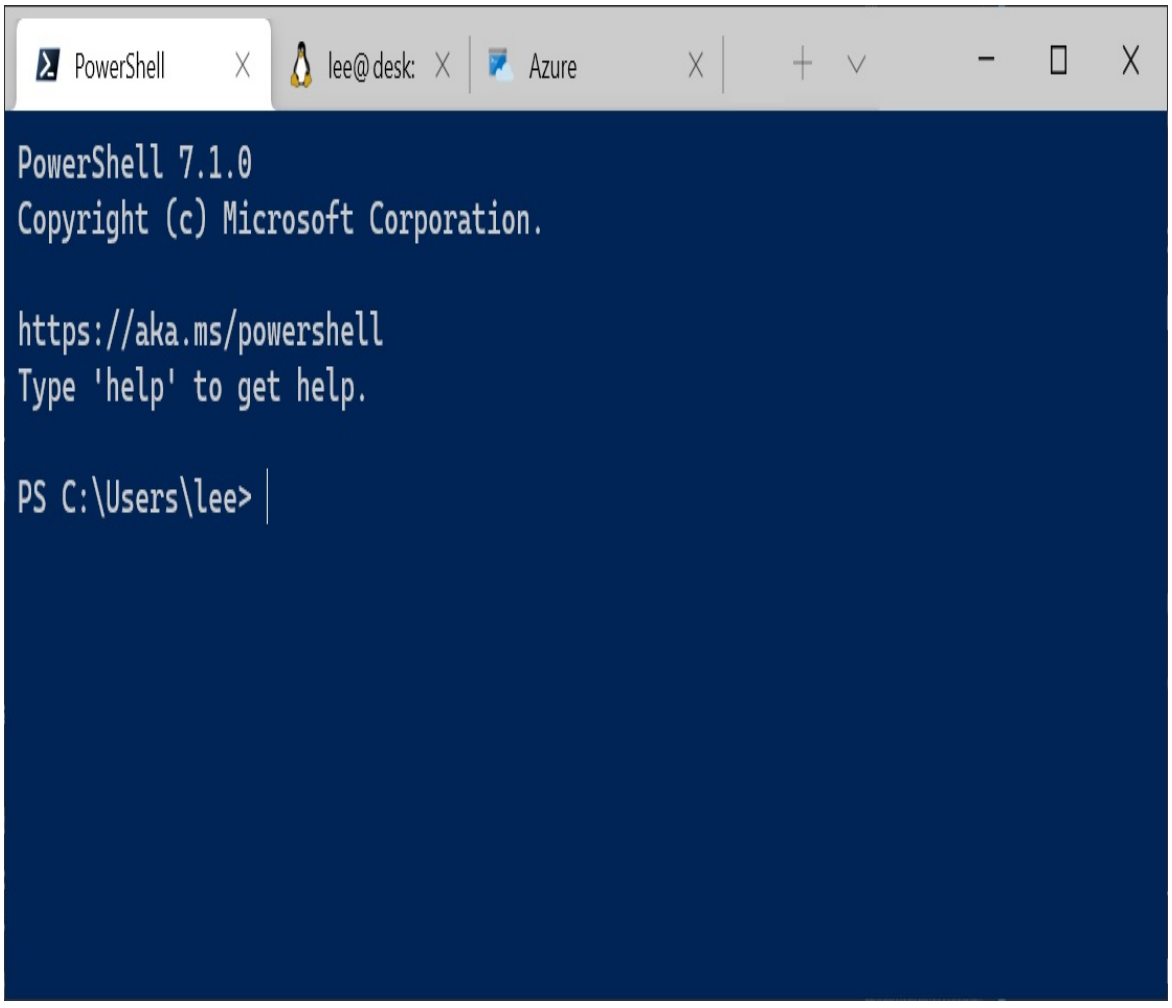

Figure 1-1. Windows Terminal running PowerShell, Bash, and even Azure Cloud Shell!

You can run many shells within tabs in Windows Terminal: PowerShell, Windows PowerShell, cmd.exe, Bash (if you've enabled the Windows Subsystem for Linux), and even a connection to Azure Cloud Shell. Windows Terminal defaults to PowerShell if you have it installed.

Customizing PowerShell on Windows Terminal

There are two changes to a default Windows Terminal + PowerShell installation that really improve the experience: taskbar pinning, and themes.

Taskbar pinning

When you launch Windows Terminal, right-click on its taskbar icon. Select “Pin to Taskbar”, and then drag the icon to the far left of the taskbar. From now on, whenever you press the Windows Key + `1` at the same time, you will either launch Windows Terminal and PowerShell (if it’s not already open), or activate it. This is an incredible way to keep your favorite shell close at hand.

Themes

Windows Powershell has a gorgeous Noble Blue theme. It is easy on the eyes, quick to identify, and sets it apart from the dozens of other shells out there. PowerShell Core did not take this color scheme with it by default, but it is still possible to customize your installation. From Windows Terminal, press `Ctrl+Comma` or click the downward arrow on the right-hand side of the tab strip to open the Settings dialog of Windows Terminal. The file that contains these settings will open in your default text editor. Under `Profiles`, find the item with `Windows.Terminal.PowershellCore` as its source, and add `Campbell PowerShell` as a `colorScheme`. The result should look like this:

```
{
    "guid": ...
    "hidden": false,
    "name": "PowerShell",
    "colorScheme": "Campbell Powershell",
    "source": "Windows.Terminal.PowershellCore"
},
```

Pay attention to quotes, colons, and commas, and you should have your PowerShell sessions looking noble again in no time!

Installing and Running PowerShell on Mac and Linux

For the most part, installing PowerShell on Mac and Linux follows the patterns that you are likely already familiar with.

On Mac, the recommended installation method is through the popular

Homebrew package manager. Homebrew is not installed by default on MacOS, but installation is quite easy. If you haven't installed Homebrew yet, you can find instructions at *https://brew.sh*.

On Linux, the installation methods vary depending on the distribution you are interested in. For the most part, installation is as simple as registering the Microsoft repository for your distribution's package manager, and then installing PowerShell. The Solution provides an example specific to Ubuntu `20.04`, but you can specific instructions for your distribution and specific version here: *https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-core-on-linux*.

1.2 Run Programs, Scripts, and Existing Tools

Problem

You rely on a lot of effort invested in your current tools. You have traditional executables, Perl scripts, VBScript, and of course, a legacy build system that has organically grown into a tangled mess of batch files. You want to use PowerShell, but you don't want to give up everything you already have.

Solution

To run a program, script, batch file, or other executable command in the system's path, enter its filename. For these executable types, the extension is optional:

```
Program.exe arguments
ScriptName.ps1 arguments
BatchFile.cmd arguments
```

To run a command that contains a space in its name, enclose its filename in single-quotes (`'`) and precede the command with an ampersand (&), known in PowerShell as the *invoke operator*:

```
& 'C:\Program Files\Program\Program.exe' arguments
```

To run a command in the current directory, place `.\` in front of its filename:

```
.\Program.exe arguments
```

To run a command with spaces in its name from the current directory, precede it with both an ampersand and `.\`:

```
& '.\Program With Spaces.exe' arguments
```

Discussion

In this case, the solution is mainly to use your current tools as you always have. The only difference is that you run them in the PowerShell interactive shell rather than *cmd.exe*.

Specifying the command name

The final three tips in the Solution merit special attention. They are the features of PowerShell that many new users stumble on when it comes to running programs. The first is running commands that contain spaces. In *cmd.exe*, the way to run a command that contains spaces is to surround it with quotes:

```
"C:\Program Files\Program\Program.exe"
```

In PowerShell, though, placing text inside quotes is part of a feature that lets you evaluate complex expressions at the prompt, as shown in Example 1-1.

Example 1-1. Evaluating expressions at the PowerShell prompt

```
PS > 1 + 1
2
PS > 26 * 1.15
29.9
PS > "Hello" + " World"
Hello World
PS > "Hello World"
Hello World
PS > "C:\Program Files\Program\Program.exe"
C:\Program Files\Program\Program.exe
PS >
```

So, a program name in quotes is no different from any other string in quotes. It's just an expression. As shown previously, the way to run a command in a string is

to precede that string with the invoke operator (&). If the command you want to run is a batch file that modifies its environment, see Recipe 3.5.

> **NOTE**
>
> By default, PowerShell's security policies prevent scripts from running. Once you begin writing or using scripts, though, you should configure this policy to something less restrictive. For information on how to configure your execution policy, see Recipe 18.1.

The second command that new users (and seasoned veterans before coffee!) sometimes stumble on is running commands from the current directory. In *cmd.exe*, the current directory is considered part of the *path*: the list of directories that Windows searches to find the program name you typed. If you are in the *C:\Programs* directory, *cmd.exe* looks in *C:\Programs* (among other places) for applications to run.

PowerShell, like most Unix shells, requires that you explicitly state your desire to run a program from the current directory. To do that, you use the `.\Program.exe` syntax, as shown previously. This prevents malicious users on your system from littering your hard drive with evil programs that have names similar to (or the same as) commands you might run while visiting that directory.

To save themselves from having to type the location of commonly used scripts and programs, many users put commonly used utilities along with their PowerShell scripts in a "tools" directory, which they add to their system's path. If PowerShell can find a script or utility in your system's path, you do not need to explicitly specify its location.

If you want PowerShell to automatically look in your current working directory for scripts, you can add a period (`.`) to your PATH environment variable.

For more information about updating your system path, see Recipe 16.2.

If you want to capture the output of a command, you can either save the results into a variable, or save the results into a file. To save the results into a variable, see Recipe 3.3. To save the results into a file, see Recipe 9.2.

Specifying command arguments

To specify arguments to a command, you can again type them just as you would in other shells. For example, to make a specified file read-only (two arguments to `attrib.exe`), simply type:

```
attrib +R c:\path\to\file.txt
```

Where many scripters get misled when it comes to command arguments is how to change them within your scripts. For example, how do you get the filename from a PowerShell variable? The answer is to define a variable to hold the argument value, and just use that in the place you used to write the command argument:

```
$filename = "c:\path\to\other\file.txt"
attrib +R $filename
```

You can use the same technique when you call a PowerShell cmdlet, script, or function:

```
$filename = "c:\path\to\other\file.txt"
Get-Acl -Path $filename
```

If you see a solution that uses the `Invoke-Expression` cmdlet to compose command arguments, it is almost certainly incorrect. The `Invoke-Expression` cmdlet takes the string that you give it and treats it like a full PowerShell script. As just one example of the problems this can cause, consider the following: filenames are allowed to contain the semicolon (`;`) character, but when `Invoke-Expression` sees a semicolon, it assumes that it is a new line of PowerShell script. For example, try running this:

```
$filename = "c:\file.txt; Write-Warning 'This could be bad'"
Invoke-Expression "Get-Acl -Path $filename"
```

Given that these dynamic arguments often come from user input, using `Invoke-Expression` to compose commands can (at best) cause unpredictable script results. Worse, it could result in damage to your system or a security vulnerability.

In addition to letting you supply arguments through variables one at a time,

PowerShell also lets you supply several of them at once through a technique known as *splatting*. For more information about splatting, see Recipe 11.14.

See Also

Recipe 3.3

Recipe 3.5

Recipe 11.14

Recipe 16.2

Recipe 18.1

1.3 Run a PowerShell Command

Problem

You want to run a PowerShell command.

Solution

To run a PowerShell command, type its name at the command prompt. For example:

```
PS > Get-Process

Handles  NPM(K)    PM(K)    WS(K)   VM(M)  CPU(s)    Id ProcessName
-------  ------    -----    -----   -----  ------    -- -----------
    133       5    11760     7668      46          1112 audiodg
    184       5    33248      508      93          1692 avgamsvr
    143       7    31852      984      97          1788 avgemc
```

Discussion

The `Get-Process` command is an example of a native PowerShell command, called a *cmdlet*. As compared to traditional commands, cmdlets provide significant benefits to both administrators and developers:

- They share a common and regular command-line syntax.

- They support rich pipeline scenarios (using the output of one command as the input of another).
- They produce easily manageable object-based output, rather than error-prone plain-text output.

Because the `Get-Process` cmdlet generates rich object-based output, you can use its output for many process-related tasks.

Every PowerShell command lets you provide input to the command through its *parameters*. For more information on providing input to commands, see Appendix A.

The `Get-Process` cmdlet is just one of the many that PowerShell supports. See Recipe 1.12 to learn techniques for finding additional commands that PowerShell supports.

For more information about working with classes from the .NET Framework, see Recipe 3.8.

See Also

Recipe 1.12

Recipe 3.8

Appendix A

1.4 Resolve Errors Calling Native Executables

Problem

You have a command line that works from *cmd.exe*, and want to resolve errors that occur from running that command in PowerShell.

Solution

Enclose any affected command arguments in single quotes to prevent them from being interpreted by PowerShell, and replace any single quotes in the command with two single quotes.

```
PS > cmd /c echo '!"#$%&''()*+,-./09:;<=>?@AZ[\]^_`az{|}~'
!"#$%&'()*+,-./09:;<=>?@AZ[\]^_`az{|}~
```

For complicated commands where this does not work, use the verbatim argument (`--%`) syntax.

```
PS > cmd /c echo 'quotes' "and" $variables @{ etc = $true }
quotes and System.Collections.Hashtable

PS > cmd --% /c echo 'quotes' "and" $variables @{ etc = $true }
'quotes' "and" $variables @{ etc = $true }
```

Discussion

One of PowerShell's primary goals has always been command consistency. Because of this, cmdlets are very regular in the way that they accept parameters. Native executables write their own parameter parsing, so you never know what to expect when working with them. In addition, PowerShell offers many features that make you more efficient at the command line: command substitution, variable expansion, and more. Since many native executables were written before PowerShell was developed, they may use special characters that conflict with these features.

As an example, the command given in the Solution uses all the special characters available on a typical keyboard. Without the quotes, PowerShell treats some of them as language features, as shown in Table 1-1.

Table 1-1. Sample of special characters

Special character	Meaning
"	The beginning (or end) of quoted text
#	The beginning of a comment
$	The beginning of a variable
&	Reserved for future use
( )	Parentheses used for subexpressions
;	Statement separator

`{ }`	Script block
`|`	Pipeline separator
`` ` ``	Escape character

When surrounded by single quotes, PowerShell accepts these characters as written, without the special meaning.

Despite these precautions, you may still sometimes run into a command that doesn't seem to work when called from PowerShell. For the most part, these can be resolved by reviewing what PowerShell passes to the command and escaping the special characters.

To see *exactly* what PowerShell passes to that command, you can view the output of the trace source called `NativeCommandParameterBinder`:

```
PS > Trace-Command NativeCommandParameterBinder {
    cmd /c echo '!"#$%&''()*+,-./09:;<=>?@AZ[\]^_`az{|}~'
} -PsHost

DEBUG: NativeCommandParameterBinder Information: 0 :  WriteLine
Argument 0: /c
DEBUG: NativeCommandParameterBinder Information: 0 :  WriteLine
Argument 1: echo
DEBUG: NativeCommandParameterBinder Information: 0 :  WriteLine
Argument 2: !#$%&'()*+,-./09:;<=>?@AZ[\]^_`az{|}~
!"#$%&'()*+,-./09:;<=>?@AZ[\]^_`az{|}~
```

If the command arguments shown in this output don't match the arguments you expect, they have special meaning to PowerShell and should be escaped.

For a complex enough command that "just used to work," though, escaping special characters is tiresome. To escape the whole command invocation, use the verbatim argument marker (`--%`) to prevent PowerShell from interpreting any of the remaining characters on the line. You can place this marker anywhere in the command's arguments, letting you benefit from PowerShell constructs where appropriate. The following example uses a PowerShell variable for some of the command arguments, but then uses verbatim arguments for the rest:

```
PS > $username = "Lee"
PS > cmd /c echo Hello $username with 'quotes' "and" $variables @{ etc
= $true }
```

```
Hello Lee with quotes and System.Collections.Hashtable
PS > cmd /c echo Hello $username `
     --% with 'quotes' "and" $variables @{ etc = $true }
Hello Lee with 'quotes' "and" $variables @{ etc = $true }
```

While in this mode, PowerShell also accepts *cmd.exe*-style environment variables—as these are frequently used in commands that "just used to work":

```
PS > $env:host = "myhost"
PS > ping %host%
Ping request could not find host %host%. Please check the name and try
again.
PS > ping --% %host%

Pinging myhost [127.0.1.1] with 32 bytes of data:
(...)
```

See Also

Appendix A

1.5 Supply Default Values for Parameters

Problem

You want to define a default value for a parameter in a PowerShell command.

Solution

Add an entry to the `PSDefaultParameterValues` hashtable.

```
PS > Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    150      13     9692       9612    39    21.43    996 audiodg
   1013      84    45572      42716   315     1.67   4596 WWAHost
(...)

PS > $PSDefaultParameterValues["Get-Process:ID"] = $pid
PS > Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
```

```
            ` ,       ` ,         ` ,    ` ,       ` ,
-------  ------    -----       -----  -----   ------     -- -----------
    584      62   132776     157940   985    13.15   9104
powershell_ise

PS > Get-Process -Id 0

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
      0       0        0         20     0                0 Idle
```

Discussion

In PowerShell, many commands (cmdlets and advanced functions) have parameters that let you configure their behavior. For a full description of how to provide input to commands, see Appendix A. Sometimes, though, supplying values for those parameters at each invocation becomes awkward or repetitive.

In early versions of PowerShell, it was the responsibility of each cmdlet author to recognize awkward or repetitive configuration properties and build support for "preference variables" into the cmdlet itself. For example, the `Send-MailMessage` cmdlet looks for the `$PSEmailServer` variable if you do not supply a value for its `-SmtpServer` parameter.

To make this support more consistent and configurable, PowerShell supports the `PSDefaultParameterValues` preference variable. This preference variable is a hashtable. Like all other PowerShell hashtables, entries come in two parts: the key and the value.

Keys in the `PSDefaultParameterValues` hashtable must match the pattern *`cmdlet`*:*`parameter`*—that is, a cmdlet name and parameter name, separated by a colon. Either (or both) may use wildcards, and spaces between the command name, colon, and parameter are ignored.

Values for the cmdlet/parameter pairs can be either a simple parameter value (a string, boolean value, integer, etc.) or a script block. Simple parameter values are what you will use most often.

If you need the default value to dynamically change based on what parameter values are provided so far, you can use a script block as the default. When you do so, PowerShell evaluates the script block and uses its result as the default value. If your script block doesn't return a result, PowerShell doesn't apply a default value.

When PowerShell invokes your script block, `$args[0]` contains information about any parameters bound so far: `BoundDefaultParameters`, `BoundParameters`, and `BoundPositionalParameters`. As one example of this, consider providing default values to the `-Credential` parameter based on the computer being connected to. Here is a function that simply outputs the credential being used:

```
function RemoteConnector
{
    param(
        [Parameter()]
        $ComputerName,

        [Parameter(Mandatory = $true)]
        $Credential)

    "Connecting as " + $Credential.UserName
}
```

Now, you can define a credential map:

```
PS > $credmap = @{}
PS > $credmap["RemoteComputer1"] = Get-Credential
PS > $credmap["RemoteComputer2"] = Get-Credential
```

Then, create a parameter default for all `Credential` parameters that looks at the `ComputerName` bound parameter:

```
$PSDefaultParameterValues["*:Credential"] = {
    if($args[0].BoundParameters -contains "ComputerName")
    {
        $cred = $credmap[$PSBoundParameters["ComputerName"]]
        if($cred) { $cred }
    }
}
```

Here is an example of this in use:

```
PS > RemoteConnector -ComputerName RemoteComputer1
Connecting as UserForRemoteComputer1
PS > RemoteConnector -ComputerName RemoteComputer2
Connecting as UserForRemoteComputer2
PS > RemoteConnector -ComputerName RemoteComputer3
```

```
PS > RemoteConnector -ComputerName RemoteComputer3

cmdlet RemoteConnector at command pipeline position 1
Supply values for the following parameters:
Credential: (...)
```

For more information about working with hashtables in PowerShell, see Appendix A.

See Also

Appendix A

1.6 Invoke a Long-Running or Background Command

Problem

You want to invoke a long-running command on a local or remote computer.

Solution

Invoke the command as a `Job` to have PowerShell run it in the background:

```
PS > Start-Job { while($true) { Get-Random; Start-Sleep 5 } } -Name
Sleeper

Id              Name            State      HasMoreData     Location
--              ----            -----      -----------     --------
1               Sleeper         Running    True            localhost

PS > Receive-Job Sleeper
671032665
1862308704
PS > Stop-Job Sleeper
```

Or, if your command is a single pipeline, place a *&* character at the end of the line to run that pipeline in the background:

```
PS > dir c:\windows\system32 -recurse &

Id     Name            PSJobTypeName   State         HasMore
```

```
Id     Name            PSJobTypeName   State         HasMoreData
--     ----            -------------   -----         -------
1      Job1            BackgroundJob   Running       True

PS > 1+1
2

PS > Receive-Job -id 1 | Select -First 5

    Directory: C:\Windows\System32

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----           12/7/2019  1:50 AM                0409
d----           11/5/2020  7:09 AM                1028
d----           11/5/2020  7:09 AM                1029
d----           11/5/2020  7:09 AM                1031
d----           11/5/2020  7:09 AM                1033
```

Discussion

PowerShell's job cmdlets provide a consistent way to create and interact with background tasks. In the Solution, we use the `Start-Job` cmdlet to launch a background job on the local computer. We give it the name of `Sleeper`, but otherwise we don't customize much of its execution environment.

In addition to allowing you to customize the job name, the `Start-Job` cmdlet also lets you launch the job under alternate user credentials or as a 32-bit process (if run originally from a 64-bit process).

As an alternative to the `Start-Job` cmdlet, you can also use the `Start-ThreadJob` cmdlet. The `Start-ThreadJob` cmdlet is a bit quicker at starting background jobs and also lets you supply and interact with live objects in the jobs that you create. However, it consumes resources of your current PowerShell process and does not let you run your job under alternate user credentials.

Once you have launched a job, you can use the other `Job` cmdlets to interact with it:

`Get-Job`

> Gets all jobs associated with the current session. In addition, the `-Before`, `-After`, `-Newest`, and `-State` parameters let you filter jobs based on

their state or completion time.

`Wait-Job`

Waits for a job until it has output ready to be retrieved.

`Receive-Job`

Retrieves any output the job has generated since the last call to `Receive-Job`.

`Stop-Job`

Stops a job.

`Remove-Job`

Removes a job from the list of active jobs.

> **NOTE**
>
> In addition to the `Start-Job` cmdlet, you can also use the `-AsJob` parameter in many cmdlets to have them perform their tasks in the background. Two of the most useful examples are the `Invoke-Command` cmdlet (when operating against remote computers) and the set of WMI-related cmdlets.

If your job generates an error, the `Receive-Job` cmdlet will display it to you when you receive the results, as shown in Example 1-2. If you want to investigate these errors further, the object returned by `Get-Job` exposes them through the `Error` property.

Example 1-2. Retrieving errors from a Job

```
PS > Start-Job -Name ErrorJob { Write-Error Error! }

Id              Name            State      HasMoreData     Location
--              ----            -----      -----------     --------
1               ErrorJob        Running    True            localhost


PS > Receive-Job ErrorJob
Error!
    + CategoryInfo          : NotSpecified: (:) [Write-Error],
WriteError
   Exception
```

```
    + FullyQualifiedErrorId :
Microsoft.PowerShell.Commands.WriteErrorExc
   eption,Microsoft.PowerShell.Commands.WriteErrorCommand

PS > $job = Get-Job ErrorJob
PS > $job | Format-List *

State         : Completed
HasMoreData   : False
StatusMessage :
Location      : localhost
Command       :  Write-Error Error!
JobStateInfo  : Completed
Finished      : System.Threading.ManualResetEvent
InstanceId    : 801e932c-5580-4c8b-af06-ddd1024840b7
Id            : 1
Name          : ErrorJob
ChildJobs     : {Job2}
Output        : {}
Error         : {}
Progress      : {}
Verbose       : {}
Debug         : {}
Warning       : {}

PS > $job.ChildJobs[0] | Format-List *
State         : Completed
StatusMessage :
HasMoreData   : False
Location      : localhost
Runspace      : System.Management.Automation.RemoteRunspace
Command       :  Write-Error Error!
JobStateInfo  : Completed
Finished      : System.Threading.ManualResetEvent
InstanceId    : 60fa85da-448b-49ff-8116-6eae6c3f5006
Id            : 2
Name          : Job2
ChildJobs     : {}
Output        : {}
Error         :
{Microsoft.PowerShell.Commands.WriteErrorException,Microso
                ft.PowerShell.Commands.WriteErrorCommand}
Progress      : {}
Verbose       : {}
Debug         : {}
Warning       : {}


PS > $job.ChildJobs[0].Error
Error!
```

```
    + CategoryInfo          : NotSpecified: (:) [Write-Error],
WriteError
   Exception
    + FullyQualifiedErrorId :
Microsoft.PowerShell.Commands.WriteErrorExc
   eption,Microsoft.PowerShell.Commands.WriteErrorCommand

PS >
```

As this example shows, jobs are sometimes containers for other jobs, called *child jobs*. Jobs created through the `Start-Job` cmdlet will always be child jobs attached to a generic container. To access the errors returned by these jobs, you instead access the errors in its first child job (called child job number zero).

In addition to long-running jobs that execute under control of the current PowerShell session, you might want to register and control jobs that run on a schedule, or independently of the current PowerShell session. PowerShell has a handful of commands to let you work with scheduled jobs like this; for more information, see Recipe 27.14.

See Also

Recipe 27.14

Recipe 28.7

Recipe 29.5

1.7 Program: Monitor a Command for Changes

As thrilling as our lives are, some days are reduced to running a command over and over and over. Did the files finish copying yet? Is the build finished? Is the site still up?

Usually, the answer to these questions comes from running a command, looking at its output, and then deciding whether it meets your criteria. And usually this means just waiting for the output to change, waiting for some text to appear, or waiting for some text to disappear.

Fortunately, Example 1-3 automates this tedious process for you.

Example 1-3. Watch-Command.ps1

```
##############################################################################
```

```
######
##
## Watch-Command
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
#############################################################################
######

<#

.SYNOPSIS

Watches the result of a command invocation, alerting you when the output
either matches a specified string, lacks a specified string, or has
simply
changed.

.EXAMPLE

PS > Watch-Command { Get-Process -Name Notepad | Measure } -UntilChanged
Monitors Notepad processes until you start or stop one.

.EXAMPLE

PS > Watch-Command { Get-Process -Name Notepad | Measure } -Until "Count
: 1"
Monitors Notepad processes until there is exactly one open.

.EXAMPLE

PS > Watch-Command {
     Get-Process -Name Notepad | Measure } -While 'Count    : \d\s*\n'
Monitors Notepad processes while there are between 0 and 9 open
(once number after the colon).

#>

[CmdletBinding(DefaultParameterSetName = "Forever")]
param(
    ## The script block to invoke while monitoring
    [Parameter(Mandatory = $true, Position = 0)]
    [ScriptBlock] $ScriptBlock,

    ## The delay, in seconds, between monitoring attempts
    [Parameter()]
    [Double] $DelaySeconds = 1,

    ## Specifies that the alert sound should not be played
```

```
    [Parameter()]
    [Switch] $Quiet,

    ## Monitoring continues only while the output of the
    ## command remains the same.
    [Parameter(ParameterSetName = "UntilChanged", Mandatory = $false)]
    [Switch] $UntilChanged,

    ## The regular expression to search for. Monitoring continues
    ## until this expression is found.
    [Parameter(ParameterSetName = "Until", Mandatory = $false)]
    [String] $Until,

    ## The regular expression to search for. Monitoring continues
    ## until this expression is not found.
    [Parameter(ParameterSetName = "While", Mandatory = $false)]
    [String] $While
)

Set-StrictMode -Version 3

$initialOutput = ""
$lastCursorTop = 0
Clear-Host

## Start a continuous loop
while($true)
{
    ## Run the provided script block
    $r = & $ScriptBlock

    ## Clear the screen and display the results
    $buffer = $ScriptBlock.ToString().Trim() + "`r`n"
    $buffer += "`r`n"
    $textOutput = $r | Out-String
    $buffer += $textOutput

    [Console]::SetCursorPosition(0, 0)
    [Console]::Write($buffer)

    $currentCursorTop = [Console]::CursorTop
    $linesToClear = $lastCursorTop - $currentCursorTop
    if($linesToClear -gt 0)
    {
        [Console]::Write((" " * [Console]::WindowWidth * $linesToClear))
    }

    $lastCursorTop = [Console]::CursorTop
    [Console]::SetCursorPosition(0, 0)
```

```
    ## Remember the initial output, if we haven't
    ## stored it yet
    if(-not $initialOutput)
    {
        $initialOutput = $textOutput
    }

    ## If we are just looking for any change,
    ## see if the text has changed.
    if($UntilChanged)
    {
        if($initialOutput -ne $textOutput)
        {
            break
        }
    }

    ## If we need to ensure some text is found,
    ## break if we didn't find it.
    if($While)
    {
        if($textOutput -notmatch $While)
        {
            break
        }
    }

    ## If we need to wait for some text to be found,
    ## break if we find it.
    if($Until)
    {
        if($textOutput -match $Until)
        {
            break
        }
    }

    ## Delay
    Start-Sleep -Seconds $DelaySeconds
}

## Notify the user
if(-not $Quiet)
{
    [Console]::Beep(1000, 1000)
}
```

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

1.8 Notify Yourself of Job Completion

Problem

You want to notify yourself when a long-running job completes.

Solution

Use the `Register-TemporaryEvent` command given in Recipe 32.3 to register for the event's `StateChanged` event:

```
PS > $job = Start-Job -Name TenSecondSleep { Start-Sleep 10 }
PS > Register-TemporaryEvent $job StateChanged -Action {
     [Console]::Beep(100,100)
     Write-Host "Job #$($sender.Id) ($($sender.Name)) complete."
}

PS > Job #6 (TenSecondSleep) complete.
PS >
```

Discussion

When a job completes, it raises a `StateChanged` event to notify subscribers that its state has changed. We can use PowerShell's event handling cmdlets to register for notifications about this event, but they are not geared toward this type of one-time event handling. To solve that, we use the `Register-TemporaryEvent` command given in Recipe 32.3.

In our example action block in the Solution, we simply emit a beep and write a message saying that the job is complete.

As another option, you can also update your `prompt` function to highlight jobs that are complete but still have output you haven't processed:

```
$psJobs = @(Get-Job -State Completed | ? { $_.HasMoreData })
if($psJobs.Count -gt 0) {
    ($psJobs | Out-String).Trim() | Write-Host -Fore Yellow }
```

For more information about events and this type of automatic event handling, see Chapter 32.

See Also

Recipe 1.2

Chapter 32

1.9 Customize Your Shell, Profile, and Prompt

Problem

You want to customize PowerShell's interactive experience with a personalized prompt, aliases, and more.

Solution

When you want to customize aspects of PowerShell, place those customizations in your personal profile script. PowerShell provides easy access to this profile script by storing its location in the `$profile` variable.

> **NOTE**
>
> By default, PowerShell's security policies prevent scripts (including your profile) from running. Once you begin writing scripts, though, you should configure this policy to something less restrictive. For information on how to configure your execution policy, see Recipe 18.1.

To create a new profile (and overwrite one if it already exists):

```
New-Item -type file -force $profile
```

To edit your profile (in the Integrated Scripting Environment):

```
ise $profile
```

To see your profile file:

```
Get-ChildItem $profile
```

Once you create a profile script, you can add a function called `prompt` that returns a string. PowerShell displays the output of this function as your command-line prompt.

```
function prompt
{
    "PS [$env:COMPUTERNAME] >"
}
```

This example prompt displays your computer name, and looks like `PS [LEE-DESK] >`.

You may also find it helpful to add aliases to your profile. Aliases let you refer to common commands by a name that you choose. Personal profile scripts let you automatically define aliases, functions, variables, or any other customizations that you might set interactively from the PowerShell prompt. Aliases are among the most common customizations, as they let you refer to PowerShell commands (and your own scripts) by a name that is easier to type.

> **NOTE**
>
> If you want to define an alias for a command but also need to modify the parameters to that command, then define a function instead. For more information, see Recipe 11.14.

For example:

```
Set-Alias new New-Object
Set-Alias iexplore 'C:\Program Files\Internet Explorer\iexplore.exe'
```

Your changes will become effective once you save your profile and restart PowerShell. Alternatively, you can reload your profile immediately by running this command:

```
. $profile
```

Functions are also very common customizations, with the most popular being the

`prompt` function.

Discussion

The Solution discusses three techniques to make useful customizations to your PowerShell environment: aliases, functions, and a hand-tailored prompt. You can (and will often) apply these techniques at any time during your PowerShell session, but your profile script is the standard place to put customizations that you want to apply to every session.

> **NOTE**
>
> To remove an alias or function, use the `Remove-Item` cmdlet:
>
> ```
> Remove-Item function:\MyCustomFunction
> Remove-Item alias:\new
> ```

Although the `Prompt` function returns a simple string, you can also use the function for more complex tasks. For example, many users update their console window title (by changing the `$host.UI.RawUI.WindowTitle` variable) or use the `Write-Host` cmdlet to output the prompt in color. If your prompt function handles the screen output itself, it still needs to return a string (for example, a single space) to prevent PowerShell from using its default. If you don't want this extra space to appear in your prompt, add an extra space at the end of your `Write-Host` command and return the backspace (`"``b"`) character, as shown in Example 1-4.

Example 1-4. An example PowerShell prompt

```
##############################################################################
######
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

Set-StrictMode -Version 3
```

```
function Prompt
{
    $id = 1
    $historyItem = Get-History -Count 1
    if($historyItem)
    {
        $id = $historyItem.Id + 1
    }

    Write-Host -ForegroundColor DarkGray "`n[$(Get-Location)]"
    Write-Host -NoNewLine "PS:$id > "
    $host.UI.RawUI.WindowTitle = "$(Get-Location)"

    "`b"
}
```

In addition to showing the current location, this prompt also shows the ID for that command in your history. This lets you locate and invoke past commands with relative ease:

```
[C:\]
PS:73 >5 * 5
25

[C:\]
PS:74 >1 + 1
2

[C:\]
PS:75 >Invoke-History 73
5 * 5
25

[C:\]
PS:76 >
```

Although the profile referenced by `$profile` is the one you will almost always want to use, PowerShell actually supports four separate profile scripts. For further details on these scripts (along with other shell customization options), see Appendix A.

See Also

Recipe 18.1

1.10 Customize PowerShell's User Input Behavior

Problem

You want to override the way that PowerShell reads and handles input at the prompt.

Solution

Use the `Set-PSReadLineOption` cmdlet to configure properties such as `EditMode` (Windows, VI, Emacs) and history management. For example, to make the continuation line for incomplete input a bit more red than usual:

```
Set-PSReadLineOption -Colors @{ ContinuationPrompt = "#663333" }
```

Use the `Set-PSReadLineKeyHandler` command to configure how PSReadLine responds to your actual keypresses. For example, to add forward and backward directory history navigation for `Alt+Comma` and `Alt+Period`:

```
Set-PSReadLineKeyHandler -Chord 'Alt+,' -ScriptBlock {
    Set-Location -
    [Microsoft.PowerShell.PSConsoleReadLine]::RevertLine()
    [Microsoft.PowerShell.PSConsoleReadLine]::AcceptLine()
}

Set-PSReadLineKeyHandler -Chord 'Alt+.' -ScriptBlock {
    Set-Location +
    [Microsoft.PowerShell.PSConsoleReadLine]::RevertLine()
    [Microsoft.PowerShell.PSConsoleReadLine]::AcceptLine()
}
```

Discussion

When PowerShell first came on the scene, Unix folks were among the first to notice. They'd enjoyed a powerful shell and a vigorous heritage of automation for years—and "when I'm forced to use Windows, PowerShell rocks" is a phrase

we've heard many times.

This natural uptake was no mistake. There are many on the team who come from a deep Unix background, and similarities to traditional Unix shells were intentional. For folks coming from other shells, though, we still hear the occasional grumble that some feature or other feels weird. Alt+P doesn't launch the built-in paging utility? Ctrl+XX doesn't move between the beginning of the line and current cursor position? Abhorrent!

In early versions of PowerShell, there was nothing you could reasonably do to address this. In those versions, PowerShell read its input from the console in what is known as *Cooked Mode*—where the Windows console subsystem handles all the keypresses, fancy F7 menus, and more. When you press Enter or Tab, PowerShell gets the text of what you have typed so far, but that's it. There is no way for it to know that you had pressed the (Unix-like) Ctrl+R, Ctrl+A, Ctrl+E, or any other keys.

In later versions of PowerShell, most of these questions have gone away with the introduction of the fantastic *PSReadLine* module that PowerShell uses for command-line input. `PSReadLine` adds rich syntax highlighting, tab completion, history navigation, and more.

The `PSReadLine` module lets you configure it to an incredible degree. The `Set-PSReadLineOption` cmdlet supports options for its user interface, input handling mode, history processing, and much more:

```
EditMode                        BellStyle
ContinuationPrompt              CompletionQueryItems
HistoryNoDuplicates             WordDelimiters
AddToHistoryHandler             HistorySearchCaseSensitive
CommandValidationHandler        HistorySaveStyle
HistorySearchCursorMovesToEnd   HistorySavePath
MaximumHistoryCount             AnsiEscapeTimeout
MaximumKillRingCount            PromptText
ShowToolTips                    ViModeIndicator
ExtraPromptLineCount            ViModeChangeHandler
DingTone                        PredictionSource
DingDuration                    Colors
```

In addition to letting you configure its runtime behavior, you can also configure how your keypresses cause it to react. To see all of the behaviors that you can

map to key presses, run `Get-PSReadLineKeyHandler`. PSReadLine offers pages of options - many of them not currently assigned to any keypress:

```
PS > Get-PSReadLineKeyHandler

Basic editing functions
=======================

Key              Function           Description
---              --------           -----------
Enter            AcceptLine         Accept the input or move to the
next line if input
                                    is missing a closing token.
Shift+Enter      AddLine            Move the cursor to the next line
without attempting
                                    to execute the input
Backspace        BackwardDeleteChar Delete the character before the
cursor
Ctrl+h           BackwardDeleteChar Delete the character before the
cursor
Ctrl+Home        BackwardDeleteLine Delete text from the cursor to the
start of the line
Ctrl+Backspace   BackwardKillWord   Move the text from the start of
the current or
                                    previous word to the cursor to the
kill ring
Ctrl+w           BackwardKillWord   Move the text from the start of
the current or
(...)
```

To configure any of these functions, use the Set-PSReadLineKeyHandler command. For example, to set `Ctrl+Shift+C` to capture colorized regions of the buffer into your clipboard, run:

```
Set-PSReadLineKeyHandler -Chord Ctrl+Shift+C -Function CaptureScreen
```

If there isn't a pre-defined function to do what you want, you can use the `-ScriptBlock` parameter to have PSReadLine run any code of your choosing when you press a key or key combination. The example given by the Solution demonstrates this by adding forward and backward directory history navigation.

To make any of these changes persist, simply add these commands to your PowerShell Profile.

Although really only for extremely advanced scenarios now that PSReadLine covers almost everything you would ever need, you can customize or augment this functionality even further through the `PSConsoleHostReadLine` function. When you define this method in the PowerShell console host, PowerShell calls that function instead of Windows' default Cooked Mode input functionality. The default version of this function launches `PSReadLine`'s `ReadLine` input handler. But if you wish to redefine this completely, that's it—the rest is up to you. If you'd like to implement a custom input method, the freedom (and responsibility) is all yours.

When you define this function, it must process the user input and return the resulting command. Example 1-5 implements a somewhat ridiculous Notepad-based user input mechanism:

Example 1-5. A Notepad-based user input mechanism

```
function PSConsoleHostReadLine
{
    $inputFile = Join-Path $env:TEMP PSConsoleHostReadLine
    Set-Content $inputFile "PS > "

    ## Notepad opens. Enter your command in it, save the file,
    ## and then exit.
    notepad $inputFile | Out-Null
    $userInput = Get-Content $inputFile
    $resultingCommand = $userInput.Replace("PS >", "")
    $resultingCommand
}
```

For more information about handling keypresses and other forms of user input, see Chapter 13.

See Also

Chapter 13

Recipe 1.9

1.11 Customize PowerShell's Command Resolution Behavior

Problem

Problem

You want to override or customize the command that PowerShell invokes before it is invoked.

Solution

Assign a script block to one or all of the `PreCommandLookupAction`, `PostCommandLook+upAction++, or ++CommandNotFoundAction ++properties of ++$executionContext.Session+State.InvokeCommand`. Example 1-6 enables easy parent directory navigation when you type multiple dots.

Example 1-6. Enabling easy parent path navigation through CommandNotFoundAction

```
##############################################################################
######
##
## Add-RelativePathCapture
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Adds a new CommandNotFound handler that captures relative path
navigation without having to explicitly call 'Set-Location'

.EXAMPLE

PS C:\Users\Lee\Documents>..
PS C:\Users\Lee>...
PS C:\>

#>

Set-StrictMode -Version 3

$executionContext.SessionState.InvokeCommand.CommandNotFoundAction = {
    param($CommandName, $CommandLookupEventArgs)
```

```
    ## If the command is only dots
    if($CommandName -match '^\.+$')
    {
        ## Associate a new command that should be invoked instead
        $CommandLookupEventArgs.CommandScriptBlock = {

            ## Count the number of dots, and run "Set-Location .." one
            ## less time.
            for($counter = 0; $counter -lt $CommandName.Length - 1;
$counter++)
            {
                Set-Location ..
            }

        ## We call GetNewClosure() so that the reference to $CommandName
can
        ## be used in the new command.
        }.GetNewClosure()

        ## Stop going through the command resolution process. This isn't
        ## strictly required in the CommandNotFoundAction.
        $CommandLookupEventArgs.StopSearch = $true
    }
}
```

Discussion

When you invoke a command in PowerShell, the engine goes through three distinct phases:

1. Retrieve the text of the command.
2. Find the command for that text.
3. Invoke the command that was found.

In PowerShell the `$executionContext.SessionState.InvokeCommand` property lets you override any of these stages with script blocks to intercept any or all of the `PreCommandLookupAction`, `PostCommandLookupAction`, or `CommandNotFoundAction` stages.

Each script block receives two parameters: the command name, and an object (`CommandLookupEventArgs`) to control the command lookup behavior. If your handler assigns a script block to the `CommandScriptBlock` property of

the `CommandLookup+Event+Args` or assigns a `CommandInfo` to the `Command` property of the `CommandLookup+Event+Args`, PowerShell will use that script block or command, respectively. If your script block sets the `StopSearch` property to `true`, PowerShell will do no further command resolution.

PowerShell invokes the `PreCommandLookupAction` script block when it knows the name of a command (i.e., `Get-Process`) but hasn't yet looked for the command itself. You can override this action if you want to react primarily based on the text of the command name or want to preempt PowerShell's regular command or alias resolution. For example, Example 1-7 demonstrates a `PreCommandLookupAction` that looks for commands with an asterisk before their name. When it sees one, it enables the `-Verbose` parameter.

Example 1-7. Customizing the PreCommandLookupAction

```
$executionContext.SessionState.InvokeCommand.PreCommandLookupAction = {
    param($CommandName, $CommandLookupEventArgs)

    ## If the command name starts with an asterisk, then
    ## enable its Verbose parameter
    if($CommandName -match "\*")
    {
        ## Remove the leading asterisk
        $NewCommandName = $CommandName -replace '\*',''

        ## Create a new script block that invokes the actual command,
        ## passes along all original arguments, and adds in the -Verbose
        ## parameter
        $CommandLookupEventArgs.CommandScriptBlock = {
            & $NewCommandName @args -Verbose

        ## We call GetNewClosure() so that the reference to
$NewCommandName
        ## can be used in the new command.
        }.GetNewClosure()
    }
}

PS > dir > 1.txt
PS > dir > 2.txt
PS > del 1.txt
PS > *del 2.txt
VERBOSE: Performing operation "Remove file" on Target
"C:\temp\tempfolder\2.txt".
```

After PowerShell executes the `PreCommandLookupAction` (if one exists and doesn't return a command), it goes through its regular command resolution. If it finds a command, it invokes the script block associated with the `PostCommandLookupAction`. You can override this action if you want to react primarily to a command that is just about to be invoked. Example 1-8 demonstrates a `PostCommandLookupAction` that tallies the commands you use most frequently.

Example 1-8. Customizing the PostCommandLookupAction

```
$executionContext.SessionState.InvokeCommand.PostCommandLookupAction = {
    param($CommandName, $CommandLookupEventArgs)

    ## Stores a hashtable of the commands we use most frequently
    if(-not (Test-Path variable:\CommandCount))
    {
        $global:CommandCount = @{}
    }

    ## If it was launched by us (rather than as an internal helper
    ## command), record its invocation.
    if($CommandLookupEventArgs.CommandOrigin -eq "Runspace")
    {
        $commandCount[$CommandName] = 1 + $commandCount[$CommandName]
    }
}

PS > Get-Variable commandCount
PS > Get-Process -id $pid
PS > Get-Process -id $pid
PS > $commandCount

Name                           Value
----                           -----
Out-Default                    4
Get-Variable                   1
prompt                         4
Get-Process                    2
```

If command resolution is unsuccessful, PowerShell invokes the `CommandNotFoundAction` script block if one exists. At its simplest, you can override this action if you want to recover from or override PowerShell's error behavior when it cannot find a command.

As a more advanced application, the `CommandNotFoundAction` lets you write PowerShell extensions that alter their behavior based on the *form* of the

name, rather than the arguments passed to it. For example, you might want to automatically launch URLs just by typing them or navigate around providers just by typing relative path locations.

The Solution gives an example of implementing this type of handler. While dynamic relative path navigation is not a built-in feature of PowerShell, it is possible to get a very reasonable alternative by intercepting the `CommandNotFoundAction`. If we see a missing command that has a pattern we want to handle (a series of dots), we return a script block that does the appropriate relative path navigation.

1.12 Find a Command to Accomplish a Task

Problem

You want to accomplish a task in PowerShell but don't know the command or cmdlet to accomplish that task.

Solution

Use the `Get-Command` cmdlet to search for and investigate commands.

To get the summary information about a specific command, specify the command name as an argument:

```
Get-Command CommandName
```

To get the detailed information about a specific command, pipe the output of `Get-Command` to the `Format-List` cmdlet:

```
Get-Command CommandName | Format-List
```

To search for all commands with a name that contains *text*, surround the text with asterisk characters:

```
Get-Command *text*
```

To search for all commands that use the `Get` verb, supply `Get` to the `-Verb` parameter:

```
Get-Command -Verb Get
```

To search for all commands that act on a service, use `Service` as the value of the `-Noun` parameter:

```
Get-Command -Noun Service
```

Discussion

One of the benefits that PowerShell provides administrators is the consistency of its command names. All PowerShell commands (called *cmdlets*) follow a regular *Verb-Noun* pattern—for example, `Get-Process`, `Get-EventLog`, and `Set-Location`. The verbs come from a relatively small set of standard verbs (as listed in Appendix J) and describe what action the cmdlet takes. The nouns are specific to the cmdlet and describe what the cmdlet acts on.

Knowing this philosophy, you can easily learn to work with groups of cmdlets. If you want to start a service on the local machine, the standard verb for that is `Start`. A good guess would be to first try `Start-Service` (which in this case would be correct), but typing `Get-Command -Verb Start` would also be an effective way to see what things you can start. Going the other way, you can see what actions are supported on services by typing `Get-Command -Noun Service`.

When you use the `Get-Command` cmdlet, PowerShell returns results from the list of all commands available on your system. If you'd instead like to search just commands from modules that you've loaded either explicitly or through autoloading, use the `-ListImported` parameter. For more information about PowerShell's autoloading of commands, see Recipe 1.28.

See Recipe 1.13 for a way to list all commands along with a brief description of what they do.

The `Get-Command` cmdlet is one of the three commands you will use most commonly as you explore PowerShell. The other two commands are `Get-Help` and `Get-Member`.

There is one important point to keep in mind when it comes to looking for a PowerShell command to accomplish a particular task. Many times, that PowerShell command does not exist, because the task is best accomplished the same way it always was—for example, `ipconfig.exe` to get IP configuration information, `netstat.exe` to list protocol statistics and current TCP/IP network connections, and many more.

For more information about the `Get-Command` cmdlet, type **`Get-Help Get-Command`**.

See Also

Recipe 1.13

1.13 Get Help on a Command

Problem

You want to learn how a specific command works and how to use it.

Solution

The command that provides help and usage information about a command is called `Get-Help`. It supports several different views of the help information, depending on your needs.

To get the summary of help information for a specific command, provide the command's name as an argument to the `Get-Help` cmdlet. This primarily includes its synopsis, syntax, and detailed description:

```
Get-Help CommandName
```

or:

```
CommandName -?
```

To get the detailed help information for a specific command, supply the `-`

`Detailed` flag to the `Get-Help` cmdlet. In addition to the summary view, this also includes its parameter descriptions and examples:

```
Get-Help CommandName -Detailed
```

To get the full help information for a specific command, supply the `-Full` flag to the `Get-Help` cmdlet. In addition to the detailed view, this also includes its full parameter descriptions and additional notes:

```
Get-Help CommandName -Full
```

To get only the examples for a specific command, supply the `-Examples` flag to the `Get-Help` cmdlet:

```
Get-Help CommandName -Examples
```

To retrieve the most up-to-date online version of a command's help topic, supply the `-Online` flag to the `Get-Help` cmdlet:

```
Get-Help CommandName -Online
```

To view a searchable, graphical view of a help topic, use the `-ShowWindow` parameter:

```
Get-Help CommandName -ShowWindow
```

To find all help topics that contain a given keyword, provide that keyword as an argument to the `Get-Help` cmdlet. If the keyword isn't also the name of a specific help topic, this returns all help topics that contain the keyword, including its name, category, and synopsis:

```
Get-Help Keyword
```

Discussion

The `Get-Help` cmdlet is the primary way to interact with the help system in PowerShell. Like the `Get-Command` cmdlet, the `Get-Help` cmdlet supports

wildcards. If you want to list all commands that have help content that matches a certain pattern (for example, ***process***), you can simply type **`Get-Help *process*`**.

If the pattern matches only a single command, PowerShell displays the help for that command. Although command wildcarding and keyword searching is a helpful way to search PowerShell help, see Recipe 1.15 for a script that lets you search the help content for a specified pattern.

While there are thousands of pages of custom-written help content at your disposal, PowerShell by default includes only information that it can automatically generate from the information contained in the commands themselves: names, parameters, syntax, and parameter defaults. You need to update your help content to retrieve the rest. The first time you run `Get-Help` as an administrator on a system, PowerShell offers to download this updated help content:

```
PS > Get-Help Get-Process

Do you want to run Update-Help?
The Update-Help cmdlet downloads the newest Help files for Windows
PowerShell modules and installs them on your computer. For more
details,
see the help topic at http://go.microsoft.com/fwlink/?LinkId=210614.

[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):
```

Answer Y to this prompt, and PowerShell automatically downloads and installs the most recent help content for all modules on your system. For more information on updatable help, see Recipe 1.14.

If you'd like to generate a list of all cmdlets and aliases (along with their brief synopses), run the following command:

```
Get-Help * -Category Cmdlet | Select-Object Name,Synopsis | Format-
Table -Auto
```

In addition to console-based help, PowerShell also offers online access to its help content. The Solution demonstrates how to quickly access online help content.

The `Get-Help` cmdlet is one of the three commands you will use most commonly as you explore PowerShell. The other two commands are `Get-Command` and `Get-Member`.

For more information about the `Get-Help` cmdlet, type **`Get-Help Get-Help`**.

See Also

Recipe 1.15

1.14 Update System Help Content

Problem

You want to update your system's help content to the latest available.

Solution

Run the `Update-Help` command. To retrieve help from a local path, use the `-SourcePath` cmdlet parameter:

```
Update-Help
```

or:

```
Update-Help -SourcePath \\helpserver\help
```

Discussion

One of PowerShell's greatest strengths is the incredible detail of its help content. Counting only the help content and `about_*` topics that describe core functionality, PowerShell's help includes approximately half a million words and would span 1,200 pages if printed.

The challenge that every version of PowerShell has been forced to deal with is that this help content is written at the same time as PowerShell itself. Given that its goal is to *help the user*, the content that's ready by the time a version of

PowerShell releases is a best-effort estimate of what users will need help with.

As users get their hands on PowerShell, they start to have questions. Some of these are addressed by the help topics, while some of them aren't. Sometimes the help is simply incorrect due to a product change during the release. To address this, PowerShell supports updatable help.

It's not only possible to update help, but in fact the `Update-Help` command is the *only* way to get help on your system. Out of the box, PowerShell provides an experience derived solely from what is built into the commands themselves: name, syntax, parameters, and default values.

The first time you run `Get-Help` as an administrator on a system, PowerShell offers to download updated help content:

```
PS > Get-Help Get-Process

Do you want to run Update-Help?
The Update-Help cmdlet downloads the newest Help files for Windows
PowerShell modules and installs them on your computer. For more
details,
see the help topic at http://go.microsoft.com/fwlink/?LinkId=210614.

[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):
```

Answer `Y` to this prompt, and PowerShell automatically downloads and installs the most recent help content for all modules on your system.

> **NOTE**
>
> If you are building a system image and want to prevent this prompt from ever appearing, set the registry key `HKLM:\Software\Microsoft\PowerShell\DisablePromptToUpdateHelp` to `1`.

In addition to the prompt-driven experience, you can call the `Update-Help` cmdlet directly.

Both experiences look at each module on your system, comparing the help you have for that module with the latest version online. For in-box modules,

PowerShell uses `download.microsoft.com` to retrieve updated help content. Other modules that you download from the Internet can use the `HelpInfoUri` module key to support their own updatable help.

By default, the `Update-Help` command retrieves its content from the Internet. If you want to update help on a machine not connected to the Internet, you can use the `-SourcePath` parameter of the `Update-Help` cmdlet. This path represents a directory or UNC path where PowerShell should look for updated help content. To populate this content, first use the `Save-Help` cmdlet to download the files, and then copy them to the source location.

For more information about PowerShell help, see Recipe 1.13.

See Also

Recipe 1.13

1.15 Program: Search Help for Text

Both the `Get-Command` and `Get-Help` cmdlets let you search for command names that match a given pattern. However, when you don't know exactly what portions of a command name you are looking for, you will more often have success searching through the help *content* for an answer. On Unix systems, this command is called `Apropos`.

The `Get-Help` cmdlet automatically searches the help database for keyword references when it can't find a help topic for the argument you supply. In addition to that, you might want to extend this even further to search for text *patterns* or even help topics that talk *about* existing help topics. PowerShell's help facilities support a version of wildcarded content searches, but don't support full regular expressions.

That doesn't need to stop us, though, as we can write the functionality ourselves.

To run this program, supply a search string to the `Search-Help` script (given in Example 1-9). The search string can be either simple text or a regular expression. The script then displays the name and synopsis of all help topics that match. To see the help content for that topic, use the `Get-Help` cmdlet.

Example 1-9. Search-Help.ps1

```
##############################################################################
######
##
## Search-Help
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Search the PowerShell help documentation for a given keyword or regular
expression. For simple keyword searches in PowerShell version two or
three,
simply use "Get-Help <keyword>"

.EXAMPLE

PS > Search-Help hashtable
Searches help for the term 'hashtable'

.EXAMPLE

PS > Search-Help "(datetime|ticks)"
Searches help for the term datetime or ticks, using the regular
expression
syntax.

#>

param(
    ## The pattern to search for
    [Parameter(Mandatory = $true)]
    $Pattern
)

$helpNames = $(Get-Help * | Where-Object { $_.Category -ne "Alias" })

## Go through all of the help topics
foreach($helpTopic in $helpNames)
{
    ## Get their text content, and
    $content = Get-Help -Full $helpTopic.Name | Out-String
    if($content -match "(.{0,30}$pattern.{0,30})")
```

```
    {
        $helpTopic | Add-Member NoteProperty Match $matches[0].Trim()
        $helpTopic | Select-Object Name,Match
    }
}
```

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

1.16 Launch PowerShell at a Specific Location

Problem

You want to launch a PowerShell session in a specific location.

Solution

Both Windows and PowerShell offer several ways to launch PowerShell in a specific location:

- Explorer's address bar
- PowerShell's command-line arguments
- Community extensions

Discussion

If you are browsing the filesystem with Windows Explorer, typing `PowerShell` into the address bar launches PowerShell in that location (as shown in Figure 1-2).

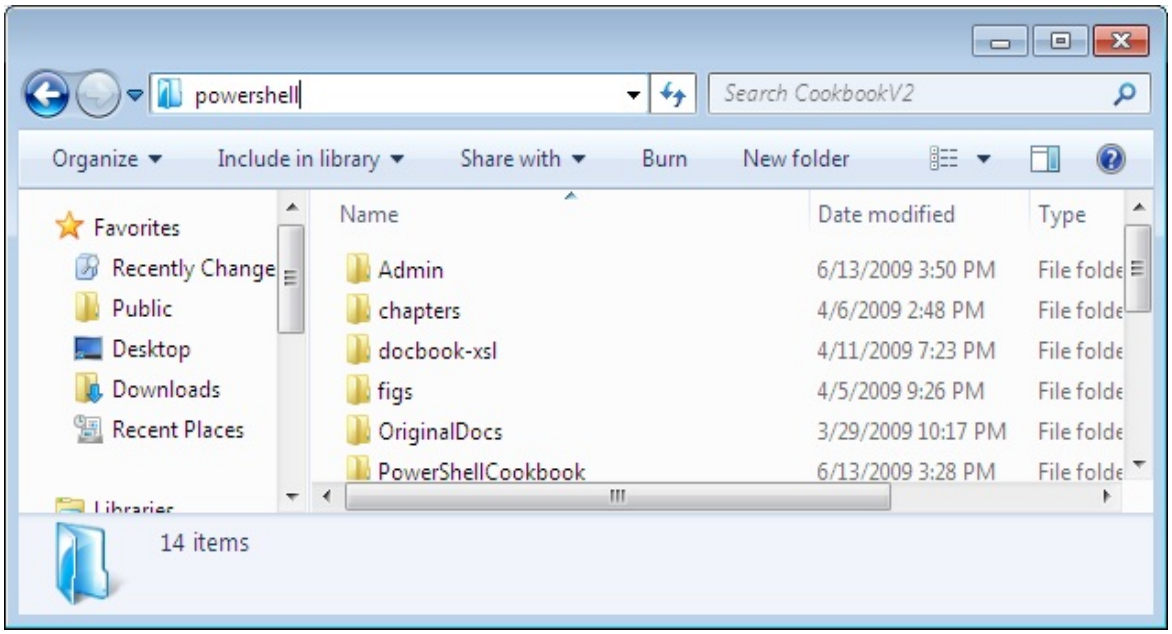

Figure 1-2. Launching PowerShell from Windows Explorer

Additionally, you can open Windows PowerShell option directly from the File menu, as shown in Figure 1-3).

For another way to launch PowerShell from Windows Explorer, Windows Terminal (if you've installed it) adds an "Open in Windows Terminal" option when you right-click on a folder from Windows Explorer.

If you aren't browsing the desired folder with Windows Explorer, you can use Start→Run (or any other means of launching an application) to launch PowerShell at a specific location. For that, use PowerShell's `-NoExit` parameter, along with the `-Command` parameter. In the `-Command` parameter, call the `Set-Location` cmdlet to initially move to your desired location.

```
pwsh -NoExit -Command Set-Location 'C:\Program Files'
```

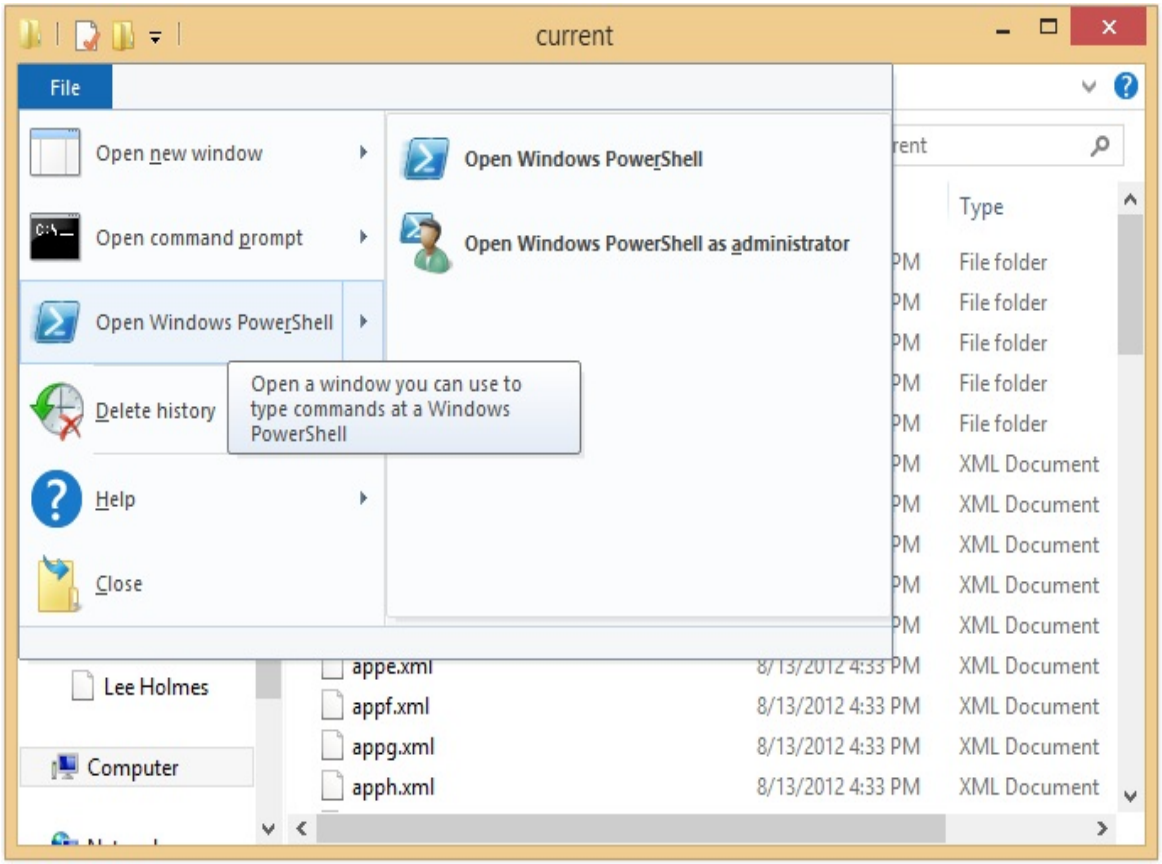

Figure 1-3. Launching PowerShell from Explorer

1.17 Invoke a PowerShell Command or Script from Outside PowerShell

Problem

You want to invoke a PowerShell command or script from a batch file, a logon script, a scheduled task, or any other non-PowerShell application.

Solution

To invoke a PowerShell command, use the `-Command` parameter:

```
PowerShell -Command Get-Process; Read-Host
```

To launch a PowerShell script, use the `-File` parameter:

```
PowerShell -File 'full path to script' arguments
```

For example:

```
PowerShell -File 'c:\shared scripts\Get-Report.ps1' Hello World
```

Discussion

By default, any arguments to *pwsh.exe* get interpreted as a script to run. If you use the `-Command` parameter, PowerShell runs the command as though you had typed it in the interactive shell, and then exits. You can customize this behavior by supplying other parameters to *pwsh.exe*, such as `-NoExit`, `-NoProfile`, and more.

> **NOTE**
>
> If you are the author of a program that needs to run PowerShell scripts or commands, PowerShell lets you call these scripts and commands much more easily than calling its command-line interface. For more information about this approach, see Recipe 17.10.

Since launching a script is so common, PowerShell provides the `-File` parameter to eliminate the complexities that arise from having to invoke a script from the `-Command` parameter. This technique lets you invoke a PowerShell script as the target of a logon script, advanced file association, scheduled task, and more.

> **NOTE**
>
> When PowerShell detects that its input or output streams have been redirected, it suppresses any prompts that it might normally display. If you want to host an interactive PowerShell prompt inside another application (such as Emacs), use `-` as the argument for the `-File` parameter. In PowerShell (as with traditional Unix shells), this implies "taken from standard input."
>
> ```
> powershell -File -
> ```

If the script is for background automation or a scheduled task, these scripts can sometimes interfere with (or become influenced by) the user's environment. For these situations, three parameters come in handy:

`-NoProfile`

Runs the command or script without loading user profile scripts. This makes the script launch faster, but it primarily prevents user preferences (e.g., aliases and preference variables) from interfering with the script's working environment.

`-WindowStyle`

Runs the command or script with the specified window style—most commonly `Hidden`. When run with a window style of `Hidden`, PowerShell hides its main window immediately. For more ways to control the window style from *within* PowerShell, see Recipe 24.3.

`-ExecutionPolicy`

Runs the command or script with a specified execution policy applied only to this instance of PowerShell. This lets you write PowerShell scripts to manage a system without having to change the system-wide execution policy. For more information about scoped execution policies, see Recipe 18.1.

If the arguments to the `-Command` parameter become complex, special character handling in the application calling PowerShell (such as *cmd.exe*) might interfere with the command you want to send to PowerShell. For this situation, PowerShell supports an `EncodedCommand` parameter: a Base64-encoded representation of the Unicode string you want to run. Example 1-10 demonstrates how to convert a string containing PowerShell commands to a Base64-encoded form.

Example 1-10. Converting PowerShell commands into a Base64-encoded form

```
$commands = '1..10 | % { "PowerShell Rocks" }'
$bytes = [System.Text.Encoding]::Unicode.GetBytes($commands)
$encodedString = [Convert]::ToBase64String($bytes)
```

Once you have the encoded string, you can use it as the value of the

`EncodedCommand` parameter, as shown in Example 1-11.

Example 1-11. Launching PowerShell with an encoded command from cmd.exe

```
Microsoft Windows [Version 10.0.19041.685]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Lee>PowerShell -EncodedCommand
MQAuAC4AMQAwACAAfAAgACUAIAB7ACAAIgBQAG8A
   dwBlAHIAUwBoAGUAbABsACAAUgBvAGMAawBzACIAIAB9AA==
PowerShell Rocks
PowerShell Rocks
PowerShell Rocks
PowerShell Rocks
PowerShell Rocks
PowerShell Rocks
PowerShell Rocks
PowerShell Rocks
PowerShell Rocks
PowerShell Rocks
```

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

Recipe 17.10

1.18 Understand and Customize PowerShell's Tab Completion

Problem

You want to customize how PowerShell reacts to presses of the Tab key (and additionally, Ctrl+Space in the case of IntelliSense in the Integrated Scripting Environment).

Solution

Create a custom function called `TabExpansion2`. PowerShell invokes this function when you press Tab, or when it invokes IntelliSense in the Integrated

Scripting Environment.

Discussion

When you press Tab, PowerShell invokes a facility known as *tab expansion*: replacing what you've typed so far with an expanded version of that (if any apply.) For example, if you type `Set-Location C:\` and then press Tab, PowerShell starts cycling through directories under `C:\` for you to navigate into.

The features offered by PowerShell's built-in tab expansion are quite rich, as shown in Table 1-2.

Table 1-2. Tab expansion features in PowerShell

Description	Example
Command completion. Completes command names when current text appears to represent a command invocation.	`Get-Ch` <Tab>
Parameter completion. Completes command parameters for the current command.	`Get-ChildItem -Pat` <Tab>
Argument completion. Completes command arguments for the current command parameter. This applies to any command argument that takes a fixed set of values (enumerations or parameters that define a `ValidateSet` attribute). In addition, PowerShell contains extended argument completion for module names, help topics, CIM / WMI classes, event log names, job IDs and names, process IDs and names, provider names, drive names, service names and display names, and trace source names.	`Set-ExecutionPolicy -ExecutionPolicy` <Tab>
History text completion. Replaces the current input with items from the command history that match the text after the # character.	`#Process` <Tab>
History ID completion. Replaces the current input with the command line from item number *`ID`* in your command history.	`#12` <Tab>
Filename completion. Replaces the current parameter value with file names that match what you've typed so far. When applied to the `Set-Location` cmdlet, PowerShell further filters results to only directories.	`+Set-Location + C:\Win`

	`dows\S` <Tab>
Operator completion. Replaces the current text with a matching operator. This includes flags supplied to the `switch` statement.	`"Hello World" -rep` <Tab> `switch - c` <Tab>
Variable completion. Replaces the current text with available PowerShell variables. In the Integrated Scripting Environment, PowerShell incorporates variables even from script content that has never been invoked.	$myGreeting = “Hello World”; $myGr <Tab>
Member completion. Replaces member names for the currently referenced variable or type. When PowerShell can infer the members from previous commands in the pipeline, it even supports member completion within script blocks.	[Console]::Ba<TAB> Get-Process | Where-Object { $_.Ha <Tab>
Type completion. Replaces abbreviated type names with their namespace-qualified name.	[PSSer<TAB> $l = New-Object List[Stri <Tab>

If you want to extend PowerShell’s tab expansion capabilities, define a function called `TabExpansion2`. You can add this to your PowerShell profile directly, or dot-source it from your profile. Example 1-12 demonstrates an example custom tab expansion function that extends the functionality already built into PowerShell.

Example 1-12. A sample implementation of TabExpansion2

```
##############################################################################
######
##
```

```powershell
## TabExpansion2
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

function TabExpansion2
{
    [CmdletBinding(DefaultParameterSetName = 'ScriptInputSet')]
    Param(
        [Parameter(ParameterSetName = 'ScriptInputSet', Mandatory =
$true, Position = 0)]
        [string] $inputScript,

        [Parameter(ParameterSetName = 'ScriptInputSet', Mandatory =
$true, Position = 1)]
        [int] $cursorColumn,

        [Parameter(ParameterSetName = 'AstInputSet', Mandatory = $true,
Position = 0)]
        [System.Management.Automation.Language.Ast] $ast,

        [Parameter(ParameterSetName = 'AstInputSet', Mandatory = $true,
Position = 1)]
        [System.Management.Automation.Language.Token[]] $tokens,

        [Parameter(ParameterSetName = 'AstInputSet', Mandatory = $true,
Position = 2)]
        [System.Management.Automation.Language.IScriptPosition]
$positionOfCursor,

        [Parameter(ParameterSetName = 'ScriptInputSet', Position = 2)]
        [Parameter(ParameterSetName = 'AstInputSet', Position = 3)]
        [Hashtable] $options = $null
    )

    End
    {
        ## Create a new 'Options' hashtable if one has not been
supplied.
        ## In this hashtable, you can add keys for the following
options, using
        ## $true or $false for their values:
        ##
        ## IgnoreHiddenShares - Ignore hidden UNC shares (such as
\\COMPUTER\ADMIN$)
        ## RelativePaths - When expanding filenames and paths, $true
```

```
forces PowerShell
        ##     to replace paths with relative paths. When $false, forces
PowerShell to
        ##     replace them with absolute paths. By default, PowerShell
makes this
        ##     decision based on what you had typed so far before
invoking tab completion.
        ## LiteralPaths - Prevents PowerShell from replacing special
file characters
        ##     (such as square brackets and back-ticks) with their
escaped equivalent.
        if(-not $options) { $options = @{} }

        ## Demonstrate some custom tab expansion completers for
parameters.
        ## This is a hash table of parameter names (and optionally
cmdlet names)
        ## that we add to the $options hashtable.
        ##
        ## When PowerShell evaluates the script block, $args gets the
        ## following: command name, parameter, word being completed,
        ## AST of the command being completed, and currently-bound
arguments.
        $options["CustomArgumentCompleters"] = @{
            "Get-ChildItem:Filter" = { "*.ps1","*.txt","*.doc" }
            "ComputerName" = {
"ComputerName1","ComputerName2","ComputerName3" }
        }

        ## Also define a completer for a native executable.
        ## When PowerShell evaluates the script block, $args gets the
        ## word being completed, and AST of the command being completed.
        $options["NativeArgumentCompleters"] = @{
            "attrib" = { "+R","+H","+S" }
        }

        ## Define a "quick completions" list that we'll cycle through
        ## when the user types '!!' followed by TAB.
        $quickCompletions = @(
            'Get-Process -Name PowerShell | ? Id -ne $pid | Stop-
Process',
            'Set-Location $pshome',
            ('$errors = $error | % { $_.InvocationInfo.Line }; Get-
History | ' +
                ' ? { $_.CommandLine -notin $errors }')
        )

        ## First, check the built-in tab completion results
        $result = $null
```

```
        if ($psCmdlet.ParameterSetName -eq 'ScriptInputSet')
        {
            $result =
[System.Management.Automation.CommandCompletion]::CompleteInput(
                <#inputScript#>  $inputScript,
                <#cursorColumn#> $cursorColumn,
                <#options#>      $options)
        }
        else
        {
            $result =
[System.Management.Automation.CommandCompletion]::CompleteInput(
                <#ast#>              $ast,
                <#tokens#>           $tokens,
                <#positionOfCursor#> $positionOfCursor,
                <#options#>          $options)
        }

        ## If we didn't get a result
        if($result.CompletionMatches.Count -eq 0)
        {
            ## If this was done at the command-line or in a remote
session,
            ## create an AST out of the input
            if ($psCmdlet.ParameterSetName -eq 'ScriptInputSet')
            {
                $ast =
[System.Management.Automation.Language.Parser]::ParseInput(
                    $inputScript, [ref]$tokens, [ref]$null)
            }

            ## In this simple example, look at the text being supplied.
            ## We could do advanced analysis of the AST here if we
wanted,
            ## but in this case just use its text. We use a regular
expression
            ## to check if the text started with two exclamations, and
then
            ## use a match group to retain the rest.
            $text = $ast.Extent.Text
            if($text -match '^!!(.*)')
            {
                ## Extract the rest of the text from the regular
expression
                ## match group.
                $currentCompletionText = $matches[1].Trim()

                ## Go through each of our quick completions and add them
to
```

```
                ## our completion results. The arguments to the
completion results
                ## are the text to be used in tab completion, a
potentially shorter
                ## version to use for display (i.e.: intellisense in the
ISE),
                ## the type of match, and a potentially more verbose
description to
                ## be used as a tool tip.
                $quickCompletions | Where-Object { $_ -match
$currentCompletionText } |
                    Foreach-Object { $result.CompletionMatches.Add(
                        (New-Object
Management.Automation.CompletionResult $_,$_,"Text",$_) )
                }
            }
        }

        return $result
    }
}
```

See Also

Recipe 10.10

Appendix A

1.19 Program: Learn Aliases for Common Commands

In interactive use, full cmdlet names (such as `Get-ChildItem`) are cumbersome and slow to type. Although aliases are much more efficient, it takes a while to discover them. To learn aliases more easily, you can modify your prompt to remind you of the shorter version of any aliased commands that you use.

This involves two steps:

1. Add the program`Get-AliasSuggestion.ps1`, shown in Example 1-13, to your *tools* directory or another directory.

Example 1-13. Get-AliasSuggestion.ps1

```
##############################################################################
```

```powershell
######
##
## Get-AliasSuggestion
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
#############################################################################
######

<#

.SYNOPSIS

Get an alias suggestion from the full text of the last command. Intended
to
be added to your prompt function to help learn aliases for commands.

.EXAMPLE

PS > Get-AliasSuggestion Remove-ItemProperty
Suggestion: An alias for Remove-ItemProperty is rp

#>

param(
    ## The full text of the last command
    $LastCommand
)

Set-StrictMode -Version 3

$helpMatches = @()

## Find all of the commands in their last input
$tokens = [Management.Automation.PSParser]::Tokenize(
    $lastCommand, [ref] $null)
$commands = $tokens | Where-Object { $_.Type -eq "Command" }

## Go through each command
foreach($command in $commands)
{
    ## Get the alias suggestions
    foreach($alias in Get-Alias -Definition $command.Content)
    {
        $helpMatches += "Suggestion: An alias for " +
            "$($alias.Definition) is $($alias.Name)"
    }
}
```

```
$helpMatches
```

1. Add the text from Example 1-14 to the `Prompt` function in your profile. If you do not yet have a `Prompt` function, see Recipe 1.9 to learn how to add one.

Example 1-14. A useful prompt to teach you aliases for common commands

```
function Prompt
{
    ## Get the last item from the history
    $historyItem = Get-History -Count 1

    ## If there were any history items
    if($historyItem)
    {
        ## Get the training suggestion for that item
        $suggestions = @(Get-AliasSuggestion $historyItem.CommandLine)
        ## If there were any suggestions
        if($suggestions)
        {
            ## For each suggestion, write it to the screen
            foreach($aliasSuggestion in $suggestions)
            {
                Write-Host "$aliasSuggestion"
            }
            Write-Host ""

        }
    }

    ## Rest of prompt goes here
    "PS [$env:COMPUTERNAME] >"
}
```

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

Recipe 1.9

1.20 Program: Learn Aliases for Common Parameters

Problem

You want to learn aliases defined for command parameters.

Solution

Use the `Get-ParameterAlias` script, as shown in Example 1-15, to return all aliases for parameters used by the previous command in your session history.

Example 1-15. Get-ParameterAlias.ps1

```
##############################################################################
######
##
## Get-ParameterAlias
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Looks in the session history, and returns any aliases that apply to
parameters of commands that were used.

.EXAMPLE

PS > dir -ErrorAction SilentlyContinue
PS > Get-ParameterAlias
An alias for the 'ErrorAction' parameter of 'dir' is ea

#>

Set-StrictMode -Version 3

## Get the last item from their session history
$history = Get-History -Count 1
if(-not $history)
{
    return
}

## And extract the actual command line they typed
$lastCommand = $history.CommandLine
```

```
## Use the Tokenizer API to determine which portions represent
## commands and parameters to those commands
$tokens = [System.Management.Automation.PsParser]::Tokenize(
    $lastCommand, [ref] $null)
$currentCommand = $null

## Now go through each resulting token
foreach($token in $tokens)
{
    ## If we've found a new command, store that.
    if($token.Type -eq "Command")
    {
        $currentCommand = $token.Content
    }

    ## If we've found a command parameter, start looking for aliases
    if(($token.Type -eq "CommandParameter") -and ($currentCommand))
    {
        ## Remove the leading "-" from the parameter
        $currentParameter = $token.Content.TrimStart("-")

        ## Determine all of the parameters for the current command.
        (Get-Command $currentCommand).Parameters.GetEnumerator() |

            ## For parameters that start with the current parameter
name,
            Where-Object { $_.Key -like "$currentParameter*" } |

            ## return all of the aliases that apply. We use "starts
with"
            ## because the user might have typed a shortened form of
            ## the parameter name.
            Foreach-Object {
                $_.Value.Aliases | Foreach-Object {
                    "Suggestion: An alias for the '$currentParameter' "
+
                    "parameter of '$currentCommand' is '$_'"
                }
            }
    }
}
```

Discussion

To make it easy to type command parameters, PowerShell lets you type only as much of the command parameter as is required to disambiguate it from other parameters of that command. In addition to shortening implicitly supported by

the shell, cmdlet authors can also define explicit aliases for their parameters—for example, `CN` as a short form for `ComputerName`.

While helpful, these aliases are difficult to discover.

If you want to see the aliases for a specific command, you can access its `Parameters` collection:

```
PS > (Get-Command New-TimeSpan).Parameters.Values | Select
Name,Aliases

Name                                               Aliases
----                                               -------
Start                                              {LastWriteTime}
End                                                {}
Days                                               {}
Hours                                              {}
Minutes                                            {}
Seconds                                            {}
Verbose                                            {vb}
Debug                                              {db}
ErrorAction                                        {ea}
WarningAction                                      {wa}
ErrorVariable                                      {ev}
WarningVariable                                    {wv}
OutVariable                                        {ov}
OutBuffer                                          {ob}
```

If you want to learn any aliases for parameters in your previous command, simply run `Get-ParameterAlias.ps1`. To make PowerShell do this automatically, add a call to `Get-ParameterAlias.ps1` in your prompt.

This script builds on two main features: PowerShell's *Tokenizer API*, and the rich information returned by the `Get-Command` cmdlet. PowerShell's Tokenizer API examines its input and returns PowerShell's interpretation of the input: commands, parameters, parameter values, operators, and more. Like the rich output produced by most of PowerShell's commands, `Get-Command` returns information about a command's parameters, parameter sets, output type (if specified), and more.

For more information about the Tokenizer API, see Recipe 10.10.

See Also

Recipe 1.2

Recipe 10.10

“A Guided Tour of PowerShell”

1.21 Access and Manage Your Console History

Problem

After working in the shell for a while, you want to invoke commands from your history, view your command history, and save your command history.

Solution

The shortcuts given in Recipe 1.9 let you manage your history, but PowerShell offers several features to help you work with your console in even more detail.

To get the most recent commands from your session, use the `Get-History` cmdlet (or its alias of `h`):

```
Get-History
```

To rerun a specific command from your session history, provide its ID to the `Invoke-History` cmdlet (or its alias of `ihy`):

```
Invoke-History ID
```

To increase (or limit) the number of commands stored in your session history, assign a new value to the `$MaximumHistoryCount` variable:

```
$MaximumHistoryCount = Count
```

To save your command history to a file, pipe the output of `Get-History` to the `Export-CliXml` cmdlet:

```
Get-History | Export-CliXml Filename
```

To add a previously saved command history to your current session history, call the `Import-CliXml` cmdlet and then pipe that output to the `Add-History` cmdlet:

```
Import-CliXml Filename | Add-History
```

To clear all commands from your session history, use the `Clear-History` cmdlet:

```
Clear-History
```

Discussion

Unlike the console history hotkeys discussed in Recipe 1.9, the `Get-History` cmdlet produces rich objects that represent information about items in your history. Each object contains that item's ID, command line, start of execution time, and end of execution time.

Once you know the ID of a history item (as shown in the output of `Get-History`), you can pass it to `Invoke-History` to execute that command again. The example prompt function shown in Recipe 1.9 makes working with prior history items easy, as the prompt for each command includes the history ID that will represent it.

> **NOTE**
>
> You can easily see how long a series of commands took to invoke by looking at the `StartExecutionTime` and `EndExecutionTime` properties. This is a great way to get a handle on exactly how little time it took to come up with the commands that just saved you hours of manual work:

```
PS C:\> Get-History 65,66 | Format-Table *

Id CommandLine             StartExecutionTime
EndExecutionTime
-- -----------             ------------------    ------------
----
65 dir                     10/13/2012 2:06:05 PM 10/13/2012
2:06:05 PM
66 Start-Sleep -Seconds 45 10/13/2012 2:06:15 PM 10/13/2012
2:07:00 PM
```

IDs provided by the `Get-History` cmdlet differ from the IDs given by the Windows console common history hotkeys (such as F7), because their history management techniques differ.

By default, PowerShell stores the last 4,096 entries of your command history. If you want to raise or lower this amount, set the `$MaximumHistoryCount` variable to the size you desire. To make this change permanent, set the variable in your PowerShell profile script.

By far, the most useful feature of PowerShell's command history is for reviewing ad hoc experimentation and capturing it in a script that you can then use over and over. For an overview of that process (and a script that helps to automate it), see Recipe 1.22.

See Also

Recipe 1.9

Recipe 1.22

Recipe 1.23

1.22 Program: Create Scripts from Your Session History

After interactively experimenting at the command line for a while to solve a multistep task, you'll often want to keep or share the exact steps you used to eventually solve the problem. The script smiles at you from your history buffer, but it is unfortunately surrounded by many more commands that you *don't* want to keep.

> **NOTE**
>
> For an example of using the `Out-GridView` cmdlet to do this graphically, see Recipe 2.4.

To solve this problem, use the `Get-History` cmdlet to view the recent commands that you've typed. Then, call `Copy-History` with the IDs of the commands you want to keep, as shown in Example 1-16.

Example 1-16. Copy-History.ps1

```
##############################################################################
######
##
## Copy-History
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Copy selected commands from the history buffer into the clipboard as a
script.

.EXAMPLE

PS > Copy-History
Copies the entire contents of the history buffer into the clipboard.

.EXAMPLE

PS > Copy-History -5
Copies the last five commands into the clipboard.

.EXAMPLE

PS > Copy-History 2,5,8,4
Copies commands 2,5,8, and 4.

.EXAMPLE

PS > Copy-History (1..10+5+6)
Copies commands 1 through 10, then 5, then 6, using PowerShell's array
slicing syntax.

#>

[CmdletBinding()]
param(
```

```
    ## The range of history IDs to copy
    [Alias("Id")]
    [int[]] $Range
)

Set-StrictMode -Version 3

$history = @()

## If they haven't specified a range, assume it's everything
if((-not $range) -or ($range.Count -eq 0))
{
    $history = @(Get-History -Count ([Int16]::MaxValue))
}
## If it's a negative number, copy only that many
elseif(($range.Count -eq 1) -and ($range[0] -lt 0))
{
    $count = [Math]::Abs($range[0])
    $history = (Get-History -Count $count)
}
## Otherwise, go through each history ID in the given range
## and add it to our history list.
else
{
    foreach($commandId in $range)
    {
        if($commandId -eq -1) { $history += Get-History -Count 1 }
        else { $history += Get-History -Id $commandId }
    }
}

## Finally, export the history to the clipboard.
$history | Foreach-Object { $_.CommandLine } | clip.exe
```

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

Recipe 2.4

1.23 Invoke a Command from Your Session History

Problem

You want to run a command from the history of your current session.

Solution

Use the `Invoke-History` cmdlet (or its `ihy` alias) to invoke a specific command by its *`ID`*:

```
Invoke-History ID
```

To search through your history for a command containing *`text`*:

```
PS > #text<Tab>
```

To repopulate your command with the text of a previous command by its *`ID`*:

```
PS > #ID<Tab>
```

Discussion

Once you've had your shell open for a while, your history buffer quickly fills with useful commands. The history management hotkeys described in Recipe 1.9 show one way to navigate your history, but this type of history navigation works only for command lines you've typed in that specific session. If you keep a persistent command history (as shown in Recipe 1.31), these shortcuts do not apply.

The `Invoke-History` cmdlet illustrates the simplest example of working with your command history. Given a specific history ID (perhaps shown in your prompt function), calling `Invoke-History` with that ID will run that command again. For more information about this technique, see Recipe 1.9.

As part of its tab-completion support, PowerShell gives you easy access to previous commands as well. If you prefix your command with the # character, tab completion takes one of two approaches:

ID completion

If you type a number, tab completion finds the entry in your command

history with that ID, and then replaces your command line with the text of that history entry. This is especially useful when you want to slightly modify a previous history entry, since `Invoke-History` by itself doesn't support that.

Pattern completion

If you type anything else, tab completion searches for entries in your command history that contain that text. Under the hood, PowerShell uses the `-like` operator to match your command entries, so you can use all of the wildcard characters supported by that operator. For more information on searching text for patterns, see Recipe 5.7.

PowerShell's tab completion is largely driven by the fully customizable `TabExpansion2` function. You can easily change this function to include more advanced functionality, or even just customize specific behaviors to suit your personal preferences. For more information, see Recipe 1.18.

See Also

Recipe 1.9

Recipe 5.7

Recipe 1.18

Recipe 1.31

1.24 Program: Search Formatted Output for a Pattern

While PowerShell's built-in filtering facilities are incredibly flexible (for example, the `Where-Object` cmdlet), they generally operate against specific properties of the incoming object. If you are searching for text in the object's formatted output, or don't know which property contains the text you are looking for, simple text-based filtering is sometimes helpful.

To solve this problem, you can pipe the output into the `Out-String` cmdlet before passing it to the `Select-String` cmdlet:

```
Get-Service | Out-String -Stream | Select-String audio
```

Or, using built-in aliases:

```
Get-Service | oss | sls audio
```

In script form, `Select-TextOutput` (shown in Example 1-17) does exactly this, and it lets you search for a pattern in the visual representation of command output.

Example 1-17. Select-TextOutput.ps1

```
##############################################################################
######
##
## Select-TextOutput
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Searches the textual output of a command for a pattern.

.EXAMPLE

PS > Get-Service | Select-TextOutput audio
Finds all references to "Audio" in the output of Get-Service

#>

param(
    ## The pattern to search for
    $Pattern
)

Set-StrictMode -Version 3
$input | Out-String -Stream | Select-String $pattern
```

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

1.25 Interactively View and Process Command Output

Problem

You want to graphically explore and analyze the output of a command.

Solution

Use the `Out-GridView` cmdlet to interactively explore the output of a command.

Discussion

The `Out-GridView` cmdlet is one of the rare PowerShell cmdlets that displays a graphical user interface. While the `Where-Object` and `Sort-Object` cmdlets are the most common way to sort and filter lists of items, the `Out-GridView` cmdlet is very effective at the style of repeated refinement that sometimes helps you develop complex queries. Figure 1-4 shows the `Out-GridView` cmdlet in action.

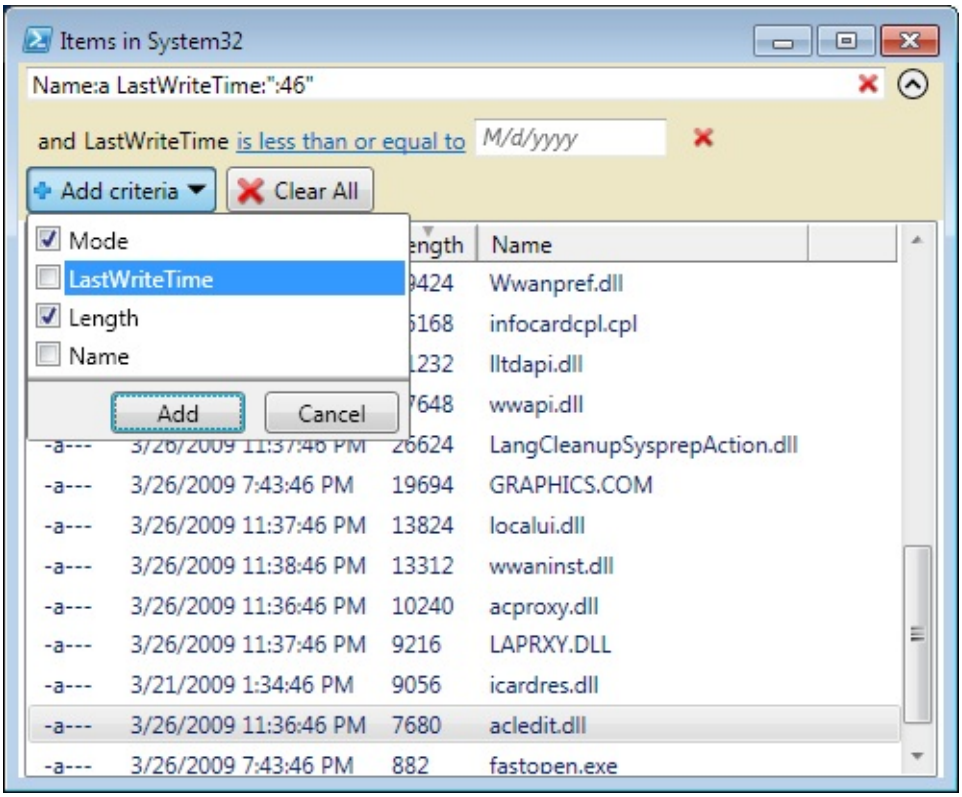

Figure 1-4. Out-GridView, ready to filter

`Out-GridView` lets you primarily filter your command output in two ways: a *quick filter* expression and a *criteria filter*.

Quick filters are fairly simple. As you type text in the topmost "Filter" window, `Out-GridView` filters the list to contain only items that match that text. If you want to restrict this text filtering to specific columns, simply provide a column name before your search string and separate the two with a colon. You can provide multiple search strings, in which case `Out-GridView` returns only rows that match all of the required strings.

> **NOTE**
>
> Unlike most filtering cmdlets in PowerShell, the quick filters in the `Out-GridView` cmdlet do not support wildcards or regular expressions. For this type of advanced query, criteria-based filtering can help.

Criteria filters give fine-grained control over the filtering used by the `Out-GridView` cmdlet. To apply a criteria filter, click the "Add criteria" button and select a property to filter on. `Out-GridView` adds a row below the quick filter field and lets you pick one of several operations to apply to this property:

- Less than or equal to
- Greater than or equal to
- Between
- Equals
- Does not equal
- Contains
- Does not contain

In addition to these filtering options, `Out-GridView` also lets you click and rearrange the header columns to sort by them.

Processing output

Once you've sliced and diced your command output, you can select any rows you want to keep and press Ctrl+C to copy them to the clipboard. `Out-GridView` copies the items to the clipboard as tab-separated data, so you can easily paste the information into a spreadsheet or other file for further processing.

In addition to supporting clipboard output, the `Out-GridView` cmdlet supports full-fidelity object filtering if you use its `-PassThru` parameter. For an example of this full-fidelity filtering, see Recipe 2.4.

See Also

Recipe 2.4

1.26 Program: Interactively View and Explore Objects

When working with unfamiliar objects in PowerShell, much of your time is

spent with the `Get-Member` and `Format-List` commands—navigating through properties, reviewing members, and more.

For ad hoc investigation, a graphical interface is often useful.

To solve this problem, Example 1-18 provides an interactive tree view that you can use to explore and navigate objects. For example, to examine the structure of a script as PowerShell sees it (its *abstract syntax tree*):

```
$ps = { Get-Process -ID $pid }.Ast
Show-Object $ps
```

For more information about parsing and analyzing the structure of PowerShell scripts, see Recipe 10.10.

Example 1-18. Show-Object.ps1

```
##############################################################################
#####
##
## Show-Object
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Provides a graphical interface to let you explore and navigate an
object.


.EXAMPLE

PS > $ps = { Get-Process -ID $pid }.Ast
PS > Show-Object $ps

#>

param(
    ## The object to examine
    [Parameter(ValueFromPipeline = $true)]
    $InputObject
)
```

```
Set-StrictMode -Version 3

Add-Type -Assembly System.Windows.Forms

## Figure out the variable name to use when displaying the
## object navigation syntax. To do this, we look through all
## of the variables for the one with the same object identifier.
$rootVariableName = dir variable:\* -Exclude InputObject,Args |
    Where-Object {
        $_.Value -and
        ($_.Value.GetType() -eq $InputObject.GetType()) -and
        ($_.Value.GetHashCode() -eq $InputObject.GetHashCode())
}

## If we got multiple, pick the first
$rootVariableName = $rootVariableName| % Name | Select -First 1

## If we didn't find one, use a default name
if(-not $rootVariableName)
{
    $rootVariableName = "InputObject"
}

## A function to add an object to the display tree
function PopulateNode($node, $object)
{
    ## If we've been asked to add a NULL object, just return
    if(-not $object) { return }

    ## If the object is a collection, then we need to add multiple
    ## children to the node

if([System.Management.Automation.LanguagePrimitives]::GetEnumerator($obj
ect))
    {
        ## Some very rare collections don't support indexing (i.e.:
$foo[0]).
        ## In this situation, PowerShell returns the parent object back
when you
        ## try to access the [0] property.
        $isOnlyEnumerable = $object.GetHashCode() -eq
$object[0].GetHashCode()

        ## Go through all the items
        $count = 0
        foreach($childObjectValue in $object)
        {
            ## Create the new node to add, with the node text of the
item and
```

```
            ## value, along with its type
            $newChildNode = New-Object Windows.Forms.TreeNode
            $newChildNode.Text = "$($node.Name)[$count] =
$childObjectValue"
            $newChildNode.ToolTipText = $childObjectValue.GetType()

            ## Use the node name to keep track of the actual property
name
            ## and syntax to access that property.
            ## If we can't use the index operator to access children,
add
            ## a special tag that we'll handle specially when displaying
            ## the node names.
            if($isOnlyEnumerable)
            {
                $newChildNode.Name = "@"
            }

            $newChildNode.Name += "[$count]"
            $null = $node.Nodes.Add($newChildNode)

            ## If this node has children or properties, add a
placeholder
            ## node underneath so that the node shows a '+' sign to be
            ## expanded.
            AddPlaceholderIfRequired $newChildNode $childObjectValue

            $count++
        }
    }
    else
    {
        ## If the item was not a collection, then go through its
        ## properties
        foreach($child in $object.PSObject.Properties)
        {
            ## Figure out the value of the property, along with
            ## its type.
            $childObject = $child.Value
            $childObjectType = $null
            if($childObject)
            {
                $childObjectType = $childObject.GetType()
            }

            ## Create the new node to add, with the node text of the
item and
            ## value, along with its type
            $childNode = New-Object Windows.Forms.TreeNode
            $childNode.Text = $child.Name + " = $childObject"
```

```
            $childNode.ToolTipText = $childObjectType

if([System.Management.Automation.LanguagePrimitives]::GetEnumerator($chi
ldObject))
            {
                $childNode.ToolTipText += "[]"
            }

            $childNode.Name = $child.Name
            $null = $node.Nodes.Add($childNode)

            ## If this node has children or properties, add a
placeholder
            ## node underneath so that the node shows a '+' sign to be
            ## expanded.
            AddPlaceholderIfRequired $childNode $childObject
        }
    }
}

## A function to add a placeholder if required to a node.
## If there are any properties or children for this object, make a
temporary
## node with the text "..." so that the node shows a '+' sign to be
## expanded.
function AddPlaceholderIfRequired($node, $object)
{
    if(-not $object) { return }


if([System.Management.Automation.LanguagePrimitives]::GetEnumerator($obj
ect) -or
        @($object.PSObject.Properties))
    {
        $null = $node.Nodes.Add( (New-Object Windows.Forms.TreeNode
"...") )
    }
}

## A function invoked when a node is selected.
function OnAfterSelect
{
    param($Sender, $TreeViewEventArgs)

    ## Determine the selected node
    $nodeSelected = $Sender.SelectedNode

    ## Walk through its parents, creating the virtual
    ## PowerShell syntax to access this property.
    $nodePath = GetPathForNode $nodeSelected
```

```
    ## Now, invoke that PowerShell syntax to retrieve
    ## the value of the property.
    $resultObject = Invoke-Expression $nodePath
    $outputPane.Text = $nodePath

    ## If we got some output, put the object's member
    ## information in the text box.
    if($resultObject)
    {
        $members = Get-Member -InputObject $resultObject | Out-String
        $outputPane.Text += "`n" + $members
    }
}

## A function invoked when the user is about to expand a node
function OnBeforeExpand
{
    param($Sender, $TreeViewCancelEventArgs)

    ## Determine the selected node
    $selectedNode = $TreeViewCancelEventArgs.Node

    ## If it has a child node that is the placeholder, clear
    ## the placeholder node.
    if($selectedNode.FirstNode -and
        ($selectedNode.FirstNode.Text -eq "..."))
    {
        $selectedNode.Nodes.Clear()
    }
    else
    {
        return
    }

    ## Walk through its parents, creating the virtual
    ## PowerShell syntax to access this property.
    $nodePath = GetPathForNode $selectedNode

    ## Now, invoke that PowerShell syntax to retrieve
    ## the value of the property.
    Invoke-Expression "`$resultObject = $nodePath"

    ## And populate the node with the result object.
    PopulateNode $selectedNode $resultObject
}

## A function to handle key presses on the tree view.
## In this case, we capture ^C to copy the path of
## the object property that we're currently viewing.
```

```
function OnTreeViewKeyPress
{
    param($Sender, $KeyPressEventArgs)

    ## [Char] 3 = Control-C
    if($KeyPressEventArgs.KeyChar -eq 3)
    {
        $KeyPressEventArgs.Handled = $true

        ## Get the object path, and set it on the clipboard
        $node = $Sender.SelectedNode
        $nodePath = GetPathForNode $node
        [System.Windows.Forms.Clipboard]::SetText($nodePath)

        $form.Close()
    }
    elseif([System.Windows.Forms.Control]::ModifierKeys -eq "Control")
    {
        if($KeyPressEventArgs.KeyChar -eq '+')
        {
            $SCRIPT:currentFontSize++
            UpdateFonts $SCRIPT:currentFontSize

            $KeyPressEventArgs.Handled = $true
        }
        elseif($KeyPressEventArgs.KeyChar -eq '-')
        {
            $SCRIPT:currentFontSize--
            if($SCRIPT:currentFontSize -lt 1) { $SCRIPT:currentFontSize
= 1 }
            UpdateFonts $SCRIPT:currentFontSize

            $KeyPressEventArgs.Handled = $true
        }
    }
}

## A function to handle key presses on the form.
## In this case, we handle Ctrl-Plus and Ctrl-Minus
## to adjust font size.
function OnKeyUp
{
    param($Sender, $KeyUpEventArgs)

    if([System.Windows.Forms.Control]::ModifierKeys -eq "Control")
    {
        if($KeyUpEventArgs.KeyCode -in 'Add','OemPlus')
        {
            $SCRIPT:currentFontSize++
            UpdateFonts $SCRIPT:currentFontSize
```

```
            $KeyUpEventArgs.Handled = $true
        }
        elseif($KeyUpEventArgs.KeyCode -in 'Subtract','OemMinus')
        {
            $SCRIPT:currentFontSize--
            if($SCRIPT:currentFontSize -lt 1) { $SCRIPT:currentFontSize
= 1 }
            UpdateFonts $SCRIPT:currentFontSize

            $KeyUpEventArgs.Handled = $true
        }
        elseif($KeyUpEventArgs.KeyCode -eq 'D0')
        {
            $SCRIPT:currentFontSize = 12
            UpdateFonts $SCRIPT:currentFontSize

            $KeyUpEventArgs.Handled = $true
        }
    }
}

## A function to handle mouse wheel scrolling.
## In this case, we translate Ctrl-Wheel to zoom.
function OnMouseWheel
{
    param($Sender, $MouseEventArgs)

    if(
        ([System.Windows.Forms.Control]::ModifierKeys -eq "Control") -
and
        ($MouseEventArgs.Delta -ne 0))
    {
        $SCRIPT:currentFontSize += ($MouseEventArgs.Delta / 120)
        if($SCRIPT:currentFontSize -lt 1) { $SCRIPT:currentFontSize = 1
}

        UpdateFonts $SCRIPT:currentFontSize
        $MouseEventArgs.Handled = $true
    }
}

## A function to walk through the parents of a node,
## creating virtual PowerShell syntax to access this property.
function GetPathForNode
{
    param($Node)

    $nodeElements = @()
```

```
    ## Go through all the parents, adding them so that
    ## $nodeElements is in order.
    while($Node)
    {
        $nodeElements = ,$Node + $nodeElements
        $Node = $Node.Parent
    }

    ## Now go through the node elements
    $nodePath = ""
    foreach($Node in $nodeElements)
    {
        $nodeName = $Node.Name

        ## If it was a node that PowerShell is able to enumerate
        ## (but not index), wrap it in the array cast operator.
        if($nodeName.StartsWith('@'))
        {
            $nodeName = $nodeName.Substring(1)
            $nodePath = "@(" + $nodePath + ")"
        }
        elseif($nodeName.StartsWith('['))
        {
            ## If it's a child index, we don't need to
            ## add the dot for property access
        }
        elseif($nodePath)
        {
            ## Otherwise, we're accessing a property. Add a dot.
            $nodePath += "."
        }

        ## Append the node name to the path
        $tempNodePath = $nodePath + $nodeName
        if($nodeName -notmatch '^[$\[\]a-zA-Z0-9]+$')
        {
            $nodePath += "'" + $nodeName + "'"
        }
        else
        {
            $nodePath = $tempNodePath
        }
    }

    ## And return the result
    $nodePath
}

function UpdateFonts
{
```

```
    param($fontSize)

    $treeView.Font = New-Object System.Drawing.Font "Consolas",$fontSize
    $outputPane.Font = New-Object System.Drawing.Font
"Consolas",$fontSize
}

$SCRIPT:currentFontSize = 12

## Create the TreeView, which will hold our object navigation
## area.
$treeView = New-Object Windows.Forms.TreeView
$treeView.Dock = "Top"
$treeView.Height = 500
$treeView.PathSeparator = "."
$treeView.ShowNodeToolTips = $true
$treeView.Add_AfterSelect( { OnAfterSelect @args } )
$treeView.Add_BeforeExpand( { OnBeforeExpand @args } )
$treeView.Add_KeyPress( { OnTreeViewKeyPress @args } )

## Create the output pane, which will hold our object
## member information.
$outputPane = New-Object System.Windows.Forms.TextBox
$outputPane.Multiline = $true
$outputPane.WordWrap = $false
$outputPane.ScrollBars = "Both"
$outputPane.Dock = "Fill"

## Create the root node, which represents the object
## we are trying to show.
$root = New-Object Windows.Forms.TreeNode
$root.ToolTipText = $InputObject.GetType()
$root.Text = $InputObject
$root.Name = '$' + $rootVariableName
$root.Expand()
$null = $treeView.Nodes.Add($root)

UpdateFonts $currentFontSize

## And populate the initial information into the tree
## view.
PopulateNode $root $InputObject

## Finally, create the main form and show it.
$form = New-Object Windows.Forms.Form
$form.Text = "Browsing " + $root.Text
$form.Width = 1000
$form.Height = 800
$form.Controls.Add($outputPane)
$form.Controls.Add($treeView)
```

```
$form.Add_MouseWheel( { OnMouseWheel @args } )
$treeView.Add_KeyUp( { OnKeyUp @args } )
$treeView.Select()
$null = $form.ShowDialog()
$form.Dispose()
```

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

Recipe 10.10

1.27 Record a Transcript of Your Shell Session

Problem

You want to record a log or transcript of your shell session.

Solution

To record a transcript of your shell session, run the command `Start-Transcript`. It has an optional `-Path` parameter that defaults to a filename based on the current system time. By default, PowerShell places this file in the *My Documents* directory. To stop recording the transcript of your shell system, run the command `Stop-Transcript`.

Discussion

Although the `Get-History` cmdlet is helpful, it does not record the output produced during your PowerShell session. To accomplish that, use the `Start-Transcript` cmdlet. In addition to the `Path` parameter described previously, the `Start-Transcript` cmdlet also supports parameters that let you control how PowerShell interacts with the output file.

If you don't specify a `-Path` parameter, PowerShell generates a random file name for you. If you want to process this file after stopping the transcript, PowerShell adds this as a property name to the output of either `Start-`

`Transcript` or `Stop-Transcript`:

```
PS > $myTranscript = Start-Transcript
PS > Stop-Transcript
Transcript stopped, output file is D:\Lee\PowerShell_transcript...
PS > $myTranscript | fl * -force

Path   : D:\Lee\PowerShell_transcript.LEE-
DESKTOP.kg_Vsm_o.20201217195052.txt
Length : 104

PS > $myTranscript.Path
D:\Lee\PowerShell_transcript.LEE-DESKTOP.kg_Vsm_o.20201217195052.txt
```

PowerShell transcripts start with a standard file header that includes time, user, host name, as well as several other useful items. If you specify the `-IncludeInvocationHeader` parameter either interactively or through system-wide policy, PowerShell also includes a separator between commands to assist in automatic analysis.

```
**********************
PowerShell transcript start
Start time: 20201217190500
Username: ubuntu-20-04\lee
Machine: ubuntu-20-04 (Unix 4.19.128.0)
Host Application: /opt/microsoft/powershell/7/pwsh.dll
Process ID: 1925
OS: Linux 4.19.128-microsoft-standard #1 SMP Tue Jun 23 12:58:10 UTC
2020
(...)
**********************

**********************
Command start time: 20201217190502
**********************
PS /mnt/c/Users/lee> Get-Process

 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      0     0.00       5.26       0.16     984 984 bash
      0     0.00       0.53       0.02       1   0 init
      0     0.00       0.07       0.00     982 982 init
      0     0.00       0.08       0.32     983 982 init
      0     0.00      96.52       0.64    1925 984 pwsh
      0     0.00       3.25       0.00    1873 …73 rsyslogd

**********************
```

```
**********************
Command start time: 20201217190504
**********************
PS /mnt/c/Users/lee> cat /var/log/powershell.log
(...)
```

In addition to letting you record transcripts manually, PowerShell also lets you set a system policy to record these automatically. For more information on how to set this up, see Recipe 18.2.

See Also

Recipe 18.2

1.28 Extend Your Shell with Additional Commands

Problem

You want to use PowerShell cmdlets, providers, or script-based extensions written by a third party.

Solution

If the module is part of the standard PowerShell module path, simply run the command you want.

```
Invoke-NewCommand
```

If it is not, use the `Import-Module` command to import third-party commands into your PowerShell session.

To import a module from a specific directory:

```
Import-Module c:\path\to\module
```

To import a module from a specific file (module, script, or assembly):

```
Import-Module c:\path\to\module\file.ext
```

Discussion

PowerShell supports two sets of commands that enable additional cmdlets and providers: `*-Module` and `*-PsSnapin`. Snapins were the packages for extensions in version 1 of PowerShell, and are rarely used. Snapins supported only compiled extensions and had onerous installation requirements.

Version 2 of PowerShell introduced *modules* that support everything that snapins support (and more) without the associated installation pain. That said, PowerShell version 2 also required that you remember which modules contained which commands and manually load those modules before using them. Windows now includes thousands of commands in hundreds of modules—quickly making reliance on one's memory an unsustainable approach.

Any recent version of PowerShell significantly improves the situation by autoloading modules for you. Internally, PowerShell maintains a mapping of command names to the module that contains them. Simply start using a command (which the `Get-Command` cmdlet can help you discover), and PowerShell loads the appropriate module automatically. If you wish to customize this autoloading behavior, you can use the `$PSModuleAutoLoadingPreference` preference variable.

When PowerShell imports a module with a given name, it searches through every directory listed in the `PSModulePath` environment variable, looking for the first module that contains the subdirectories that match the name you specify. Inside those directories, it looks for the module (`*.psd1`, `*.psm1`, and `*.dll`) with the same name and loads it.

> **NOTE**
>
> When autoloading modules, PowerShell prefers modules in the system's module directory over those in your personal module path. This prevents user modules from accidentally overriding core functionality. If you want a module to override core functionality, you can still use the `Import-Module` cmdlet to load the module explicitly.

When you install a module on your own system, the most common place to put it is in the *PowerShell\Modules* directory in your *My Documents* directory. In Windows PowerShell, this location will be *WindowsPowerShell\Modules*. To

have PowerShell look in another directory for modules, add it to your personal `PSModulePath` environment variable, just as you would add a *Tools* directory to your personal path.

For more information about managing system paths, see Recipe 16.2.

If you want to load a module from a directory not in `PSModulePath`, you can provide the entire directory name and module name to the `Import-Module` command. For example, for a module named `Test`, use `Import-Module` *c:\path\to*`\Test`. As with loading modules by name, PowerShell looks in `c:\temp\path\to` for a module (*.*psd1*, *.*psm1*, or *.*dll*) named `Test` and loads it.

If you know the specific module file you want to load, you can also specify the full path to that module.

If you want to find additional commands, see Recipe 1.29.

See Also

Recipe 1.9

Recipe 11.6

Recipe 16.2

Recipe 1.29

1.29 Find and Install Additional PowerShell Scripts and Modules

Problem

You want to find additional modules to extend your shell's functionality.

Solution

Use the `Find-Module` command to find interesting modules:

```
PS > Find-Module *Cookbook* | Format-List
```

```
Name                           : PowerShellCookbook
Version                        : 1.3.6
Type                           : Module
Description                    : Sample scripts from the Windows
PowerShell
                                 Cookbook
Author                         : Lee Holmes
(...)
```

Then use `Install-Module` to add them to your system.

```
Install-Module PowerShellCookbook -Scope CurrentUser
```

Similarly, use the `Find-Script` and `Install-Script` commands if the item has been published as a standalone script. If you haven't already on your machine, make sure to add `My Documents\PowerShell\Scripts` to your system path. For more information about modifying your system path, see Recipe 16.2.

```
PS > Find-Script Get-WordCluster | Install-Script -Scope CurrentUser
PS > Get-WordCluster -Count 3
"Hello","World","Jello","Mellow","Jealous","Wordy","Sword"

Representatitive Items
---------------- -----
Wordd            {World, Wordy, Sword}
Jealou           {Jello, Jealous}
Hellow           {Hello, Mellow}
```

Discussion

The PowerShell Gallery is the worldwide hub for publishing and sharing PowerShell scripts and modules. It contains thousands of modules: official corporate releases by Microsoft and many other companies, popular community projects like the DbaTools module for SQL management, and fun whimsical ones like OutConsolePicture to display images as ANSI graphics.

Its web interface lets you search, browse, and explore through *https://www.powershellgallery.com*, but of course that's not the way you use it through PowerShell.

In PowerShell, the `Find-Module` and `Install-Module` commands let you

interact with the PowerShell Gallery and install modules from it. You can find modules by name, tags, and even Just Enough Administration role capabilities.

When you first try to install a module from the PowerShell Gallery, PowerShell will provide a warning:

```
PS > Install-Module someModule -Scope CurrentUser

Untrusted repository
You are installing the modules from an untrusted repository. If you
trust this repository, change its InstallationPolicy value by running
the Set-PSRepository cmdlet. Are you sure you want to install the
modules from 'PSGallery'?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "N"):
```

Common to all other code sharing repositories out there, there are no restrictions on who can publish to the PowerShell Gallery or what they can publish. If a module is reported through the abuse reporting mechanisms and found to be malicious or against the gallery's Terms of Service it will of course be removed. But other than that - you should not consider the gallery to be vetted, approved, or otherwise implicitly trustworthy. To acknowledge this and remove the warning from future module installations, you can can declare the PowerShell Gallery to be trusted on your machine:

```
Set-PSRepository -Name PSGallery -InstallationPolicy Trusted
```

In addition to the public PowerShell Gallery, PowerShell can also talk to private galleries (including file shares!) as well. PowerShell uses the NuGet protocol. For more information about creating a private PowerShell Gallery, see *https://docs.microsoft.com/en-us/powershell/scripting/gallery/how-to/working-with-local-psrepositories*.

See Also

Recipe 16.2

https://docs.microsoft.com/en-us/powershell/scripting/gallery/how-to/working-with-local-psrepositories

1.30 Use Commands from Customized Shells

Problem

You want to use the commands from a PowerShell-based product that launches a customized version of the PowerShell console, but in a regular PowerShell session.

Solution

Launch the customized version of the PowerShell console, and then use the `Get-Module` and `Get-PsSnapin` commands to see what additional modules and/or snapins it loaded.

Discussion

As described in Recipe 1.28, PowerShell modules and snapins are the two ways that third parties can distribute and add additional PowerShell commands. Products that provide customized versions of the PowerShell console do this by launching PowerShell with one of three parameters:

- `-PSConsoleFile`, to load a console file that provides a list of snapins to load.
- `-Command`, to specify an initial startup command (that then loads a snapin or module)
- `-File`, to specify an initial startup script (that then loads a snapin or module)

Regardless of which one it used, you can examine the resulting set of loaded extensions to see which ones you can import into your other PowerShell sessions.

Detecting loaded snapins

The `Get-PsSnapin` command returns all snapins loaded in the current session. It always returns the set of core PowerShell snapins, but it will also return any additional snapins loaded by the customized environment. For example, if the name of a snapin you recognize is

Product.Feature.Commands, you can load that into future PowerShell sessions by typing `Add-PsSnapin` *`Product.Feature.Commands`*. To automate this, add the command into your PowerShell profile.

If you are uncertain of which snapin to load, you can also use the `Get-Command` command to discover which snapin defines a specific command:

```
PS > Get-Command Get-Counter | Select PsSnapin

PSSnapIn
--------
Microsoft.PowerShell.Diagnostics
```

Detecting loaded modules

Like the `Get-PsSnapin` command, the `Get-Module` command returns all modules loaded in the current session. It returns any modules you've added so far into that session, but it will also return any additional modules loaded by the customized environment. For example, if the name of a module you recognize is *`ProductModule`*, you can load that into future PowerShell sessions by typing `Import-Module` *`ProductModule`*. To automate this, add the command into your PowerShell profile.

If you are uncertain of which module to load, you can also use the `Get-Command` command to discover which module defines a specific command:

```
PS > Get-Command Start-BitsTransfer | Select Module

Module
------
BitsTransfer
```

See Also

Recipe 1.28

1.31 Save State Between Sessions

Problem

You want to save state or history between PowerShell sessions.

Solution

Subscribe to the `PowerShell.Exiting` engine event to have PowerShell invoke a script or script block that saves any state you need.

To have PowerShell save your command history, place a call to `Enable-+History+Persistence` in your profile, as in Example 1-19.

Example 1-19. Enable-HistoryPersistence.ps1

```
##############################################################################
######
##
## Enable-HistoryPersistence
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Reloads any previously saved command history, and registers for the
PowerShell.Exiting engine event to save new history when the shell
exits.

#>

Set-StrictMode -Version 3

## Load our previous history
$GLOBAL:maximumHistoryCount = 32767
$historyFile = (Join-Path (Split-Path $profile) "commandHistory.clixml")
if(Test-Path $historyFile)
{
    Import-CliXml $historyFile | Add-History
}

## Register for the engine shutdown event
$null = Register-EngineEvent -SourceIdentifier `
    ([System.Management.Automation.PsEngineEvent]::Exiting) -Action {

    ## Save our history
```

```
    $historyFile = (Join-Path (Split-Path $profile)
"commandHistory.clixml")
    $maximumHistoryCount = 1kb

    ## Get the previous history items
    $oldEntries = @()
    if(Test-Path $historyFile)
    {
        $oldEntries = Import-CliXml $historyFile -ErrorAction
SilentlyContinue
    }

    ## And merge them with our changes
    $currentEntries = Get-History -Count $maximumHistoryCount
    $additions = Compare-Object $oldEntries $currentEntries `
        -Property CommandLine | Where-Object { $_.SideIndicator -eq "=>"
} |
        Foreach-Object { $_.CommandLine }

    $newEntries = $currentEntries | ? { $additions -contains
$_.CommandLine }

    ## Keep only unique command lines. First sort by CommandLine in
    ## descending order (so that we keep the newest entries,) and then
    ## re-sort by StartExecutionTime.
    $history = @($oldEntries + $newEntries) |
        Sort -Unique -Descending CommandLine | Sort StartExecutionTime

    ## Finally, keep the last 100
    Remove-Item $historyFile
    $history | Select -Last 100 | Export-CliXml $historyFile
}
```

Discussion

PowerShell provides easy script-based access to a broad variety of system, engine, and other events. You can register for notification of these events and even automatically process any of those events. In this example, we subscribe to the only one currently available, which is called `PowerShell.Exiting`. PowerShell generates this event when you close a session.

This script could do anything, but in this example we have it save our command history and restore it when we launch PowerShell. Why would we want to do this? Well, with a rich history buffer, we can more easily find and reuse commands we've previously run. For two examples of doing this, see Examples Recipe 1.21 and Recipe 1.23.

Example 1-19 takes two main actions. First, we load our stored command history (if any exists). Then, we register an automatic action to be processed whenever the engine generates its `PowerShell.Exiting` event. The action itself is relatively straightforward, although exporting our new history does take a little finesse. If you have several sessions open at the same time, each will update the saved history file when it exits. Since we don't want to overwrite the history saved by the other shells, we first reload the history from disk and combine it with the history from the current shell.

Once we have the combined list of command lines, we sort them and pick out the unique ones before storing them back in the file.

For more information about working with PowerShell engine events, see Recipe 32.3.

See Also

Recipe 1.2

Recipe 1.21

Recipe 32.3

Chapter 2. Pipelines

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please visit *https://www.powershellcookbook.com/4th_ed_techreview*. You can also reach out to the author at powershellcookbook@leeholmes.com.

2.0 Introduction

One of the fundamental concepts in a shell is called the *pipeline*. It also forms the basis of one of PowerShell's most significant advances. A pipeline is a big name for a simple concept—a series of commands where the output of one becomes the input of the next. A pipeline in a shell is much like an assembly line in a factory: it successively refines something as it passes between the stages, as shown in Example 2-1.

Example 2-1. A PowerShell pipeline

```
Get-Process | Where-Object WorkingSet -gt 500kb | Sort-Object -
Descending Name
```

In PowerShell, you separate each stage in the pipeline with the pipe (`|`) character.

In Example 2-1, the `Get-Process` cmdlet generates objects that represent actual processes on the system. These process objects contain information about the process's name, memory usage, process ID, and more. As the `Get-Process` cmdlet generates output, it passes it along. Simultaneously, the `Where-Object` cmdlet gets to work directly with those processes, testing easily for those that use more than 500 KB of memory. It passes those along

immediately as it processes them, allowing the `Sort-Object` cmdlet to also work directly with those processes and sort them by name in descending order.

This brief example illustrates a significant advancement in the power of pipelines: PowerShell passes full-fidelity objects along the pipeline, not their text representations.

In contrast, all other shells pass data as plain text between the stages. Extracting meaningful information from plain-text output turns the authoring of pipelines into a black art. Expressing the previous example in a traditional Unix-based shell is exceedingly difficult, and it is nearly impossible in *cmd.exe*.

Traditional text-based shells make writing pipelines so difficult because they require you to deeply understand the peculiarities of output formatting for each command in the pipeline, as shown in Example 2-2.

Example 2-2. A traditional text-based pipeline

```
lee@ubuntu-20-04:~$ ps -F | awk '{ if($5 > 500) print }' | sort -r -k
64,70
UID          PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
lee         8175  7967  0   965  1036   0 21:51 pts/0    00:00:00 ps -F
lee         7967  7966  0  1173  2104   0 21:38 pts/0    00:00:00 -bash
```

In this example, you have to know that, for every line, group number five represents the memory usage. You have to know another language (that of the `awk` tool) to filter by that column. Finally, you have to know the column range that contains the process name (columns 64 to 70 on this system) and then provide that to the `sort` command. And that's just a simple example.

An object-based pipeline opens up enormous possibilities, making system administration both immensely more simple and more powerful.

2.1 Chain Commands Based on their Success or Error

Problem

You wish to chain together multiple commands based on the success of previous commands in the pipeline

Solution

Use the && and || pipeline chain operators:

```
PS > Invoke-Command localhost { "Some command output" } && "Connection
successful!"
Some command output
Connection successful!

PS > Invoke-Command missing_computer { "Some command output" } &&
"Connection successful!"
OpenError: [missing_computer] Connecting to remote server
missing_computer failed with (...)

PS > Invoke-Command missing_computer { "Some command output" } ||
"Connection failed."
OpenError: [missing_computer] Connecting to remote server
missing_computer failed with (...)
Connection failed.
```

Discussion

If you wish to chain together multiple commands based on the success of other commands in the pipeline, you can use PowerShell's pipeline chain operators. The && operator only executes the next command if the previous command was successful. The || operator only executes the next command if the previous command failed.

For the pipeline chain operators, success of a command is determined by the $? ("dollar hook") automatic variable. For more information about the $? automatic variable, see Recipe 15.1.

See Also

Recipe 15.1

2.2 Filter Items in a List or Command Output

Problem

You want to filter the items in a list or command output.

Solution

Use the `Where-Object` cmdlet to select items in a list (or command output) that match a condition you provide. The `Where-Object` cmdlet has the standard aliases `where` and ?.

To list all running processes that have "search" in their name, use the `-like` operator to compare against the process's `Name` property:

```
Get-Process | Where-Object { $_.Name -like "*Search*" }
```

To list all stopped services, use the `-eq` operator to compare against the service's `Status` property:

```
Get-Service | Where-Object { $_.Status -eq "Stopped" }
```

To list all processes not responding, test the `Responding` property:

```
Get-Process | Where-Object { -not $_.Responding }
```

For simple comparisons on properties, you can omit the script block syntax and use the comparison parameters of `Where-Object` directly:

```
Get-Process | Where-Object Name -like "*Search*"
```

Discussion

For each item in its input (which is the output of the previous command), the `Where-Object` cmdlet evaluates that input against the script block that you specify. If the script block returns `True`, then the `Where-Object` cmdlet passes the object along. Otherwise, it does not. A script block is a series of PowerShell commands enclosed by the `{` and `}` characters. You can write any PowerShell commands inside the script block. In the script block, the `$_` (or `$PSItem`) variable represents the current input object. For each item in the incoming set of objects, PowerShell assigns that item to the `$_` (or `$PSItem`) variable and then runs your script block. In the preceding examples, this incoming object represents the process, file, or service that the previous cmdlet generated.

This script block can contain a great deal of functionality, if desired. It can combine multiple tests, comparisons, and much more. For more information about script blocks, see Recipe 11.4. For more information about the type of comparisons available to you, see Appendix A.

For simple filtering, the syntax of using script blocks in the `Where-Object` cmdlet may sometimes seem overbearing. For these scenarios, `Where-Object` offers parameters that directly support parameters to apply simple comparisons like `-Eq`, `-Match`, `-In`, and more.

In addition to the script block syntax offered by the `Where-Object` cmdlet, Powershell also offers a version built into the language itself: the `where()` method. This is slightly faster for very large data collections, although the time it takes to collect those items (such as getting the list of files in a directory) normally dwarfs any time it takes to filter them. The `where()` method does offer several additional useful modes, however, through, its second parameter.

Get the first part of a list:

```
PS > (1..10).where( { $_ -eq 5 }, "Until" )
1
2
3
4
```

Get the second part of a list:

```
PS > (1..10).where( { $_ -eq 5 }, "SkipUntil" )
5
6
7
8
9
10
```

Split a list:

```
PS > $even,$odd = (1..10).where( { $_ % 2 -eq 0 }, "Split" )
PS > $even -join ","
2,4,6,8,10
PS > $odd -join ","
1,3,5,7,9
```

For complex filtering (for example, the type you would normally rely on a mouse to do with files in an Explorer window), writing the script block to express your intent may be difficult or even infeasible. If this is the case, Recipe 2.4 shows a script that can make manual filtering easier to accomplish.

For more information about the `Where-Object` cmdlet, type **`Get-Help Where-Object`**. For more information about the `where()` method, type **`Get-Help about_Arrays`**

See Also

Recipe 2.4

Recipe 11.4

Appendix A

2.3 Group and Pivot Data by Name

Problem

You want to easily access items in a list by a property name.

Solution

Use the `Group-Object` cmdlet (which has the standard alias `group`) with the `-AsHash` and `-AsString` parameters. This creates a hashtable with the selected property (or expression) used as keys in that hashtable:

```
PS > $h = dir | group -AsHash -AsString Length
PS > $h

Name                           Value
----                           -----
746                            {ReplaceTest.ps1}
499                            {Format-String.ps1}
20494                          {test.dll}

PS > $h["499"]

    Directory: C:\temp
```

```
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         10/18/2009   9:57 PM        499 Format-String.ps1


PS > $h["746"]

    Directory: C:\temp

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         10/18/2009   9:51 PM        746 ReplaceTest.ps1
```

Discussion

In some situations, you might find yourself repeatedly calling the `Where-Object` cmdlet to interact with the same list or output:

```
PS > $processes = Get-Process
PS > $processes | Where-Object { $_.Id -eq 1216 }

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     62       3     1012       3132    50     0.20   1216 dwm


PS > $processes | Where-Object { $_.Id -eq 212 }

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    614      10    28444       5484   117     1.27    212
SearchIndexer
```

In these situations, you can instead use the `-AsHash` parameter of the `Group-Object` cmdlet. When you use this parameter, PowerShell creates a hashtable to hold your results. This creates a map between the property you are interested in and the object it represents:

```
PS > $processes = Get-Process | Group-Object -AsHash Id
PS > $processes[1216]

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     62       3     1012       3132    50     0.20   1216 dwm

PS > $processes[212]
```

```
PS > $processes[212]

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    610      10    28444       5488   117     1.27    212
SearchIndexer
```

For simple types of data, this approach works well. Depending on your data, though, using the `-AsHash` parameter alone can create difficulties.

The first issue you might run into arises when the value of a property is `$null`. Hashtables in PowerShell (and the .NET Framework that provides the underlying support) do not support `$null` as a value, so you get a misleading error message:

```
PS > "Hello",(Get-Process -id $pid) | Group-Object -AsHash Id
Group-Object : The objects grouped by this property cannot be expanded
since there is a duplication of the key. Please give a valid property
and try
again.
```

A second issue crops up when more complex data gets stored within the hashtable. This can unfortunately be true even of data that *appears* to be simple:

```
PS > $result = dir | Group-Object -AsHash Length
PS > $result

Name                           Value
----                           -----
746                            {ReplaceTest.ps1}
499                            {Format-String.ps1}
20494                          {test.dll}

PS > $result[746]
(Nothing appears)
```

This missing result is caused by an incompatibility between the information in the hashtable and the information you typed. This is normally not an issue in hashtables that you create yourself, because you provided all of the information to populate them. In this case, though, the `Length` values stored in the hashtable come from the directory listing and are of the type `Int64`. An explicit cast resolves the issue but takes a great deal of trial and error to discover:

```
PS > $result[ [int64] 746 ]

    Directory: C:\temp

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         10/18/2009   9:51 PM        746 ReplaceTest.ps1
```

It is difficult to avoid both of these issues, so the `Group-Object` cmdlet also offers an `-AsString` parameter to convert all of the values to their string equivalents. With that parameter, you can always assume that the values will be treated as (and accessible by) strings:

```
PS > $result = dir | Group-Object -AsHash -AsString Length
PS > $result["746"]

    Directory: C:\temp

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         10/18/2009   9:51 PM        746 ReplaceTest.ps1
```

For more information about the `Group-Object` cmdlet, type **`Get-Help Group-Object`**. For more information about PowerShell hashtables, see Recipe 7.13.

See Also

Recipe 7.13

Appendix A

2.4 Interactively Filter Lists of Objects

There are times when the scriptblock syntax of `Where-Object` cmdlet is too powerful. In those situations, the simplified property access parameters provides a much simpler alternative. There are also times when the `Where-Object` cmdlet is too simple—when expressing your selection logic as code is more cumbersome than selecting it manually. In those situations, an interactive filter can be much more effective.

PowerShell makes this interactive filtering incredibly easy through the `-PassThru` parameter of the `Out-GridView` cmdlet. For example, you can use this parameter after experimenting with commands for a while to create a simple script. Simply highlight the lines you want to keep, and press OK:

```
PS > $script = Get-History | Foreach-Object CommandLine | Out-GridView
-PassThru
PS > $script | Set-Content c:\temp\script.ps1
```

By default, the `Out-GridView` cmdlet lets you select multiple items at once before pressing OK. If you'd rather constrain the selection to a single element, use `Single` as the value of the `-OutputMode` parameter.

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

Recipe 2.2

2.5 Work with Each Item in a List or Command Output

Problem

You have a list of items and want to work with each item in that list.

Solution

Use the `Foreach-Object` cmdlet (which has the standard aliases `foreach` and %) to work with each item in a list.

To apply a calculation to each item in a list, use the `$_` (or `$PSItem`) variable as part of a calculation in the script block parameter:

```
PS > 1..10 | Foreach-Object { $_ * 2 }
2
4
6
```

```
0
8
10
12
14
16
18
20
```

To run a program on each file in a directory, use the `$_` (or `$PSItem`) variable as a parameter to the program in the script block parameter:

```
Get-ChildItem *.txt | Foreach-Object { attrib -r $_ }
```

To access a method or property for each object in a list, access that method or property on the `$_` (or `$PSItem`) variable in the script block parameter. In this example, you get the list of running processes called `notepad`, and then wait for each of them to exit:

```
$notepadProcesses = Get-Process notepad
$notepadProcesses | Foreach-Object { $_.WaitForExit() }
```

Discussion

Like the `Where-Object` cmdlet, the `Foreach-Object` cmdlet runs the script block that you specify for each item in the input. A script block is a series of PowerShell commands enclosed by the `{` and `}` characters. For each item in the set of incoming objects, PowerShell assigns that item to the `$_` (or `$PSItem`) variable, one element at a time. In the examples given by the Solution, the `$_` (or `$PSItem`) variable represents each file or process that the previous cmdlet generated.

> **NOTE**
>
> The first example in the Solution demonstrates a neat way to generate ranges of numbers:`1..10`
>
> This is PowerShell's array range syntax, which you can learn more about in Recipe 7.3.

This script block can contain a great deal of functionality, if desired. You can

combine multiple tests, comparisons, and much more. For more information about script blocks, see Recipe 11.4. For more information about the type of comparisons available to you, see Appendix A.

In addition to the script block supported by the `Foreach-Object` cmdlet to process each element of the pipeline, it also supports script blocks to be executed at the beginning and end of the pipeline. For example, consider the following code to measure the sum of elements in an array:

```
$myArray = 1,2,3,4,5
$sum = 0
$myArray | Foreach-Object { $sum += $_ }
$sum
```

You can simplify this to:

```
$myArray | Foreach-Object -Begin {
    $sum = 0 } -Process { $sum += $_ } -End { $sum }
```

Since you can also specify the `-Begin`, `-Process`, and `-End` parameters by position, this can simplify even further to:

```
$myArray | Foreach-Object { $sum = 0 } { $sum += $_ } { $sum }
```

For simple scenarios (such as retrieving only a single property), the script-block-based syntax can get a little ungainly:

```
Get-Process | Foreach-Object { $_.Name }
```

In PowerShell, the `Foreach-Object` cmdlet (and by extension its % alias) also supports parameters to simplify property and method access dramatically:

```
Get-Process | Foreach-Object Name
Get-Process | % Name | % ToUpper
```

As with the `Where-Object` cmdlet, PowerShell offers a `foreach()` method on collections that let you perform many of these same tasks:

```
## Property access
```

```
(Get-Process).foreach("Name")

## Script block invocation
$sum = 0
(1..10).foreach( { $sum += $_} )

## Type conversion
$bytes = (1..10).foreach( [Byte] )
```

In addition to using the `Foreach-Object` cmdlet to support full member invocation, the PowerShell language has a quick way to easily enumerate properties. Just as you are able to access a property on a single element, PowerShell lets you use a similar syntax to access that property on each item of a collection:

```
PS > Start-Process PowerShell
PS > Start-Process PowerShell
PS > $processes = Get-Process -Name PowerShell
PS > $processes[0].Id
7928

PS > $processes.Id
7928
13120
```

While writing more advanced pipelines, you might sometimes find yourself writing a `Where-Object` or `Foreach-Object` script block within another script block that is already processing pipeline input. In this situation, you lose access to the outer `$_` (or `$PSItem`) variable within the inner script block:

```
## Get all processes
Get-Process | Foreach-Object {
    ## Get all of their modules (loaded DLLs)
    $_.Modules | Foreach-Object {
        ## If the DLL is loaded from AppData
        if($_.FileName -match 'AppData') {
            ## Desired behavior: Output the process name
            ## Actual behavior: Outputs the module name
            $_
        }
    }
}
```

To solve this problem, PowerShell supports the `-PipelineVariable` parameter. When you add this parameter to a command, PowerShell saves the command's current pipeline output into the variable name that you specify in addition to the `$_` variable. At this point you can use it from within other nested script blocks freely without it being overwritten:

```
## Get all processes
Get-Process -PipelineVariable currentProcess | Foreach-Object {
    ## Get all of their modules (loaded DLLs)
    $_.Modules | Foreach-Object {
        ## If the DLL is loaded from AppData
        if($_.FileName -match 'AppData') {
            ## Output the process name
            $currentProcess
        }
    } | Select-Object -First 1
}
```

The `Foreach-Object` cmdlet isn't the only way to perform actions on items in a list. The PowerShell scripting language supports several other keywords, such as `for`, (a different) `foreach`, `do`, and `while`. For information on how to use those keywords, see Recipe 4.4.

For more information about the `Foreach-Object` cmdlet, type **`Get-Help Foreach-Object`**.

For more information about dealing with pipeline input in your own scripts, functions, and script blocks, see Recipe 11.18.

See Also

Recipe 4.4

Recipe 7.3

Recipe 11.4

Recipe 11.18

Appendix A

2.6 Automate Data-Intensive Tasks

Problem

You want to invoke a simple task on large amounts of data.

Solution

If only one piece of data changes (such as a server name or username), store the data in a text file. Use the `Get-Content` cmdlet to retrieve the items, and then use the `Foreach-Object` cmdlet (which has the standard aliases `foreach` and %) to work with each item in that list. Example 2-3 illustrates this technique.

Example 2-3. Using information from a text file to automate data-intensive tasks

```
PS > Get-Content servers.txt
SERVER1
SERVER2
PS > $computers = Get-Content servers.txt
PS > $computers | Foreach-Object {
     Get-CimInstance Win32_OperatingSystem -Computer $_ }

SystemDirectory : C:\WINDOWS\system32
Organization    :
BuildNumber     : 2600
Version         : 5.1.2600

SystemDirectory : C:\WINDOWS\system32
Organization    :
BuildNumber     : 2600
Version         : 5.1.2600
```

If it becomes cumbersome (or unclear) to include the actions in the `Foreach-Object` cmdlet, you can also use the `foreach` scripting keyword, as illustrated in Example 2-4.

Example 2-4. Using the foreach scripting keyword to make a looping statement easier to read

```
$computers = Get-Content servers.txt

foreach($computer in $computers)
{
    ## Get the information about the operating system from WMI
    $system = Get-CimInstance Win32_OperatingSystem -Computer $computer

    ## Determine if it is running Windows XP
    if($system.Version -eq "5.1.2600")
    {
```

```
        "$computer is running Windows XP"
    }
}
```

If several aspects of the data change per task (for example, both the CIM class and the computer name for computers in a large report), create a CSV file with a row for each task. Use the `Import-Csv` cmdlet to import that data into PowerShell, and then use properties of the resulting objects as multiple sources of related data. Example 2-5 illustrates this technique.

Example 2-5. Using information from a CSV to automate data-intensive tasks

```
PS > Get-Content WmiReport.csv
ComputerName,Class
LEE-DESK,Win32_OperatingSystem
LEE-DESK,Win32_Bios
PS > $data = Import-Csv WmiReport.csv
PS > $data

ComputerName                            Class
------------                            -----
LEE-DESK                                Win32_OperatingSystem
LEE-DESK                                Win32_Bios

PS > $data |
    Foreach-Object { Get-CimInstance $_.Class -Computer $_.ComputerName
}

SystemDirectory : C:\WINDOWS\system32
Organization    :
BuildNumber     : 2600
Version         : 5.1.2600

SMBIOSBIOSVersion : ASUS A7N8X Deluxe ACPI BIOS Rev 1009
Manufacturer      : Phoenix Technologies, LTD
Name              : Phoenix - AwardBIOS v6.00PG
SerialNumber      : xxxxxxxxxxx
Version           : Nvidia - 42302e31
```

Discussion

One of the major benefits of PowerShell is its capability to automate repetitive tasks. Sometimes these repetitive tasks are action-intensive (such as system maintenance through registry and file cleanup) and consist of complex sequences of commands that will always be invoked together. In those situations, you can write a script to combine these operations to save time and reduce errors.

Other times, you need only to accomplish a single task (for example, retrieving the results of a WMI query) but need to invoke that task repeatedly for a large amount of data. In those situations, PowerShell's scripting statements, pipeline support, and data management cmdlets help automate those tasks.

One of the options given by the Solution is the `Import-Csv` cmdlet. The `Import-Csv` cmdlet reads a CSV file and, for each row, automatically creates an object with properties that correspond to the names of the columns. Example 2-6 shows the results of a CSV that contains a `ComputerName` and `Class` header.

Example 2-6. The Import-Csv cmdlet creating objects with ComputerName and Class properties

```
PS > $data = Import-Csv WmiReport.csv
PS > $data

ComputerName                            Class
------------                            -----
LEE-DESK                                Win32_OperatingSystem
LEE-DESK                                Win32_Bios

PS > $data[0].ComputerName
LEE-DESK
```

As the Solution illustrates, you can use the `Foreach-Object` cmdlet to provide data from these objects to repetitive cmdlet calls. It does this by specifying each parameter name, followed by the data (taken from a property of the current CSV object) that applies to it.

> **NOTE**
>
> If you already have the comma-separated values in a variable (rather than a file), you can use the `ConvertFrom-Csv` cmdlet to convert these values to objects.

While this is the most general solution, many cmdlet parameters can automatically retrieve their value from incoming objects if any property of that object has the same name. This enables you to omit the `Foreach-Object` and property mapping steps altogether. Parameters that support this feature are said to support *value from pipeline by property name*. The `Move-Item` cmdlet is

one example of a cmdlet with parameters that support this, as shown by the `Accept pipeline input?` rows in Example 2-7.

Example 2-7. Help content of the Move-Item cmdlet showing a parameter that accepts value from pipeline by property name

```
PS > Get-Help Move-Item -Full
(...)
PARAMETERS

    -path <string[]>
        Specifies the path to the current location of the items. The
default
        is the current directory. Wildcards are permitted.

        Required?                    true
        Position?                    1
        Default value                <current location>
        Accept pipeline input?       true (ByValue, ByPropertyName)
        Accept wildcard characters?  true

    -destination <string>
        Specifies the path to the location where the items are being
moved.
        The default is the current directory. Wildcards are permitted,
but
        the result must specify a single location.

        To rename the item being moved, specify a new name in the value
of
        Destination.

        Required?                    false
        Position?                    2
        Default value                <current location>
        Accept pipeline input?       true (ByPropertyName)
        Accept wildcard characters?  True
        (...)
```

If you purposefully name the columns in the CSV to correspond to parameters that take their value from pipeline by property name, PowerShell can do some (or all) of the parameter mapping for you. Example 2-8 demonstrates a CSV file that moves items in bulk.

Example 2-8. Using the Import-Csv cmdlet to automate a cmdlet that accepts value from pipeline by property name

```
PS > Get-Content ItemMoves.csv
Path,Destination
```

```
test.txt,Test1Directory
test2.txt,Test2Directory
PS > dir test.txt,test2.txt | Select Name

Name
----
test.txt
test2.txt

PS > Import-Csv ItemMoves.csv | Move-Item
PS > dir Test1Directory | Select Name

Name
----
test.txt

PS > dir Test2Directory | Select Name
Name
----
test2.txt
```

For more information about the `Foreach-Object` cmdlet and `foreach` scripting keyword, see Recipe 2.5. For more information about working with CSV files, see Recipe 10.7. For more information about working with Windows Management Instrumentation (WMI), see Chapter 28.

See Also

Recipe 2.5

Recipe 10.7

Chapter 28

2.7 Intercept Stages of the Pipeline

Problem

You want to intercept or take some action at different stages of the PowerShell pipeline.

Solution

Use the `New-CommandWrapper` script given in Recipe 11.23 to wrap the `Out-Default` command, and place your custom functionality in that.

Discussion

For any pipeline, PowerShell adds an implicit call to the `Out-Default` cmdlet at the end. By adding a command wrapper over this function we can heavily customize the pipeline processing behavior.

When PowerShell creates a pipeline, it first calls the `BeginProcessing()` method of each command in the pipeline. For advanced functions (the type created by the `New-CommandWrapper` script), PowerShell invokes the `Begin` block. If you want to do anything at the beginning of the pipeline, then put your customizations in that block.

For each object emitted by the pipeline, PowerShell sends that object to the `ProcessRecord()` method of the next command in the pipeline. For advanced functions (the type created by the `New-CommandWrapper` script), PowerShell invokes the `Process` block. If you want to do anything for each element in the pipeline, put your customizations in that block.

Finally, when PowerShell has processed all items in the pipeline, it calls the `EndProcessing()` method of each command in the pipeline. For advanced functions (the type created by the `New-CommandWrapper` script), PowerShell invokes the `End` block. If you want to do anything at the end of the pipeline, then put your customizations in that block.

For two examples of this approach, see Recipe 2.8 and Recipe 11.22.

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

Recipe 2.8

Recipe 11.22

Recipe 11.23

2.8 Automatically Capture Pipeline Output

Problem

You want to automatically capture the output of the last command without explicitly storing its output in a variable.

Solution

Use the `PSDefaultParameterValues` automatic variable to set the `-OutVariable` parameter value of the `Out-Default` command to a variable name of your choice:

```
$PSDefaultParameterValues["Out-Default:OutVariable"] = "__"
```

Discussion

Once each object in a command has passed through the pipeline, it eventually reaches the end. If your script does not capture this output, PowerShell provides it to the `Out-Default` cmdlet which is then responsible for figuring out how to format and display the output.

Like all cmdlets, the `Out-Default` cmdlet supports an `-OutVariable` parameter that lets you store its output into a variable:

```
PS > 1..3 | Out-Default -OutVariable myOutput
1
2
3

PS > $myOutput
1
2
3
```

Knowing this, we can use PowerShell's `$PSDefaultParameterValues` infrastructure to make `Out-Default` do this every time. The Solution uses two underscore characters as the variable name to look like the single underscore that represents the current pipeline input in PowerShell, but you can use any variable name you want:

```
PS > $PSDefaultParameterValues["Out-Default:OutVariable"] =
"lastOutput"

PS > 1..3
1
2
3

PS > $lastOutput
1
2
3
```

For more information about providing default values to cmdlet parameters, see Recipe 1.5.

See Also

Recipe 1.5

Recipe 2.7

Recipe 11.23

2.9 Capture and Redirect Binary Process Output

Problem

You want to run programs that transfer complex binary data between themselves.

Solution

Use the `Invoke-BinaryProcess` script to invoke the program, as shown in Example 2-9. If it is the source of binary data, use the `-RedirectOutput` parameter. If it consumes binary data, use the `-RedirectInput` parameter.

Example 2-9. Invoke-BinaryProcess.ps1

```
##############################################################################
######
##
## Invoke-BinaryProcess
```

```
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Invokes a process that emits or consumes binary data.

.EXAMPLE

PS > Invoke-BinaryProcess binaryProcess.exe -RedirectOutput -
ArgumentList "-Emit" |
       Invoke-BinaryProcess binaryProcess.exe -RedirectInput -
ArgumentList "-Consume"

#>

param(
    ## The name of the process to invoke
    [string] $ProcessName,

    ## Specifies that input to the process should be treated as
    ## binary
    [Alias("Input")]
    [switch] $RedirectInput,

    ## Specifies that the output of the process should be treated
    ## as binary
    [Alias("Output")]
    [switch] $RedirectOutput,

    ## Specifies the arguments for the process
    [string] $ArgumentList
)

Set-StrictMode -Version 3

## Prepare to invoke the process
$processStartInfo = New-Object System.Diagnostics.ProcessStartInfo
$processStartInfo.FileName = (Get-Command $processname).Definition
$processStartInfo.WorkingDirectory = (Get-Location).Path
if($argumentList) { $processStartInfo.Arguments = $argumentList }
$processStartInfo.UseShellExecute = $false

## Always redirect the input and output of the process.
```

```
## Sometimes we will capture it as binary, other times we will
## just treat it as strings.
$processStartInfo.RedirectStandardOutput = $true
$processStartInfo.RedirectStandardInput = $true

$process = [System.Diagnostics.Process]::Start($processStartInfo)

## If we've been asked to redirect the input, treat it as bytes.
## Otherwise, write any input to the process as strings.
if($redirectInput)
{
    $inputBytes = @($input)
    $process.StandardInput.BaseStream.Write($inputBytes, 0,
$inputBytes.Count)
    $process.StandardInput.Close()
}
else
{
    $input | % { $process.StandardInput.WriteLine($_) }
    $process.StandardInput.Close()
}

## If we've been asked to redirect the output, treat it as bytes.
## Otherwise, read any input from the process as strings.
if($redirectOutput)
{
    $byteRead = -1
    do
    {
        $byteRead = $process.StandardOutput.BaseStream.ReadByte()
        if($byteRead -ge 0) { $byteRead }
    } while($byteRead -ge 0)
}
else
{
    $process.StandardOutput.ReadToEnd()
}
```

Discussion

When PowerShell launches a native application, one of the benefits it provides is allowing you to use PowerShell commands to work with the output. For example:

```
  PS > (ipconfig)[7]
     Link-local IPv6 Address . . . . . : fe80::20f9:871:8365:f368%8
  PS > (ipconfig)[8]
     IPv4 Address. . . . . . . . . . . : 10.211.55.3
```

PowerShell enables this by splitting the output of the program on its newline characters, and then passing each line independently down the pipeline. This includes programs that use the Unix newline (`\n`) as well as the Windows newline (`\r\n`).

If the program outputs binary data, however, that reinterpretation can corrupt data as it gets redirected to another process or file. For example, some programs communicate between themselves through complicated binary data structures that cannot be modified along the way. This is common in some image editing utilities and other non-PowerShell tools designed for pipelined data manipulation.

We can see this through an example *BinaryProcess.exe* application that either emits binary data or consumes it. Here is the C# source code to the *BinaryProcess.exe* application:

```
using System;
using System.IO;

public class BinaryProcess
{
    public static void Main(string[] args)
    {
        if(args[0] == "-consume")
        {
            using(Stream inputStream = Console.OpenStandardInput())
            {
                for(byte counter = 0; counter < 255; counter++)
                {
                    byte received = (byte) inputStream.ReadByte();
                    if(received != counter)
                    {
                        Console.WriteLine(
                            "Got an invalid byte: {0}, expected {1}.",
                            received, counter);
                        return;
                    }
                    else
                    {
                        Console.WriteLine(
                            "Properly received byte: {0}.", received,
counter);
                    }
                }
```

```
            }
        }

        if(args[0] == "-emit")
        {
            using(Stream outputStream = Console.OpenStandardOutput())
            {
                for(byte counter = 0; counter < 255; counter++)
                {
                    outputStream.WriteByte(counter);
                }
            }
        }
    }
}
```

When we run it with the `-emit` parameter, PowerShell breaks the output into three objects:

```
PS > $output = .\binaryprocess.exe -emit
PS > $output.Count
3
```

We would expect this output to contain the numbers 0 through 254, but we see that it does not:

```
PS > $output | Foreach-Object { "------------";
    $_.ToCharArray() | Foreach-Object { [int] $_ } }
------------
0
1
2
3
4
5
6
7
8
9
------------
11
12
------------
14
15
16
17
```

```
18
19
20
21
22
(...)
255
214
220
162
163
165
8359
402
225
```

At number 10, PowerShell interprets that byte as the end of the line, and uses that to split the output into a new element. It does the same for number 13. Things appear to get even stranger when we get to the higher numbers and PowerShell starts to interpret combinations of bytes as Unicode characters from another language.

The Solution resolves this behavior by managing the output of the binary process directly. If you supply the `-Redirect+Input++ parameter, the script assumes an incoming stream of binary data and passes it to the program directly. If you supply the ++-Redirect+Output` parameter, the script assumes that the output is binary data, and likewise reads it from the process directly.

See Also

Recipe 1.2

Chapter 3. Variables and Objects

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please visit *https://www.powershellcookbook.com/4th_ed_techreview*. You can also reach out to the author at powershellcookbook@leeholmes.com.

3.0 Introduction

As touched on in Chapter 2, PowerShell makes life immensely easier by keeping information in its native form: *objects*. Users expend most of their effort in traditional shells just trying to resuscitate information that the shell converted from its native form to plain text. Tools have evolved that ease the burden of working with plain text, but that job is still significantly more difficult than it needs to be.

Since PowerShell builds on Microsoft's .NET Framework, native information comes in the form of .NET *objects*—packages of information and functionality closely related to that information.

Let's say that you want to get a list of running processes on your system. In other shells, your command (such as `tlist.exe` or `/bin/ps`) generates a plain-text report of the running processes on your system. To work with that output, you send it through a bevy of text processing tools—if you are lucky enough to have them available.

PowerShell's `Get-Process` cmdlet generates a list of the running processes on your system. In contrast to other shells, though, these are full-fidelity `System.+Diag+nostics.Process` objects straight out of the .NET

Framework. The .NET Framework documentation describes them as objects that "[provide] access to local and remote processes, and [enable] you to start and stop local system processes." With those objects in hand, PowerShell makes it trivial for you to access properties of objects (such as their process name or memory usage) and to access functionality on these objects (such as stopping them, starting them, or waiting for them to exit).

3.1 Display the Properties of an Item as a List

Problem

You have an item (for example, an error record, directory item, or .NET object), and you want to display detailed information about that object in a list format.

Solution

To display detailed information about an item, pass that item to the `Format-List` cmdlet. For example, to display an error in list format, type the following commands:

```
$currentError = $error[0]
$currentError | Format-List -Force
```

Discussion

Many commands by default display a summarized view of their output in a table format, for example, the `Get-Process` cmdlet:

```
PS > Get-Process PowerShell

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    920      10    43808      48424   183     4.69   1928 powershell
    149       6    18228       8660   146     0.48   1940 powershell
    431      11    33308      19072   172            2816 powershell
```

In most cases, the output actually contains a great deal more information. You can use the `Format-List` cmdlet to view it:

```
PS > Get-Process PowerShell | Format-List *

Name                       : pwsh
Id                         : 14820
PriorityClass              : Normal
FileVersion                : 7.1.0.0
HandleCount                : 940
TotalProcessorTime         : 00:00:25.7500000
VM                         : 2204249919488
WS                         : 81596416
Path                       : C:\Program
Files\WindowsApps\Microsoft.PowerShell_7.1.0.0_x64__8wekyb3d8bbwe\pwsh
.exe
CommandLine                :
C:\Users\lee\AppData\Local\Microsoft\WindowsApps\Microsoft.PowerShell_
8wekyb3d8bbwe\pwsh.e
                             xe
Parent                     : System.Diagnostics.Process
(WindowsTerminal)
Company                    : Microsoft Corporation
CPU                        : 25.765625
ProductVersion             : 7.1.0 SHA:
cbb7d40f684fdeb56cc276340b3b7435ac649d8f
Description                : pwsh
Product                    : PowerShell(...)
```

The `Format-List` cmdlet is one of the four PowerShell formatting cmdlets. These cmdlets are `Format-Table`, `Format-List`, `Format-Wide`, and `Format-Custom`. The `Format-List` cmdlet takes input and displays information about that input as a list.

By default, PowerShell takes the list of properties to display from the **.format.ps1xml* files in PowerShell's installation directory. In many situations, you'll only get a small set of the properties:

```
PS > Get-Process pwsh | Format-List

Id      : 2816
Handles : 431
CPU     :
Name    : pwsh

Id      : 5244
Handles : 665
CPU     : 10.296875
Name    : pwsh
```

To display all properties of the item, type **`Format-List *`**. If you type `Format-List *` but still do not get a list of the item's properties, then the item is defined in the **.format.ps1xml* files, but does not define anything to be displayed for the list command. In that case, type **`Format-List -Force`**.

One common stumbling block in PowerShell's formatting cmdlets comes from putting them in the middle of a script or pipeline:

```
PS > Get-Process PowerShell | Format-List | Sort 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.
```

Internally, PowerShell's formatting commands generate a new type of object: `Microsoft.PowerShell.Commands.Internal.Format.*`. When these objects make it to the end of the pipeline, PowerShell automatically sends them to an output cmdlet: by default, `Out-Default`. These `Out-*` cmdlets assume that the objects arrive in a certain order, so doing anything with the output of the formatting commands causes an error in the output system.

To resolve this problem, try to avoid calling the formatting cmdlets in the middle of a script or pipeline. When you do this, the output of your script no longer lends itself to the object-based manipulation so synonymous with PowerShell.

If you want to use the formatted output directly, send the output through the `Out-String` cmdlet as described in Recipe 1.24.

For more information about the `Format-List` cmdlet, type **`Get-Help Format-List`**.

3.2 Display the Properties of an Item as a Table

Problem

You have a set of items (for example, error records, directory items, or .NET objects), and you want to display summary information about them in a table

format.

Solution

To display summary information about a set of items, pass those items to the `Format-Table` cmdlet. This is the default type of formatting for sets of items in PowerShell and provides several useful features.

To use PowerShell's default formatting, pipe the output of a cmdlet (such as the `Get-Process` cmdlet) to the `Format-Table` cmdlet:

```
Get-Process | Format-Table
```

To display specific properties (such as `Name` and `WorkingSet`) in the table formatting, supply those property names as parameters to the `Format-Table` cmdlet:

```
Get-Process | Format-Table Name,WS
```

To instruct PowerShell to format the table in the most readable manner, supply the `-Auto` flag to the `Format-Table` cmdlet. PowerShell defines `WS` as an alias of the `WorkingSet` property for processes:

```
Get-Process | Format-Table Name,WS -Auto
```

To define a custom column definition (such as a process's `WorkingSet` in megabytes), supply a custom formatting expression to the `Format-Table` cmdlet:

```
$fields = "Name",@{
    Label = "WS (MB)"; Expression = {$_.WS / 1mb}; Align = "Right"}
Get-Process | Format-Table $fields -Auto
```

Discussion

The `Format-Table` cmdlet is one of the four PowerShell formatting cmdlets. These cmdlets are `Format-Table`, `Format-List`, `Format-Wide`, and `Format-Custom`. The `Format-Table` cmdlet takes input and displays

information about that input as a table. By default, PowerShell takes the list of properties to display from the **.format.ps1xml* files in PowerShell's installation directory. You can display all properties of the items if you type **`Format-Table *`**, although this is rarely a useful view.

The `-Auto` parameter to `Format-Table` is a helpful way to automatically format the table in the most readable way possible. It does come at a cost, however. To figure out the best table layout, PowerShell needs to examine each item in the incoming set of items. For small sets of items, this doesn't make much difference, but for large sets (such as a recursive directory listing) it does. Without the `-Auto` parameter, the `Format-Table` cmdlet can display items as soon as it receives them. With the `-Auto` flag, the cmdlet displays results only after it receives all the input.

Perhaps the most interesting feature of the `Format-Table` cmdlet is illustrated by the last example: the ability to define completely custom table columns. You define a custom table column similarly to the way that you define a custom column list. Rather than specify an existing property of the items, you provide a hashtable. That hashtable includes up to three keys: the column's label, a formatting expression, and alignment. The `Format-Table` cmdlet shows the label as the column header and uses your expression to generate data for that column. The label must be a string, the expression must be a script block, and the alignment must be either`"Left"`, `"Center"`, or `"Right"`. In the expression script block, the `$_` (or `$PSItem`) variable represents the current item being formatted.

> **NOTE**
>
> The `Select-Object` cmdlet supports a similar hashtable to add calculated properties, but uses `Name` (rather than `Label`) as the key to identify the property. After realizing how confusing this was, the PowerShell team updated both cmdlets to accept both `Name` and `Label`.

The expression shown in the last example takes the working set of the current item and divides it by 1 megabyte (1 MB).

One common stumbling block in PowerShell's formatting cmdlets comes from putting them in the middle of a script or pipeline:

putting them in the middle of a script or pipeline:

```
PS > Get-Process | Format-Table | Sort 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.
```

Internally, PowerShell's formatting commands generate a new type of object: `Microsoft.PowerShell.Commands.Internal.Format.*`. When these objects make it to the end of the pipeline, PowerShell then automatically sends them to an output cmdlet: by default, `Out-Default`. These `Out-*` cmdlets assume that the objects arrive in a certain order, so doing anything with the output of the formatting commands causes an error in the output system.

To resolve this problem, try to avoid calling the formatting cmdlets in the middle of a script or pipeline. When you do this, the output of your script no longer lends itself to the object-based manipulation so synonymous with PowerShell.

If you want to use the formatted output directly, send the output through the `Out-String` cmdlet as described in Recipe 1.24.

For more information about the `Format-Table` cmdlet, type **`Get-Help Format-Table`**. For more information about hashtables, see Recipe 7.13. For more information about script blocks, see Recipe 11.4.

See Also

Recipe 1.24

Recipe 7.13

Recipe 11.4

3.3 Store Information in Variables

Problem

You want to store the output of a pipeline or command for later use or to work with it in more detail.

Solution

To store output for later use, store the output of the command in a variable. You can access this information later, or even pass it down the pipeline as though it were the output of the original command:

```
PS > $result = 2 + 2
PS > $result
4

PS > $output = ipconfig
PS > $output | Select-String "Default Gateway" | Select -First 1

   Default Gateway . . . . . . . . . : 192.168.11.1

PS > $processes = Get-Process
PS > $processes.Count
85
PS > $processes | Where-Object { $_.ID -eq 0 }

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
      0       0        0         16     0               0 Idle
```

Discussion

Variables in PowerShell (and all other scripting and programming languages) let you store the output of something so that you can use it later. A variable name starts with a dollar sign ($) and can be followed by nearly any character. A small set of characters have special meaning to PowerShell, so PowerShell provides a way to make variable names that include even these.

For more information about the syntax and types of PowerShell variables, see Appendix A.

You can store the result of any pipeline or command in a variable to use it later. If that command generates simple data (such as a number or string), then the variable contains simple data. If the command generates rich data (such as the objects that represent system processes from the `Get-Process` cmdlet), then

the variable contains that list of rich data. If the command (such as a traditional executable) generates plain text (such as the output of traditional executable), then the variable contains plain text.

> **NOTE**
>
> If you've stored a large amount of data into a variable but no longer need that data, assign a new value (such as `$null`) to that variable. That will allow PowerShell to release the memory it was using to store that data.

In addition to variables that you create, PowerShell automatically defines several variables that represent things such as the location of your profile file, the process ID of PowerShell, and more. For a full list of these automatic variables, type **`Get-Help about_automatic_variables`**.

See Also

Appendix A

3.4 Access Environment Variables

Problem

You want to use an environment variable (such as the system path or the current user's name) in your script or interactive session.

Solution

PowerShell offers several ways to access environment variables.

To list all environment variables, list the children of the `env` drive:

```
Get-ChildItem env:
```

To get an environment variable using a more concise syntax, precede its name with +$+env:

```
$env:variablename
```

(For example, `$env:username`.)

To get an environment variable using its provider path, supply `env:` or `Environment::` to the `Get-ChildItem` cmdlet:

```
Get-ChildItem env:variablename
Get-ChildItem Environment::variablename
```

Discussion

PowerShell provides access to environment variables through its *environment provider*. Providers let you work with data stores (such as the registry, environment variables, and aliases) much as you would access the filesystem.

By default, PowerShell creates a drive (called `env`) that works with the *environment provider* to let you access environment variables. The environment provider lets you access items in the `env:` drive as you would any other drive: `dir env:\variablename` or `dir env:variablename`. If you want to access the provider directly (rather than go through its drive), you can also type `dir Environment::variablename`.

However, the most common (and easiest) way to work with environment variables is by typing `$env:variablename`. This works with any provider but is most typically used with environment variables.

This is because the environment provider shares something in common with several other providers—namely, support for the `*-Content` set of core cmdlets (see Example 3-1).

Example 3-1. Working with content on different providers

```
PS > "hello world" > test
PS > Get-Content test
hello world
PS > Get-Content c:test
hello world
PS > Get-Content variable:ErrorActionPreference
Continue
PS > Get-Content function:more
param([string[]]$paths)
$OutputEncoding = [System.Console]::OutputEncoding
```

```
if($paths)
{
    foreach ($file in $paths)
    {
        Get-Content $file | more.com
    }
}
else
{
    $input | more.com
}
PS > Get-Content env:systemroot
C:\WINDOWS
```

For providers that support the content cmdlets, PowerShell lets you interact with this content through a special variable syntax (see Example 3-2).

Example 3-2. Using PowerShell's special variable syntax to access content

```
PS > $function:more
param([string[]]$paths); if(($paths -ne $null) -and ($paths.length -ne
0)) { ...
       Get-Content $local:file | Out-Host -p } } else { $input | Out-
Host ...
PS > $variable:ErrorActionPreference
Continue
PS > $c:test
hello world
PS > $env:systemroot
C:\WINDOWS
```

This variable syntax for content management lets you both get and set content:

```
PS > $function:more = { $input | less.exe }
PS > $function:more
$input | less.exe
```

Now, when it comes to accessing complex provider paths using this method, you'll quickly run into naming issues (even if the underlying file exists):

```
PS > $c:\temp\test.txt
Unexpected token '\temp\test.txt' in expression or statement.
At line:1 char:17
+ $c:\temp\test.txt <<<<
```

The solution to that lies in PowerShell's escaping support for complex variable names. To define a complex variable name, enclose it in braces:

```
PS > ${1234123!@#$!@#$12$!@#$@!} = "Crazy Variable!"
PS > ${1234123!@#$!@#$12$!@#$@!}
Crazy Variable!
PS > dir variable:\1*

Name                                Value
----                                -----
1234123!@#$!@#$12$!@#$@!            Crazy Variable!
```

The following is the content equivalent (assuming that the file exists):

```
PS > ${c:\temp\test.txt}
hello world
```

Since environment variable names do not contain special characters, this `Get-Content` variable syntax is the best (and easiest) way to access environment variables.

For more information about working with PowerShell variables, see Appendix A. For more information about working with environment variables, type **`Get-Help About_Environment_Variable`**.

See Also

Appendix A

3.5 Program: Retain Changes to Environment Variables Set by a Batch File

When a batch file modifies an environment variable, *cmd.exe* retains this change even after the script exits. This often causes problems, as one batch file can accidentally pollute the environment of another. That said, batch file authors sometimes intentionally change the global environment to customize the path and other aspects of the environment to suit a specific task.

However, environment variables are private details of a process and disappear when that process exits. This makes the environment customization scripts mentioned earlier stop working when you run them from PowerShell—just as they fail to work when you run them from another *cmd.exe* (for example,

`cmd.exe /c MyEnvironmentCustomizer.cmd`).

The script in Example 3-3 lets you run batch files that modify the environment and retain their changes even after *cmd.exe* exits. It accomplishes this by storing the environment variables in a text file once the batch file completes, and then setting all those environment variables again in your PowerShell session.

To run this script, type `Invoke-CmdScript` *`Scriptname.cmd`* or `Invoke-CmdScript` *`Scriptname.bat`*—whichever extension the batch files uses.

> **NOTE**
>
> If this is the first time you've run a script in PowerShell, you will need to configure your Execution Policy. For more information about selecting an execution policy, see Recipe 18.1.

Notice that this script uses the full names for cmdlets: `Get-Content`, `Foreach-Object`, `Set-Content`, and `Remove-Item`. This makes the script readable and is ideal for scripts that somebody else will read. It is by no means required, though. For quick scripts and interactive use, shorter aliases (such as `gc`, `%`, `sc`, and `ri`) can make you more productive.

Example 3-3. Invoke-CmdScript.ps1

```
##############################################################################
######
##
## Invoke-CmdScript
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Invoke the specified batch file (and parameters), but also propagate any
environment variable changes back to the PowerShell environment that
called it.
```

```
.EXAMPLE

PS > type foo-that-sets-the-FOO-env-variable.cmd
@set FOO=%*
echo FOO set to %FOO%.

PS > $env:FOO
PS > Invoke-CmdScript "foo-that-sets-the-FOO-env-variable.cmd" Test

C:\Temp>echo FOO set to Test.
FOO set to Test.

PS > $env:FOO
Test

#>

param(
    ## The path to the script to run
    [Parameter(Mandatory = $true)]
    [string] $Path,

    ## The arguments to the script
    [string] $ArgumentList
)

Set-StrictMode -Version 3

$tempFile = [IO.Path]::GetTempFileName()

## Store the output of cmd.exe.  We also ask cmd.exe to output
## the environment table after the batch file completes
cmd /c " `"$Path`" $argumentList && set > `"$tempFile`" "

## Go through the environment variables in the temp file.
## For each of them, set the variable in our local environment.
Get-Content $tempFile | Foreach-Object {
    if($_ -match "^(.*?)=(.*)$")
    {
        Set-Content "env:\$($matches[1])" $matches[2]
    }
}

Remove-Item $tempFile
```

For more information about running scripts, see Recipe 1.2.

See Also

Recipe 1.2

Recipe 18.1

3.6 Control Access and Scope of Variables and Other Items

Problem

You want to control how you define (or interact with) the visibility of variables, aliases, functions, and drives.

Solution

PowerShell offers several ways to access variables.

To create a variable with a specific scope, supply that scope before the variable name:

```
$SCOPE:variable = value
```

To access a variable at a specific scope, supply that scope before the variable name:

```
$SCOPE:variable
```

To create a variable that remains even after the script exits, create it in the `GLOBAL` scope:

```
$GLOBAL:variable = value
```

To change a scriptwide variable from within a function, supply `SCRIPT` as its scope name:

```
$SCRIPT:variable = value
```

Discussion

PowerShell controls access to variables, functions, aliases, and drives through a mechanism known as *scoping*. The *scope* of an item is another term for its visibility. You are always in a scope (called the *current* or *local* scope), but some actions change what that means.

When your code enters a nested prompt, script, function, or script block, PowerShell creates a new scope. That scope then becomes the local scope. When it does this, PowerShell remembers the relationship between your old scope and your new scope. From the view of the new scope, the old scope is called the *parent scope*. From the view of the old scope, the new scope is called a *child scope*. Child scopes get access to all the variables in the parent scope, but changing those variables in the child scope doesn't change the version in the parent scope.

> **NOTE**
>
> Trying to change a scriptwide variable from a function is often a "gotcha" because a function is a new scope. As mentioned previously, changing something in a child scope (the function) doesn't affect the parent scope (the script). The rest of this discussion describes ways to change the value for the entire script.

When your code exits a nested prompt, script, function, or script block, the opposite happens. PowerShell removes the old scope, then changes the local scope to be the scope that originally created it—the parent of that old scope.

Some scopes are so common that PowerShell gives them special names:

Global

: The outermost scope. Items in the global scope are visible from all other scopes.

Script

: The scope that represents the current script. Items in the script scope are visible from all other scopes in the script.

Local

: The current scope.

When you define the scope of an item, PowerShell supports two additional scope names that act more like options: `Private` and `AllScope`. When you define an item to have a `Private` scope, PowerShell does not make that item directly available to child scopes. PowerShell does not *hide* it from child scopes, though, as child scopes can still use the `-Scope` parameter of the `Get-Variable` cmdlet to get variables from parent scopes. When you specify the `AllScope` option for an item (through one of the `*-Variable`, `*-Alias`, or `*-Drive` cmdlets), child scopes that change the item also affect the value in parent scopes.

With this background, PowerShell provides several ways for you to control access and scope of variables and other items.

Variables

To define a variable at a specific scope (or access a variable at a specific scope), use its scope name in the variable reference. For example:

```
$SCRIPT:myVariable = value
```

As illustrated in Appendix A, the `*-Variable` set of cmdlets also lets you specify scope names through their `-Scope` parameter.

Functions

To define a function at a specific scope (or access a function at a specific scope), use its scope name when creating the function. For example:

```
function GLOBAL:MyFunction { ... }
GLOBAL:MyFunction args
```

Aliases and drives

To define an alias or drive at a specific scope, use the `Option` parameter of the `*-Alias` and `*-Drive` cmdlets. To access an alias or drive at a specific scope, use the `Scope` parameter of the `*-Alias` and `*-Drive` cmdlets.

For more information about scopes, type **`Get-Help About-Scope`**.

See Also

Appendix A

3.7 Program: Create a Dynamic Variable

When working with variables and commands, some concepts feel too minor to deserve an entire new command or function, but the readability of your script suffers without them.

A few examples where this becomes evident are date math (*yesterday* becomes `(Get-Date).AddDays(-1)`) and deeply nested variables (*windowTitle* becomes `$host.UI.RawUI.WindowTitle`).

> **NOTE**
>
> There are innovative solutions on the Internet that use PowerShell's debugging facilities to create a breakpoint that changes a variable's value whenever you attempt to read from it. While unique, this solution causes PowerShell to think that any scripts that rely on the variable are in debugging mode. This, unfortunately, prevents PowerShell from enabling some important performance optimizations in those scripts.

Although we could write our own extensions to make these easier to access, `Get-Yesterday`, `Get-WindowTitle`, and `Set-WindowTitle` feel too insignificant to deserve their own commands.

PowerShell lets you define your own types of variables by extending its `PSVariable` class, but that functionality is largely designed for developer scenarios, and not for scripting scenarios. Example 3-4 resolves this quandary by using PowerShell classes to create a new variable type (`DynamicVariable`) that supports dynamic script actions when you get or set the variable's value.

Example 3-4. New-DynamicVariable.ps1

```
##############################################################################
######
##
## New-DynamicVariable
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######
```

```powershell
<#

.SYNOPSIS

Creates a variable that supports scripted actions for its getter and
setter

.EXAMPLE

PS > .\New-DynamicVariable GLOBAL:WindowTitle `
     -Getter { $host.UI.RawUI.WindowTitle } `
     -Setter { $host.UI.RawUI.WindowTitle = $args[0] }

PS > $windowTitle
Administrator: pwsh.exe
PS > $windowTitle = "Test"
PS > $windowTitle
Test

#>

using namespace System
using namespace System.Collections.ObjectModel
using namespace System.Management.Automation

param(
    ## The name for the dynamic variable
    [Parameter(Mandatory = $true)]
    $Name,

    ## The scriptblock to invoke when getting the value of the variable
    [Parameter(Mandatory = $true)]
    [ScriptBlock] $Getter,

    ## The scriptblock to invoke when setting the value of the variable
    [ScriptBlock] $Setter
)

Set-StrictMode -Version Latest

class DynamicVariable : PSVariable
{
    DynamicVariable(
        [string] $Name,
        [ScriptBlock] $ScriptGetter,
        [ScriptBlock] $ScriptSetter)
            : base($Name, $null, "AllScope")
    {
        $this.getter = $scriptGetter
```

```
        $this.setter = $scriptSetter
    }
    hidden [ScriptBlock] $getter;
    hidden [ScriptBlock] $setter;

    [Object] get_Value()
    {
        if($this.getter -ne $null)
        {
            $results = $this.getter.Invoke()
            if($results.Count -eq 1)
            {
                return $results[0];
            }
            else
            {
                $returnResults = New-Object 'PSObject[]' $results.Count
                $results.CopyTo($returnResults, 0)
                return $returnResults;
            }
        }
        else { return $null; }
    }

    [void] set_Value([Object] $Value)
    {
        if($this.setter -ne $null) { $this.setter.Invoke($Value); }
    }
}

## If we've already defined the variable, remove it.
if(Test-Path variable:\$name)
{
    Remove-Item variable:\$name -Force
}

## Set the new variable, along with its getter and setter.
$executioncontext.SessionState.PSVariable.Set(
    ([DynamicVariable]::New($name, $getter, $setter)))
```

3.8 Work with .NET Objects

Problem

You want to use and interact with one of the features that makes PowerShell so powerful: its intrinsic support for .NET objects.

Solution

PowerShell offers ways to access methods (both static and instance) and properties.

To call a static method on a class, place the type name in square brackets, and then separate the class name from the method name with two colons:

```
[ClassName]::MethodName(parameter list)
```

To call a method on an object, place a dot between the variable that represents that object and the method name:

```
$objectReference.MethodName(parameter list)
```

To access a static property on a class, place the type name in square brackets, and then separate the class name from the property name with two colons:

```
[ClassName]::PropertyName
```

To access a property on an object, place a dot between the variable that represents that object and the property name:

```
$objectReference.PropertyName
```

Discussion

One feature that gives PowerShell its incredible reach into both system administration and application development is its capability to leverage Microsoft's enormous and broad .NET Framework. The .NET Framework is a large collection of classes. Each class embodies a specific concept and groups closely related functionality and information. Working with the .NET Framework is one aspect of PowerShell that introduces a revolution to the world of management shells.

An example of a class from the .NET Framework is `System.Diagnostics.Process`—the grouping of functionality that "provides access to local and remote processes, and enables you to start and stop local system processes."

> **NOTE**
>
> The terms *type* and *class* are often used interchangeably.

Classes contain *methods* (which let you perform operations) and *properties* (which let you access information).

For example, the `Get-Process` cmdlet generates `System.Diagnostics.Process` objects, not a plain-text report like traditional shells. Managing these processes becomes incredibly easy, as they contain a rich mix of information (properties) and operations (methods). You no longer have to parse a stream of text for the ID of a process; you can just ask the object directly!

```
PS > $process = Get-Process Notepad
PS > $process.Id
3872
```

Static methods

```
[ClassName]::MethodName(parameter list)
```

Some methods apply only to the concept the class represents. For example, retrieving all running processes on a system relates to the general concept of processes instead of a specific process. Methods that apply to the class/type as a whole are called *static methods*.

For example:

```
PS > [System.Diagnostics.Process]::GetProcessById(0)
```

This specific task is better handled by the `Get-Process` cmdlet, but it demonstrates PowerShell's capability to call methods on .NET classes. It calls the static `GetProcessById` method on the `System.Diagnostics.Process` class to get the process with the ID of 0. This generates the following output:

```
Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName
```

```
------- ------ ----- ----- ----- ------ -- -----------
      0      0     0    16     0         0 Idle
```

Instance methods

```
$objectReference.MethodName(parameter list)
```

Some methods relate only to specific, tangible realizations (called instances) of a class. An example of this would be stopping a process actually running on the system, as opposed to the general concept of processes. If `$objectReference` refers to a specific `System.Diagnostics.Process` (as output by the `Get-Process` cmdlet, for example), you may call methods to start it, stop it, or wait for it to exit. Methods that act on instances of a class are called *instance methods*.

> **NOTE**
>
> The term *object* is often used interchangeably with the term *instance*.

For example:

```
PS > $process = Get-Process Notepad
PS > $process.WaitForExit()
```

stores the Notepad process into the `$process` variable. It then calls the `WaitForExit()` instance method on that specific process to pause PowerShell until the process exits. To learn about the different sets of parameters (overloads) that a given method supports, type that method name without any parameters:

```
PS > $now = Get-Date
PS > $now.ToString

OverloadDefinitions
-------------------
string ToString()
string ToString(string format)
string ToString(System.IFormatProvider provider)
string ToString(string format, System.IFormatProvider provider)
```

```
string IFormattable.ToString(string format, System.IFormatProvider
formatProvider)
string IConvertible.ToString(System.IFormatProvider provider)
```

For both static methods and instance methods, you may sometimes run into situations where PowerShell either generates an error or fails to invoke the method you expected. In this case, review the output of the `Trace-Command` cmdlet, with `MemberResolution` as the trace type (see Example 3-5).

Example 3-5. Investigating PowerShell's method resolution

```
PS > Trace-Command MemberResolution -PsHost {
    [System.Diagnostics.Process]::GetProcessById(0) }

DEBUG: MemberResolution Information: 0 : cache hit, Calling Method:
static
 System.Diagnostics.Process GetProcessById(int processId)
DEBUG: MemberResolution Information: 0 : Method argument conversion.
DEBUG: MemberResolution Information: 0 :     Converting parameter "0" to
"System.Int32".
DEBUG: MemberResolution Information: 0 : Checking for possible
references.

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
      0       0        0         12     0                0 Idle
```

If you are adapting a C# example from the Internet and PowerShell can't find a method used in the example, the method may have been added through a relatively rare technique called *explicit interface implementation*. If this is the case, you can cast the object to that interface before calling the method:

```
$sourceObject = 123
$result = ([IConvertible] $sourceObject).ToUint16($null)
```

Static properties

```
[ClassName]::PropertyName
```

or:

```
[ClassName]::PropertyName = value
```

Like static methods, some properties relate only to information about the concept

that the class represents. For example, the `System.DateTime` class "represents an instant in time, typically expressed as a date and time of day." It provides a `Now` static property that returns the current time:

```
PS > [System.DateTime]::Now
Saturday, June 2, 2010 4:57:20 PM
```

This specific task is better handled by the `Get-Date` cmdlet, but it demonstrates PowerShell's capability to access properties on .NET objects.

Although they are relatively rare, some types let you set the value of some static properties as well: for example, the `[System.Environment]::CurrentDirectory` property. This property represents the process's current directory—which represents PowerShell's startup directory, as opposed to the path you see in your prompt.

Instance properties

```
$objectReference.PropertyName
```

or:

```
$objectReference.PropertyName = value
```

Like instance methods, some properties relate only to specific, tangible realizations (called *instances*) of a class. An example of this would be the day of an actual instant in time, as opposed to the general concept of dates and times. If `$objectReference` refers to a specific `System.DateTime` (as output by the `Get-Date` cmdlet or `[System.DateTime]::Now`, for example), you may want to retrieve its day of week, day, or month. Properties that return information about instances of a class are called *instance properties*.

For example:

```
PS > $today = Get-Date
PS > $today.DayOfWeek
Saturday
```

This example stores the current date in the `$today` variable. It then calls the

`DayOfWeek` instance property to retrieve the day of the week for that specific date.

Dynamically accessing methods and properties

When you are working with a .NET type, you might have some advanced scenarios where you do not know a method or property name when you are writing your script, but do know it at runtime. Even in these rare situations, PowerShell still lets you access these members through dynamic member invocation. To access a property or method with a dynamic name, simply store that name in a variable and access it as you would any other method or property:

```
PS > $propertyName = "Length"
PS > "Hello World".$propertyName
11
PS > $methodName = "SubString"
PS > "Hello World".$methodName(6)
World
PS > $staticProperty = "OSVersion"
PS > [Environment]::$staticProperty

Platform ServicePack Version      VersionString
-------- ----------- -------      -------------
 Win32NT             10.0.19041.0 Microsoft Windows NT 10.0.19041.0
```

With this knowledge, the next questions are: “How do I learn about the functionality available in the .NET Framework?” and “How do I learn what an object does?”

For an answer to the first question, see Appendix F for a hand-picked list of the classes in the .NET Framework most useful to system administrators. For an answer to the second, see Recipe 3.12 and Recipe 3.13.

See Also

Recipe 3.12

Recipe 3.13

Appendix F

3.9 Create an Instance of a .NET Object

Problem

You want to create an instance of a .NET object to interact with its methods and properties.

Solution

Use the `New-Object` cmdlet to create an instance of an object.

To create an instance of an object using its default constructor, use the `New-Object` cmdlet with the class name as its only parameter:

```
PS > $generator = New-Object System.Random
PS > $generator.NextDouble()
0.853699042859347
```

To create an instance of an object that takes parameters for its constructor, supply those parameters to the `New-Object` cmdlet. In some instances, the class may exist in a separate library not loaded in PowerShell by default, such as the `System.Windows.Forms` assembly. In that case, you must first load the assembly that contains the class:

```
Add-Type -Assembly System.Windows.Forms
$image = New-Object System.Drawing.Bitmap source.gif
$image.Save("source_converted.jpg", "JPEG")
```

As an alternative to the `New-Object` cmdlet, you can also use PowerShell's `new()` method:

```
$image = [System.Drawing.Bitmap]::new("source.gif")
```

To create an object and use it at the same time (without saving it for later), wrap the call to `New-Object` in parentheses:

```
PS > (New-Object Net.WebClient).DownloadString("http://live.com")
```

If you plan to work with several classes from the same .NET namespace, the `using` statement can make your code easier to read and type:

```
using namespace System.Collections

$arrayList = New-Object ArrayList
$queue = [Queue]::new()
```

Discussion

Many cmdlets (such as `Get-Process` and `Get-ChildItem`) generate live .NET objects that represent tangible processes, files, and directories. However, PowerShell supports much more of the .NET Framework than just the objects that its cmdlets produce. These additional areas of the .NET Framework supply a huge amount of functionality that you can use in your scripts and general system administration tasks.

> **NOTE**
>
> To create an instance of a generic object, see Example 3-6.

When it comes to using most of these classes, the first step is often to create an instance of the class, store that instance in a variable, and then work with the methods and properties on that instance. To create an instance of a class, you use the `New-Object` cmdlet. The first parameter to the `New-Object` cmdlet is the type name, and the second parameter is the list of arguments to the constructor, if it takes any. The `New-Object` cmdlet supports PowerShell's *type shortcuts*, so you never have to use the fully qualified type name. For more information about type shortcuts, see Appendix A.

In addition to the `New-Object` cmdlet, you can also use the `new()` method that PowerShell supports as though it were a static method on that type: surround the type name with square brackets, add two colons, and then invoke the method with any parameters:

```
PS > [System.Drawing.Point]::new(10, 20)

IsEmpty  X  Y
-------  -  -
  False 10 20
```

Most objects support several different constructors that let you create objects in different ways. The official documentation on MSDN is usually the best place to get detailed information about these constructors, but PowerShell offers a handy shortcut by calling its `new()` method without parenthesis (like you would examine overloads of other methods):

```
PS > [System.Drawing.Point]::New

OverloadDefinitions
-------------------
System.Drawing.Point new(int x, int y)
System.Drawing.Point new(System.Drawing.Size sz)
System.Drawing.Point new(int dw)
```

A common pattern when working with .NET objects is to create them, set a few properties, and then use them. The `-Property` parameter of the `New-Object` cmdlet lets you combine these steps:

```
$startInfo = New-Object Diagnostics.ProcessStartInfo -Property @{
    'Filename' = "pwsh.exe";
    'WorkingDirectory' = $pshome;
    'Verb' = "RunAs"
}
[Diagnostics.Process]::Start($startInfo)
```

Or even more simply through PowerShell's built-in type conversion:

```
$startInfo = [Diagnostics.ProcessStartInfo] @{
    'Filename' = "pwsh.exe";
    'WorkingDirectory' = $pshome;
    'Verb' = "RunAs"
}
```

When calling the `New-Object` cmdlet directly, you might encounter difficulty when trying to specify a parameter that itself is a list. Assuming `$byte` is an array of bytes:

```
PS > [byte[]] $bytes = 1..10
PS > $memoryStream = New-Object System.IO.MemoryStream $bytes
New-Object : Cannot find an overload for ".ctor" and the argument
count: "10".
At line:1 char:27
+ $memoryStream = New-Object <<<< System.IO.MemoryStream $bytes
```

```
+ $memoryStream = New-Object <<<< System.IO.MemoryStream $bytes
```

To solve this, create the object using the `new()` keyword:

```
[System.IO.MemoryStream]::New($bytes)
```

The workarounds for New-Object are more complicated, but also work:

```
PS > $parameters = ,$bytes
PS > $memoryStream = New-Object System.IO.MemoryStream $parameters
```

or:

```
PS > $memoryStream = New-Object System.IO.MemoryStream @(,$bytes)
```

Load types from another assembly

PowerShell makes most common types available by default. However, many are available only after you load the library (called the assembly) that defines them. The MSDN documentation for a class includes the assembly that defines it. For more information about loading types from another assembly, please see Recipe 17.8.

For a hand-picked list of the classes in the .NET Framework most useful to system administrators, see Appendix F. To learn more about the functionality that a class supports, see Recipe 3.12.

For more information about the `New-Object` cmdlet, type **`Get-Help New-Object`**. For more information about the `Add-Type` cmdlet, type **`Get-Help Add-Type`**.

See Also

Recipe 3.8

Recipe 3.12

Recipe 17.8

Appendix F

Example 3-6

3.10 Create Instances of Generic Objects

When you work with the .NET Framework, you'll often run across classes that have the primary responsibility of managing other objects. For example, the `System.+Collections.+ArrayList` class lets you manage a dynamic list of objects. You can add objects to an `ArrayList`, remove objects from it, sort the objects inside, and more. These objects can be any type of object: `String` objects, integers, `DateTime` objects, and many others. However, working with classes that support arbitrary objects can sometimes be a little awkward. One example is *type safety*. If you accidentally add a `String` to a list of integers, you might not find out until your program fails.

Although the issue becomes largely moot when you're working only inside PowerShell, a more common complaint in strongly typed languages (such as C#) is that you have to remind the environment (through explicit casts) about the type of your object when you work with it again:

```
// This is C# code
System.Collections.ArrayList list =
    new System.Collections.ArrayList();
list.Add("Hello World");

string result = (String) list[0];
```

To address these problems, the .NET Framework includes a feature called *generic types*: classes that support arbitrary types of objects but let you specify *which type* of object. In this case, a collection of strings:

```
// This is C# code
System.Collections.ObjectModel.Collection<String> list =
    new System.Collections.ObjectModel.Collection<String>();
list.Add("Hello World");

string result = list[0];
```

PowerShell supports generic parameters by placing them between square brackets, as demonstrated in Example 3-6.

Example 3-6. Creating a generic object

```
PS > $coll = New-Object System.Collections.ObjectModel.Collection[Int]
PS > $coll.Add(15)
PS > $coll.Add("Test")
```

```
PS > $coll.Add("Test")
Cannot convert argument "0", with value: "Test", for "Add" to type
"System
.Int32": "Cannot convert value "Test" to type "System.Int32". Error:
"Input
string was not in a correct format.""
At line:1 char:10
+ $coll.Add <<<< ("Test")
    + CategoryInfo          : NotSpecified: (:) [], MethodException
    + FullyQualifiedErrorId :
MethodArgumentConversionInvalidCastArgument
```

For a generic type that takes two or more parameters, provide a comma-separated list of types, enclosed in quotes (see Example 3-7).

Example 3-7. Creating a multiparameter generic object

```
PS > $map = New-Object
"System.Collections.Generic.Dictionary[String,Int]"
PS > $map.Add("Test", 15)
PS > $map.Add("Test2", "Hello")
Cannot convert argument "1", with value: "Hello", for "Add" to type
"System
.Int32": "Cannot convert value "Hello" to type "System.Int32". Error:
"Input string was not in a correct format.""
At line:1 char:9
+ $map.Add <<<< ("Test2", "Hello")
    + CategoryInfo          : NotSpecified: (:) [], MethodException
    + FullyQualifiedErrorId :
MethodArgumentConversionInvalidCastArgument
```

3.11 Use a COM Object

Problem

You want to create a COM object to interact with its methods and properties.

Solution

Use the `New-Object` cmdlet (with the `-ComObject` parameter) to create a COM object from its *`ProgID`*. You can then interact with the methods and properties of the COM object as you would any other object in PowerShell.

```
$object = New-Object -ComObject ProgId
```

For example:

```
PS > $sapi = New-Object -Com Sapi.SpVoice
PS > $sapi.Speak("Hello World")
```

Discussion

Historically, many applications have exposed their scripting and administration interfaces as COM objects. While .NET APIs (and PowerShell cmdlets) are by far the most common, interacting with COM objects is still a routine administrative task.

As with classes in the .NET Framework, it is difficult to know what COM objects you can use to help you accomplish your system administration tasks. For a hand-picked list of the COM objects most useful to system administrators, see Appendix H.

For more information about the `New-Object` cmdlet, type **`Get-Help New-Object`**.

See Also

Appendix H

3.12 Learn About Types and Objects

Problem

You have an instance of an object and want to know what methods and properties it supports.

Solution

The most common way to explore the methods and properties supported by an object is through the `Get-Member` cmdlet.

To get the instance members of an object you've stored in the $*`object`* variable, pipe it to the `Get-Member` cmdlet:

```
$object | Get-Member
Get-Member -InputObject $object
```

To get the static members of an object you've stored in the *$object* variable, supply the `-Static` flag to the `Get-Member` cmdlet:

```
$object | Get-Member -Static
Get-Member -Static -InputObject $object
```

To get the static members of a specific type, pipe that type to the `Get-Member` cmdlet, and also specify the `-Static` flag:

```
[Type] | Get-Member -Static
Get-Member -InputObject [Type]
```

To get members of the specified member type (for example, `Method` or `Property`) from an object you have stored in the *$object* variable, supply that member type to the `-MemberType` parameter:

```
$object | Get-Member -MemberType MemberType
Get-Member -MemberType MemberType -InputObject $object
```

Discussion

The `Get-Member` cmdlet is one of the three commands you will use most commonly as you explore PowerShell. The other two commands are `Get-Command` and `Get-Help`.

> **NOTE**
>
> To interactively explore an object's methods and properties, see Recipe 1.26.

If you pass the `Get-Member` cmdlet a collection of objects (such as an `Array` or `ArrayList`) through the pipeline, PowerShell extracts each item from the collection and then passes them to the `Get-Member` cmdlet one by one. The `Get-Member` cmdlet then returns the members of each unique type that it receives. Although helpful the vast majority of the time, this sometimes causes

difficulty when you want to learn about the members or properties of the collection class itself.

If you want to see the properties of a collection (as opposed to the elements it contains), provide the collection to the `-InputObject` parameter instead. Alternatively, you can wrap the collection in an array (using PowerShell's *unary comma operator*) so that the collection class remains when the `Get-Member` cmdlet unravels the outer array:

```
PS > $files = Get-ChildItem
PS > ,$files | Get-Member

   TypeName: System.Object[]

Name                MemberType     Definition
----                ----------     ----------
Count               AliasProperty  Count = Length
Address             Method         System.Object& Address(Int32 )
(...)
```

For another way to learn detailed information about types and objects, see Recipe 3.13.

For more information about the `Get-Member` cmdlet, type **`Get-Help Get-Member`**.

See Also

Recipe 1.26

Recipe 3.13

3.13 Get Detailed Documentation About Types and Objects

Problem

You have a type of object and want to know detailed information about the methods and properties it supports.

Solution

The documentation for the .NET Framework [available here] is the best way to get detailed documentation about the methods and properties supported by an object. That exploration generally comes in two stages:

1. Find the type of the object.
   To determine the type of an object, you can either use the type name shown by the `Get-Member` cmdlet (as described in Recipe 3.12) or call the `GetType()` method of an object (if you have an instance of it):

   ```
   PS > $date = Get-Date
   PS > $date.GetType().ToString()
   System.DateTime
   ```

2. Enter that type name into the search box here.

Discussion

When the `Get-Member` cmdlet does not provide the information you need, the MSDN documentation for a type is a great alternative. It provides much more detailed information than the help offered by the `Get-Member` cmdlet—usually including detailed descriptions, related information, and even code samples. MSDN documentation focuses on developers using these types through a language such as C#, though, so you may find interpreting the information for use in PowerShell to be a little difficult at first.

Typically, the documentation for a class first starts with a general overview, and then provides a hyperlink to the members of the class—the list of methods and properties it supports.

> **NOTE**
>
> To get to the documentation for the members quickly, search for them more explicitly by adding the term "members" to your MSDN search term: "*`typename`* members."

Documentation for the members of a class lists the class's methods and properties, as does the output of the `Get-Member` cmdlet. The S icon

represents static methods and properties. Click the member name for more information about that method or property.

Public constructors

This section lists the constructors of the type. You use a constructor when you create the type through the `New-Object` cmdlet. When you click on a constructor, the documentation provides all the different ways that you can create that object, including the parameter list that you will use with the `New-Object` cmdlet.

Public fields/public properties

This section lists the names of the fields and properties of an object. The S icon represents a static field or property. When you click on a field or property, the documentation also provides the type returned by this field or property.

For example, you might see the following in the definition for `System.DateTime.Now`:

```
public static DateTime Now { get; }
```

`Public` means that the `Now` property is public—that you can access it from PowerShell. `Static` means that the property is static (as described in Recipe 3.8). `DateTime` means that the property returns a `DateTime` object when you call it. +get++;++ means that you can get information from this property but cannot set the information. Many properties support a +set`;` as well (such as the `IsReadOnly` property on `System.IO.FileInfo`), which means that you can change its value.

Public methods

This section lists the names of the methods of an object. The S icon represents a static method. When you click on a method, the documentation provides all the different ways that you can call that method, including the parameter list that you will use to call that method in PowerShell.

For example, you might see the following in the definition for `System.DateTime.AddDays()`:

```
C#
public DateTime AddDays (
    double value
)
```

`Public` means that the `AddDays` method is public—that you can access it from PowerShell. `DateTime` means that the method returns a `DateTime` object when you call it. The text `double value` means that this method requires a parameter (of type `double`). In this case, that parameter determines the number of days to add to the `DateTime` object on which you call the method.

See Also

Recipe 3.8

Recipe 3.12

3.14 Add Custom Methods and Properties to Objects

Problem

You have an object and want to add your own custom properties or methods (*members*) to that object.

Solution

Use the `Add-Member` cmdlet to add custom members to an object.

Discussion

The `Add-Member` cmdlet is extremely useful in helping you add custom members to individual objects. For example, imagine that you want to create a report from the files in the current directory, and that report should include each file's owner. The `Owner` property is not standard on the objects that `Get-ChildItem` produces, but you could write a small script to add them, as shown

in Example 3-8.

Example 3-8. A script that adds custom properties to its output of file objects

```
##############################################################################
######
##
## Get-OwnerReport
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Gets a list of files in the current directory, but with their owner
added
to the resulting objects.

.EXAMPLE

PS > Get-OwnerReport | Format-Table Name,LastWriteTime,Owner
Retrieves all files in the current directory, and displays the
Name, LastWriteTime, and Owner

#>

Set-StrictMode -Version 3

$files = Get-ChildItem
foreach($file in $files)
{
    $owner = (Get-Acl $file).Owner
    $file | Add-Member NoteProperty Owner $owner
    $file
}
```

For more information about running scripts, see Recipe 1.2.

The most common type of information to add to an object is static information in a `NoteProperty`. `Add-Member` even uses this as the default if you omit it:

```
PS > $item = Get-Item C:\
PS > $item | Add-Member VolumeName "Operating System"
PS > $item.VolumeName
Operating System
```

```
OperatingSystem
```

In addition to note properties, the `Add-Member` cmdlet supports several other property and method types, including `AliasProperty`, `ScriptProperty`, `CodeProperty`, `CodeMethod`, and `ScriptMethod`. For a more detailed description of these other property types, see Appendix A, as well as the help documentation for the `Add-Member` cmdlet.

> **NOTE**
>
> To create entirely new objects (instead of adding information to existing ones), see Recipe 3.15.

Although the `Add-Member` cmdlet lets you customize specific objects, it does not let you customize all objects of that type. For information on how to do that, see Recipe 3.16.

Calculated properties

Calculated properties are another useful way to add information to output objects. If your script or command uses a `Format-Table` or `Select-Object` command to generate its output, you can create additional properties by providing an expression that generates their value. For example:

```
Get-ChildItem |
  Select-Object Name,
      @{Name="Size (MB)"; Expression={ "{0,8:0.00}" -f ($_.Length /
1MB) } }
```

In this command, we get the list of files in the directory. We use the `Select-Object` command to retrieve its name and a calculated property called `Size (MB)`. This calculated property returns the size of the file in megabytes, rather than the default (bytes).

> **NOTE**
>
> The `Format-Table` cmdlet supports a similar hashtable to add calculated properties, but uses `Label` (rather than `Name`) as the key to identify the property. To eliminate the confusion

this produced, the PowerShell team updated the two cmdlets to accept both `Name` and `Label`.

For more information about the `Add-Member` cmdlet, type **`Get-Help Add-Member`**.

For more information about adding calculated properties, type **`Get-Help Select-Object`** or **`Get-Help Format-Table`**.

See Also

Recipe 1.2

Recipe 3.15

Recipe 3.16

Appendix A

3.15 Create and Initialize Custom Objects

Problem

You want to return structured results from a command so that users can easily sort, group, and filter them.

Solution

Use the `[PSCustomObject]` type cast to a new `PSCustomObject`, supplying a hashtable with the custom information as its value, as shown in Example 3-9.

Example 3-9. Creating a custom object

```
$output = [PSCustomObject] @{
    'User' = 'DOMAIN\User';
    'Quota' = 100MB;
    'ReportDate' = Get-Date;
}
```

If you want to create a custom object with associated functionality, write a PowerShell class in a module, and create an instance of that class:

```
using module c:\modules\PlottingObject.psm1

$obj = [PlottingObject]::new()
$obj.Move(10,10)
$obj.Points = SineWave
while($true) { $obj.Rotate(10); $obj.Draw(); Sleep -m 20 }
```

Discussion

When your script outputs information to the user, always prefer richly structured data over hand-formatted reports. By emitting custom objects, you give the end user as much control over your script's output as PowerShell gives you over the output of its own commands.

Despite the power afforded by the output of custom objects, user-written scripts have frequently continued to generate plain-text output. This can be partly blamed on PowerShell's previously cumbersome support for the creation and initialization of custom objects, as shown in Example 3-10.

Example 3-10. Creating a custom object in PowerShell version 1

```
$output = New-Object PsObject
Add-Member -InputObject $output NoteProperty User 'DOMAIN\user'
Add-Member -InputObject $output NoteProperty Quota 100MB
Add-Member -InputObject $output NoteProperty ReportDate (Get-Date)

$output
```

In PowerShell version 1, creating a custom object required creating a new object (of the type `PsObject`), and then calling the `Add-Member` cmdlet multiple times to add the desired properties. PowerShell version 2 made this immensely easier by adding the `-Property` parameter to the `New-Object` cmdlet, which applied to the `PSObject` type as well. PowerShell version 3 made this as simple as possible by directly supporting the `[PSCustomObject]` type cast.

While creating a `PSCustomObject` makes it easy to create data-centric objects (often called *property bags*), it does not let you add functionality to those objects. When you need functionality as well, the next step is to create a PowerShell class (see Example 3-11). Like many other languages, PowerShell classes support constructors, public properties, public methods, as well as internal helper variables and methods.

Example 3-11. Creating a module that exports a custom class

```
##############################################################################
######
##
## PlottingObject.psm1
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Demonstrates a module that implements a custom class

.EXAMPLE

function SineWave { -15..15 | % { ,($_,(10 * [Math]::Sin($_ / 3))) } }
function Box { -5..5 | % { ($_,-5),($_,5),(-5,$_),(5,$_) } }

using module PlottingObject

$obj = [PlottingObject]::New(@())
$obj.Points = Box
$obj.Move(10,10)
while($true) { $obj.Rotate(10); $obj.Draw(); Start-Sleep -m 20 }

$obj = [PlottingObject]::New((SineWave))
while($true) { $obj.Rotate(10); $obj.Draw(); Start-Sleep -m 20 }

#>

class PlottingObject
{
    ## Constructors - one with no arguments, and another that takes a
    ## set of initial points.
    PlottingObject()
    {
        $this.Points = @()
    }

    PlottingObject($initialPoints)
    {
        $this.Points = $initialPoints
    }

    ## An external property holding the points to plot
    $Points = @()
```

```
    ## Internal variables
    hidden $x = 0
    hidden $y = 0
    hidden $angle = 0
    hidden $xScale = -50,50
    hidden $yScale = -50,50
    hidden $windowWidth = [Console]::WindowWidth
    hidden $windowHeight = [Console]::WindowHeight

    ## A public method to rotate the points by a certain amount
    [void] Rotate([int] $angle)
    {
        $this.angle += $angle
    }

    ## A public method to move the points by a certain amount
    [void] Move([int] $xDelta, [int] $yDelta)
    {
        $this.x += $xDelta
        $this.y += $yDelta
    }

    ## A public method to draw the given points
    [void] Draw()
    {
        $degToRad = 180 * [Math]::Pi

        ## Go through each of the supplied points,
        ## move them the amount specified, and then rotate them
        ## by the angle specified
        $frame = foreach($point in $this.Points)
        {
            $pointX,$pointY = $point
            $pointX = $pointX + $this.x
            $pointY = $pointY + $this.y

            $newX = $pointX * [Math]::Cos($this.angle / $degToRad ) -
                $pointY * [Math]::Sin($this.angle / $degToRad )
            $newY = $pointY * [Math]::Cos($this.angle / $degToRad ) +
                $pointX * [Math]::Sin($this.angle / $degToRad )

            $this.PutPixel($newX, $newY, 'O')
        }

        ## Draw the origin
        $frame += $this.PutPixel(0, 0, '+')

        Clear-Host
        Write-Host "`e[?25l" -NoNewline
```

```
        Write-Host $frame -NoNewline
    }

    ## A helper function to draw a pixel on the screen
    hidden [string] PutPixel([int] $x, [int] $y, [char] $character)
    {
        $scaledX = ($x - $this.xScale[0]) / ($this.xScale[1] -
$this.xScale[0])
        $scaledX = [int] ($scaledX * $this.windowHeight * 2.38)

        $scaledY = (($y * 4 / 3) - $this.yScale[0]) / ($this.yScale[1] -
$this.yScale[0])
        $scaledY = [int] ($scaledY * $this.windowHeight)

        return "`e[$scaledY;${scaledX}H$character"
    }
}
```

For more information about creating modules, see Recipe 11.6. For more information about the syntax of PowerShell classes, see Appendix A.

See Also

Recipe 7.13

Recipe 11.6

Appendix A

3.16 Add Custom Methods and Properties to Types

Problem

You want to add your own custom properties or methods to all objects of a certain type.

Solution

Use the `Update-TypeData` cmdlet to add custom members to all objects of a type.

```
Update-TypeData -TypeName AddressRecord `
    -MemberType AliasProperty -Membername Cell -Value Phone
```

Alternatively, use custom type extension files.

Discussion

Although the `Add-Member` cmdlet is extremely useful in helping you add custom members to individual objects, it requires that you add the members to each object that you want to interact with. It does not let you automatically add them to all objects of that type. For that purpose, PowerShell supports another mechanism—*custom type extensions*.

The simplest and most common way to add members to all instances of a type is through the `Update-TypeData` cmdlet. This cmdlet supports aliases, notes, script methods, and more:

```
$r = [PSCustomObject] @{
    Name = "Lee";
    Phone = "555-1212";
    SSN = "123-12-1212"
}
$r.PSTypeNames.Add("AddressRecord")
Update-TypeData -TypeName AddressRecord `
    -MemberType AliasProperty -Membername Cell -Value Phone
```

Custom type extensions let you easily add your own features to any type exposed by the system. If you write code (for example, a script or function) that primarily interacts with a single type of object, then that code might be better suited as an extension to the type instead.

For example, imagine a script that returns the free disk space on a given drive. That might be helpful as a script, but instead you might find it easier to make PowerShell's `PSDrive` objects themselves tell you how much free space they have left.

In addition to the `Update-TypeData` approach, PowerShell supports type extensions through XML-based type extension files. Since type extension files are XML files, make sure that your customizations properly encode the characters that have special meaning in XML files, such as <, >, and &.

For more information about the features supported by these formatting XML files, type **`Get-Help about_format.ps1xml`**.

Getting started

If you haven't done so already, the first step in creating a type extension file is to create an empty one. The best location for this is probably in the same directory as your custom profile, with the filename *Types.Custom.ps1xml*, as shown in Example 3-12.

Example 3-12. Sample Types.Custom.ps1xml file

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
</Types>
```

Next, add a few lines to your PowerShell profile so that PowerShell loads your type extensions during startup:

```
$typeFile = (Join-Path (Split-Path $profile) "Types.Custom.ps1xml")
Update-TypeData -PrependPath $typeFile
```

By default, PowerShell loads several type extensions from the *Types.ps1xml* file in PowerShell's installation directory. The `Update-TypeData` cmdlet tells PowerShell to also look in your *Types.Custom.ps1xml* file for extensions. The `-PrependPath` parameter makes PowerShell favor your extensions over the built-in ones in case of conflict.

Once you have a custom types file to work with, adding functionality becomes relatively straightforward. As a theme, these examples do exactly what we alluded to earlier: add functionality to PowerShell's `PSDrive` type.

> **NOTE**
>
> PowerShell does this automatically. Type **`Get-PSDrive`** to see the result.

To support this, you need to extend your custom types file so that it defines additions to the `System.Management.Automation.PSDriveInfo` type, shown in Example 3-13.

`System.Management.Automation.PSDriveInfo` is the type that the `Get-PSDrive` cmdlet generates.

Example 3-13. A template for changes to a custom types file

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
  <Type>
    <Name>System.Management.Automation.PSDriveInfo</Name>
    <Members>
        add members such as <ScriptProperty> here
    <Members>
  </Type>
</Types>
```

Add a ScriptProperty

A `ScriptProperty` lets you add properties (that get and set information) to types, using PowerShell script as the extension language. It consists of three child elements: the `Name` of the property, the *getter* of the property (via the `GetScriptBlock` child), and the *setter* of the property (via the `SetScriptBlock` child).

In both the `GetScriptBlock` and `SetScriptBlock` sections, the `$this` variable refers to the current object being extended. In the `SetScriptBlock` section, the `$args[0]` variable represents the value that the user supplied as the righthand side of the assignment.

Example 3-14 adds an `AvailableFreeSpace` `ScriptProperty` to `PSDriveInfo`, and should be placed within the members section of the template given in Example 3-13. When you access the property, it returns the amount of free space remaining on the drive. When you set the property, it outputs what changes you must make to obtain that amount of free space.

Example 3-14. A ScriptProperty for the PSDriveInfo type

```
<ScriptProperty>
  <Name>AvailableFreeSpace</Name>
  <GetScriptBlock>
    ## Ensure that this is a FileSystem drive
    if($this.Provider.ImplementingType -eq
       [Microsoft.PowerShell.Commands.FileSystemProvider])
    {
       ## Also ensure that it is a local drive
       $driveRoot = $this.Root
       $fileZone = [System.Security.Policy.Zone]::CreateFromUrl(`
```

```
                $driveRoot).SecurityZone
        if($fileZone -eq "MyComputer")
        {
           $drive = New-Object System.IO.DriveInfo $driveRoot
           $drive.AvailableFreeSpace
        }
     }
  </GetScriptBlock>
  <SetScriptBlock>
   ## Get the available free space
   $availableFreeSpace = $this.AvailableFreeSpace

   ## Find out the difference between what is available, and what they
   ## asked for.
   $spaceDifference = (([long] $args[0]) - $availableFreeSpace) / 1MB

   ## If they want more free space than they have, give that message
   if($spaceDifference -gt 0)
   {
       $message = "To obtain $args bytes of free space, " +
          " free $spaceDifference megabytes."
       Write-Host $message
    }
   ## If they want less free space than they have, give that message
   else
   {
       $spaceDifference = $spaceDifference * -1
       $message = "To obtain $args bytes of free space, " +
           " use up $spaceDifference more megabytes."
        Write-Host $message
    }
  </SetScriptBlock>
</ScriptProperty>
```

Add an AliasProperty

An `AliasProperty` gives an alternative name (alias) for a property. The referenced property does not need to exist when PowerShell processes your type extension file, since you (or another script) might later add the property through mechanisms such as the `Add-Member` cmdlet.

Example 3-15 adds a `Free` `AliasProperty` to `PSDriveInfo`, and it should also be placed within the members section of the template given in Example 3-13. When you access the property, it returns the value of the `AvailableFreeSpace` property. When you set the property, it sets the value of the `AvailableFreeSpace` property.

Example 3-15. An AliasProperty for the PSDriveInfo type

```
<AliasProperty>
  <Name>Free</Name>
  <ReferencedMemberName>AvailableFreeSpace</ReferencedMemberName>
</AliasProperty>
```

Add a ScriptMethod

A `ScriptMethod` lets you define an action on an object, using PowerShell script as the extension language. It consists of two child elements: the `Name` of the property and the `Script`.

In the script element, the `$this` variable refers to the current object you are extending. Like a standalone script, the `$args` variable represents the arguments to the method. Unlike standalone scripts, `ScriptMethod`s do not support the `param` statement for parameters.

Example 3-16 adds a `Remove ScriptMethod` to `PSDriveInfo`. Like the other additions, place these customizations within the members section of the template given in Example 3-13. When you call this method with no arguments, the method simulates removing the drive (through the `-WhatIf` option to `Remove-PSDrive`). If you call this method with `$true` as the first argument, it actually removes the drive from the PowerShell session.

Example 3-16. A ScriptMethod for the PSDriveInfo type

```
<ScriptMethod>
  <Name>Remove</Name>
  <Script>
    $force = [bool] $args[0]
    ## Remove the drive if they use $true as the first parameter
    if($force)
    {
       $this | Remove-PSDrive
    }
    ## Otherwise, simulate the drive removal
    else
    {
       $this | Remove-PSDrive -WhatIf
    }
  </Script>
</ScriptMethod>
```

Add other extension points

PowerShell supports several additional features in the types extension file, including `CodeProperty`, `NoteProperty`,`CodeMethod`, and `MemberSet`. Although not generally useful to end users, developers of PowerShell providers and cmdlets will find these features helpful. For more information about these additional features, see the PowerShell SDK or the MSDN documentation.

3.17 Define Custom Formatting for a Type

Problem

You want to emit custom objects from a script and have them formatted in a specific way.

Solution

Use a custom format extension file to define the formatting for that type, followed by a call to the `Update-FormatData` cmdlet to load them into your session:

```
$formatFile = Join-Path (Split-Path $profile) "Format.Custom.Ps1Xml"
Update-FormatData -PrependPath $typesFile
```

If a file-based approach is not an option, use the `Formats` property of the `[Runspace]::DefaultRunspace.InitialSessionState` type to add new formatting definitions for the custom type.

Discussion

When PowerShell commands produce output, this output comes in the form of richly structured objects rather than basic streams of text. These richly structured objects stop being of any use once they make it to the screen, though, so PowerShell guides them through one last stage before showing them on screen: *formatting* and *output*.

The formatting and output system is based on the concept of *views*. Views can take several forms: table views, list views, complex views, and more. The most

common view type is a table view. This is the form you see when you use `Format-Table` in a command, or when an object has four or fewer properties.

As with the custom type extensions described in Recipe 3.16, PowerShell supports both file-based and in-memory updates of type formatting definitions.

The simplest and most common way to define formatting for a type is through the `Update-FormatData` cmdlet, as shown in the Solution. The `Update-FormatData` cmdlet takes paths to *Format.ps1xml* files as input. There are many examples of formatting definitions in the PowerShell installation directory that you can use. To create your own formatting customizations, use these files as a source of examples, but do not modify them directly. Instead, create a new file and use the `Update-FormatData` cmdlet to load your customizations.

For more information about the features supported by these formatting XML files, type **`Get-Help about_format.ps1xml`**.

In addition to file-based formatting, PowerShell makes it possible (although not easy) to create formatting definitions from scratch. Example 3-17 provides a script to simplify this process.

Example 3-17. Add-FormatData.ps1

```
##############################################################################
######
##
## Add-FormatData
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
######

<#

.SYNOPSIS

Adds a table formatting definition for the specified type name.

.EXAMPLE

PS > $r = [PSCustomObject] @{
    Name = "Lee";
    Phone = "555-1212";
    SSN = "123-12-1212"
```

```
}
PS > $r.PSTypeNames.Add("AddressRecord")
PS > Add-FormatData -TypeName AddressRecord -TableColumns Name, Phone
PS > $r

Name Phone
---- -----
Lee  555-1212

#>

param(
    ## The type name (or PSTypeName) that the table definition should
    ## apply to.
    $TypeName,

    ## The columns to be displayed by default
    [string[]] $TableColumns
)

Set-StrictMode -Version 3

## Define the columns within a table control row
$rowDefinition = New-Object Management.Automation.TableControlRow

## Create left-aligned columns for each provided column name
foreach($column in $TableColumns)
{
    $rowDefinition.Columns.Add(
        (New-Object Management.Automation.TableControlColumn "Left",
            (New-Object Management.Automation.DisplayEntry
$column,"Property")))
}

$tableControl = New-Object Management.Automation.TableControl
$tableControl.Rows.Add($rowDefinition)

## And then assign the table control to a new format view,
## which we then add to an extended type definition. Define this view
for the
## supplied custom type name.
$formatViewDefinition = New-Object
Management.Automation.FormatViewDefinition "TableView",$tableControl
$extendedTypeDefinition = New-Object
Management.Automation.ExtendedTypeDefinition $TypeName
$extendedTypeDefinition.FormatViewDefinition.Add($formatViewDefinition)

## Add the definition to the session, and refresh the format data
[Runspace]::DefaultRunspace.InitialSessionState.Formats.Add($extendedTyp
eDefinition)
```

```
Update-FormatData
```

Chapter 4. Looping and Flow Control

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please visit *https://www.powershellcookbook.com/4th_ed_techreview*. You can also reach out to the author at powershellcookbook@leeholmes.com.

4.0 Introduction

As you begin to write scripts or commands that interact with unknown data, the concepts of looping and flow control become increasingly important.

PowerShell's looping statements and commands let you perform an operation (or set of operations) without having to repeat the commands themselves. This includes, for example, doing something a specified number of times, processing each item in a collection, or working until a certain condition comes to pass.

PowerShell's flow control and comparison statements let you adapt your script or command to unknown data. They let you execute commands based on the value of that data, skip commands based on the value of that data, and more.

Together, looping and flow control statements add significant versatility to your PowerShell toolbox.

4.1 Make Decisions with Comparison and Logical Operators

Problem

You want to compare some data with other data and make a decision based on that comparison.

Solution

Use PowerShell's logical operators to compare pieces of data and make decisions based on them.

Comparison operators

`-eq`, `-ne`, `-ge`, `-gt`, `-in`, `-notin`, `-lt`, `-le`, `-like`, `-notlike`, `-match`, `-notmatch`, `-contains`, `-notcontains`, `-is`, `-isnot`

Logical operators

`-and`, `-or`, `-xor`, `-not`

For a detailed description (and examples) of these operators, see Appendix A.

Discussion

PowerShell's logical and comparison operators let you compare pieces of data or test data for some condition. An operator either compares two pieces of data (a *binary* operator) or tests one piece of data (a *unary* operator). All comparison operators are binary operators (they compare two pieces of data), as are most of the logical operators. The only unary logical operator is the `-not` operator, which returns the `true/false` opposite of the data that it tests.

Comparison operators compare two pieces of data and return a result that depends on the specific comparison operator. For example, you might want to check whether a collection has at least a certain number of elements:

```
PS > (dir).Count -ge 4
True
```

or check whether a string matches a given regular expression:

```
PS > "Hello World" -match "H.*World"
True
```

Most comparison operators also adapt to the type of their input. For example, when you apply them to simple data such as a string, the `-like` and `-match` comparison operators determine whether the string matches the specified pattern. When you apply them to a collection of simple data, those same comparison operators return all elements in that collection that match the pattern you provide.

> **NOTE**
>
> The `-match` operator takes a regular expression as its argument. One of the more common regular expression symbols is the `$` character, which represents the end of line. The `$` character also represents the start of a PowerShell variable, though! To prevent PowerShell from interpreting characters as language terms or escape sequences, place the string in single quotes rather than double quotes:
>
> ```
> PS > "Hello World" -match "Hello"
> True
> PS > "Hello World" -match 'Hello$'
> False
> ```

By default, PowerShell's comparison operators are case-insensitive. To use the case-sensitive versions, prefix them with the character `c`:

```
-ceq, -cne, -cge, -cgt, -cin, -clt, -cle, -clike, -cnotlike,
-cmatch, -cnotmatch, -ccontains, -cnotcontains
```

For a detailed description of the comparison operators, their case-sensitive counterparts, and how they adapt to their input, see Appendix A.

Logical operators combine `true` or `false` statements and return a result that depends on the specific logical operator. For example, you might want to check whether a string matches the wildcard pattern you supply *and* that it is longer than a certain number of characters:

```
PS > $data = "Hello World"
PS > ($data -like "*llo W*") -and ($data.Length -gt 10)
True
PS > ($data -like "*llo W*") -and ($data.Length -gt 20)
False
```

Some of the comparison operators actually incorporate aspects of the logical operators. Since using the opposite of a comparison (such as `-like`) is so common, PowerShell provides comparison operators (such as `-notlike`) that save you from having to use the `-not` operator explicitly.

For a detailed description of the individual logical operators, see Appendix A.

Comparison operators and logical operators (when combined with flow control statements) form the core of how we write a script or command that adapts to its data and input.

See also Appendix A for detailed information about these statements.

For more information about PowerShell's operators, type **`Get-Help About_Operators`**.

See Also

Appendix A

4.2 Adjust Script Flow Using Conditional Statements

Problem

You want to control the conditions under which PowerShell executes commands or portions of your script.

Solution

Use PowerShell's `if`, `elseif`, and `else` conditional statements to control the flow of execution in your script.

For example:

```
$temperature = 90

if($temperature -le 0)
{
```

```
    "Balmy Canadian Summer"
}
elseif($temperature -le 32)
{
    "Freezing"
}
elseif($temperature -le 50)
{
    "Cold"
}
elseif($temperature -le 70)
{
    "Warm"
}
else
{
    "Hot"
}
```

Discussion

Conditional statements include the following:

`if` *statement*

: Executes the script block that follows it if its *condition* evaluates to `true`

`elseif` *statement*

: Executes the script block that follows it if its *condition* evaluates to `true` and none of the conditions in the `if` or `elseif` statements before it evaluate to `true`

`else` *statement*

: Executes the script block that follows it if none of the conditions in the `if` or `elseif` statements before it evaluate to `true`

In addition to being useful for script control flow, conditional statements are often a useful way to assign data to a variable. PowerShell makes this very easy by letting you assign the results of a conditional statement directly to a variable:

```
$result = if(Get-Process -Name notepad) { "Running" } else { "Not
running" }
```

For very simple conditional statements such as this, you can also use

PowerShell's ternary operator:

```
$result = (Get-Process -Name notepad*) ? "Running" : "Not running"
```

For more information about these flow control statements, type **`Get-Help About_Flow_Control`**.

4.3 Manage Large Conditional Statements with Switches

Problem

You want to find an easier or more compact way to represent a large `if` … `elseif` … `else` conditional statement.

Solution

Use PowerShell's `switch` statement to more easily represent a large `if` … `elseif` … `else` conditional statement.

For example:

```
$temperature = 20

switch($temperature)
{
   { $_ -lt 32 }   { "Below Freezing"; break }
   32              { "Exactly Freezing"; break }
   { $_ -le 50 }   { "Cold"; break }
   { $_ -le 70 }   { "Warm"; break }
   default         { "Hot" }
}
```

Discussion

PowerShell's `switch` statement lets you easily test its input against a large number of comparisons. The `switch` statement supports several options that allow you to configure how PowerShell compares the input against the conditions—such as with a wildcard, regular expression, or even an arbitrary

script block. Since scanning through the text in a file is such a common task, PowerShell's `switch` statement supports that directly. These additions make PowerShell `switch` statements a great deal more powerful than those in C and C++.

As another example of the `switch` statement in action, consider how to determine the SKU of the current operating system. For example, is the script running on Windows 7 Ultimate? Windows Server Cluster Edition? The `Get-CimInstance` cmdlet lets you determine the operating system SKU, but unfortunately returns its result as a simple number. A `switch` statement lets you map these numbers to their English equivalents based on the official documentation listed at this site:

```
##############################################################################
########
##
## Get-OperatingSystemSku
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
##
##############################################################################
########

<#

.SYNOPSIS

Gets the sku information for the current operating system

.EXAMPLE

PS > Get-OperatingSystemSku
Professional with Media Center

#>

param($Sku =
    (Get-CimInstance Win32_OperatingSystem).OperatingSystemSku)

Set-StrictMode -Version 3

switch ($Sku)
{
    0   { "An unknown product"; break; }
```

```
    1   { "Ultimate"; break; }
    2   { "Home Basic"; break; }
    3   { "Home Premium"; break; }
    4   { "Enterprise"; break; }
    5   { "Home Basic N"; break; }
    6   { "Business"; break; }
    7   { "Server Standard"; break; }
    8   { "Server Datacenter (full installation)"; break; }
    9   { "Windows Small Business Server"; break; }
    10  { "Server Enterprise (full installation)"; break; }
    11  { "Starter"; break; }
    12  { "Server Datacenter (core installation)"; break; }
    13  { "Server Standard (core installation)"; break; }
    14  { "Server Enterprise (core installation)"; break; }
    15  { "Server Enterprise for Itanium-based Systems"; break; }
    16  { "Business N"; break; }
    17  { "Web Server (full installation)"; break; }
    18  { "HPC Edition"; break; }
    19  { "Windows Storage Server 2008 R2 Essentials"; break; }
    20  { "Storage Server Express"; break; }
    21  { "Storage Server Standard"; break; }
    22  { "Storage Server Workgroup"; break; }
    23  { "Storage Server Enterprise"; break; }
    24  { "Windows Server 2008 for Windows Essential Server
Solutions"; break; }
    25  { "Small Business Server Premium"; break; }
    26  { "Home Premium N"; break; }
    27  { "Enterprise N"; break; }
    28  { "Ultimate N"; break; }
    29  { "Web Server (core installation)"; break; }
    30  { "Windows Essential Business Server Management Server";
break; }
    31  { "Windows Essential Business Server Security Server"; break;
}
    32  { "Windows Essential Business Server Messaging Server"; break;
}
    33  { "Server Foundation"; break; }
    34  { "Windows Home Server 2011"; break; }
    35  { "Windows Server 2008 without Hyper-V for Windows Essential
Server Solutions"; break; }
    36  { "Server Standard without Hyper-V"; break; }
    37  { "Server Datacenter without Hyper-V (full installation)";
break; }
    38  { "Server Enterprise without Hyper-V (full installation)";
break; }
    39  { "Server Datacenter without Hyper-V (core installation)";
break; }
    40  { "Server Standard without Hyper-V (core installation)";
break; }
    41  { "Server Enterprise without Hyper-V (core installation)";
```

```
break; }
    42  { "Microsoft Hyper-V Server"; break; }
    43  { "Storage Server Express (core installation)"; break; }
    44  { "Storage Server Standard (core installation)"; break; }
    45  { "Storage Server Workgroup (core installation)"; break; }
    46  { "Storage Server Enterprise (core installation)"; break; }
    46  { "Storage Server Enterprise (core installation)"; break; }
    47  { "Starter N"; break; }
    48  { "Professional"; break; }
    49  { "Professional N"; break; }
    50  { "Windows Small Business Server 2011 Essentials"; break; }
    51  { "Server For SB Solutions"; break; }
    52  { "Server Solutions Premium"; break; }
    53  { "Server Solutions Premium (core installation)"; break; }
    54  { "Server For SB Solutions EM"; break; }
    55  { "Server For SB Solutions EM"; break; }
    56  { "Windows MultiPoint Server"; break; }
    59  { "Windows Essential Server Solution Management"; break; }
    60  { "Windows Essential Server Solution Additional"; break; }
    61  { "Windows Essential Server Solution Management SVC"; break; }
    62  { "Windows Essential Server Solution Additional SVC"; break; }
    63  { "Small Business Server Premium (core installation)"; break;
}
    64  { "Server Hyper Core V"; break; }
    72  { "Server Enterprise (evaluation installation)"; break; }
    76  { "Windows MultiPoint Server Standard (full installation)";
break; }
    77  { "Windows MultiPoint Server Premium (full installation)";
break; }
    79  { "Server Standard (evaluation installation)"; break; }
    80  { "Server Datacenter (evaluation installation)"; break; }
    84  { "Enterprise N (evaluation installation)"; break; }
    95  { "Storage Server Workgroup (evaluation installation)"; break;
}
    96  { "Storage Server Standard (evaluation installation)"; break;
}
    98  { "Windows 8 N"; break; }
    99  { "Windows 8 China"; break; }
    100 { "Windows 8 Single Language"; break; }
    101 { "Windows 8"; break; }
    103 { "Professional with Media Center"; break; }

    default {"UNKNOWN: " + $SKU }
}
```

Although used as a way to express large conditional statements more cleanly, a `switch` statement operates much like a large sequence of `if` statements, as opposed to a large sequence of `if` … `elseif` … `elseif` … `else`

statements. Given the input that you provide, PowerShell evaluates that input against *each* of the comparisons in the `switch` statement. If the comparison evaluates to `true`, PowerShell then executes the script block that follows it. Unless that script block contains a `break` statement, PowerShell continues to evaluate the following comparisons.

For more information about PowerShell's `switch` statement, see Appendix A or type **`Get-Help About_Switch`**.

See Also

Appendix A

4.4 Repeat Operations with Loops

Problem

You want to execute the same block of code more than once.

Solution

Use one of PowerShell's looping statements (`for`, `foreach`, `while`, and `do`) or PowerShell's `Foreach-Object` cmdlet to run a command or script block more than once. For a detailed description of these looping statements, see Appendix A. For example:

`for` loop

```
for($counter = 1; $counter -le 10; $counter++)
{
  "Loop number $counter"
}
```

`foreach` loop

```
foreach($file in dir)
{
  "File length: " + $file.Length
}
```

`Foreach-Object` cmdlet

```
Get-ChildItem | Foreach-Object { "File length: " + $_.Length }
```

`while` loop

```
$response = ""
while($response -ne "QUIT")
{
  $response = Read-Host "Type something"
}
```

`do..while` loop

```
$response = ""
do
{
  $response = Read-Host "Type something"
} while($response -ne "QUIT")
```

`do..until` loop

```
$response = ""
do
{
  $response = Read-Host "Type something"
} until($response -eq "QUIT")
```

Discussion

Although any of the looping statements can be written to be functionally equivalent to any of the others, each lends itself to certain problems.

You usually use a `for` loop when you need to perform an operation an exact number of times. Because using it this way is so common, it is often called a *counted* `for` *loop*.

You usually use a `foreach` loop when you have a collection of objects and want to visit each item in that collection. If you do not yet have that entire collection in memory (as in the `dir` collection from the `foreach` example shown earlier), the `Foreach-Object` cmdlet is usually a more efficient alternative.

Unlike the `foreach` loop, the `Foreach-Object` cmdlet lets you process each element in the collection *as PowerShell generates it*. This is an important distinction; asking PowerShell to collect the entire output of a large command (such as `Get-Content` *`hugefile.txt`*) in a `foreach` loop can easily drag down your system.

Like pipeline-oriented functions, the `Foreach-Object` cmdlet lets you define commands to execute before the looping begins, during the looping, and after the looping completes:

```
PS > "a","b","c" | Foreach-Object `
    -Begin { "Starting"; $counter = 0 } `
    -Process { "Processing $_"; $counter++ } `
    -End { "Finishing: $counter" }

Starting
Processing a
Processing b
Processing c
Finishing: 3
```

> **TIP**
>
> To invoke multiple operations in your loop at the same time, use the -parallel switch of `Foreach-Object`. For more information, see Recipe 4.5.

The `while` and `do..while` loops are similar, in that they continue to execute the loop as long as its condition evaluates to `true`. A `while` loop checks for this before running your script block, whereas a `do..while` loop checks the condition after running your script block. A `do..until` loop is exactly like a `do..while` loop, except that it exits when its condition returns `$true`, rather than when its condition returns `$false`.

For a detailed description of these looping statements, see Appendix A or type **`Get-Help About_For`**, **`Get-Help About_Foreach`**, **`Get-Help about_While`**, or **`Get-Help about_Do`**.

See Also

Appendix A

Recipe 4.5

4.5 Process Time-Consuming Action in Parallel

Problem

You have a set of data or actions that you want to run at the same time.

Solution

Use the `-parallel` switch of the `Foreach-Object` cmdlet:

```
PS > Measure-Command { 1..5 | Foreach-Object { Start-Sleep -Seconds 5
} }

(...)
TotalSeconds      : 25.0247856
(...)

PS > Measure-Command { 1..5 | Foreach-Object -parallel { Start-Sleep -
Seconds 5 } }

(...)
TotalSeconds      : 5.1354752
(...)
```

Discussion

There are times in PowerShell when you can significantly speed up a long-running operation by running parts of it at the same time. Perfect opportunities for this are scenarios where your script spends most of its time waiting on network resources (such as downloading files or web pages), or slow operations (such as restarting a series of slow services).

In these scenarios, you can use the `-parallel` parameter of `Foreach-Object` to perform these actions at the same time. Under the covers, PowerShell uses background jobs to run each branch. It caps the number of branches running at the same time to whatever you specify in the `-ThrottleLimit` parameter, with a default of `5`.

> **NOTE**
>
> If the reason you want multiple commands in parallel is to accomplish some task quickly across a large set of machines, you should instead use `Invoke-Command`. For more information, see Recipe 29.5.

Since PowerShell runs these branches as background jobs, you need to use either the `$USING` syntax to bring outside variables into this background job (PowerShell brings `$_` by default), or provide the variables in the `-ArgumentList` parameter. For example:

```
PS > $greeting = "World"
PS > 1..5 | Foreach-Object -parallel { "Hello $greeting" }
Hello
Hello
Hello
Hello
Hello

PS > 1..5 | Foreach-Object -parallel { "Hello $USING:greeting" }
Hello World
Hello World
Hello World
Hello World
Hello World
```

PowerShell runs these background jobs in your main PowerShell process, so you

PowerShell runs these background jobs in your main PowerShell process, so you can act on input as live instances:

```
$processes = 1..10 | Foreach-Object { Start-Process notepad -PassThru
}
$processes | Foreach-Object -parallel { $_.Kill() }
```

If you need the branches of your parallel loop to communicate back to your main shell, the recommended approach is to accomplish this through script block output and then have your main shell process the results. It is tempting to do this with live objects, but beware that the path is treacherous and difficult. Let's take a simple example - running a parallel operation to increment a counter.

It might initially seem like you should use:

```
$counter = 0
1..10 | Foreach-Object -parallel {
    $myCounter = $USING:counter
    $myCounter = $myCounter + 1
}
```

However, when you type `$counter = $counter + 1` in PowerShell, PowerShell updates the `$counter` variable in the current scope. If you want to change an object from a background job, you need to do so by setting a property on a live object rather than trying to replace the object. Fortunately, PowerShell has a type called `[ref]` for this kind of scenario:

```
$counter = [ref] 0
1..10 | Foreach-Object -parallel {
    $myCounter = $USING:counter
    $myCounter.Value = $myCounter.Value + 1
}
```

Initially, this seems to work:

```
PS > $counter

Value
-----
   10
```

Now that we're proud of ourselves, let's really do this in parallel:

```
PS > $counter = [ref] 0
PS > 1..10000 | Foreach-Object -throttlimit 100 -parallel {
>>     $myCounter = $USING:counter
>>     $myCounter.Value = $myCounter.Value + 1
>> }
PS > $counter

Value
-----
 9992
```

Oops! Because we've done this with massive parallelism, `$myCounter.Value` can change at any time during the parts of the pipeline where PowerShell runs `$myCounter.Value = $myCounter.Value + 1`. This is called a *race condition*, and is common to any language that lets code from multiple simultaneous blocks of code run at the same time. To get rid of the weird intermediate states, we have to use the Interlocked Increment class from the .Net framework:

```
$counter = [ref] 0
1..10000 | Foreach-Object -throttlimit 100 -parallel {
    $myCounter = $USING:counter
    $null = [Threading.Interlocked]::Increment($myCounter)
}
```

Which correctly gives us:

```
PS > $counter

Value
-----
10000
```

These problems are gnarly, and bite even professional programmers with regularity. The best practice to handle this class of issue is to avoid the area altogether by not processing or operating on shared state.

See Also

Recipe 4.4

Recipe 29.5

4.6 Add a Pause or Delay

Problem

You want to pause or delay your script or command.

Solution

To pause until the user presses the Enter key, use the `pause` command :

```
PS > pause
Press Enter to continue...:
```

To pause until the user presses any key, use the `ReadKey()` method on the `$host` object:

```
PS > $host.UI.RawUI.ReadKey()
```

To pause a script for a given amount of time, use the `Start-Sleep` cmdlet:

```
PS > Start-Sleep 5
PS > Start-Sleep -Milliseconds 300
```

Discussion

When you want to pause your script until the user presses a key or for a set amount of time, `pause` and `Start-Sleep` are the two cmdlets you are most likely to use.

> **NOTE**
>
> If you want to retrieve user input rather than just pause, the `Read-Host` cmdlet lets you read input from the user. For more information, see Recipe 13.1.

In other situations, you may sometimes want to write a loop in your script that runs at a constant speed—such as once per minute or 30 times per second. That is typically a difficult task, as the commands in the loop might take up a significant amount of time, or even an inconsistent amount of time.

significant amount of time, or even an inconsistent amount of time.

In the past, many computer games suffered from solving this problem incorrectly. To control their game speed, game developers added commands to slow down their game. For example, after much tweaking and fiddling, the developers might realize that the game plays correctly on a typical machine if they make the computer count to 1 million every time it updates the screen. Unfortunately, the speed of these commands (such as counting) depends heavily on the speed of the computer. Since a fast computer can count to 1 million much more quickly than a slow computer, the game ends up running much more quickly (often to the point of incomprehensibility) on faster computers!

To make your loop run at a regular speed, you can measure how long the commands in a loop take to complete, and then delay for whatever time is left, as shown in Example 4-1.

Example 4-1. Running a loop at a constant speed

```
$loopDelayMilliseconds = 650
while($true)
{
   $startTime = Get-Date

   ## Do commands here
   "Executing"

   $endTime = Get-Date
   $loopLength = ($endTime - $startTime).TotalMilliseconds
   $timeRemaining = $loopDelayMilliseconds - $loopLength

   if($timeRemaining -gt 0)
   {
      Start-Sleep -Milliseconds $timeRemaining
   }
}
```

For more information about the `Start-Sleep` cmdlet, type **`Get-Help Start-Sleep`**.

See Also

Recipe 13.1

Chapter 5. Strings and Unstructured Text

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please visit *https://www.powershellcookbook.com/4th_ed_techreview*. You can also reach out to the author at powershellcookbook@leeholmes.com.

5.0 Introduction

Creating and manipulating text has long been one of the primary tasks of scripting languages and traditional shells. In fact, Perl (the language) started as a simple (but useful) tool designed for text processing. It has grown well beyond those humble roots, but its popularity provides strong evidence of the need it fills.

In text-based shells, this strong focus continues. When most of your interaction with the system happens by manipulating the text-based output of programs, powerful text processing utilities become crucial. These text parsing tools, such as `awk`, `sed`, and `grep`, form the keystones of text-based systems management.

In PowerShell's object-based environment, this traditional tool chain plays a less critical role. You can accomplish most of the tasks that previously required these tools much more effectively through other PowerShell commands. However, being an object-based shell does not mean that PowerShell drops all support for text processing. Dealing with strings and unstructured text continues to play an important part in a system administrator's life. Since PowerShell lets you manage the majority of your system in its full fidelity (using cmdlets and

objects), the text processing tools can once again focus primarily on actual text processing tasks.

5.1 Create a String

Problem

You want to create a variable that holds text.

Solution

Use PowerShell string variables as a way to store and work with text.

To define a string that supports variable expansion and escape characters in its definition, surround it with double quotes:

```
$myString = "Hello World"
```

To define a literal string (one that does not interpret variable expansion or escape characters), surround it with single quotes:

```
$myString = 'Hello World'
```

Discussion

String literals come in two varieties: *literal* (*nonexpanding*) and *expanding* strings. To create a literal string, place single quotes (`$myString = 'Hello World'`) around the text. To create an expanding string, place double quotes (`$myString = "Hello World"`) around the text.

In a literal string, all the text between the single quotes becomes part of your string. In an expanding string, PowerShell expands variable names (such as `$replacementString`) and escape sequences (such as `` `n ``) with their values (such as the content of `$replacementString` and the newline character, respectively).

For a detailed explanation of the escape sequences and replacement rules inside PowerShell strings, see Appendix A.

One exception to the “all text in a literal string is literal” rule comes from the quote characters themselves. In either type of string, PowerShell lets you place two of that string’s quote characters together to add the quote character itself:

```
$myString = "This string includes ""double quotes"" because it
combined quote
characters."
$myString = 'This string includes ''single quotes'' because it
combined quote
characters.'
```

This helps prevent escaping atrocities that would arise when you try to include a single quote in a single-quoted string. For example:

```
$myString = 'This string includes ' + "'" + 'single quotes' + "'"
```

> **NOTE**
>
> This example shows how easy PowerShell makes it to create new strings by adding other strings together. This is an attractive way to build a formatted report in a script but should be used with caution. Because of the way that the .NET Framework (and therefore PowerShell) manages strings, adding information to the end of a large string this way causes noticeable performance problems. If you intend to create large reports, see Recipe 5.16.

See Also

Recipe 5.16

Appendix A

5.2 Create a Multiline or Formatted String

Problem

You want to create a variable that holds text with newlines or other explicit formatting.

Solution

Use a PowerShell *here string* to store and work with text that includes newlines and other formatting information.

```
$myString = @"
This is the first line
of a very long string. A "here string"
lets you create blocks of text
that span several lines.
"@
```

Discussion

PowerShell begins a here string when it sees the characters `@"` followed by a newline. It ends the string when it sees the characters `"@` on their own line. These seemingly odd restrictions let you create strings that include quote characters, newlines, and other symbols that you commonly use when you create large blocks of preformatted text.

> **NOTE**
>
> These restrictions, while useful, can sometimes cause problems when you copy and paste PowerShell examples from the Internet. Web pages often add spaces at the end of lines, which can interfere with the strict requirements of the beginning of a here string. If PowerShell produces an error when your script defines a here string, check that the here string does not include an errant space after its first quote character.

Like string literals, here strings may be literal (and use single quotes) or expanding (and use double quotes).

5.3 Place Special Characters in a String

Problem

You want to place special characters (such as tab and newline) in a string variable.

Solution

In an expanding string, use PowerShell's escape sequences to include special characters such as tab and newline.

```
PS > $myString = "Report for Today`n---------------"
PS > $myString
Report for Today
---------------
```

Discussion

As discussed in Recipe 5.1, PowerShell strings come in two varieties: literal (or nonexpanding) and expanding strings. A literal string uses single quotes around its text, whereas an expanding string uses double quotes around its text.

In a literal string, all the text between the single quotes becomes part of your string. In an expanding string, PowerShell expands variable names (such as `$ENV:SystemRoot`) and escape sequences (such as `` `n``) with their values (such as the `SystemRoot` environment variable and the newline character).

> **NOTE**
>
> Unlike many languages that use a backslash character (\) for escape sequences, PowerShell uses a backtick (`) character. This stems from its focus on system administration, where backslashes are ubiquitous in pathnames.

For a detailed explanation of the escape sequences and replacement rules inside PowerShell strings, see Appendix A.

See Also

Recipe 5.1

Appendix A

5.4 Insert Dynamic Information in a String

Problem

You want to place dynamic information (such as the value of another variable) in a string.

Solution

In an expanding string, include the name of a variable in the string to insert the value of that variable:

```
PS > $header = "Report for Today"
PS > $myString = "$header`n----------------"
PS > $myString
Report for Today
----------------
```

To include information more complex than just the value of a variable, enclose it in a subexpression:

```
PS > $header = "Report for Today"
PS > $myString = "$header`n$('-' * $header.Length)"
PS > $myString
Report for Today
----------------
```

Discussion

Variable substitution in an expanding string is a simple enough concept, but subexpressions deserve a little clarification.

A *subexpression* is the dollar sign character, followed by a PowerShell command (or set of commands) contained in parentheses:

```
$(subexpression)
```

When PowerShell sees a subexpression in an expanding string, it evaluates the subexpression and places the result in the expanding string. In the Solution, the expression `'-' * $header.Length` tells PowerShell to make a line of dashes `$header.Length` long.

Another way to place dynamic information inside a string is to use PowerShell's string formatting operator, which uses the same rules that .NET string formatting does:

```
PS > $header = "Report for Today"
PS > $myString = "{0}`n{1}" -f $header,('-' * $header.Length)
PS > $myString
Report for Today
----------------
```

For an explanation of PowerShell's formatting operator, see Recipe 5.6. For more information about PowerShell's escape characters, type **`Get-Help About_Escape_Characters`** or type **`Get-Help About_Special_Characters`**.

See Also

Recipe 5.6

5.5 Prevent a String from Including Dynamic Information

Problem

You want to prevent PowerShell from interpreting special characters or variable names inside a string.

Solution

Use a nonexpanding string to have PowerShell interpret your string exactly as entered. A nonexpanding string uses the single quote character around its text.

```
PS > $myString = 'Useful PowerShell characters include: $, `, " and {
}'
PS > $myString
Useful PowerShell characters include: $, `, " and { }
```

If you want to include newline characters as well, use a nonexpanding *here string*, as in Example 5-1.

Example 5-1. A nonexpanding here string that includes newline characters

```
PS > $myString = @'
Tip of the Day
--------------
```

```
-------------
Useful PowerShell characters include: $, `, ', " and { }
'@

PS > $myString
Tip of the Day
Useful PowerShell characters include: $, `, ', " and { }
```

Discussion

In a literal string, all the text between the single quotes becomes part of your string. This is in contrast to an expanding string, where PowerShell expands variable names (such as `$myString`) and escape sequences (such as `` `n ``) with their values (such as the content of `$myString` and the newline character).

> **NOTE**
>
> Nonexpanding strings are a useful way to manage files and folders containing special characters that might otherwise be interpreted as escape sequences. For more information about managing files with special characters in their name, see Recipe 20.7.

As discussed in Recipe 5.1, one exception to the "all text in a literal string is literal" rule comes from the quote characters themselves. In either type of string, PowerShell lets you place two of that string's quote characters together to include the quote character itself:

```
$myString = "This string includes ""double quotes"" because it
combined quote
characters."
$myString = 'This string includes ''single quotes'' because it
combined quote
characters.'
```

See Also

Recipe 5.1

Recipe 20.7

5.6 Place Formatted Information in a String

Problem

You want to place formatted information (such as right-aligned text or numbers rounded to a specific number of decimal places) in a string.

Solution

Use PowerShell's formatting operator to place formatted information inside a string:

```
PS > $formatString = "{0,8:D4} {1:C}`n"
PS > $report = "Quantity Price`n"
PS > $report += "---------------`n"
PS > $report += $formatString -f 50,2.5677
PS > $report += $formatString -f 3,9
PS > $report
Quantity Price
---------------
    0050 $2.57
    0003 $9.00
```

Discussion

PowerShell's string formatting operator (`-f`) uses the same string formatting rules as the `String.Format()` method in the .NET Framework. It takes a format string on its left side and the items you want to format on its right side.

In the Solution, you format two numbers: a quantity and a price. The first number (`{0}`) represents the quantity and is right-aligned in a box of eight characters (`,8`). It is formatted as a decimal number with four digits (`:D4`). The second number (`{1}`) represents the price, which you format as currency (`:C`).

> **NOTE**
>
> If you find yourself hand-crafting text-based reports, STOP! Let PowerShell's built-in commands do all the work for you. Instead, emit custom objects so that your users can work with your script as easily as they work with regular PowerShell commands. For more information, see Recipe 3.15.

For a detailed explanation of PowerShell's formatting operator, see Appendix A.

For a detailed list of the formatting rules, see Appendix D.

Although primarily used to control the layout of information, the string-formatting operator is also a readable replacement for what is normally accomplished with string concatenation:

```
PS > $number1 = 10
PS > $number2 = 32
PS > "$number2 divided by $number1 is " + $number2 / $number1
32 divided by 10 is 3.2
```

The string formatting operator makes this much easier to read:

```
PS > "{0} divided by {1} is {2}" -f $number2, $number1, ($number2 /
$number1)
32 divided by 10 is 3.2
```

If you want to support named replacements (rather than index-based replacements), you can use the `Format-String` script given in Recipe 5.17.

In addition to the string formatting operator, PowerShell provides three formatting commands (`Format-Table`, `Format-Wide`, and `Format-List`) that let you easily generate formatted reports. For detailed information about those cmdlets, see Appendix A.

See Also

Recipe 3.15

Appendix A

Appendix D

5.7 Search a String for Text or a Pattern

Problem

You want to determine whether a string contains another string, or you want to find the position of a string within another string.

Solution

PowerShell provides several options to help you search a string for text.

Use the `-like` operator to determine whether a string matches a given DOS-like wildcard:

```
PS > "Hello World" -like "*llo W*"
True
```

Use the `-match` operator to determine whether a string matches a given regular expression:

```
PS > "Hello World" -match '.*l[l-z]o W.*$'
True
```

Use the `Contains()` method to determine whether a string contains a specific string:

```
PS > "Hello World".Contains("World")
True
```

Use the `IndexOf()` method to determine the location of one string within another:

```
PS > "Hello World".IndexOf("World")
6
```

Discussion

Since PowerShell strings are fully featured .NET objects, they support many string-oriented operations directly. The `Contains()` and `IndexOf()` methods are two examples of the many features that the `String` class supports. To learn what other functionality the `String` class supports, see Recipe 3.12.

> **NOTE**
>
> To search entire files for text or a pattern, see Recipe 9.4.

Although they use similar characters, simple wildcards and regular expressions serve significantly different purposes. Wildcards are much simpler than regular expressions, and because of that, more constrained. While you can summarize the rules for wildcards in just four bullet points, entire books have been written to help teach and illuminate the use of regular expressions.

> **NOTE**
>
> A common use of regular expressions is to search for a string that spans multiple lines. By default, regular expressions do not search across lines, but you can use the *singleline* (`?s`) option to instruct them to do so:
>
> ```
> PS > "Hello `n World" -match "Hello.*World"
> False
> PS > "Hello `n World" -match "(?s)Hello.*World"
> True
> ```

Wildcards lend themselves to simple text searches, whereas regular expressions lend themselves to more complex text searches.

For a detailed description of the `-like` operator, see Appendix A. For a detailed description of the `-match` operator, see Appendix A. For a detailed list of the regular expression rules and syntax, see Appendix B.

One difficulty sometimes arises when you try to store the result of a PowerShell command in a string, as shown in Example 5-2.

Example 5-2. Attempting to store output of a PowerShell command in a string

```
PS > Get-Help Get-ChildItem

NAME
    Get-ChildItem

SYNOPSIS
    Gets the items and child items in one or more specified locations.

(...)

PS > $helpContent = Get-Help Get-ChildItem
PS > $helpContent -match "location"
False
```

The `-match` operator searches a string for the pattern you specify but seems to fail in this case. This is because all PowerShell commands generate objects. If you don't store that output in another variable or pass it to another command, PowerShell converts the output to a text representation before it displays it to you. In Example 5-2, `$helpContent` is a fully featured object, not just its string representation:

```
PS > $helpContent.Name
Get-ChildItem
```

To work with the text-based representation of a PowerShell command, you can explicitly send it through the `Out-String` cmdlet. The `Out-String` cmdlet converts its input into the text-based form you are used to seeing on the screen:

```
PS > $helpContent = Get-Help Get-ChildItem | Out-String -Stream
PS > $helpContent -match "location"
True
```

For a script that makes searching textual command output easier, see Recipe 1.24.

See Also

Recipe 1.24

Recipe 3.12

Appendix A

Appendix B

5.8 Replace Text in a String

Problem

You want to replace a portion of a string with another string.

Solution

PowerShell provides several options to help you replace text in a string with other text.

Use the `Replace()` method on the string itself to perform simple replacements:

```
PS > "Hello World".Replace("World", "PowerShell")
Hello PowerShell
```

Use PowerShell's regular expression `-replace` operator to perform more advanced regular expression replacements:

```
PS > "Hello World" -replace '(.*) (.*)','$2 $1'
World Hello
```

Discussion

The `Replace()` method and the `-replace` operator both provide useful ways to replace text in a string. The `Replace()` method is the quickest but also the most constrained. It replaces every occurrence of the exact string you specify with the exact replacement string that you provide. The `-replace` operator provides much more flexibility because its arguments are regular expressions that can match and replace complex patterns.

> **NOTE**
>
> For an approach that uses input and output examples to learn automatically how to replace text in a string, see Recipe 5.14.

Given the power of the regular expressions it uses, the `-replace` operator carries with it some pitfalls of regular expressions as well.

First, the regular expressions that you use with the `-replace` operator often contain characters (such as the dollar sign, which represents a group number) that PowerShell normally interprets as variable names or escape characters. To prevent PowerShell from interpreting these characters, use a nonexpanding string (single quotes) as shown in the Solution.

Another, less common pitfall is wanting to use characters that have special meaning to regular expressions as part of your replacement text. For example:

```
PS > "Power[Shell]" -replace "[Shell]","ful"
Powfulr[fulfulfulfulful]
```

That's clearly not what we intended. In regular expressions, square brackets around a set of characters means "match any of the characters inside of the square brackets." In our example, this translates to "Replace the characters S, h, e, and l with 'ful'."

To avoid this, we can use the regular expression escape character to escape the square brackets:

```
PS > "Power[Shell]" -replace "\[Shell\]","ful"
Powerful
```

However, this means knowing all of the regular expression special characters and modifying the input string. Sometimes we don't control that, so the `[Regex]::Escape()` method comes in handy:

```
PS > "Power[Shell]" -replace ([Regex]::Escape("[Shell]")),"ful"
Powerful
```

For extremely advanced regular expression replacement needs, you can use a script block to accomplish your replacement tasks, as described in Recipe 32.6. For example, to capitalize the first character (`\w`) after a word boundary (`\b`):

```
PS > "hello world" -replace '\b(\w)',{ $_.Value.ToUpper() }
Hello World
```

For more information about the `-replace` operator, see Appendix A and Appendix B.

See Also

Recipe 5.14

Appendix A

Appendix B

5.9 Split a String on Text or a Pattern

Problem

You want to split a string based on some literal text or a regular expression pattern.

Solution

Use PowerShell's `-split` operator to split on a sequence of characters or specific string:

```
PS > "a-b-c-d-e-f" -split "-c-"
a-b
d-e-f
```

To split on a pattern, supply a regular expression as the first argument:

```
PS > "a-b-c-d-e-f" -split "b|[d-e]"
a-
-c-
-
-f
```

Discussion

To split a string, many beginning scripters already comfortable with C# use the `String.Split()` and `[Regex]::Split()` methods from the .NET Framework. While still available in PowerShell, PowerShell's `-split` operator provides a more natural way to split a string into smaller strings. When used with no arguments (the *unary* split operator), it splits a string on whitespace characters, as in Example 5-3.

Example 5-3. PowerShell's unary split operator

```
PS > -split "Hello World `t How `n are you?"
Hello
World
How
are
```

you?

When used with an argument, it treats the argument as a regular expression and then splits based on that pattern.

```
PS > "a-b-c-d-e-f" -split 'b|[d-e]'
a-
-c-
-
-f
```

If the replacement pattern avoids characters that have special meaning in a regular expression, you can use it to split a string based on another string.

```
PS > "a-b-c-d-e-f" -split '-c-'
a-b
d-e-f
```

If the replacement pattern has characters that have special meaning in a regular expression (such as the `.` character, which represents "any character"), use the `-split` operator's `SimpleMatch` option, as in Example 5-4.

Example 5-4. PowerShell's SimpleMatch split option

```
PS > "a.b.c" -split '.'
(A bunch of newlines. Something went wrong!)



PS > "a.b.c" -split '.',0,"SimpleMatch"
a
b
c
```

For more information about the `-split` operator's options, type **`Get-Help about_split`**.

While regular expressions offer an enormous amount of flexibility, the `-split` operator gives you ultimate flexibility by letting you supply a script block for a split operation. For each character, it invokes the script block and splits the string based on the result. In the script block, `$_` (or `$PSItem`) represents the current character. For example, Example 5-5 splits a string on even numbers.

Example 5-5. Using a script block to split a string

```
PS > "1234567890" -split { ($_ % 2) -eq 0 }
1
3
5
7
9
```

When you're using a script block to split a string, `$_` represents the current character. For arguments, `$args[0]` represents the entire string, and `$args[1]` represents the index of the string currently being examined.

To split an entire file by a pattern, use the `-Delimiter` parameter of the `Get-Content` cmdlet:

```
PS > Get-Content test.txt
Hello
World
PS > (Get-Content test.txt)[0]
Hello
PS > Get-Content test.txt -Delimiter l
Hel
l
o
Worl
d
PS > (Get-Content test.txt -Delimiter l)[0]
Hel
PS > (Get-Content test.txt -Delimiter l)[1]
l
PS > (Get-Content test.txt -Delimiter l)[2]
o
Worl
PS > (Get-Content test.txt -Delimiter l)[3]
d
```

For more information about the `-split` operator, see Appendix A or type **`++Get-Help about0split++`**.

See Also

Appendix A

Appendix B

5.10 Combine Strings into a Larger String

Problem

You want to combine several separate strings into a single string.

Solution

Use PowerShell's *unary* `-join` operator to combine separate strings into a larger string using the default empty separator:

```
PS > -join ("A","B","C")
ABC
```

If you want to define the operator that PowerShell uses to combine the strings, use PowerShell's *binary* `-join` operator:

```
PS > ("A","B","C") -join "`r`n"
A
B
C
```

To use a cmdlet for features not supported by the `-join` operator, use the `Join-String` cmdlet:

```
PS > 1..5 | Join-String -DoubleQuote -Separator ','
"1","2","3","4","5"
```

Discussion

The `+-join+` operator provides a natural way to combine strings. When used with no arguments (the *unary* join operator), it joins the list using the default empty separator. When used between a list and a separator (the *binary* join operator), it joins the strings using the provided separator.

Aside from its performance benefit, the `-join` operator solves an extremely common difficulty that arises from trying to combine strings by hand.

When first writing the code to join a list with a separator (for example, a comma and a space), you usually end up leaving a lonely separator at the beginning or

end of the output:

```
PS > $list = "Hello","World"
PS > $output = ""
PS >
PS > foreach($item in $list)
{
    $output += $item + ", "
}

PS > $output
Hello, World,
```

You can resolve this by adding some extra logic to the `foreach` loop:

```
PS > $list = "Hello","World"
PS > $output = ""
PS >
PS > foreach($item in $list)
{
    if($output -ne "") { $output += ", " }
    $output += $item
}

PS > $output
Hello, World
```

Or, save yourself the trouble and use the `-join` operator directly:

```
PS > $list = "Hello","World"
PS > $list -join ", "
Hello, World
```

If you have advanced needs not covered by the `-join` operator, the .NET methods such as `[String]::Join()` are of course available in PowerShell.

For a more structured way to join strings into larger strings or reports, see Recipe 5.6.

See Also

Recipe 5.6

5.11 Convert a String to Uppercase or Lowercase

Problem

You want to convert a string to uppercase or lowercase.

Solution

Use the `ToUpper()` or `ToLower()` methods of the string to convert it to uppercase or lowercase, respectively.

To convert a string to uppercase, use the `ToUpper()` method:

```
PS > "Hello World".ToUpper()
HELLO WORLD
```

To convert a string to lowercase, use the `ToLower()` method:

```
PS > "Hello World".ToLower()
hello world
```

Discussion

Since PowerShell strings are fully featured .NET objects, they support many string-oriented operations directly. The `ToUpper()` and `ToLower()` methods are two examples of the many features that the `String` class supports. To learn what other functionality the `String` class supports, see Recipe 3.12.

Neither PowerShell nor the methods of the .NET `String` class directly support capitalizing only the first letter of a word. If you want to capitalize only the first character of a word or sentence, try the following commands:

```
PS > $text = "hello"
PS > $newText = $text.Substring(0,1).ToUpper() +
   $text.Substring(1)
$newText

Hello
```

You can also use an advanced regular expression replacement, as described in

You can also use an advanced regular expression replacement, as described in Recipe 32.6:

```
"hello world" -replace '\b(\w)',{ $_.Value.ToUpper() }
```

One thing to keep in mind as you convert a string to uppercase or lowercase is your motivation for doing it. One of the most common reasons is for comparing strings, as shown in Example 5-6.

Example 5-6. Using the ToUpper() method to normalize strings

```
## $text comes from the user, and contains the value "quit"
if($text.ToUpper() -eq "QUIT") { ... }
```

Unfortunately, explicitly changing the capitalization of strings fails in subtle ways when your script runs in different cultures. Many cultures follow different capitalization and comparison rules than you may be used to. For example, the Turkish language includes two types of the letter *I*: one with a dot and one without. The uppercase version of the lowercase letter *i* corresponds to the version of the capital *I* with a dot, not the capital *I* used in `QUIT`. Those capitalization rules cause the string comparison code in Example 5-6 to fail in the Turkish culture.

Recipe 13.8 shows us this quite clearly:

```
PS > Use-Culture tr-TR { "quit".ToUpper() -eq "QUIT" }
False
PS > Use-Culture tr-TR { "quIt".ToUpper() -eq "QUIT" }
True
PS > Use-Culture tr-TR { "quit".ToUpper() }
QUİT
```

For comparing some input against a hardcoded string in a case-insensitive manner, the better solution is to use PowerShell's `-eq` operator without changing any of the casing yourself. The `-eq` operator is case-insensitive and culture-neutral by default:

```
PS > $text1 = "Hello"
PS > $text2 = "HELLO"
PS > $text1 -eq $text2
True

PS > Use-Culture tr-TR { "quit" -eq "QUIT" }
```

```
True
```

For more information about writing culture-aware scripts, see Recipe 13.6.

See Also

Recipe 3.12

Recipe 13.6

Recipe 32.6

5.12 Trim a String

Problem

You want to remove leading or trailing spaces from a string or user input.

Solution

Use the `Trim()` method of the string to remove all leading and trailing whitespace characters from that string.

```
PS > $text = " `t Test String`t `t"
PS > "|" + $text.Trim() + "|"
|Test String|
```

Discussion

The `Trim()` method cleans all whitespace from the beginning *and* end of a string. If you want just one or the other, you can call the `TrimStart()` or `TrimEnd()` method to remove whitespace from the beginning or the end of the string, respectively. If you want to remove specific characters from the beginning or end of a string, the `Trim()`, `TrimStart()`, and `TrimEnd()` methods provide options to support that. To trim a list of specific characters from the end of a string, provide that list to the method, as shown in Example 5-7.

Example 5-7. Trimming a list of characters from the end of a string

```
PS > "Hello World".TrimEnd('d','l','r','o','W',' ')
```

```
He
```

> **NOTE**
>
> At first blush, the following command that attempts to trim the text `"World"` from the end of a string appears to work incorrectly:
>
> ```
> PS > "Hello World".TrimEnd(" World")
> He
> ```
>
> This happens because the `TrimEnd()` method takes a list of characters to remove from the end of a string. PowerShell automatically converts a string to a list of characters if required, and in this case converts your string to the characters `W`, `o`, `r`, `l`, `d`, and a space. These are in fact the same characters as were used in Example 5-7, so it has the same effect.

If you want to replace text anywhere in a string (and not just from the beginning or end), see Recipe 5.8.

See Also

Recipe 5.8

5.13 Format a Date for Output

Problem

You want to control the way that PowerShell displays or formats a date.

Solution

To control the format of a date, use one of the following options:

- The `Get-Date` cmdlet's `-Format` parameter:

```
PS > Get-Date -Date "05/09/1998 1:23 PM" -Format FileDateTime
19980509T1323000000

PS > Get-Date -Date "05/09/1998 1:23 PM" -Format "dd-MM-yyyy @
hh:mm:ss"
09-05-1998 @ 01:23:00
```

```
09-05-1998 @ 01:23:00
```

- PowerShell's string formatting (`-f`) operator:

```
PS > $date = [DateTime] "05/09/1998 1:23 PM"
PS > "{0:dd-MM-yyyy @ hh:mm:ss}" -f $date
09-05-1998 @ 01:23:00
```

- The object's `ToString()` method:

```
PS > $date = [DateTime] "05/09/1998 1:23 PM"
PS > $date.ToString("dd-MM-yyyy @ hh:mm:ss")
09-05-1998 @ 01:23:00
```

- The `Get-Date` cmdlet's `-UFormat` parameter, which supports Unix date format strings:

```
PS > Get-Date -Date "05/09/1998 1:23 PM" -UFormat "%d-%m-%Y @
%I:%M:%S"
09-05-1998 @ 01:23:00
```

Discussion

One of the common needs for converting a data into a string is for use in file names, directory names, and similar situations. For these incredibly common scenarios, the `Get-Date` cmdlet offers four easy options for its `-Format` parameter: `FileDate`, `FileDateUniversal`, `FileDateTime`, and `FileDateTimeUniversal`. These return representations of the date ("`19980509`") or date and time ("`19980509T1323000000`") in either local or universal time zones.

In addition to these standard format strings, the `-Format` parameter also supports standard .NET `DateTime` format strings. These format strings let you display dates in one of many standard formats (such as your system's short or long date patterns), or in a completely custom manner. For more information on how to specify standard .NET `DateTime` format strings, see Appendix E.

If you are already used to the Unix-style date formatting strings (or are converting an existing script that uses a complex one), the `-UFormat` parameter of the `Get-Date` cmdlet may be helpful. It accepts the format strings accepted

by the Unix `date` command, but does not provide any functionality that standard .NET date formatting strings cannot.

When working with the string version of dates and times, be aware that they are the most common source of *internationalization* issues—problems that arise from running a script on a machine with a different culture than the one it was written on. In North America, "05/09/1998" means "May 9, 1998." In many other cultures, though, it means "September 5, 1998." Whenever possible, use and compare `DateTime` objects (rather than strings) to other `DateTime` objects, as that avoids these cultural differences. Example 5-8 demonstrates this approach.

Example 5-8. Comparing DateTime objects with the -gt operator

```
PS > $dueDate = [DateTime] "01/01/2006"
PS > if([DateTime]::Now -gt $dueDate)
{
    "Account is now due"
}

Account is now due
```

> **NOTE**
>
> PowerShell *always* assumes the North American date format when it interprets a `DateTime` constant such as `[DateTime] "05/09/1998"`. This is for the same reason that all languages interpret numeric constants (such as `12.34`) in the North American format. If it did otherwise, nearly every script that dealt with dates and times would fail on international systems.

For more information about the `Get-Date` cmdlet, type **`Get-Help Get-Date`**. For more information about dealing with dates and times in a culture-aware manner, see Recipe 13.6.

See Also

Recipe 13.6

Appendix E

5.14 Convert a String Between One Format and

5.14 Convert a String Between One Format and Another

Problem

You have a series of text strings, and you want to convert it into another format.

Solution

Use the `Convert-String` cmdlet:

```
PS D:\> $phoneNumbers = "5551212","4524587","2112132","8752113"
PS D:\> $replacementExamples = "5551212=(425) 555-1212","4524587=(425)
452-4587"
PS D:\> $phoneNumbers | Convert-String -Example $replacementExamples
(425) 555-1212
(425) 452-4587
(425) 211-2132
(425) 875-2113
```

Discussion

The `Convert-String` cmdlet takes input text in one format and converts it to an output format. Unlike features in PowerShell that do this through regular expressions and capture groups and other complicated topics, the `Convert-String` cmdlet only requires that you provide it examples of data as it started, along with how it should look after the conversion is complete.

The `Convert-String` cmdlet, along with the `ConvertFrom-String` cmdlet are based on the Flash Fill technology that you can find in Excel. They are two of the things that are likely as close to magic as you will ever find in a shell. Rather than ask you to specify the exact series of steps you want to take to transform the text input, `Convert-String` instead learns these operations on your behalf.

In addition to the “Original=Replacement” format of examples, you can also supply objects (such as hashtables or PSCustomObjects) that have `Before` and `After` properties:

```
$examples =
```

```
    @{ Before = "Get-AclMisconfiguration.ps1"
       After = "Gets the AclMisconfiguration from the system" },
    @{ Before = "Get-AliasSuggestion.ps1"
       After = "Gets the AliasSuggestion from the system" }

PS > dir scripts\Get-* | Foreach-Object Name

Get-AclMisconfiguration.ps1
Get-AliasSuggestion.ps1
Get-Answer.ps1
Get-Arguments.ps1
Get-Characteristics.ps1
Get-Clipboard.ps1
Get-DetailedSystemInformation.ps1
(...)

PS > dir scripts\Get-* | Foreach-Object Name | Convert-String -Example
$examples

Gets the AclMisconfiguration from the system
Gets the AliasSuggestion from the system
Gets the Answer from the system
Gets the Arguments from the system
Gets the Characteristics from the system
Gets the Clipboard from the system
Gets the DetailedSystemInformation from the system
(...)
```

As with hand-written regular expressions or `String.Replace()` calls, `ConvertFrom-String` can sometimes make mistakes in understanding your intention. You can normally resolve these by providing more examples. Once you have a set of examples that you know express your intention, these examples will continue to work for similar text in the future.

For more information about using the `String.Replace()` method or regular expressions to modify strings, see Recipe 5.8.

See Also

Recipe 5.8

Recipe 5.15

5.15 Convert Text Streams to Objects

Problem

You have raw, unstructured text, and want to parse it into PowerShell objects

Solution

Use the `-Delimiter` parameter of the `ConvertFrom-String` cmdlet to parse data in simple column formats. PowerShell automatically generates property names if you do not specify them, and automatically converts the strings into more appropriate data types if possible:

```
$delimiter = "[ ]+(?=\d|Services|Console)"
$output = tasklist.exe | Select -Skip 3 | ConvertFrom-String -
Delimiter $delimiter

PS > $output | Where-Object P2 -lt 1000 | Format-Table

P1                   P2 P3       P4 P5
--                   -- --       -- --
System Idle Process   0 Services  0 8 K
System                4 Services  0 2,072 K
Secure System        72 Services  0 39,256 K
Registry            132 Services  0 99,088 K
smss.exe            524 Services  0 1,076 K
(...)
```

You can also use the `-Delimiter` parameter to parse entire strings. Any text matched by your capture groups will be present as the second property and beyond, which you can name as you like:

```
PS > $expression = 'FirstName=(.*);LastName=(.*)'
PS > $parsed = "FirstName=Lee;LastName=Holmes" |
    ConvertFrom-String -Delimiter $expression -Property
Ignored,FName,LName
PS > $parsed.FName
Lee
PS > $parsed.LName
Holmes
```

Use the `-Template` parameter to parse data automatically based on the tagging that you've added to example text in the template:

```
$template = @"
{FName*:Lee} {LName:Holmes}
{FName*:John} {LName:Smith}
"@

"Lee Holmes","Adam Smith","Some Body","Another Person" |
    ConvertFrom-String -TemplateContent $template

FName   LName
-----   -----
Lee     Holmes
Adam    Smith
Some    Body
Another Person
```

Discussion

One of the strongest features of PowerShell is its object-based pipeline. You don't waste your energy creating, destroying, and recreating the object representation of your data. In other shells, you lose the full-fidelity representation of data when the pipeline converts it to pure text. You can regain some of it through excessive text parsing, but not all of it.

However, you still often have to interact with low-fidelity input that originates from outside PowerShell. Text-based data files and legacy programs are two examples.

PowerShell offers great support for all of the three text-parsing staples you might be aware of from other shells:

Sed

> Replaces text. For that functionality, PowerShell offers the `-replace` operator and `Convert-String` cmdlet.

Grep

> Searches text. For that functionality, PowerShell offers the `Select-String` cmdlet, among others.

The third traditional text-parsing tool, *Awk*, lets you chop a line of text into more intuitive groupings. For this, PowerShell offers the incredibly powerful `ConvertFrom-String` cmdlet.

In its simplest form, you can use the `ConvertFrom-String` cmdlet to parse

column-oriented output based on a delimiter that you provide. The delimiter defaults to runs of whitespace, but you can also provide strings of your choosing or much more detailed regular expressions. PowerShell will also convert the text into more appropriate data types (such as integers and dates), if possible.

For more complicated needs, the `ConvertFrom-String` cmdlet supports example-driven parsing. As with the `Convert-String` cmdlet, this is about as close to magic as you'll ever experience in a shell. Rather than forcing you to write complicated parsers by hand, the `ConvertFrom-String` cmdlet automatically learns how to extract data based on how you've tagged data in your example template.

Let's consider trying to parse an address book:

```
Record
------

FName: Lee
LName: Holmes

Record
------

FName: Adam
LName: Smith

Record
------

FName: Some
LName: Body

Last updated: 05/09/2021
```

To have `ConvertFrom-String` parse it, we need to give it a template. A good way to think about templates is to imagine taking some sample output, highlighting regions of the sample output with a mouse, and then naming those regions.

In a template, the left curly brace `{` represents the start of your selection, and the right curly brace `}` represents the end of your selection. To name your selection, you provide a property name and a colon right after the opening brace. So, `PowerShell Rocks` becomes `{FName:PowerShell}`

`{LName:Rocks}`.

Let's start creating a template. In a new file, start with this as an example, and save it as `addressbook.template.txt` (the name is up to you):

```
{Record:Record
------

FName: Some
LName: Body}

Last updated: {LastUpdated:05/09/2021}
```

When you run `ConvertFrom-String` on this input and template, we get:

```
PS > $book = Get-Content addressbook.txt |
    ConvertFrom-String -TemplateFile addressbook.template.txt
PS > $book.LastUpdated
05/09/2021

PS > $book.Record

Record
------

FName: Lee
LName: Holmes
```

There were several records, though. To tell `ConvertFrom-String` that the input contained multiple of a certain pattern, use an asterisk after the property name:

```
{Record*:Record
------

FName: Some
LName: Body}

Last updated: {LastUpdated:05/09/2021}
```

If we run this, we see that ConvertFrom-String hasn't quite figured out the record format. So let's give it another example:

```
{Record*:Record
```

```
------

FName: Some
LName: Body}

{Record*:Record
------

FName: Adam
LName: Smith}

Last updated: {LastUpdated:05/09/2021}
```

And now, `ConvertFrom-String` understands records and a footer:

```
PS > (Get-Content addressbook.txt |
    ConvertFrom-String -TemplateFile addressbook.template.txt)

Record
------
Record...
Record...
Record...

PS > (Get-Content addressbook.txt |
    ConvertFrom-String -TemplateFile
addressbook.template.txt).LastUpdated

05/09/2021
```

To tell `ConvertFrom-String` about the inner structure of a record, we simply tag it and name it as well. Update the first record in your template:

```
(...)
FName: {FName:Some}
LName: {LName:Body}}
(...)
```

And now `ConvertFrom-String` fully understands our database format.

```
PS > (Get-Content addressbook.txt |
    ConvertFrom-String -TemplateFile addressbook.template.txt)

Record
------
{@{FName=Lee; LName=Holmes}}
```

```
{@{FName=Adam; LName=Smith}}
{@{FName=Some; LName=Body}}

PS > (Get-Content addressbook.txt |
    ConvertFrom-String -TemplateFile
addressbook.template.txt).Record[0].FName
Lee
```

As our final magic trick, let's tell PowerShell that `LastUpdate` is a `[DateTime]`. Update your template to include:

```
(...)
Last updated: {[DateTime] LastUpdated:05/09/2021}
(...)
```

Which gives an amazing result:

```
PS > (Get-Content addressbook.txt |
    ConvertFrom-String -TemplateFile
addressbook.template.txt).LastUpdated

Sunday, May 9, 2021 12:00:00 AM
```

See Also

Recipe 1.2

Recipe 5.14

5.16 Generate Large Reports and Text Streams

Problem

You want to write a script that generates a large report or large amount of data.

Solution

The best approach to generating a large amount of data is to take advantage of PowerShell's streaming behavior whenever possible. Opt for solutions that pipeline data between commands:

```
Get-ChildItem C:\*.txt -Recurse | Out-File c:\temp\AllTextFiles.txt
```

rather than collect the output at each stage:

```
$files = Get-ChildItem C:\*.txt -Recurse
$files | Out-File c:\temp\AllTextFiles.txt
```

If your script generates a large text report (and streaming is not an option), use the `StringBuilder` class:

```
$output = New-Object System.Text.StringBuilder
Get-ChildItem C:\*.txt -Recurse |
    Foreach-Object { [void] $output.AppendLine($_.FullName) }
$output.ToString()
```

rather than simple text concatenation:

```
$output = ""
Get-ChildItem C:\*.txt -Recurse | Foreach-Object { $output +=
$_.FullName }
$output
```

Discussion

In PowerShell, combining commands in a pipeline is a fundamental concept. As scripts and cmdlets generate output, PowerShell passes that output to the next command in the pipeline as soon as it can. In the Solution, the `Get-ChildItem` commands that retrieve all text files on the *C:* drive take a very long time to complete. However, since they *begin* to generate data almost immediately, PowerShell can pass that data on to the next command as soon as the `Get-ChildItem` cmdlet produces it. This is true of any commands that generate or consume data and is called *streaming*. The pipeline completes almost as soon as the `Get-ChildItem` cmdlet finishes producing its data and uses memory very efficiently as it does so.

The second `Get-ChildItem` example (which collects its data) prevents PowerShell from taking advantage of this streaming opportunity. It first stores all the files in an array, which, because of the amount of data, takes a long time and an enormous amount of memory. Then, it sends all those objects into the

output file, which takes a long time as well.

However, most commands can consume data produced by the pipeline directly, as illustrated by the `Out-File` cmdlet. For those commands, PowerShell provides streaming behavior as long as you combine the commands into a pipeline. For commands that do not support data coming from the pipeline directly, the `Foreach-Object` cmdlet (with the aliases of `foreach` and %) lets you work with each piece of data as the previous command produces it, as shown in the `StringBuilder` example.

Creating large text reports

When you generate large reports, it is common to store the entire report into a string, and then write that string out to a file once the script completes. You can usually accomplish this most effectively by streaming the text directly to its destination (a file or the screen), but sometimes this is not possible.

Since PowerShell makes it so easy to add more text to the end of a string (as in `$output += $_.FullName`), many initially opt for that approach. This works great for small-to-medium strings, but it causes significant performance problems for large strings.

NOTE

As an example of this performance difference, compare the following:

```
PS > Measure-Command {
   $output = New-Object Text.StringBuilder
   1..10000 |
       Foreach-Object { $output.Append("Hello World") }
}

(...)
TotalSeconds : 2.3471592

PS > Measure-Command {
   $output = ""
   1..10000 | Foreach-Object { $output += "Hello World" }
}

(...)
TotalSeconds      : 4.9884882
```

In the .NET Framework (and therefore PowerShell), strings never change after you create them. When you add more text to the end of a string, PowerShell has to build a *new* string by combining the two smaller strings. This operation takes a long time for large strings, which is why the .NET Framework includes the `System.Text.StringBuilder` class. Unlike normal strings, the `StringBuilder` class assumes that you will modify its data—an assumption that allows it to adapt to change much more efficiently.

5.17 Generate Source Code and Other Repetitive Text

Problem

You want to simplify the creation of large amounts of repetitive source code or other text.

Solution

Use PowerShell's string formatting operator (`-f`) to place dynamic information inside of a preformatted string, and then repeat that replacement for each piece of dynamic information.

Discussion

Code generation is a useful technique in nearly any technology that produces output from some text-based input. For example, imagine having to create an HTML report to show all of the processes running on your system at that time. In this case, "code" is the HTML code understood by a web browser.

HTML pages start with some standard text (`<html>`, `<head>`, `<body>`), and then you would likely include the processes in an HTML `<table>`. Each row would include columns for each of the properties in the process you're working with.

Generating this by hand would be mind-numbing and error-prone. Instead, you can write a function to generate the code for the row:

```
function Get-HtmlRow($process)
{
    $template = "<TR> <TD>{0}</TD> <TD>{1}</TD> </TR>"
    $template -f $process.Name,$process.ID
}
```

and then generate the report in milliseconds, rather than hours:

```
"<HTML><BODY><TABLE>" > report.html
Get-Process | Foreach-Object { Get-HtmlRow $_ } >> report.html
"</TABLE></BODY></HTML>" >> report.html
Invoke-Item .\report.html
```

In addition to the formatting operator, you can sometimes use the `String.Replace` method:

```
$string = @'
Name is __NAME__
Id is __ID__
'@

$string = $string.Replace("__NAME__", $process.Name)
$string = $string.Replace("__ID__", $process.Id)
```

This works well (and is very readable) if you have tight control over the data you'll be using as replacement text. If it is at all possible for the replacement text to contain one of the special tags (`__NAME__` or `__ID__`, for example), then they will *also* get replaced by further replacements and corrupt your final output.

To avoid this issue, you can use the `Format-String` script shown in Example 5-9.

Example 5-9. Format-String.ps1

```
##############################################################################
######
##
## Format-String
##
## From Windows PowerShell Cookbook (O'Reilly)
## by Lee Holmes (http://www.leeholmes.com/guide)
```

```powershell
##
#############################################################################
######

<#

.SYNOPSIS

Replaces text in a string based on named replacement tags

.EXAMPLE

PS > Format-String "Hello {NAME}" @{ NAME = 'PowerShell' }
Hello PowerShell

.EXAMPLE

PS > Format-String "Your score is {SCORE:P}" @{ SCORE = 0.85 }
Your score is 85.00 %

#>

param(
    ## The string to format. Any portions in the form of {NAME}
    ## will be automatically replaced by the corresponding value
    ## from  the supplied hashtable.
    $String,

    ## The named replacements to use in the string
    [hashtable] $Replacements
)

Set-StrictMode -Version 3

$currentIndex = 0
$replacementList = @()

if($String -match "{{|}}")
{
    throw "Escaping of replacement terms are not supported."
}

## Go through each key in the hashtable
foreach($key in $replacements.Keys)
{
    ## Convert the key into a number, so that it can be used by
    ## String.Format
    $inputPattern = '{(.*)' + $key + '(.*)}'
    $replacementPattern = '{${1}' + $currentIndex + '${2}}'
    $string = $string -replace $inputPattern,$replacementPattern
```

```
    $replacementList += $replacements[$key]

    $currentIndex++
}

## Now use String.Format to replace the numbers in the
## format string.
$string -f $replacementList
```

PowerShell includes several commands for code generation that you've probably used without recognizing their "code generation" aspect. The `ConvertTo-Html` cmdlet applies code generation of incoming objects to HTML reports. The `ConvertTo-Csv` cmdlet applies code generation to CSV files. The `ConvertTo-Xml` cmdlet applies code generation to XML files.

Code generation techniques seem to come up naturally when you realize you are writing a report, but they are often missed when writing source code of another programming or scripting language. For example, imagine you need to write a C# function that outputs all of the details of a process. The `System.Diagnostics.Process` class has a lot of properties, so that's going to be a long function. Writing it by hand is going to be difficult, so you can have PowerShell do most of it for you.

For any object (for example, a process that you've retrieved from the `Get-Process` command), you can access its `PsObject.Properties` property to get a list of all of its properties. Each of those has a `Name` property, so you can use that to generate the C# code:

```
$process.PsObject.Properties |
    Foreach-Object {
        'Console.WriteLine("{0}: " + process.{0});' -f $_.Name }
```

This generates more than 60 lines of C# source code, rather than having you do it by hand:

```
Console.WriteLine("Name: " + process.Name);
Console.WriteLine("Handles: " + process.Handles);
Console.WriteLine("VM: " + process.VM);
Console.WriteLine("WS: " + process.WS);
Console.WriteLine("PM: " + process.PM);
Console.WriteLine("NPM: " + process.NPM);
Console.WriteLine("Path: " + process.Path);
Console.WriteLine("Company: " + process.Company);
```

```
Console.WriteLine("Company: " + process.Company);
Console.WriteLine("CPU: " + process.CPU);
Console.WriteLine("FileVersion: " + process.FileVersion);
Console.WriteLine("ProductVersion: " + process.ProductVersion);
(...)
```

Similar benefits come from generating bulk SQL statements, repetitive data structures, and more.

PowerShell code generation can still help you with large-scale administration tasks, even when PowerShell is not available. Given a large list of input (for example, a complex list of files to copy), you can easily generate a *cmd.exe* batch file or Unix shell script to automate the task. Generate the script in PowerShell, and then invoke it on the system of your choice!

About the Author

Lee Holmes is a developer on the Microsoft Windows PowerShell team, and has been an authoritative source of information about PowerShell since its earliest betas. His vast experience with Windows PowerShell enables him to integrate both the “how” and the “why” into discussions. Lee’s involvement with the PowerShell and administration community (via newsgroups, mailing lists, and blogs) gives him a great deal of insight into the problems faced by all levels of administrators and PowerShell users alike.