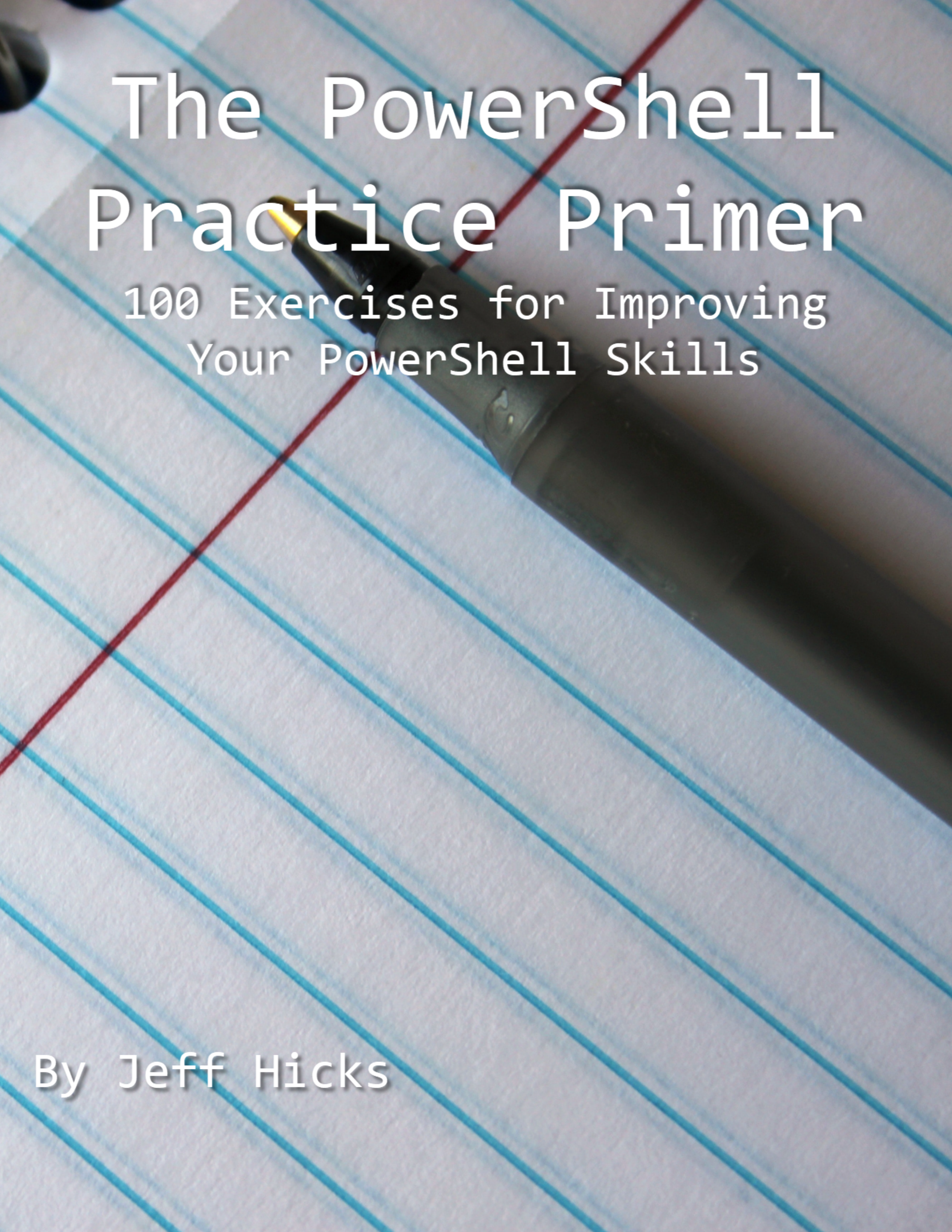
The PowerShell Practice Primer

100 Exercises for Improving Your PowerShell Skills

By Jeff Hicks



The PowerShell Practice Primer

100+ Exercises for Improving Your PowerShell Skills

Jeff Hicks

This book is for sale at http://leanpub.com/psprimer

This version was published on 2021-03-02

Tweet This Book!

Please help Jeff Hicks by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought the #PowerShell Practice Primer to improve my fluency in PowerShell. 100 Exercises to go!

The suggested hashtag for this book is #PSPrimer.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#PSPrimer

Contents

About This Book i
Dedication ii
Acknowledgements iii
A Note on Code Listings iv
Foreword vi
Preface vii
How To Use This Book ix
A PowerShell Refresher 1
Part 1 - PowerShell Principles 12
Part 1 Exercises 13
Part 1 Solutions 22
Part 2 - PowerShell Providers 34
Part 2 Exercises 35
Part 2 Solutions 43
Part 3 - PowerShell Structures 54
Part 3 Exercises 55
Part 3 Solutions 64
Part 4 - WMI and CIM 78
Part 4 Exercises 79
Part 4 Solutions 88

Part 5 - PowerShell 7 103

Part 5 Exercises 104

Part 5 Solutions 108

Afterword 114

About the Author 115

Release Notes 116

About This Book

This book is being offered on Leanpub.com[1], an "Agile" online publishing platform. That means the book is published as I write it, and *that* means I'll be able to revise it as needed in the future. Early adopters of this book will get parts of the book as I complete them. Those of you who purchase a copy at the end will get the complete and finished book. And *everyone* can get an updated version should I need to provide any updates or corrections. I do not anticipate offering editions elsewhere such as Amazon.com.

If you purchased this book, I sincerely thank you. Writing any sort of book requires a time commitment and believe it or not, I'm not in it for the money! Your purchase price is important to keeping the lights on, food on the table and connected to the Internet. It also helps in a small way so that I can continue projects like this book. Please treat your copy of the book as your own personal copy - it isn't to be uploaded anywhere, and you aren't meant to give copies to other people. I appreciate your respecting my rights and not making unauthorized copies of this work.

If you got this book for free from somewhere other than Leanpub, know that you are making it difficult for me to write books. When I can't make even a small amount of money from my books, I'm discouraged from writing them. If you realize you have an "unauthorized" copy and find this book useful, I would greatly appreciate you purchasing a legitimate copy from Leanpub.com.

Please note that this book is not authorized for classroom use unless a unique copy has been purchased for each student. No one is authorized or licensed to manually reproduce the PDF version of this book for use in any kind of class or training environment.

---

[1]http://leanpub.com

Dedication

I would like to dedicate this book to my amazing wife, Beth, who continues to make everything possible.

I also dedicate this work to everyone who has decided to take the leap and learn something new, such as PowerShell, in hopes of advancing their careers, improving their skills or simply because they are curious. It is thanks to people like you that *I* have the career that I do.

Acknowledgements

I’d like to thank the keen eye of my technical reviewer, Michael Bender. Michael is a long-time friend, PowerShell expert, and teacher. He helped keep me on track and focused on how to make this book work for you. Of course, any technical mistakes are my own.

A Note on Code Listings

If you've read other PowerShell books from Leanpub, you probably have seen this disclaimer for code samples. I know it adds a wrinkle to your learning experience, but it can't be helped. The code formatting in this book only allows for about 80 characters per line before things start wrapping. I've tried my best to keep the code samples within that limit, although sometimes you may see awkward formatting as a result.

For example:

```
Get-CimInstance -ComputerName $computer -Classname Win32_logicalDisk -Filter "drivet\
ype=3" -property  DeviceID,Size,FreeSpace
```

Here, you can see the default action in a Leanpub book for a too-long line - it gets word-wrapped, and a backslash inserted at the wrap point. I try to avoid these situations, but sometimes they are unavoidable. When I *do* avoid them, it may be by using awkward formatting, such as using backticks (‘).

```
Get-CimInstance -ComputerName $computer `
                -Classname Win32_logicalDisk `
                -Filter "drivetype=3" `
                -property 'DeviceID','Size','FreeSpace'
```

In this situation, I've given up on neatly aligning everything to prevent a wrap situation. Ugly, but a necessary evil when printing code samples. When *you* write PowerShell expressions, you should not be limited by these constraints.

If you are reading this book on a Kindle, tablet, or other e-reader, then I hope you'll understand that all code formatting bets are off the table. There's no way to know what the formatting will look like due to how each reader might format the page.

The intended solutions in this book are commands you would type in an interactive PowerShell console, so you can just keep typing. There is no reason for you to have to use a backtick to "break" a command. Simply type out your command. If you want to break a long line to make it easier to read without a lot of horizontal scrolling, you can press `Enter` after any of these characters:

- Open parenthesis (
- Open curly brace {

- Pipe |
- Comma ,
- Semicolon ;
- Equal sign =

This is probably not a complete list, but breaking after any of these characters makes the most sense. I'm assuming that most of the solutions in this book will be short enough that formatting a PowerShell expression won't be an issue.

Foreword

For ages, I've always felt that the best way to learn something was to get your hands dirty and actually *work with* that something. Traditional book, video, and classroom learning is all well and good, but *doing* something is what really cements it for most people.

Well, that's what Jeff has done with this book. It's a fantastic companion to anyone learning PowerShell and wanting to fill in the inevitable, tiny gaps that almost every form of education invariably leaves behind. You'll run smack into many of PowerShell's "gotchas", giving you an opportunity to work through them in your own place and way, which is absolutely the best way to learn.

If you're looking to make the jump between "book-learned neophyte" and "practicing PowerSheller" I think you've come to the right place. Take your time with these exercises - don't just read them and rush to the solution. Really work through them. You'll be changing your brain just a bit, into one that *thinks* in PowerShell more easily, which will make *accomplishing* things in PowerShell easier.

Good Luck!

Don Jones Co-Founder, PowerShell.org[2]

[2]https://powershell.org

Preface

I've been writing, teaching, and thinking about PowerShell since the Monad code-name days. Over the last 15 years, I've watched many people struggle to learn PowerShell. And it *can* be challenging, no doubt about it. Learning PowerShell is no different than if I asked you to learn conversational Hungarian. The only way you can gain proficiency is by constant exposure and repetition. As with learning any language, you master it by *doing*.

Many people will start their PowerShell learning journey with a book, such as *Learn Windows PowerShell in a Month of Lunches*[3] that I co-authored with Don Jones. One of the reasons that book is successful is because we give you things to **do**. But once you've finished the book, what's next? Sure, there are other books, YouTube clips, and training courses to read and watch. But I think the best way to learn is to **do**. That is the premise behind this book.

A *primer* is any early educational book used to teach essential skills such as reading. For years I've thought about putting together a book of PowerShell exercises that would enhance the learning process and might even be a little fun. But I also knew that this was a project no traditional publisher was likely to pick up. Fast forward several years - now more people are learning PowerShell than ever before, and there is an Agile publishing platform in Leanpub.com. Combine this with my years of experience teaching and writing about PowerShell, and I think there is a fantastic learning opportunity.

I have organized the exercises in this primer by topic and complexity. Each section begins with simple problems and progress in complexity. I recommend you start at the beginning and work your way through the book, especially as some exercises may build on previous challenges. My goal is not to teach you 100-plus different ways to use PowerShell, although that is a nice benefit. My real hope is that in figuring out a solution to each exercise, the process you go through, which builds your PowerShell muscle memory, is the ultimate objective. I think you'll find that the more complicated problems at the end of the book won't be as intimidating because you will have built up your PowerShell muscles and proficiency.

The provided solutions **are not intended** to be authoritative. They are *suggested* solutions that follow PowerShell best practices. Although, since the scenario is that you are running commands from the console prompt, you are welcome to use shortcuts like aliases and shell variables. You will see that in many of the suggested answers. You may devise other solutions that I hadn't even considered. That is one of the terrific features of PowerShell. If your solution works at a PowerShell command prompt and follows community best-practices, then you completed the task.

The exercises are written primarily for Windows PowerShell, although you should be able to use

[3] https://www.manning.com/books/learn-windows-powershell-in-a-month-of-lunches-third-edition?query=learn%20powershell

PowerShell 7.x since the fundamentals don't change between versions. Part 5 though, is written specifically for PowerShell 7.x.

Once you fully understand the PowerShell paradigm, language, and syntax, everything you might want to do with it will fall into place. I hope this primer gets you started down that path.

Jeff Hicks

What I hear, I forget;

What I see, I remember;

What I do, I understand.

— Confucius

How To Use This Book

What You Need

I developed the exercises in the book with Windows PowerShell 5.1 on a Windows 10 desktop. Many, if not most, of the exercises should also work on PowerShell 7.x, although you may arrive at slightly different answers. You should have administrator rights, and the ability to modify things like files and the registry. I've tried to avoid the need for a domain or member servers. My assumption is that you can always query the local computer when asked using `localhost`, `$env:computername` or the actual name. If you have a lab environment where you can query one or more remote computers, that is great, but I didn't want to make that a requirement.

Learning Strategies

The most important learning strategy will be using PowerShell's help system. All of the commands I am using, or feel you will use, have full help, often with several examples. Don't forget that you can use wildcards with `Get-Help` or the `help` function. PowerShell help also includes a large number of topic or conceptual help files that all start with the keyword `about`. You should also be familiar with using cmdlets like `Get-Member` and `Get-Command`. Learning how to *discover* in PowerShell is a critical skill.

I developed and designed this book with the assumption that you would work through it from start to finish. I have tried to arrange the content in such a way that very basic exercises are first with complexity increasing throughout the book. You may also find that some exercises build on earlier ones, so it makes sense to go through the book in order.

Because the *real* learning is the process you go through to solve each exercise, you may prefer to print out the exercise chapter from each part so that you have some place to keep notes. I've included such an area for each problem. When you are finished, you can then compare your answers with the provided solutions and "grade" your work. When you purchased the book, you should have gotten an Extras package which includes a `.ps1` file version of each part. This file is pre-formatted with comment blocks and regions that should work in the PowerShell ISE or Visual Studio Code[4]. You can use the file to keep notes, and you get the added benefit of having a PowerShell editing tool to help you develop a solution.

[4]https://code.visualstudio.com/

As you read through the solutions, you'll see that many of them use PowerShell aliases. I'm expecting many of your solutions will also use aliases. In the context of this book this is completely acceptable. The goal is for you to work from a PowerShell prompt to achieve a given task. Aliases are OK to use because *you* are typing them and (presumably) you know what they mean. It is only when you get to writing PowerShell scripts that using aliases is something to avoid. But you aren't scripting in this book, so alias away!

I trust that you will take whatever time is necessary to solve each exercise. Sure, you can always peek at the solution - where necessary, I've included hints so that you aren't totally lost. If you are still unsure about how or why it works after seeing the solution and reading whatever notes might accompany it, your first step should be to re-read the full cmdlet help, including examples. As a last resort, you can contact me through the Feedback section of the book's Leanpub page or use the book's forum.

A PowerShell Refresher

To get the most out of this book, I'm assuming you already have a basic understanding of Windows PowerShell fundamentals. The rule of thumb that the PowerShell community appears to have adopted is that you have read the *Learn PowerShell in a Month of Lunches* book or have equivalent experience. Even so, I want to make sure we have a common baseline of PowerShell knowledge. This brief introduction is intended as a crash-course or refresher on fundamental PowerShell skills. I am expecting that everything in this chapter will sound familiar. If there is something that is unfamiliar or new, I encourage you to take a bit of time to brush up on that topic before continuing with this book.

The goal of this book is to increase your proficiency in using PowerShell from the console in an interactive session. *No scripting experience or skills are required.*

Even though this refresher is based on experiences and commands related to Windows PowerShell 5.1, the concepts apply equally to PowerShell 7.x that you might run on Windows, Linux, or a Mac. Some details might change on those platforms, such as remoting, but the core PowerShell concepts do not.

All About the Objects

What makes PowerShell different, and I would argue so effective, is that it is focused on working with **objects** instead of **text**. It is silly to argue the merits of other tools like *bash* vs PowerShell because each tool is designed for its environment. The former is the right tool in a Linux environment because it is designed to exist in a text-based world. The latter is intended for Windows, which is object-oriented.

PowerShell is designed to work with *objects*. An object is a *thing* defined by software. We don't care how it was created. For our purposes, objects have two characteristics that matter. Objects have one or more *properties*. A property is something that describes the object. For example, imagine a carrot object. This object has properties such as *color* and _length). In PowerShell, we can access an object's properties using a dotted notation: `object.property`.

```
PS C:\> $carrot.color
orange
PS C:\> $carrot.length
6
```

Some objects might also have *methods*. A method is some action that either the object can do, or you can do to the object. Continuing with our carrot example, it might have a `Peel()` method. Some methods might take parameters. Perhaps the carrot object has a `Cut([int]Size)` method that needs to know how much to cut. Methods too, are invoked using dotted notation.

```
PS C:\> $carrot.peel()
PS C:\> $carrot.Cut(2)
```

Whenever you invoke a method on an object, you need to include the parentheses, even if there are no parameters. You may or may not get a result depending on the method. However, you should rarely need to invoke a method on an object. Ideally, there will be a command you can execute that will run that method "under the hood".

PowerShell Wants to Help

Be aware that PowerShell will automatically extend objects in PowerShell. You will often see methods and properties that PowerShell adds and don't belong to the original or native object.

To move back to a more practical example, think of a Windows service. It has several properties that describe it:

- Service name
- Display Name
- Status
- Start mode
- Account name

Now think of the actions a service can do, or you can do to a service:

- Start
- Stop
- Pause
- Resume

Fortunately for us, PowerShell has commands to handle those actions. Instead of figuring out how to invoke the `Stop()` method, there is a command that will do it for us.

```
Stop-Service -name bits
```

The commands I've been referring to are known as *cmdlets*. A cmdlet is single purpose command that works with objects. Cmdlets often emit objects to the PowerShell pipeline and can be designed to consume objects. Cmdlet behavior can be controlled by the use of *parameters*. In my `Stop-Service` example, -Name is a parameter using the value `bits`. This instructs the cmdlet to only stop the `bits` service.

Pipelines Rule

PowerShell's real value is that objects can be passed from one command to another via a *pipeline*. The vertical bar (`|`) character represents the pipeline. At the end of the pipeline, PowerShell displays the remaining object. You don't have to write any complicated code to parse text output. You can simply let the cmdlets do their thing. Think of cmdlets as building blocks that you can join together to achieve a result.

```
Get-Process |
Sort-Object -property PagedMemorySize -descending |
Select-Object -first 5
```

PowerShell cmdlets use a `Verb-Noun` naming convention that make it easy for you to visualize the pipeline process. In the example above, I am getting all processes with the first cmdlet. `Get-Process` writes a collection of process objects to the pipeline. This object has a property called *PageMemorySize*. The second part of the pipeline sorts the process objects in descending order on that property. All of the sorted process objects are then piped to the last command, which is selecting the first five objects. The result is a display of the five top processes using the most paged memory. I didn't have to do any programming or scripting. The objects moved down the pipeline from command to command until there was nothing left to do.

Of course, this technique only works when you are piping objects that a cmdlet is designed to handle. If you try something like this:

```
Get-Service win* | Stop-Process
```

It will likely fail, which makes sense if you think about it. I'm getting all services that start with `win` and then sending them to `Stop-Process`, which does exactly what the name suggests. Since a service object is not the same thing as a process object this won't work.

Sometimes, objects will change in the pipeline. Which is not a bad thing.

```
PS C:\> dir C:\Data -file -Recurse | measure-object -Property length -sum`


Count             : 174
Average           :
Sum               : 289755995
Maximum           :
Minimum           :
StandardDeviation :
Property          : Length
```

The first part of the pipelined expression is getting a bunch of file objects from C:Data. I am then sending these objects to the `Measure-Object` cmdlet, which adds up values from the `Length` property of the file object. The command writes a `MeasureInfo` object to the pipeline, not a file object. All I had to do was figure out what cmdlets to join together in a pipelined expression to get the desired result. In fact, it is pretty easy to continue fine-tuning this PowerShell expression.

```
PS C:\> dir $env:temp -file -Recurse | Measure-Object -Property length -sum |
Select-Object -property Count,Sum

Count       Sum
-----       ---
  274 116839081
```

I added another step to the pipeline to send the measurement object to `Select-Object` and select the properties from *that* object that matter to me. Again, no scripting and no programming. Instead, I wrote a simple 3 step expression that added the file size of 274 files and displayed an object with the results.

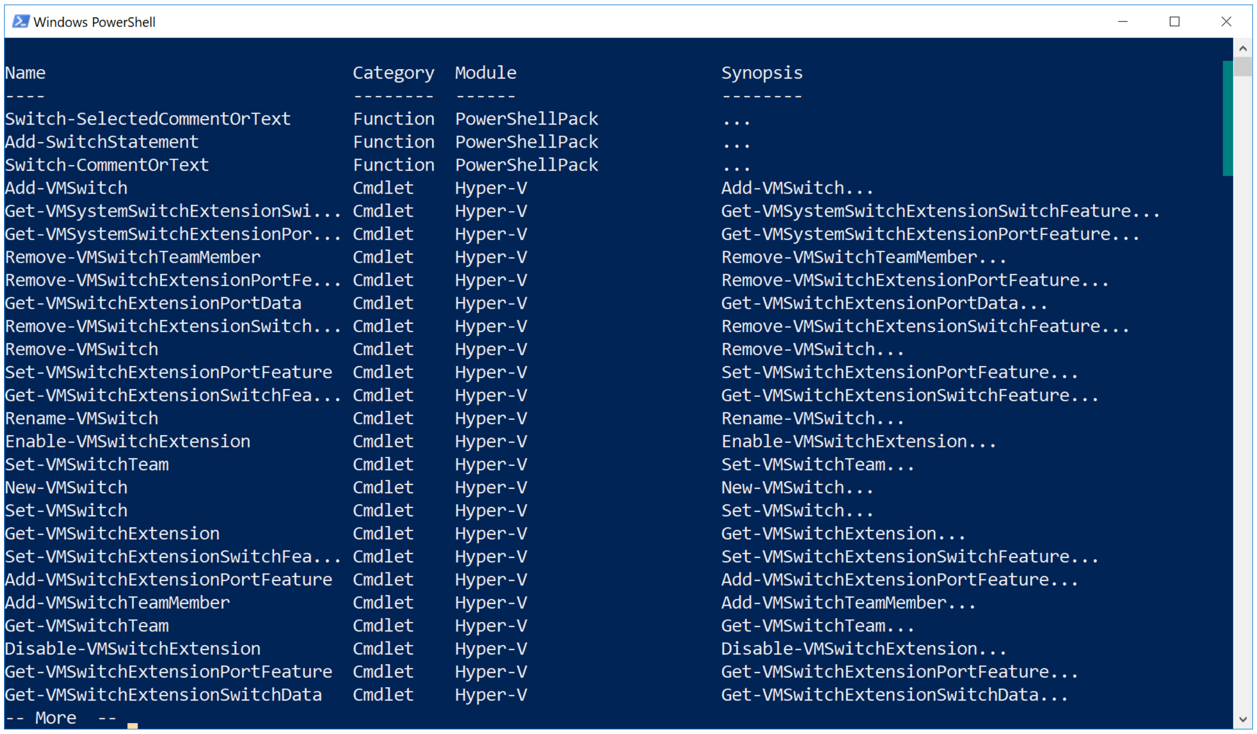
Key Commands

As you are learning PowerShell, and frankly, for as long as you use PowerShell, there are a few cmdlets that you will use and rely on constantly. Understanding how to use these cmdlets will improve your PowerShell experience.

Get-Help

Without a doubt, the most important command you need to know and get in the habit of using often is `Get-Help`. Or use the *help* function, which is essentially a wrapper around `Get-Help` that pipes output to the command line *more* command so that you can view help in “pages”. Even though the PowerShell ISE doesn’t support the *more* command, you can still use the *help* function.

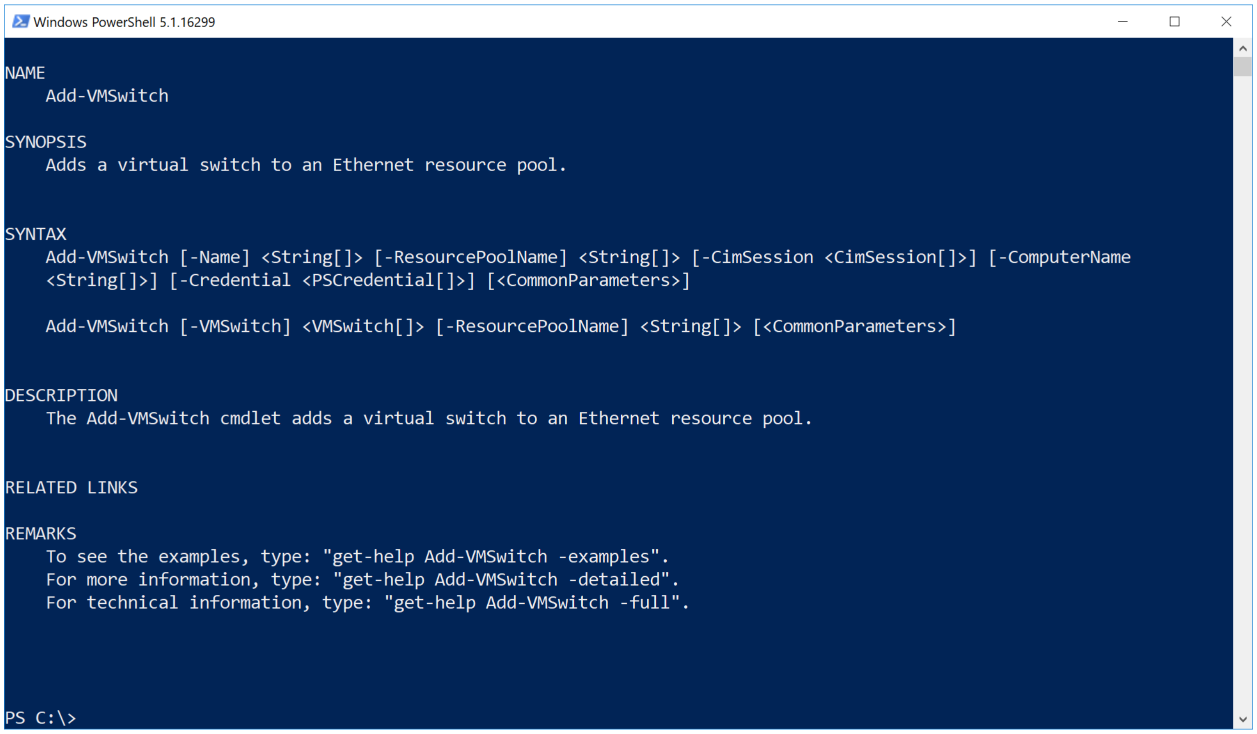
The Help command recognized wildcards which is often a good place to start. Suppose you are looking for commands to work with Hyper-V virtual switches. You might start with a broad search.



```
Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
Switch-SelectedCommentOrText      Function  PowerShellPack            ...
Add-SwitchStatement               Function  PowerShellPack            ...
Switch-CommentOrText              Function  PowerShellPack            ...
Add-VMSwitch                      Cmdlet    Hyper-V                   Add-VMSwitch...
Get-VMSystemSwitchExtensionSwi... Cmdlet    Hyper-V                   Get-VMSystemSwitchExtensionSwitchFeature...
Get-VMSystemSwitchExtensionPor... Cmdlet    Hyper-V                   Get-VMSystemSwitchExtensionPortFeature...
Remove-VMSwitchTeamMember         Cmdlet    Hyper-V                   Remove-VMSwitchTeamMember...
Remove-VMSwitchExtensionPortFe... Cmdlet    Hyper-V                   Remove-VMSwitchExtensionPortFeature...
Get-VMSwitchExtensionPortData     Cmdlet    Hyper-V                   Get-VMSwitchExtensionPortData...
Remove-VMSwitchExtensionSwitch... Cmdlet    Hyper-V                   Remove-VMSwitchExtensionSwitchFeature...
Remove-VMSwitch                   Cmdlet    Hyper-V                   Remove-VMSwitch...
Set-VMSwitchExtensionPortFeature  Cmdlet    Hyper-V                   Set-VMSwitchExtensionPortFeature...
Get-VMSwitchExtensionSwitchFea... Cmdlet    Hyper-V                   Get-VMSwitchExtensionSwitchFeature...
Rename-VMSwitch                   Cmdlet    Hyper-V                   Rename-VMSwitch...
Enable-VMSwitchExtension          Cmdlet    Hyper-V                   Enable-VMSwitchExtension...
Set-VMSwitchTeam                  Cmdlet    Hyper-V                   Set-VMSwitchTeam...
New-VMSwitch                      Cmdlet    Hyper-V                   New-VMSwitch...
Set-VMSwitch                      Cmdlet    Hyper-V                   Set-VMSwitch...
Get-VMSwitchExtension             Cmdlet    Hyper-V                   Get-VMSwitchExtension...
Set-VMSwitchExtensionSwitchFea... Cmdlet    Hyper-V                   Set-VMSwitchExtensionSwitchFeature...
Add-VMSwitchExtensionPortFeature  Cmdlet    Hyper-V                   Add-VMSwitchExtensionPortFeature...
Add-VMSwitchTeamMember            Cmdlet    Hyper-V                   Add-VMSwitchTeamMember...
Get-VMSwitchTeam                  Cmdlet    Hyper-V                   Get-VMSwitchTeam...
Disable-VMSwitchExtension         Cmdlet    Hyper-V                   Disable-VMSwitchExtension...
Get-VMSwitchExtensionPortFeature  Cmdlet    Hyper-V                   Get-VMSwitchExtensionPortFeature...
Get-VMSwitchExtensionSwitchData   Cmdlet    Hyper-V                   Get-VMSwitchExtensionSwitchData...
-- More  --
```

Wild card help search

Looking through the list, you see several commands that have VMSwitch as the noun. Now, you have something to dig into further.



```
NAME
    Add-VMSwitch

SYNOPSIS
    Adds a virtual switch to an Ethernet resource pool.


SYNTAX
    Add-VMSwitch [-Name] <String[]> [-ResourcePoolName] <String[]> [-CimSession <CimSession[]>] [-ComputerName
    <String[]>] [-Credential <PSCredential[]>] [<CommonParameters>]

    Add-VMSwitch [-VMSwitch] <VMSwitch[]> [-ResourcePoolName] <String[]> [<CommonParameters>]


DESCRIPTION
    The Add-VMSwitch cmdlet adds a virtual switch to an Ethernet resource pool.


RELATED LINKS

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



PS C:\>
```

Getting help for a command

Hopefully, you recall that there is more help content that you can read. I often have to remind students in my classes to look at full help and examples.

```
Get-Help Add-VMSwitch -full
```

The *-Full* parameter will display all help, including parameter details and examples. If you only want to see examples, use the *-Examples* parameter.

```
help Get-Service -examples
```

Of you can get information on just a parameter:

```
help Get-Service -parameter computername
```

You can also use wildcards if you aren't confident about the parameter name.

```
help Get-Service -parameter *name
```

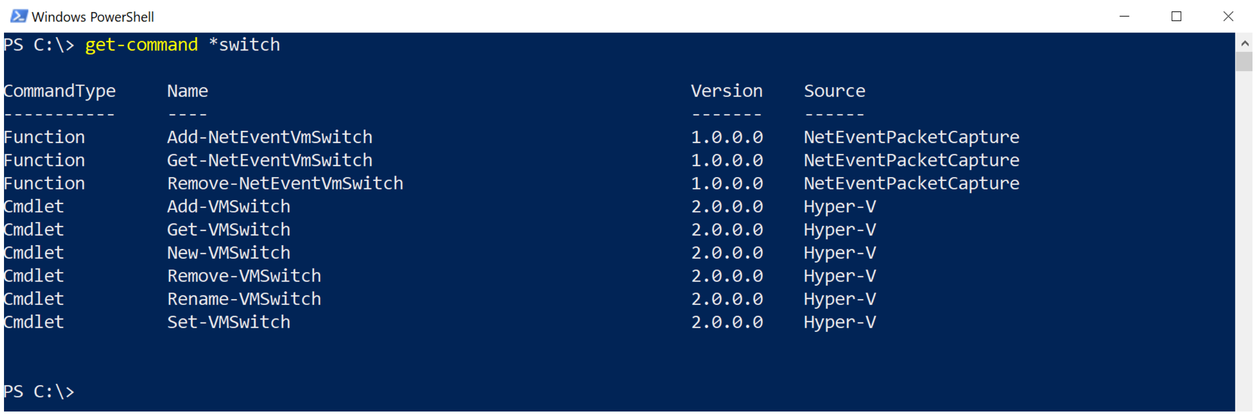
Finally, you can get the most up-to-date help online.

```
help Get-Eventlog -online
```

I can't stress enough the importance of getting in the habit of reading the help. For as long as I have been using PowerShell, I still discover useful tidbits in the help documentation. I have had situations where I've hit a snag, decide to look at the help, and find a solution right in the examples! The first time I looked at help I probably skimmed it, or it wasn't relevant so my brain didn't process it. But now, it jumps right out. I'm not saying this will happen to you, but PowerShell help offers a tremendous amount of information, don't forget the *about* topics that you simply can't afford to ignore.

Get-Command

Even though you can use PowerShell's help system to find commands, the `Get-Command` cmdlet might be a better choice. Revisiting my scenario above about finding commands to work with Hyper-V virtual switches. A simple wildcard search offers a lot of information.



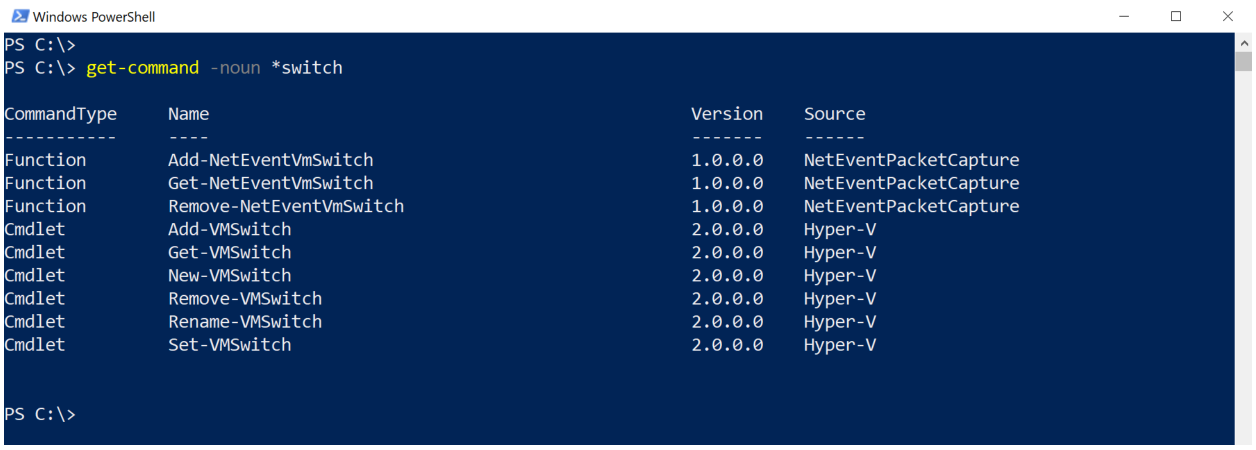
```
PS C:\> get-command *switch

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Add-NetEventVmSwitch                               1.0.0.0    NetEventPacketCapture
Function        Get-NetEventVmSwitch                               1.0.0.0    NetEventPacketCapture
Function        Remove-NetEventVmSwitch                            1.0.0.0    NetEventPacketCapture
Cmdlet          Add-VMSwitch                                       2.0.0.0    Hyper-V
Cmdlet          Get-VMSwitch                                       2.0.0.0    Hyper-V
Cmdlet          New-VMSwitch                                       2.0.0.0    Hyper-V
Cmdlet          Remove-VMSwitch                                    2.0.0.0    Hyper-V
Cmdlet          Rename-VMSwitch                                    2.0.0.0    Hyper-V
Cmdlet          Set-VMSwitch                                       2.0.0.0    Hyper-V


PS C:\>
```

Searching for commands

Or, you can take advantage of the `Get-Command` to find commands by noun.



```
PS C:\>
PS C:\> get-command -noun *switch

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Add-NetEventVmSwitch                               1.0.0.0    NetEventPacketCapture
Function        Get-NetEventVmSwitch                               1.0.0.0    NetEventPacketCapture
Function        Remove-NetEventVmSwitch                            1.0.0.0    NetEventPacketCapture
Cmdlet          Add-VMSwitch                                       2.0.0.0    Hyper-V
Cmdlet          Get-VMSwitch                                       2.0.0.0    Hyper-V
Cmdlet          New-VMSwitch                                       2.0.0.0    Hyper-V
Cmdlet          Remove-VMSwitch                                    2.0.0.0    Hyper-V
Cmdlet          Rename-VMSwitch                                    2.0.0.0    Hyper-V
Cmdlet          Set-VMSwitch                                       2.0.0.0    Hyper-V


PS C:\>
```

Searching for commands by noun

Once you've identified a command, you can read its help.

```
help add-vmswitch -full
```

If you haven't put `2+2` together yet, I encourage you to read full help and examples for `Get-Command`!

Get-Member

The last of the Big Trhee cmdlets is `Get-Member`. Because PowerShell is based on the idea of objects, you need to be able to discover how an object is defined. Don't assume that the output you see from running a command is all there is or that the headings you see are the actual property names. PowerShell tries to be helpful and presents default output that is easy to read and provides the most information that someone thought you might want to see. The `Get-Process` cmdlet is a great example. When you run the command by itself, PowerShell gives you a beautifully formatted table with nicely formatted values. But there is much more going on. Pipe `Get-Process` to `Get-Member` to see for yourself.

The property names are of the most importance to you. Once you know what they are, you can use them everywhere.

```
Get-Process |
Select-Object -property Name,ID,WS,Handles |
Sort-Object -property WS -Descending
```

`Get-Member` is a great troubleshooting tool as objects can change in the pipeline. Your first command in a pipelined expression might start with one type of object, but at the end, you can get something entirely different. If you don't get the result you expect, I often tell students to repeat the command, stripping off the last step and piping to `Get-Member` to verify what PowerShell is sending down the pipeline.

Terminology

Let's wrap up this refresher by taking a quick look at some PowerShell terms.

Variable

A *variable* is a container or placeholder for some other value or group of values. For the most part, the variable itself has no meaning. It is what's inside that counts. You reference variables by placing a `$` in front of the name.

Array

An *array* is a collection or group of objects. They don't have to be the same type, although they usually are. PowerShell will "unroll" an array when you access it. Or you can reference individual items in an array by their index number, starting at `0`. Or you can start at the end with `-1`.

```
PS C:\> $a = "alice","bob","carol","david"
PS C:\> $a
alice
bob
carol
david
PS C:\> $a[1]
bob
PS C:\>$a[-2]
carol
```

Do you think you can find a help topic on arrays?

Hashtable

A *hashtable* is a key/value pair. In VBScript, we called this a dictionary object. Hashtables are used extensively in PowerShell. Once the hashtable is defined, you can access values by using the key as property. You can add and subtract entries in a hashtable or change the value of a given key. My favorite trick with hashtables is that you can turn them into objects that get written to the PowerShell pipeline.

```
PS C:\> $h = @{Name="Jeff"
>> Size = 123
>> Computername = $env:computername
>> Version = $psversiontable.PSVersion
>> }
PS C:\> $h

Name                           Value
----                           -----
Computername                   VALKYRIE-1
Name                           Jeff
Version                        5.1.19041.1
Size                           123


PS C:\> $h.size
123
PS C:\> $h.size = 456
PS C:\> $h.size
456
PS C:\> new-object -TypeName psobject -Property $h
```

```
Computername Name Version       Size
------------ ---- -------       ----
VALKYRIE-1    Jeff 5.1.19041.1   456

PS C:\>
```

Cmdlet

A *cmdlet* is PowerShell's core unit of functionality. This is almost always a compiled piece of code that is designed to do one small thing. You can often figure out that thing based on its name, since all cmdlets follow a standard *Verb-Noun* naming convention. You can use `Get-Command` to discover what cmdlets are available on your computer and `Get-Help` to learn how to use them.

Function

A *function* is another type of command, often written in PowerShell's scripting language. Most of the functions you will encounter in PowerShell are indistinguishable from cmdlets. They should follow the same naming convention, work with objects in the pipeline, have help documentation and be just as discoverable. Eventually, you will learn to write PowerShell functions.

Parameter

A *parameter* customizes or changes the behavior of a cmdlet or function. In PowerShell, when using a parameter in an expression, the parameter name is preceded by a dash. The value is separated by a space. Some parameters are positional, meaning you don't have to type them. For example, you can run this command:

```
Get-Service winrm -ComputerName $env:computername
```

The value of "winrm" is technically the value for the `Name` parameter, which is positional. PowerShell assumes that whatever you type immediately after `Get-Service` is the name of a service. But the `Computername` parameter is not positional, so to use it, you have to include the parameter name. Don't view parameter names as an obstacle. If you get in the habit of using tab completion, you won't even think about it.

Module

A *module* is a package of related cmdlets and functions. Many of the modules you use are from Microsoft. But they can come from other vendors such as VMware or members of the PowerShell community. Most often, they are installed from the PowerShell Gallery. As your PowerShell skills grow, you'll eventually learn to create your functions and modules. Since a module is a common "thing" in PowerShell, I bet you can find some commands to manage modules.

Summary

I’m assuming that you already have some basic PowerShell skills and experience. I wanted this chapter to serve as a refresher, in case you’ve set it aside for awhile. And if you are a savvy reader, you’ll realize that if I find these things important, I’m probably going to make sure *you* know them as well. There might even be a related question or two coming up.

Mastering PowerShell, or even gaining some degree of proficiency, means using it every day. Reading, re-reading, doing, and re-doing all combine bit by bit to make you a better PowerShell professional.

Part 1 - PowerShell Principles

Any journey towards mastering PowerShell has to begin with basic principals and concepts. You need to know how to use the help system. You need to know how the pipeline works on a very basic level. You need to know how to filter and work with object output. One of PowerShell's assets is that once you learn how to sort, it doesn't matter if you are sorting a number, a file, or an Active Directory user object. The commands are the same..

The exercises in this section aim to help you solidify your understanding of essential PowerShell principles and concepts.

Part 1 Exercises

Exercise 1

Get all services where the display name begins with ‘Windows’.

Hint

Service objects have several *name* properties.

Work Area and Notes

Exercise 2

Get a list of all classic event logs on your computer.

Work Area and Notes

Exercise 3

Find and display all of the commands on your computer that start with ‘Remove’.

Hint

Remember that cmdlets have a `Verb-Noun` naming convention.

Work Area and Notes

Exercise 4

What PowerShell command would you use to reboot one or more remote computers?

Hint

Think about what standard PowerShell verb might be used in the command.

Work Area and Notes

Exercise 5

How would you display all available modules installed on your computer?

Work Area and Notes

Exercise 6

How would you restart the BITS service on your computer *and* see the result?

Hint

Not all commands write to the pipeline by default.

Work Area and Notes

Exercise 7

List all the files in the `%TEMP%` directory and all subdirectories.

Hint

You can get values from the ENV PSDrive like this: `$env:name`.

Work Area and Notes

Exercise 8

Display the access control list (ACL) for Notepad.exe.

Hint

You might consider displaying it as a list to make it easier to read.

Work Area and Notes

Exercise 9

How could you learn more about regular expressions in PowerShell?

Hint

There is more to PowerShell help than cmdlet help.

Work Area and Notes

Exercise 10

Get the last 10 error entries from the System event log on your computer.

Work Area and Notes

Exercise 11

Show all of the ‘get’ commands in the PSReadline module.

Hint

The focus is on the commands, not the module.

Work Area and Notes

Exercise 12

Display the installed version of PowerShell.

Hint

There isn’t a cmdlet that you can run.

Work Area and Notes

Exercise 13

How would you start a new instance of Windows PowerShell without loading any profile scripts?

Hint

You need to know the PowerShell executable name.

Work Area and Notes

Exercise 14

How many aliases are defined in your current PowerShell session?

Hint

You will want to measure some objects.

Work Area and Notes

Exercise 15

List all processes on your computer that have a working set size greater than or equal to 50MB and sort by working set size in descending order.

Hint

Break this down into separate steps.

Work Area and Notes

Exercise 16

List all files in `%TEMP%` that were modified in the last 24 hours and display the full file name, its size, and the time it was last modified. Write a PowerShell expression that doesn't rely on hard-coded values.

Hint

You might find this easier to do in several steps, such as first calculating the datetime value.

Work Area and Notes

Exercise 17

Get all files in your Documents folder that are at least 1MB in size and older than 90 days. Export the full file name, size, creation date, and last modified date to a CSV file. You may have to adjust the exercise based on files you have available.

Hint

Don't feel you need to do this as a one-line command. Break this into steps.

Work Area and Notes

Exercise 18

Using files in your `%TEMP%` folder, display the total number of each files by their extension in descending order.

Hint

You will need to group your results and remember that objects can change in the pipeline.

Work Area and Notes

Exercise 19

Create an XML file of all processes running under your credentials.

Hint

Export filtered objects to an XML file that you could re-import into PowerShell.

Work Area and Notes

Exercise 20

Using the XML file, you created in the previous question, import the XML data into your PowerShell session and produce a formatted table report with processes grouped by the associated company name.

Hint

`Get-Member` is your friend.

Work Area and Notes

Exercise 21

Get 10 random numbers between 1 and 50 and multiply each number by itself.

Hint

You'll have to do something *for* a certain number of times.

Work Area and Notes

Exercise 22

Get a list of event logs on the local computer and create an HTML file that includes ‘Computername’ as a heading. You can decide if you want to rename other headings to match the original cmdlet output once you have a solution working.

Hint

You’ll need to reverse-engineer property names and be specific about what you want to convert. You’ll also need to re-calculate the entry count.

Work Area and Notes

Exercise 23

Get modules in the PowerShell Gallery that are related to teaching.

Work Area and Notes

Exercise 24

Get all running services on the local machine and export the data to a JSON file. Omit the required and dependent services. Verify by re-importing the JSON file.

Hint

You will need to exclude when you select properties.

Work Area and Notes

Exercise 25

Test the local computer to see if port 80 is open.

Hint

If you think about it, port 80 is related to a network connection.

Work Area and Notes

Suggested solutions are in the next chapter.

Part 1 Solutions

Exercise 1

Get all services where the display name begins with 'Windows'

Solution

```
Get-Service -displayname Windows*
```

Comments

Wildcards can make your commands easy to use. You could also have used `Win*` to save a little typing.

Exercise 2

Get a list of all classic event logs on your computer.

Solution

```
Get-Eventlog -List

#or

Get-WinEvent -listlog * | Where-object {$_.IsClassicLog}
```

Comments

This is a problem where you need to read the help and examples. In Windows PowerShell, you can use `Get-Eventlog`. However, this cmdlet is not in PowerShell 7.x. Instead, you used `Get-WinEvent` to work with event logs. You could have piped the first part of the expression to `Get-Member` to discover the `IsClassicLog` property.

Exercise 3

Find and display all of the commands on your computer that start with Remove.

Solution

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

Comments

You could have also retrieved names by wild card: `Get-Command -name Remove-*`.

This syntax produces results for all command types. If you wanted to limit the result to cmdlets only, you could use an expression like `Get-Command -commandtype cmdlet -name Remove-*`.

Exercise 4

What PowerShell command would you use to reboot one or more remote computers?

Solution

```
Restart-Computer
```

Comments

PowerShell's Verb-Noun naming convention should make it easier to discover commands.

Exercise 5

How would you display all available modules installed on your computer?

Solution

```
Get-Module -ListAvailable
```

Comments

`Get-Module` will only show you modules that have been imported to your PowerShell session. You need to use the `-ListAvailable` parameter to force PowerShell to check all locations for available modules.

Exercise 6

How would you restart the BITS service on your computer and see the result?

Solution

```
Restart-Service bits -passthru
```

Comments

When you see `Passthru` on a cmdlet, that is a clue that the command does not write anything to the pipeline by default.

Exercise 7

List all the files in the `%TEMP%` directory and all subdirectories.

Solution

```
Get-ChildItem -path $env:temp -file -recurse
```

Comments

Using `-File` will only return files. You could have omitted it to get a more traditional listing. If you know in advance the path to your `%TEMP%` folder, you could have manually typed it out.

Exercise 8

Display the access control list (ACL) for Notepad.exe

Solution

```
Get-Acl -path C:\windows\notepad.exe | Format-List
```

Comments

Without `Format-List`, it is a bit difficult to read the output. As an alternative, you could use the pipeline and `Select-Object`.

```
Get-Acl -path C:\Windows\notepad.exe | Select-Object -expandproperty access
```

Or get the string version:

```
Get-Acl C:\windows\notepad.exe | Select-Object -expandproperty accesstostring
```

Exercise 9

How could you learn more about regular expressions in PowerShell?

Solution

```
Help about_regular_expressions
```

Comments

There is no -online support for the about help topics.

Exercise 10

Get the last 10 error entries from the System event log on your computer.

Solution

```
Get-EventLog -LogName System -Newest 10 -EntryType Error

#or

Get-WinEvent -FilterHashtable @{Logname="System";Level=2} -MaxEvents 10
```

Comments

If you don't get results from the System log, you can try the Application event log. You might also try getting errors and warnings. Assuming you have enough events, how many do you get? Why do you think you get the result that you do?

In PowerShell 7.x, you have to use the slightly more complicated `Get-WinEvent`. But it has better filtering capabilities and seems to perform better. The tricky part is figuring out the filtering. It helps to pipe an event record to `Select-Object -property *` to discover property names and values.

Exercise 11

Show all of the 'get' commands in the PSReadline module

Solution

```
Get-Command -Module PSReadline -Verb get
```

Comments

You and also use this technique to find commands by Noun if you want to give it a try.

Exercise 12

Display the installed version of PowerShell

Solution

```
$PSVersionTable
```

Comments

This is a built-in and automatic variable that provides version information on multiple items.

Exercise 13

How would you start a new instance of PowerShell without loading any profile scripts?

Solution

```
powershell -noprofile
```

Comments

This is a handy way to launch a new, clean PowerShell session quickly. Type `exit` to end the session and return to your original session.

Exercise 14

How many aliases are defined in your PowerShell session?

Solution

```
Get-Alias | Measure-Object
```

Comments

The suggested solution would suffice. But you could get more granular with a command like:

```
Get-Alias | Measure-Object | Select-Object -ExpandProperty count
```

Exercise 15

List all processes on your computer that have a working set size greater than or equal to 50MB and sort by working set size in descending order.

Solution

```
Get-Process | Where-Object {$_.WorkingSet -ge 50MB} |
Sort-Object -Property workingset -Descending
```

Comments

If you visualize what you need to accomplish because PowerShell command names are intuitive, it shouldn't be too difficult. Learn about the unit shortcuts like MB, KB, and GB. They can simplify your PowerShell work.

Exercise 16

List all files in `%TEMP%` *that were modified in the last 24 hours and display the full file name, its size, and the time it was last modified. Write a command that doesn't rely on hard-coded values.*

Solution

```
$cut = (Get-Date).AddHours(-24)
Get-ChildItem -Path $env:TEMP -file -Recurse |
Where-Object {$_.LastWriteTime -ge $cut} |
Select-Object -Property Fullname,Length,LastWriteTime
```

Comments

You can always nest expressions, such as putting the `Get-Date` command inside the `Where-Object` filter, but it isn't required. Sometimes multiple steps are easier to develop than a single, monolithic and complex pipelined expression.

Exercise 17

Get all files in your Documents folder that are at least 1MB in size and older than 90 days. Export the full file name, size, creation date, and last modified date to a CSV file. You may have to adjust the exercise based on files you have available.

Solution

```
$cut = (Get-Date).AddDays(-90)
$files = Get-ChildItem -Path C:\users\jeff\Documents -file -Recurse |
Where-Object {$_.length -ge 1MB -AND $_.LastWriteTime -le $cut}
$files | Select Fullname,Length,CreationTime,LastWriteTime |
Export-CSV -Path myfiles.csv
```

Comments

Your filtering scriptblock can be as complex as necessary. Notice our use of the -AND operator. When exporting objects, you need to be specific about what properties you want to serialize.

Exercise 18

Using files in your %TEMP% folder, display the total number of each files by their extension in descending order.

Solution

```
Get-ChildItem -Path $env:temp -file -Recurse |
Group-Object -Property Extension -NoElement |
Sort-Object -property Count -Descending
```

Comments

Since we don't care about the actual files, we can suppress them with the `NoElement` parameter of `Group-Object`. Notice that even though the first command writes file objects to the pipeline, `Group-Object` writes a different type of object, which can be piped to `Sort-Object`.

Exercise 19

Create an XML file of all processes running under your credentials.

Solution

```
Get-Process -IncludeUserName |
where-object {$_.Username -eq "$($env:USERDOMAIN)\$($env:USERNAME)" } |
Export-Clixml -Path myprocs.xml
```

Comments

You could hard-code your credentials. This solution creates a name from environment variables. You could also have invoked the native `whoami` command and used the result:

```
Get-Process -IncludeUserName | Where-Object {$_.Username -eq (whoami)} |
Export-Clixml -Path myprocs.xml
```

Exercise 20

Using the XML file you created in the previous question, import the XML data into your PowerShell session and produce a formatted table report with processes grouped by the associated company name.

Solution

```
Import-Clixml .\myprocs.xml | Sort-Object -property Company |
Format-Table -groupby company
```

Comments

You need to sort all the objects before your format your results as a table.

Exercise 21

Get 10 random numbers between 1 and 50 and multiply each number by itself.

Solution

```
for ($i=1;$i -le 10;$i++) {
  $x = Get-Random -Maximum 50 -Minimum 1
  $x * $x
}
```

Comments

There are several ways you could have accomplished this goal. Using the For loop might be the most explicit, but you could also use the range operator to get the numbers 1 to 10 and do something for each number.

```
1..10 | foreach {
 $x = Get-Random -Maximum 50 -Minimum 1
 $x * $x
 }
```

Exercise 22

Get a list of event logs on the local computer and create an HTML file that includes the computer name as a heading. You can decide if you want to rename the headings to match once you have a solution working.

Solution

```
Get-EventLog -list |
Select-Object -Property MaximumKilobytes,MinimumRetentionDays,
@{Name="Count";Expression={$_.entries.count}},LogDisplayname |
ConvertTo-Html -PreContent "<H1>$($env:computername)</H1>" `
-title "Event Log Report" |
Out-File -FilePath Eventlog.html
```

Comments

Using custom hashtables with `Select-Object` is a great way to define properties that meet your needs. Ideally, you would also want to include a CSS value to make the resulting HTML file pretty.

In PowerShell 7, you would need to use `Get-WinEvent`, which has different property names. You might use an expression like this:

```
Get-WinEvent -listlog * |
Select-Object -Property LogName,LogMode,RecordCount,
@{Name="MaxSizeMB";Expression={$_.MaximumSizeInBytes/1mb -as [int]}} |
ConvertTo-Html -PreContent "<H1>$($env:computername)</H1>" `
-title "Event Log Report" |
Out-File -FilePath Eventlog.html
```

I had to use the backtick to break a long line to fit the printed page better. You wouldn't need to do that.

Exercise 23

Get modules in the PowerShell Gallery that are related to teaching.

Solution

```
Find-Module -Tag teaching -Repository PSGallery
```

Comments

You might get prompted to update the Nuget provider. You can also find modules by name, which is helpful because you can use wildcards. Unfortunately, tags do not support wildcards.

Exercise 24

Get all running services on the local machine and export the data to a JSON file. Omit the required and dependent services. Verify by re-importing the JSON file.

Solution

```
Get-Service | Where-Object {$_.status -eq 'running'} |
Select-Object -Property * -ExcludeProperty *Services* |
ConvertTo-Json | Set-Content -path running.json

Get-Content -path running.json | ConvertFrom-Json
```

Comments

`ConvertTo-Json` doesn't create the file on its own. You could also have used `Out-File`.

Exercise 25

Test the local computer to see if port 80 is open.

Solution

```
Test-NetConnection -ComputerName localhost -CommonTCPPort HTTP
```

Comments

If you want to suppress the warning message, include the common `-WarningAction` parameter and set the value to `SilentlyContinue`.

Part 2 - PowerShell Providers

Another useful PowerShell feature is its use of *providers*. A PowerShell provider is piece of compiled .NET code. The provider exposes different technologies or services to PowerShell. Almost always, this exposure is through a *PSDrive*. It is this concept of a provider that makes it possible to learn one command, like `Get-ChildItem` that can work in the file system and the registry with very minor differences.

The exercises in this section are devoted to ensuring you understand how to find what PowerShell providers are available and how to use them. You should also expect practice problems to continue building your proficiency with core PowerShell commands and concepts.

Part 2 Exercises

Exercise 1

Assuming you haven't modified your PowerShell session with a profile script, what are the default PSDrives for the Registry provider?

Hint

You could get PSProvider information or PSDrives.

Work Area and Notes

Exercise 2

How many certificates are installed in the root certificate store for the local machine?

Hint

You can treat certificates like files.

Work Area and Notes

Exercise 3

Query the local registry to display the registered owner and organization.

Hint

You might want to find the correct keys using `regedit.exe`.

Work Area and Notes

Exercise 4

How many functions are defined in your current PowerShell session?

Hint

You can list functions just like files.

Work Area and Notes

Exercise 5

List all applications installed under the Uninstall section of the registry. Give yourself a challenge and filter out those with a GUID for a name.

Hint

Use `regedit.exe` to find the correct path.

Work Area and Notes

Exercise 6

Modify the registered organization value in the registry. Verify the change. Then go ahead and change it back to the original value.

Hint

Save the current value to a variable.

Work Area and Notes

Exercise 7

What PSProvider supports transactions?

Hint

You should know how to find provider details by now.

Work Area and Notes

Exercise 8

How would you find code-signing certificates installed on your computer?

Hint

Some cmdlet help is provider aware.

Work Area and Notes

Exercise 9

Turn `%PATH%` into a list of directories.

Hint

You will need to split an environmental variable.

Work Area and Notes

Exercise 10

Create a new registry key under HKEY_CURRENT_USER called `PowerShell Training`. Then create values under it for your name, computer name, the current date, and PowerShell version. You should be able to get all of these values from PowerShell.

Hint

Remember that registry keys are items and values are item properties.

Work Area and Notes

Exercise 11

Using PowerShell, delete the `PowerShell Training` registry setting you created in the previous exercise.

Work Area and Notes

Exercise 12

Create a PSDrive called `Download` for the Downloads directory under your user directory.

Work Area and Notes

Exercise 13

Get all functions that **don't** support cmdletbinding.

Work Area and Notes

Exercise 14

Get the default WSMan port values.

Work Area and Notes

Exercise 15

Set the Digest Authentication setting for WSMan to $False. If it is already False, then set it to True. Revert the change if you need to.

Work Area and Notes

Exercise 16

Create a new environment variable in PowerShell called `Pictures` that points to your Pictures folder. Does this setting persist?

Work Area and Notes

Exercise 17

Make a **persistent** environmental variable called Pictures that points to your Pictures folder. Verify it in PowerShell.

Hint

You can define environment settings in the registry for just you.

Work Area and Notes

Exercise 18

Create a backup copy of your user environmental variables found in the registry to EnvBackup.

Hint

You cannot copy items across providers, so stick to the current location.

Work Area and Notes

Exercise 19

Delete the persistent Pictures environmental variable you created earlier and recreate it using `%USERPROFILE%` as an expandable variable.

Work Area and Notes

Exercise 20

Export your user-specific persistent environment settings to a CSV file that you can use outside of PowerShell.

Hint

You don't need any type information or PowerShell-specific values.

Work Area and Notes

Exercise 21

List all expired certificates showing the certificate's friendly name, when it expired, the issuer, and path.

Hint

There is provider-specific help that will assist you.

Work Area and Notes

Exercise 22

Create a hashtable of all your PSDrives grouped on the provider name.

Hint

The provider property is a nested object.

Work Area and Notes

Exercise 23

Create the folder structure `a\b\c\d\e\f` under your `%TEMP%` directory.

Work Area and Notes

Exercise 24

Create the file `data.txt` in the folder `%temp%\foo\bar\xyz`.

Hint

You should know how to create new items by now.

Work Area and Notes

Exercise 25

Using transactions, create the registry key `PSPrimer\Data` under `HKEY_LOCAL_MACHINE\Software`. Then create a string setting called `version` with a value of 1. Next, create a dword setting called `random` with a value of a random number between 10 and 99. Finally, create a value called `free` with the amount of free space on your C drive. If you are using PowerShell 7, create the registry items without using transactions.

Hint

You may need to read about transactions. You can get free space with `Get-PSDrive`. The solution will be at least 5 lines of PowerShell.

Work Area and Notes

Suggested solutions are in the next chapter.

Part 2 Solutions

Exercise 1

Assuming you haven't modified your PowerShell session with a profile script, what are the default PSDrives for the Registry provider?

Solution

```
HKLM and HCKU
```

Comments

You will see drive assignments by running `Get-PSProvider`, or you could have run `Get-PSDrive`. If you read the help, you'd also see you could narrow this down like this: `Get-PSDrive -PSProvider Registry`.

Exercise 2

How many certificates are installed in the root certificate store for the local machine?

Solution

```
Get-ChildItem Cert:\LocalMachine\root | Measure-Object
```

Comments

You can use the same file system navigation cmdlets with any PSDrive.

Exercise 3

Query the local registry to display the registered owner and organization.

Solution

```
Get-ItemPropertyValue -path 'HKLM:\SOFTWARE\Microsoft\Windows NT\`
CurrentVersion' -Name registeredowner,registeredorganization
```

Comments

Remember that you don't need to use the backtick.

Registry keys have item properties. You could have also used an expression like:

```
Get-ItemProperty 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion' |
Select-Object -property Registered*
```

Exercise 4

How many functions are defined in your current PowerShell session?

Solution

```
dir function: | measure
```

Comments

The `Function:` PSDrive exposes a lot of useful information. You could also have done this:

```
(dir function:).count
```

Exercise 5

List all applications installed under the Uninstall section of the registry. Give yourself a challenge and filter out those with a GUID for a name.

Solution

```
dir HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall |
Select-object -property PSChildName
```

Comments

If you had piped one of the entries to `Get-Member`, you would have discovered that PSChildName gives cleaner output. To filter out GUID-based names, you could use something like this:

```
dir HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall |
Where-Object {$_.pschildname -notmatch "^{"} |
Select-object -property PSChildname
```

Exercise 6

Modify the registered organization value in the registry. Verify the change. Then go ahead and change it back to the original value.

Solution

```
$ro = Get-ItemPropertyValue -path 'HKLM:\SOFTWARE\Microsoft\Windows NT\`
CurrentVersion' -Name RegisteredOrganization
Set-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion'`
 -Name RegisteredOrganization -Value PSPrimer -PassThru
Set-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion'`
 -Name RegisteredOrganization -Value $ro -PassThru
```

Comments

Use `-Passthru` for cmdlets that don't write to the pipeline by default. The `Get-ItemPropertyValue` makes it easier to get just the value so you can re-use it.

Exercise 7

What PSProvider supports transactions?

Solution

```
Registry
```

Comments

In a default PowerShell session, the Registry is the only PSProvider that supports transactions which you can see with `Get-PSProvider`. Or you could have filtered like this:

```
Get-PSProvider | Where-Object {$_.Capabilities -match "transactions"}
```

If you were trying this in PowerShell 7.x, you most likely found no providers. It appears this feature is gone in PowerShell 7.x.

Exercise 8

How would you find code-signing certificates installed on your computer?

Solution

```
dir Cert:\CurrentUser -Recurse -CodeSigningCert
```

Comments

If you run this command from the C: drive, you won’t get a result.

```
help dir -param code*
```

But specify a path (change location) and see what happens!

```
help dir -param code* -path Cert:
```

Exercise 9

Turn %PATH% into a list of directories.

Solution

```
$env:path -split ";"
```

Comments

It is easiest to reference any environmental variable using syntax like `$env:computername` or `$env:windir`. On non-Windows machines in PowerShell 7, or to use code that works cross-platform, you will need to use a .NET Framework reference.

```
$env:PATH -split [System.IO.Path]::PathSeparator
```

Exercise 10

Create a new registry key under HKEY_CURRENT_USER called `PowerShell Training`*. Then create values under it for your name, computer name, the current date, and PowerShell version. You should be able to get all of these values from PowerShell.*

Solution

```
New-Item -Path HKCU: -Name 'PowerShell Training'
Set-ItemProperty -Path 'HKCU:\PowerShell Training' -Name Name `
-Value $env:USERNAME
Set-ItemProperty -Path 'HKCU:\PowerShell Training' -Name Computername `
-Value $env:COMPUTERNAME
Set-ItemProperty -Path 'HKCU:\PowerShell Training' -Name Date `
-Value (Get-Date).ToShortDateString()
Get-item 'HKCU:\PowerShell Training'
```

Comments

Look at help and examples for `Set-ItemProperty` to see how to set other registry values like DWORD.

Exercise 11

Using PowerShell, delete the PowerShell Training registry setting you created in the previous exercise.

Solution

```
Remove-Item 'HKCU:\PowerShell Training'
```

Comments

If there had been child keys, you might have needed to use `-Recurse` and `-Force`.

Exercise 12

Create a PSDrive called Download for the Downloads directory under your user directory.

Solution

```
New-PSDrive -Name Download -PSProvider FileSystem `
-Root $env:userprofile\downloads
```

Comments

Put a command like this in your profile to always have this drive “mapping”.

Exercise 13

*Get all functions that **don’t** support cmdletbinding.*

Solution

```
Get-ChildItem function: | where-object {-not $_.CmdletBinding}
```

Comments

If you pipe an item from the `Function:` PSDrive to `Get-Member`, you would have discovered a property to use.

Exercise 14

Get the default WSMan port values.

Solution

```
Get-ChildItem WSMan:\localhost\Service\DefaultPorts |
Select-Object -property Name,Value
```

Comments

You can get single values like this:

```
Get-ChildItem WSMan:\localhost\Service\DefaultPorts\HTTP
```

Exercise 15

Set the Digest authentication setting for WSMan to $False. If it is already False, then set it to True. Revert the change if you need to.

Solution

```
Set-Item -Path WSMan:\localhost\Client\Auth\Digest -Value $false
```

Comments

An alternative to modifying settings using the `WSMan:` PSDrive is to use one of the WSManInstance cmdlets:

```
Set-WSManInstance -ResourceURI winrm/config/client/auth -ValueSet `
@{Digest=$False}
```

This is an advanced topic that I wouldn't expect you to discover.

Exercise 16

Create a new environment variable in PowerShell called Pictures that points to your Pictures folder. Does this setting persist?

Solution

```
New-Item -Path env: -Name Pictures -Value $env:userprofile\pictures
```

Comments

Any change you make to the `ENV:` PSDrive are not persistent and only affect your current PowerShell session. To make something permanent go to the next exercise.

Exercise 17

Make a persistent environment variable called Pictures that points to your Pictures folder. Verify it in PowerShell.

Solution

```
New-ItemProperty -Path HKCU:\Environment -Name Pictures -Value `
$env:userprofile\pictures
```

Comments

You won't see the new setting until you restart PowerShell. This problem assumes you can discover where environment settings are stored in the Registry.

Exercise 18

Create a backup copy of your user environmental variables found in the registry to EnvBackup.

Solution

```
Copy-Item HKCU:\Environment -Destination HKCU:\EnvBackup -PassThru
```

Comments

This location has no child items, so there was no need to use -Recurse or -Container.

Exercise 19

Delete the persistent Pictures environmental variable you created earlier and recreate it using `%USERPROFILE%` *as an expandable variable.*

Solution

```
Remove-ItemProperty -Path HKCU:\Environment -Name Pictures
New-ItemProperty -Path HKCU:\Environment -Name Pictures -Value
'%USERPROFILE%\Pictures' -PropertyType ExpandString
```

Comments

You may need to use `regedit.exe` to verify the use of `%USERPROFILE%` as PowerShell will automatically expand it.

Exercise 20

Export your user-specific persistent environment settings to a CSV file that you can use outside of PowerShell.

Solution

```
Get-ItemProperty -Path HKCU:\Environment | Select * -exclude PS* |
Export-Csv -Path c:\MyEnv.csv -NoTypeInformation
```

Comments

The suggested answer is specifically excluding properties like PSPath and PSProvider. If you had a setting like `PSFoo` that would get excluded.

Exercise 21

List all expired certificates showing the certificate's friendly name when it expired, the issuer and path

Solution

```
dir Cert:\ -Recurse -ExpiringInDays 0 |
Select-Object -property FriendlyName,NotAfter,Issuer,Subject,
@{Name="Path";Expression={Join-Path -path CERT: `
-childpath (Convert-Path $_.pspath)}}
```

Comments

You might be tempted to filter with `Where-Object` on the `NotAfter` property, but `Get-ChildItem` has a provider-specific parameter that makes this easier. You could have gotten by with PSPath, but I wanted something a bit friendlier.

Exercise 22

Create a hashtable of all your PSDrives grouped on the provider name.

Solution

```
$h = Get-PSDrive | Group-Object {$_.provider.name} -AsHashTable -AsString
$h.FileSystem
```

Comments

You could have grouped simply on the Provider property, but that would have made it more difficult to access hashtable properties. If you read help for `Group-Object`, you would have seen at least one example grouping on a custom property.

Exercise 23

Create the folder structure a\b\c\d\e\f under your %TEMP% directory.

Solution

```
New-Item -Path "$env:temp\a\b\c\d\e\f" -ItemType Directory
```

Comments

PowerShell will create the entire directory tree, but you need to specify that you are creating a directory.

Exercise 24

Create the file data.txt in the folder `%temp%\foo\bar\xyz`.

Solution

```
New-Item -Path $env:temp\foo\bar\xyz -ItemType Directory
New-Item -Path $env:temp\foo\bar\xyz\data.txt -ItemType File
```

Comments

You can only create a new file in a directory structure that already exists.

Exercise 25

_Using transactions, create the registry key `PSPrimer\Data` under `HKEY_LOCAL_MACHINE\Software`. Create a string setting called `version` with a value of 1. Create a dword setting called random with a value of a random number between 10 and 99. Create a value called `free` with the amount of free space on your C drive._If you are using PowerShell 7, create the registry items without using transactions.

Solution

```
Start-Transaction
New-Item -path HKLM:\SOFTWARE -name 'PSPrimer' -UseTransaction
New-Item -path HKLM:\SOFTWARE\PSPrimer -name 'Data' -UseTransaction
New-ItemProperty -Path HKLM:\SOFTWARE\PSPrimer\Data -Name version -Value 1 `
-PropertyType String -UseTransaction
New-ItemProperty -Path HKLM:\SOFTWARE\PSPrimer\Data -Name random -Value `
(Get-Random -Minimum 10 -Maximum 99) -PropertyType dword -UseTransaction
New-ItemProperty -Path HKLM:\SOFTWARE\PSPrimer\Data -Name free -Value `
(Get-PSDrive C).Free -UseTransaction
Complete-Transaction
```

Comments

It is interesting to note that *sometimes* the registry PSProvider can autodetect the right registry type. To be safe, it doesn't hurt to specify the property type. If you want to delete the entry from your PowerShell console, you can probably figure it out, or I'll give your brain a rest:

```
Remove-Item -Path  HKLM:\SOFTWARE\PSPrimer\ -Recurse
```

Sadly, in PowerShell 7.x, you can no longer use transactions.

Part 3 - PowerShell Structures

Whether you are working with PowerShell interactively in a console, or you have started writing simple PowerShell scripts, there are some common PowerShell structures and operators that you need to understand. Working with arrays, hashtables, and switches is the foundation of your PowerShell expertise.

In this section, solutions might require a bit more than a one-line answer, although you still don't need to write scripts. Hopefully, by the end, you'll have a better grasp of these critical concepts.

Part 3 Exercises

Exercise 1

Create an array of the numbers 1 to 20 and then get the 5th element of the array.

Hint

You may want to read help about arrays.

Work Area and Notes

Exercise 2

Initialize an empty array. Add 10 random numbers between 10 and 100. Finally, get the sum total of all numbers in the array.

Hint

You may want to get some help about arrays.

Work Area and Notes

Exercise 3

Create a hashtable with keys for your computer name, PowerShell version, and the current date (without the time). Don't use any hardcoded values. Save the hashtable to a variable.

Hint

You might want to get some help about hashtables.

Work Area and Notes

Exercise 4

Using the hashtable from the previous exercise, add a key for Name using the values of Bits and WinRM. Remove the Date and PowerShell version keys. Finally, splat the hashtable to `Get-Service`.

Hint

You will be adding an array of values.

Work Area and Notes

Exercise 5

Create an ordered hashtable with keys for your computer name, all the running processes, the top 5 processes using the most working set, and the total size of your `%TEMP%` folder in bytes. Complete the exercise by creating a custom object from the hashtable.

Hint

You'll need one-line pipelined expressions to get the values of some of these keys.

Work Area and Notes

Exercise 6

Recreate the hashtable from the previous exercise, but this time create a custom object at the same time.

Hint

You will need a type accelerator.

Work Area and Notes

Exercise 7

Get all event logs on your computer that have entries and show the log name, maximum size, the total number of entries, and a property called Computername to reflect the computer name.

Hint

Run the initial command with your computer name. You'll need need to select the properties you want.

Work Area and Notes

Exercise 8

Recursively go through all files in `%TEMP%`, or a directory of your choice. The result of your PowerShell expression should show the number of files per file extension and their total size in bytes.

Hint

You will want to process each group of extensions and select the properties to display.

Work Area and Notes

Exercise 9

Create an array of letters A through E. Get a random item from the array and assign it to $X. Then create a PowerShell construct that displays "alpha","bravo","charlie","delta","echo" based on the value of X.

Hint

This will be easier to write an answer in the PowerShell ISE or VS Code. You may need to switch things up a bit.

Work Area and Notes

Exercise 10

Take the string "PowerShell Forever" and display the characters in alphabetic order.

Hint

A string is an array of characters that you can join together any way you want.

Work Area and Notes

Exercise 11

Create a hashtable of services based on their startup type.

Hint

You can create this with a cmdlet.

Work Area and Notes

Exercise 12

Using the hashtable you created in the previous exercise, get the display name of the last disabled service.

Hint

You can use `object.property` notation to drill down.

Work Area and Notes

Exercise 13

Get all the even numbers between 1 and 50.

Hint

You'll need to test each number with a modulo expression.

Work Area and Notes

Exercise 14

Starting with `$i` equal to 0, loop 5 times, each time incrementing `$i` by 5. At the end, does `$i` equal 25?

Work Area and Notes

Exercise 15

Display the top 25 processes that have been running the longest. Include the process ID, process name, start time, how long it has been running, and the path.

Hint

Not every process has a start time, so you'll need to filter those out. Not every process will have a path, and you can calculate a runtime by subtracting dates.

Work Area and Notes

Exercise 16

Get 25 random numbers between 10 and 100. Multiply each one by 3, then get the total sum of these numbers, their average, the largest number, and the smallest.

Hint

Get your random numbers from a range.

Work Area and Notes

Exercise 17

Create these aliases:

- np (notepad.exe)
- tx (tzutil.exe)
- ct (control.exe)

Export them to a JSON file. Delete the new aliases you just created and recreate them from the JSON file.

Hint

There are no alias-specific commands for removing, so you'll have to find another way. And there is a quirk with converting from JSON in Windows PowerShell. Convert first, and then recreate the aliases in a separate command. You can simplify the process by only exporting the data you need and in the right format.

Work Area and Notes

Exercise 18

Find all files in `%TEMP%`, or a folder of your choice, that are either less than 100 bytes or greater than 1 megabyte.

Hint

You can create a compound filter.

Work Area and Notes

Exercise 19

Take the output from the previous exercise and save it as a standard XML file in UTF-8 format. The filename should be in `%TEMP%` and include a datetime, like YearMonthDay. and the computer name.

Hint

This is not using the Clixml cmdlets. `Get-Date` can give you the file date you need.

Work Area and Notes

Exercise 20

Using `Invoke-RestMethod`, get the latest RSS entries from `https://jdhitsolutions.com/blog/feed` and display the title, link, and when it was published, as a datetime value, in a grid view. For bonus points, see if you can include a comma-separated list of category names.

Hint

You'll need to treat something AS something else. Properties with special characters need to be quoted.

Work Area and Notes

Exercise 21

Using the previous solution as a starting point, select a single entry from `Out-Gridview` and open the link in your web browser.

Hint

In Windows 10, you should be able to start a url on an individual basis.

Work Area and Notes

Exercise 22

Get a unique list of commands run from your command history. Bonus points if you can filter out any help commands.

Hint

You may need to run some commands to build up history. Uniqueness is case-sensitive.

Work Area and Notes

Exercise 23

Using the DNSClientCache cmdlet, export all records other than AAAA to a CSV file. Your export should include the computer name, the entry, its type, the time to live, and its data. Instead of a comma, use a semicolon as the separator.

Hint

You can specify the local computer name, and filtering early is always best.

Work Area and Notes

Exercise 24

Using your birthday, write an object to the pipeline that shows your birthday (and time if you know or want to make something up), your current age in years as a round number, the timespan you've been alive, the day of the week you were born, and what day you can retire at age 65. Other than your birthday, you should be able to calculate everything.

Hint

You should be able to let PowerShell handle any culture-specific datetime formatting.

Work Area and Notes

Exercise 25

Assuming you have a few third-party applications or utilities running, prepare a formatted report of all processes that are not from Microsoft and copy to the clipboard. Paste into Notepad to verify.

Hint

Not all processes have a company name defined. You can only copy strings to the clipboard. `Group-Object` is not part of the solution.

Work Area and Notes

Suggested solutions are in the next chapter.

Part 3 Solutions

Exercise 1

Create an array of the numbers 1 to 20 and then get the 5th element of the array.

Solution

```
$arr = 1..15
$arr[4]
```

Comments

Don't forget that arrays start counting from an index number of `0`.

Exercise 2

Initialize an empty array. Add 10 random numbers between 10 and 100. Finally, get the sum total of all numbers in the array.

Solution

```
$arr=@()
1..10 | ForEach-Object { $arr+=(Get-Random -Minimum 10 -Maximum 100)}
$arr | Measure-Object -sum
```

Comments

You could also have used a `For` counter or a `ForEach` loop.

Exercise 3

Create a hashtable with keys for your computer name, PowerShell version, and the current date (without the time). Don't use any hardcoded values. Save the hashtable to a variable.

Solution

```
$h = @{
  Computername = $env:computername
  PSVersion = $PSVersionTable.PSVersion
  Date = (Get-Date).ToShortDateString()
}

$h
```

Comments

With a normal hashtable, PowerShell will display the keys in an indeterminate order.

Exercise 4

Using the hashtable from the previous exercise, add a key for Name using the values of Bits and WinRM. Remove the Date and PowerShell version keys. Finally, splat the hashtable to Get-Service.

Solution

```
$h.add("Name",@("Bits","WinRM"))
$h.Remove("date")
$h.remove("psversion")
Get-Service @h
```

Comments

Splatting is a technique you will use all the time, especially once you start scripting. Don't forget that splatting does not include the $ sign.

Exercise 5

Create an ordered hashtable with keys for your computer name, all the running processes, the top 5 processes using the most working set, and the total size of your %TEMP% folder in bytes. Complete the exercise by creating a custom object from the hashtable.

Solution

```
$h = [ordered]@{
  Computername = $env:computername
  Services = (Get-Service | Where-Object {$_.status -eq 'running'})
  Processes = (Get-Process | Sort-Object -Property WS -Descending |
  Select-Object -first 5)
  TempSize = (Get-ChildItem -Path $env:temp -file -Recurse |
  Measure-Object -Property length -sum).sum
}

New-Object -TypeName PSObject -Property $h
```

Comments

With an `[ordered]` hashtable, your keys will always be displayed in the order you defined them.

Exercise 6

Recreate the hashtable from the previous exercise, but this time create a custom object at the same time.

Solution

```
[pscustomobject]@{
  Computername = $env:computername
  Services = (Get-Service | Where-Object {$_.status -eq 'running'})
  Processes = (Get-Process | Sort-Object -Property WS -Descending
  | Select-Object -first 5)
  TempSize = (Get-ChildItem -Path $env:temp -file -Recurse |
  Measure-Object -Property length -sum).sum
}
```

Comments

This type of object implies an `[ordered]` hashtable so your properties will always be displayed in order. But the end result is an object which means you can't easily modify it the way you would a hashtable.

Exercise 7

Get all event logs on your computer that have entries and show the log name, maximum size, the total number of entries, and a property called Computername to reflect the computer name.

Solution

```
Get-Eventlog -list -computername $env:computername |
Where-Object {$_.entries.count -gt 0} |
Select-Object -Property Log,MaximumKilobytes,
@{Name="Entries";Expression={$_.entries.count}},
@{Name="Computername";Expression={$_.machinename}}
```

Comments

Defining custom properties with hashtables and `Select-Object` is a very common practice and something I'd encourage you to get familiar with. Remember that `$_` represents the current object in the pipeline.

In PowerShell 7.x, since `Get-EventLog` is deprecated, you might have come up with an answer like this using `Get-WinEvent`.

```
Get-Winevent -ListLog * |
Where-Object {$_.recordcount -gt 0} |
Select-Object -Property LogName,Maximum*,RecordCount,
@{Name="Computername";Expression={$env:COMPUTERNAME}}
```

While `Get-WinEvent` can query remote computers, it doesn't capture the computer name.

Exercise 8

Recursively go through all files in %TEMP% or a directory of your choice. The result of your PowerShell expression should show the number of files per file extension and their total size in bytes.

Solution

```
Get-ChildItem $env:temp -file -Recurse |
Group-Object -Property extension |
Select-Object Name,Count,
@{Name="Size";Expression={($_.group |
Measure-Object -Property length -sum).sum}}
```

Comments

This is a great example of how objects change in the pipeline, and how with a one-line command, you can do more than simply get data. You get information.

Exercise 9

Create an array of letters A through E. Get a random item from the array and assign it to $X. Then create a PowerShell construct that displays "alpha","bravo","charlie","delta","echo" based on the value of X.

Solution

```
$arr = "a","b","c","d","e"
$x = $arr | Get-Random -Count 1
Switch ($x) {

"a" { "alpha" }
"b" { "bravo" }
"c" { "charlie" }
"d" { "delta" }
"e" { "echo" }
}
```

Comments

You could have used an `If/ElseIf` construct, but after 2-3 items, I think it can get a bit unwieldy. A `Switch` construct is easier to read and offers more options.

Exercise 10

Take the string "PowerShell Forever" and display the characters in alphabetic order.

Solution

```
$text = "PowerShell Forever"
$arr = $text.ToCharArray()
$sort =$arr | foreach { $_ -as [string]} | Sort-Object
$sort -join ""
```

Comments

This answer is a little tricky. You probably created the character array with no problem. But if you sorted that and then joined, you most likely didn't get the result you were expecting. A `[char]` is actually a numeric value that represents an alphanumeric character. PowerShell handles all of that for you, so you see text. In my solution, I need to treat each `[char]` item as a string and *then* sort it.

An alternative would be to use the `-Split` operator and turn the string into an alphanumeric array.

```
$text = "PowerShell Forever"
$sort = $text -split "" | Sort-Object
$sort -join ""
```

I'll let you figure out a way to drop white-space characters.

Exercise 11

Create a hashtable of services based on their startup type.

Solution

```
$h = Get-Service | Group-Object -property starttype -AsHashTable -AsString
$h
```

Comments

Because some items in PowerShell are enums, or you can't always trust that what you see is actually a string, creating a hashtable with `Group-Object` and specifying the keys as strings is recommended.

Exercise 12

Using the hashtable you created in the previous exercise, get the display name of the last disabled service.

Solution

```
$h.Disabled[-1].DisplayName
```

Comments

This is a great example of PowerShell's object nature. PowerShell lets you treat the hashtable as an object with each key as a "property".

Exercise 13

Get all the even numbers between 1 and 50.

Solution

```
1..50 | where {$_%2 -eq 0}
```

Comments

Because the modulo operation is either 0 or 1, the result can be treated as a Boolean. Here's an alternative:

```
1..50 | where {-Not ($_%2)}
```

Exercise 14

Starting with $i equal to 0, loop 5 times, each time incrementing $i by 5. At the end, does $i equal 25?

Solution

```
1..5 | ForEach-Object -Begin { $i=0 } -Process { $i+=5} -end {$i}
```

Comments

You could have also used a `Do` loop.

```
$i=0
do {
  $i+=5
} until ($i -ge 25)
$i
```

Exercise 15

Display the top 25 processes that have been running the longest. Include the process ID, process name, start time, how long it has been running, and path.

Solution

```
Get-Process | Where-Object {$_.Starttime} | Sort-Object starttime |
Select-Object -first 25 -Property ID,Name,StartTime,
@{Name="Runtime";Expression={(Get-Date) - $_.starttime}},Path
```

Comments

You could also have filtered like this:

```
get-process | where starttime
```

By using `Select-Object` instead of `Format-Table`, you give yourself the option to do something else with the results. If you want a table, then pipe the command to `Format-Table`.

Exercise 16

Get 25 random numbers between 10 and 100. Multiply each one by 3, then get the total sum of these numbers, their average, the largest number, and the smallest.

Solution

```
10..100 | Get-Random -Count 25 | ForEach-Object { $_ * 3 } |
Measure-Object -sum -Average -Maximum -Minimum
```

Comments

Most of the time, we want to let PowerShell's pipeline and cmdlets do the work of working with multiple items. But sometimes you need to do something on an individual basis.

Exercise 17

Create these aliases: np (notepad.exe), tx (tzutil.exe), ct (control.exe). Export them to a JSON file. Delete the new aliases you just created and recreate them from the JSON file.

Solution

```
new-alias np notepad.exe -Description "user-defined"
new-alias tz tzutil.exe -Description "user-defined"
new-alias ct control.exe -Description "user-defined"

Get-Alias np,tz,ct | Select-Object  Name,
@{Name="Value";Expression={$_.ResolvedCommandName}},description |
ConvertTo-Json | Out-File c:\work\myaliases.json

dir alias:\ -include np,tz,ct -Recurse | Remove-Item

$in = Get-Content C:\work\myaliases.json | ConvertFrom-Json
$in | New-Alias -Description "user-defined" -PassThru
```

Comments

Because `New-Alias` will use parameter binding by property name, I exported the alias properties with the matching names, which makes import easier. My solution includes the description, although I wasn't expecting you to use it.

To delete aliases, you need to use the `ALIAS` PSDrive.

From a practical perspective, if you wanted to export aliases, you would normally use the `Export-Alias` cmdlet and export to a CSV file or script file. But then you wouldn't have learned as much as you did with this exercise!

Exercise 18

Find all files in %TEMP%, or a folder of your choice, that are either less than 100 bytes or greater than 1 megabyte.

Solution

```
dir $env:temp -Recurse -file |
Where-Object {($_.length -lt 100) -OR ($_.length -gt 1MB)}
```

Comments

It isn't necessary to enclose your comparisons in parentheses, but I find it helps to represent visually what the PowerShell expression is doing. This is a good practice as your filtering expressions become more complex.

Exercise 19

Take the output from the previous exercise and save it as a standard XML file in UTF-8 format. The filename should be in %TEMP% and include a datetime like YearMonthDay, and the computer name.

Solution

```
$filename = "$(Get-Date -format filedate)_$($env:computername).xml"
$file = Join-Path -Path $env:temp -ChildPath $filename

dir $env:temp -Recurse -file |
Where-Object {$_.length -lt 100 -OR $_.length -gt 1MB} |
ConvertTo-Xml -as Stream |
Out-File -FilePath $file -Encoding utf8
```

Comments

Instead of using sub-expressions, you could have used the `-f` operator to construct the file path. `Join-Path` is recommended when creating full filenames.

Exercise 20

Using Invoke-RestMethod, get the latest RSS entries from https://jdhitsolutions.com/blog/feed and display the title, link, and when it was published, as a datetime value, in a grid view. For bonus points, see if you can include a comma-separated list of category names.

Solution

```
Invoke-RestMethod -uri https://jdhitsolutions.com/blog/feed |
Select-Object -property Title,Link,
@{Name="Published";Expression = {$_.pubdate -as [datetime]}},
@{Name="Categories";Expression = {$_.category.'#cdata-section' -join ","}} |
Out-Gridview
```

Comments

You could also have treated the PubDate property as a datetime object like this:

```
[datetime]$_.pubdate
```

If you had explored the result with `Get-Member`, you would have discovered that each category object had a `#cdata-section` property which needs to be quoted because of the hash character.

Exercise 21

Using the previous solution as a starting point, select a single entry from `Out-Gridview` *and open the link in your web browser.*

Solution

```
Invoke-RestMethod -uri https://jdhitsolutions.com/blog/feed |
Select-Object -property Title,Link,
@{Name="Published";Expression = {[datetime]$_.pubdate}},
@{Name="Categories";Expression = {$_.category.'#cdata-section' -join ","}} |
Out-Gridview -Title "Select a post to read" -OutputMode Single |
Foreach-object { start-process $_.link }
```

Comments

You could have allowed multiple selections, and each one would still have opened. `Start-Process` doesn't accept pipelined input, so you need to use `Foreach-Object`.

Exercise 22

Get a unique list of commands run from your command history. Bonus points if you can filter out any help commands.

Solution

```
Get-History | Where-Object {$_.commandline -notmatch "help"} |
Sort Commandline | ForEach-Object {$_.CommandLine.ToLower() } | Get-Unique
```

Comments

You need to sort the property you are filtering on uniqueness. You could also have come up with this streamlined solution:

```
(Get-History).CommandLine.ToLower() |
Where-Object {$_ -notmatch "help"} |
Sort-Object | Get-Unique
```

Or even this, using the `Where()` method.

```
(Get-History).where({$_.commandline -notmatch "help"}).Commandline.toLower() |
Sort-Object | Get-Unique
```

Exercise 23

Using the DNSClientCache cmdlet, export all records other than AAAA to a CSV file. Your export should include the computername, the entry, its type, the time to live, and its data. Instead of a comma, use a semicolon as the separator.

Solution

```
Get-DnsClientCache -CimSession $env:computername -Type A,CNAME,PTR |
Select-Object Entry,Name,Type,TimeToLive,Data,PSComputername |
Export-CSV -Path c:\work\cache.csv -Delimiter ";"

Import-CSV c:\work\cache.csv -delimiter ";"
```

Comments

You could have selected all entries and then filtered on the type, assuming you realized that that actual value is numeric (28).

Exercise 24

Using your birthday, write an object to the pipeline that shows your birthday (and time if you know or want to make something up), your current age in years as a round number, the timespan you've been alive, the day of the week you were born, and what day you can retire at age 65. Other than your birthday, you should be able to calculate everything.

Solution

```
$bday = Get-Date "12/25/1970 1:23AM"
$age = (Get-Date) - $bday

[pscustomobject]@{
 Birthday = $bday
 Years = $age.totaldays/365 -as [int]
 Timespan = $age
 WeekDay = $bday.DayOfWeek
 Retire = $bday.AddYears(65).ToShortDateString()
}
```

Comments

I wish this were my real birthday! You could have used `$age.tostring()`, but then you would have been stuck with a string for the Age property. Better to let PowerShell handle the formatting and leaving you with the timespan object should you need it.

Exercise 25

Assuming you have a few third-party applications or utilities running, prepare a formatted report of all processes that are not from Microsoft and copy to the clipboard. Paste into Notepad to verify.

Solution

```
Get-Process |
Where-Object {$_.company -notmatch "microsoft" -AND $_.company} |
Sort-Object -property company | Format-Table -GroupBy Company |
Out-String | Set-Clipboard
```

Comments

With the right regular expression skills, you could have come up with a more concise pattern. Remember that when you group with `Format-Table`, you need to sort on the grouped property first. Finally, you could have piped to `clip.exe`, but we have a cmdlet, so why not use it?

Part 4 - WMI and CIM

I debated on how to structure this book and organize the exercises. There are so many areas. But I decided that since this book is intended for PowerShell beginners, I should focus on a core PowerShell skill, and one of those is working with Windows Management Instrumentation (WMI). WMI provides a wealth of systems management information, and PowerShell makes it incredibly easy to access.

In this final section, I'll test your knowledge of how WMI is structured and different ways of accessing it. You'll also find a few problems that pull everything from this book together.

Note: Even though you could use `Get-WmiObject` and related cmdlets for most of this section, I encourage you to get in the habit of using `Get-CimInstance` and related cmdlets. You can query the same WMI information. But the CIM cmdlets use PowerShell remoting which is much more firewall friendlier. PowerShell `7`.x doesn't even support `Get-WMIObject`, so you might as well start using the CIM cmdlets.

If you are looking for even more detail on using WMI with PowerShell, you might consider picking up a copy of PowerShell and WMI[5] by Richard Siddaway. Even though this is an older book, the WMI concepts haven't changed. There is also some good WMI/CIM content in PowerShell in Depth[6] which I co-wrote with Don Jones and Richard Siddaway.

[5]https://www.manning.com/books/powershell-and-wmi
[6]https://www.manning.com/books/powershell-in-depth-second-edition

Part 4 Exercises

Exercise 1

Display all classes in the root\cimv2 namespace that start with `win32_`.

Hint

The CIM cmdlets include one specific to this task.

Work Area and Notes

Exercise 2

What are the methods of the Win32_LogicalDisk class?

Hint

You may need to use `Get-Member` to discover what to use.

Work Area and Notes

Exercise 3

List all of the properties and current values of the WMI class for the operating system.

Hint

Once you find the class, it should be simple to select all the properties to display.

Work Area and Notes

Exercise 4

Using WMI, list all services that are set to start automatically but are not running. Show the display name only.

Hint

Don't assume that property names are the same as `Get-Service`.

Work Area and Notes

Exercise 5

Using WMI, write an object to the pipeline that shows the computer name, when it last started and how long it has been up and running.

Hint

You'll need to calculate a timespan. This is easiest when using the CIM cmdlets.

Work Area and Notes

Exercise 6

Get memory information from your computer that shows how much total physical memory and virtual memory is available and how much free of each. Don't forget to include the computer name.

Hint

The operating system needs to know this information.

Work Area and Notes

Exercise 7

Using the solution from the previous exercise, revise it to display memory values in GB, and calculate a percentage free of physical and virtual memory.

Hint

Memory values are already in KB.

Work Area and Notes

Exercise 8

Create a formatted table that shows all running services, the account they are running under, and their display name, service name, and start mode. Group the results by the account name and sorted by the display name.

Work Area and Notes

Exercise 9

PowerShell includes a number of cmdlets that are based on CIM. Find something that can get you volume information for Drive C and show the volume name, size, and free space in GB.

Hint

Don't assume that what you see are the actual property names.

Work Area and Notes

Exercise 10

Get information about the installed antivirus product on your computer.

Hint

You are going to have to find the correct security namespace. The class name should be obvious.

Work Area and Notes

Exercise 11

Using WMI, get a list of event log files that shows the log name, the path to the log file, when it was last modified, the number of entries, the size of the log, and what percentage of the log is in use.

Hint

You will need to compare the file size to its maximum size.

Work Area and Notes

Exercise 12

List the user accounts in the local administrator group.

Hint

You will need to find all win32_useraccounts associated with the group.

Work Area and Notes

Exercise 13

Using the previous solution as a starting point, list all users and groups that are members in the local administrators group. You may need to create an empty test group and manually add it.

Hint

`Get-CimAssociatedInstance` won't let you specify more than one resultant class.

Work Area and Notes

Exercise 14

Find all cmdlets and functions that accept CIMSession as a parameter.

Hint

Cmdlets and functions are different types of commands.

Work Area and Notes

Exercise 15

Get partition details for drive C that includes its partition number, size in GB, and its type.

Hint

Look for a command instead of inventing your own.

Work Area and Notes

Exercise 16

Create several CIM sessions to your local computer or remote test computers if you have them. Using the sessions get information about installed processors.

Hint

Take advantage of the PowerShell pipeline.

Work Area and Notes

Exercise 17

Using your CIM sessions from the previous exercise, create a report that shows the computer name, drive letter, drive label, size, free space (in GB), and percent free for all fixed drives.

Work Area and Notes

Exercise 18

Using your previous solution, create an HTML report with data formatted as a table. Include a title and header. View your file in a browser.

Hint

You can use HTML tags like `<H1>` in your content.

Work Area and Notes

Exercise 19

Query WMI to get a list of all processes, excluding System and System Idle Process, and display the ID, name, command line, executable path, working set size, creation date, and owner.

Hint

You'll need to use a compound legacy filter to eliminate items that don't match.

Work Area and Notes

Exercise 20

Create a report for each network adapter that is currently up. The report should show the adapter name, a description, link speed, MAC, and IPv4 address.

Hint

You can do this in a one-line command, although you'll have to use a nested expression to get the IP address.

Work Area and Notes

Exercise 21

List all installed products that have a name defined. Select the name, vendor, description, and when the product was installed. Bonus points if you can convert the date value to a `[datetime]`.

Hint

You can use a wildcard filter to see if the name has a value. The WMI wildcard is %.

Work Area and Notes

Exercise 22

This is a multi-step exercise. Using WMI, create a new environment variable for yourself called `PSPrimer` with a value of `1.0`. Verify you created it. Next, set the value to `2.0` and verify. Finally, delete the variable.

Hint

You will need to set several properties for your new instance.

Work Area and Notes

Exercise 23

Create an HTML report of all environment variables grouped by user name. Ideally, the report will have the user name in a tag like `<H2>` followed by a table showing the variable name and value.

Hint

You can do this is with `2-3` lines of PowerShell. You will have to use HTML fragments and need to account for the angle brackets around System. You will need to remove them. Your final HTML doesn't need a body. Your fragments can be post-content.

Work Area and Notes

Exercise 24

WMI has a large number of performance counters. Find one that provides formatted results of operating system performance and display all of the file-related properties, the processor queue length, and system uptime. Bonus points if you can turn the uptime value into something more meaningful. Extra bonus points if you can do this for multiple machines using splatting.

Hint

You are looking for a System related class. The uptime value is in seconds.

Work Area and Notes

Exercise 25

There are multiple TCPIP performance counter classes. Get performance counter data for all of them and display the result as a list grouped by each counter class name. As your last bonus, come up with a variation that writes an object to the pipeline, which includes the counter class. The result should be something you could export to XML or JSON.

Hint

You can pipe class names to `Get-Ciminstance`. But you'll have to extract the class name for sorting and grouping.

Work Area and Notes

Suggested solutions are in the next chapter.

Part 4 Solutions

Exercise 1

_Display all classes in the root\cimv2 namespace that start with `win32_`.

Solution

```
Get-CimClass -Namespace root\cimv2 -ClassName win32_*
```

Comments

The cmdlet defaults to the `root\cimv2` namespace, so you don't have to include it explicitly. You can use wildcards to narrow down your search for classes.

Exercise 2

What are the methods of the Win32_LogicalDisk class?

Solution

```
$class = Get-Cimclass -ClassName win32_logicaldisk
$class.CimClassMethods
```

Comments

In Windows PowerShell, you could have queried for instances of this class with `Get-WmiObject` retrieved methods with `Get-Member`.

```
Get-WmiObject win32_logicaldisk | Get-Member -MemberType methods
```

This approach would *not* have worked using `Get-CimInstance`:

```
Get-CimInstance win32_logicaldisk | Get-Member -MemberType methods
```

Exercise 3

List all of the properties current values of the WMI class for the operating system.

Solution

```
Get-CimInstance -ClassName win32_operatingsystem | Select-Object *
```

Comments

If you just wanted the property names, you could use `Get-CimClass`:

```
Get-Cimclass -ClassName win32_operatingsystem |
Select-Object -ExpandProperty CimClassProperties
```

Exercise 4

Using WMI, list all services that are set to start automatically but are not running. Show the display name only.

Solution

```
Get-CimInstance -ClassName win32_service -filter "startmode='auto' AND `
state <>'running'" | Select-Object -property Displayname
```

Comments

The tricky thing about WMI filters is that they use the legacy operators. And you should always filter early in your PowerShell expression, especially when you are querying many remote computers, or a WMI class that might return many results.

Once you understand the PowerShell's object nature, or if you only want the values not an object with a single property of `Displayname`, you could do this:

```
(Get-CimInstance -ClassName win32_service -filter "startmode='auto' AND `
state <>'running'").Displayname
```

Finally, you can improve performance with `Get-CimInstance` by requesting only the properties you intend to use.

```
(Get-CimInstance -ClassName win32_service -filter "startmode='auto' AND `
state <>'running'" -property displayname).Displayname
```

You'll appreciate the performance edge as you scale-out.

Exercise 5

Using WMI, write an object to the pipeline that shows the computer name, when it last started how long it has been up and running.

Solution

```
Get-CimInstance -ClassName win32_operatingsystem `
-ComputerName $env:computername |
Select-Object -property @{Name="Computername";Expression={$_.CSName}},
LastBootUpTime,
@{Name="Uptime";Expression = {(Get-Date) - $_.lastbootuptime}}
```

Comments

You could have selected `PSComputername`, assuming you specified a computer name. Personally, I prefer sticking to the standard `Computername` property name, so I create a custom property,

Exercise 6

Get memory information from your computer that shows how much total physical memory and virtual memory is available and how much free of each. Don't forget to include the computer name.

Solution

```
Get-CimInstance win32_operatingsystem -computername $env:computername |
Select-Object -property @{Name="Computername";Expression={$_.CSName}},
TotalVisibleMemorySize,
FreePhysicalMemory,TotalVirtualMemorySize,FreeVirtualMemory
```

Comments

You could also have simplified by using wildcards for the property names:

```
Get-CimInstance win32_operatingsystem -computername $env:computername |
Select-Object -property @{Name="Computername";Expression={$_.CSName}},TotalV*,
Free*Memory
```

Exercise 7

Using the solution from the previous exercise, revise it to display memory values in GB and calculate a percentage free of physical and virtual memory.

Solution

```
Get-CimInstance win32_operatingsystem -computername $env:computername |
Select-Object -property @{Name="Computername";Expression={$_.CSName}},
@{Name="MemoryGB";Expression = {$_.TotalVisibleMemorySize/1mb -as [int]}},
@{Name="FreeMemoryGB";Expression = {$_.FreePhysicalMemory/1mb}},
@{Name="PctFree";
Expression = {($_.freephysicalmemory/$_.totalvisiblememorysize)*100}},
@{Name="TotalVMSizeGB";Expression = {$_.TotalVirtualMemorySize/1mb -as [int]}},
@{Name="FreeVMMB";Expression ={$_.FreeVirtualMemory/1mb}},
@{Name="PctFreeVM";
Expression = { ($_.freeVirtualMemory/$_.totalvirtualmemorysize )*100 }}
```

Comments

This solution is technically a one-line expression (with unfortunate line breaks to fit the page) that clearly demonstrates how you can make PowerShell give you the information you want in the form you need. Once you learn about the `[math]` class and its static members, you can refine it into something like this:

```
Get-CimInstance win32_operatingsystem -computername $env:computername |
Select-Object -property @{Name="Computername";Expression={$_.CSName}},
@{Name="MemoryGB";Expression = {$_.TotalVisibleMemorySize/1mb -as [int]}},
@{Name="FreeMemoryGB";Expression = {[math]::Round($_.FreePhysicalMemory/1mb,2)}},
@{Name="PctFree";
Expression = {[math]::round(($_.freephysicalmemory/$_.totalvisiblememorysize)`
*100,4)}},
@{Name="TotalVMSizeGB";Expression = {$_.TotalVirtualMemorySize/1mb -as [int]}},
@{Name="FreeVMMB";Expression ={[math]::Round($_.FreeVirtualMemory/1mb,2)}},
@{Name="PctFreeVM";
Expression = { [math]::Round(($_.freeVirtualMemory/$_.totalvirtualmemorysize )`
*100,4)}}
```

Since this solution is writing a custom object to the pipeline, here is an alternative that creates the same result but might be easier to read.

```
Get-CimInstance win32_operatingsystem -computername $env:computername |
ForEach-Object {
   [pscustomobject]@{
      Computername = $_.CSName
      MemoryGB = $_.TotalVisibleMemorySize/1mb -as [int]
      FreeMemoryGB = [math]::Round($_.FreePhysicalMemory/1mb,2)
      PctFree =[math]::round(($_.freephysicalmemory/$_.totalvisiblememorysize)`
      *100,4)
      TotalVMSizeGB =$_.TotalVirtualMemorySize/1mb -as [int]
      FreeVMMB = [math]::Round($_.FreeVirtualMemory/1mb,2)
      PctFreeVM = [math]::Round(($_.freeVirtualMemory/$_.totalvirtualmemorysize)`
      *100,4)
    }
}
```

If I were building a PowerShell function around this code, I'd use this technique because I think it is easier to read and debug.

The other tricky part to this problem is the raw value of some of the properties like `TotalVisibleMemorySize`. Most of the time, you expect those values to be in bytes. But in the case of this class, they are in kilobytes. If you check the documentation at https://docs.microsoft.com/en-us/windows/desktop/cimwin32prov/win32-operatingsystem[7], you'll see that important detail. You would also figure it out by trial and error running the expression on a machine where you know how much memory in GB is installed. I believe I too, started with dividing by 1GB only to discover I had made a bad assumption.

[7]https://docs.microsoft.com/en-us/windows/desktop/cimwin32prov/win32-operatingsystem

Exercise 8

Create a formatted table that shows all running services, the account they are running under, their display name, service name, start mode. Group the results by the account name sorted by the display name.

Solution

```
Get-CimInstance -ClassName win32_service -filter "state='running'" |
Sort-Object Startname,Displayname |
Format-Table -GroupBy Startname -Property Name,Displayname,StartMode
```

Comments

You need to sort on the property you are grouping by when using `Format-Table`, so you might as well do all your sorting at once. Note that I used the WMI filter instead of using `Where-Object`. This is the preferred practice when querying WMI.

Exercise 9

PowerShell includes a number of cmdlets that are based on CIM. Find something that can get you volume information for Drive C show the volume name, size, free space in GB.

Solution

```
Get-Volume -DriveLetter C | Select-Object -Property DriveLetter,
FileSystemLabel,@{Name="SizeGB";Expression={$_.size/1gb -as [int]}},
@{Name="FreeGB";Expression={$_.SizeRemaining/1gb}}
```

Comments

When you run `Get-Volume`, PowerShell uses its formatting rules for the object. But as soon as you select properties, you are creating a custom object, so you need to take formatting and presentation into your own hands.

Exercise 10

Get information about the installed antivirus product on your computer.

Solution

```
Get-Ciminstance -Namespace root/SecurityCenter2 -ClassName AntiVirusProduct
```

Comments

There are other security-related classes you may also want to explore.

Exercise 11

Using WMI, get a list of event log files that shows the log name, the path to the log file, when it was last modified, the number of entries, the size of the log, what percentage of the log is in use.

Solution

```
Get-CimInstance win32_nteventlogfile | Select-Object -Property LogFileName,
Name,NumberOfRecords,FileSize,MaxFileSize,
@{Name="PctUsed";Expression={($_.filesize/$_.maxfilesize)*100}}
```

Comments

If you wanted to filter on the percent used property, you would need to use `Where-Object` as WQL (WMI Query Language) has no provision for calculating values on the fly.

Exercise 12

List the user accounts in the local administrator group.

Solution

```
Get-CimInstance win32_group -filter "name = 'administrators' AND `
LocalAccount = 'true'" |
Get-CimAssociatedInstance -ResultClassName win32_useraccount
```

Comments

This command won't take any nested groups into account. It will only show user accounts directly in the group. Another approach would be to use the `Get-LocalGroupMember` command.

```
Get-LocalGroupMember administrators
```

Exercise 13

Using the previous solution as a starting point, list all users groups that are members in the local administrators group. You may need to create an empty test group and manually add it.

Solution

```
Get-CimInstance win32_group -filter "name = 'administrators' AND `
LocalAccount = 'true'" |
Get-CimAssociatedInstance | Where-Object {$_.CimClass -match "User|Group"} |
Select-Object -property Name,Domain,Caption,SID,CimClass
```

Comments

You have run `Get-CimInstance | Get-CimAssociatedInstance` twice. If you were querying many remote computers, that might be preferable. The two resulting classes are different types of objects, and since it is better to only write one type of object to the pipeline, I'm creating a custom object by selecting common properties for both users and groups.

Exercise 14

Find all cmdlets and functions that accept CIMSession as a parameter.

Solution

```
Get-Command -CommandType Cmdlet,function -ParameterName Cimsession
```

Comments

Many of these commands will also have `Computername` parameter, which will usually create a temporary CIMSession.

Exercise 15

Get partition details for drive C that includes its partition number, size in GB, and its type.

Solution

```
Get-Partition -DriveLetter C | Select-Object -Property Driveletter,
PartitionNumber,Type,@{Name="SizeGB";Expression = {$_.Size/1GB -as [int]}}
```

Comments

You could have directly queried the Win32_Partition space, or even indirectly.

```
Get-CimInstance win32_logicaldisk -filter "deviceid='c:'" |
Get-CimAssociatedInstance -ResultClassName Win32_DiskPartition
```

Exercise 16

Create several CIM sessions to your local computer or remote test computers if you have them. Using the sessions get information about installed processors.

Solution

```
New-CimSession -ComputerName $env:computername,$(hostname),localhost
Get-CimSession | Get-CimInstance Win32_Processor
```

Comments

Using CIMSessions is handy when some connections might require different credentials or connection protocols.

Exercise 17

Using your CIM sessions from the previous exercise, create a report that shows the computer name, drive letter, drive label, size, free space (in GB), and percent free for all fixed drives.

Solution

```
Get-CimSession | Get-CimInstance -ClassName Win32_LogicalDisk `
-Filter "drivetype=3" |
Select-Object -property @{Name="Computername";Expression={$_.Systemname}},
DeviceID,VolumeName,
@{Name="SizeGB";Expression={$_.Size/1gb -as [int]}},
@{Name="FreeGB";Expression={$_.Freespace/1gb -as [int]}},
@{Name="PctFree";Expression = {($_.Freespace/$_.size)*100}}
```

Comments

You might need to use the `Win32_LogicalDisk` class for older computers.

Exercise 18

Using your previous solution, create an HTML report with data formatted as a table. Include a title and header. View your file in a browser.

Solution

```
Get-CimSession | Get-CimInstance -ClassName Win32_LogicalDisk `
-Filter "drivetype=3" |
Select-Object -property @{Name="Computername";Expression={$_.Systemname}},
DeviceID,VolumeName,
@{Name="SizeGB";Expression={$_.Size/1gb -as [int]}},
@{Name="FreeGB";Expression={$_.Freespace/1gb -as [int]}},
@{Name="PctFree";Expression = {($_.Freespace/$_.size)*100}} |
ConvertTo-HTML -Title "Drive Report" -PreContent `
"<h1>Company Drive Report</h1>" | Out-File c:\work\drivereport.htm

invoke-item C:\work\drivereport.htm
```

Comments

`ConvertTo-Html` should default to a table, but you can always be explicit with the `-As` parameter.

Exercise 19

Query WMI to get a list of all processes, excluding System System Idle Process, and display the ID, name, command line, executable path, working set size, creation date, and owner.

Solution

```
Get-CimInstance win32_process -filter "name <> 'system' AND `
name <> 'system idle process'" |
Select-Object -property ProcessID,Name,Commandline,ExecutablePath,
WorkingSetSize,CreationDate,@{Name="Owner";
Expression={  $o = $_ | Invoke-CimMethod -methodname GetOwner ;
"$($o.domain)\$($o.user)"
}}
```

Comments

You could have filtered with the `LIKE` operator, but it might have also filtered out other processes that had `System` as part of the name. The compound filter I am using is much more selective.

Exercise 20

Create a report for each network adapter that is currently up. The report should show the adapter name, a description, link speed, MAC, and IPv4 address.

Solution

```
Get-NetAdapter | where-object {$_.status -eq 'up'} |
Select-Object -Property Name,InterfaceDescription,LinkSpeed,MACAddress,
@{Name="IPAddress";Expression = {$_ | Get-NetIPAddress -AddressFamily IPv4}}
```

Comments

This solution is a good example that demonstrates how CIM information is related. You could also have queried the legacy `win32_networkadapter` or `win32_networkadapterconfiguration` classes.

```
Get-CimInstance win32_networkadapter -filter "netenabled='true'" |
Select-Object -property NetConnectionID,description,Speed,MacAddress,
@{Name="IPAddress";Expression = { ($_ | Get-CimAssociatedInstance `
-ResultClassName Win32_NetworkAdapterConfiguration).IPAddress}}
```

Exercise 21

List all installed products that have a name defined. Select the name, vendor, description, and when the product was installed. Bonus points if you can convert the date value to a [datetime].

Solution

```
Get-CimInstance win32_product -filter "name like '%'" |
Select-Object -property Name,Vendor,Version,Description,InstallDate
```

Comments

The WMI filter I am using says, “Get anything that has a name like something”. If the `Name` property is blank, it should get filtered out. Personally, I try to avoid using a `Like` filter if I can for performance reasons, but sometimes you have no choice.

As for the bonus, you could have parsed the `InstallDate` string a number of ways, including using regular expressions. Here’s one approach that should also respect culture-specific date formatting.

```
Get-CimInstance win32_product -filter "name like '%'" |
Select-Object  Name,Vendor,Version,Description,
@{Name="Installed" ;
Expression = {$_.InstallDate.Insert(4,"/").insert(7,"/") -as [datetime]}}
```

The expression scriptblock is inserting slashes into the `InstallDate` string which makes it easier to treat as a datetime object.

Exercise 22

This is a multi-step exercise. Using WMI, create a new environment variable for yourself called PSPrimer with a value of 1.0. Verify you created it. Next, set the value to 2.0 and verify. Finally, delete the variable.

Solution

```
New-CimInstance -ClassName win32_environment -Property @{Name="PSPrimer";
VariableValue="1.0";username="$env:userdomain\$env:username"}

Get-CimInstance win32_environment -filter "name='psprimer'"

Get-CimInstance win32_environment -filter "name='psprimer'" |
Set-CimInstance -Property @{VariableValue="2.0"} -PassThru

Get-CimInstance win32_environment -filter "name='psprimer'" |
Remove-CimInstance
```

Comments

If you read help and examples, this should have been an easy exercise.

Exercise 23

Create an HTML report of all environment variables grouped by user name. Ideally, the report will have the user name in a tag like `<H2>` *followed by a table showing the variable name and value.*

Solution

```
$frag = Get-CimInstance win32_environment -ComputerName $env:computername |
Group-Object -property Username | ForEach-Object {
 #strip off the angle brackets
 $name = $_.name.replace(">","").replace("<","")
 $_.group  | Select-Object Name,VariableValue |
ConvertTo-Html -PreContent "<h2>$name</h2>" -Fragment
}

ConvertTo-Html -PostContent $frag -Title "Environment" -preContent `
"<h1>$env:computername</h1>" | Out-File c:\work\env.html
```

Comments

If the name doesn't have angle brackets, then the `Replace` method won't do anything. You could also have broken this down into more granular steps used the `ForEach` enumerator, which is probably the approach I would have taken if I were turning this into a re-usable script or function.

Exercise 24

WMI has a large number of performance counters. Find one that provides formatted results of operating system performance and display all of the file-related properties, the processor queue length, and system uptime. Bonus points if you can turn the uptime value into something more meaningful. Extra bonus points if you can do this for multiple machines using splatting.

Solution

```
Get-CimInstance Win32_PerfFormattedData_PerfOS_System |
Select-Object -property File*,ProcessorQueueLength,SystemUptime
```

Comments

Hopefully, you came up with at least my preferred answer. You can convert the uptime value into a timespan.

```
Get-CimInstance Win32_PerfFormattedData_PerfOS_System |
Select-Object -property File*,ProcessorQueueLength,
@{Name="Uptime";Expression={New-TimeSpan -Seconds $_.systemuptime}}
```

And a splatted solution for multiple computers might look like this:

```
$get = @{
 Class = 'Win32_PerfFormattedData_PerfOS_System'
 Computername = $env:computername,'localhost',$(hostname)
 Property = (Get-Cimclass Win32_PerfFormattedData_PerfOS_System).`
  cimclassproperties.name | where {$_ -match "file|uptime|queue"}
 Erroraction = "Stop"
}

$select = 'File*','*queue*',
@{Name="Uptime";Expression={New-TimeSpan -Seconds $_.systemuptime}},
@{Name="Computername";Expression={$_.PSComputername.toUpper()}}

Get-CimInstance @get  | Select-Object -property $select
```

Make sure you understand what this code is doing.

Exercise 25

There are multiple TCPIP performance classes. Get counter data for all of them and display the result as a list grouped by each counter class name. As your last bonus, come up with a variation that writes an object to the pipeline which includes the counter class. This should be something you could export to XML or JSON.

Solution

```
Get-CimClass win32_PerfFormattedData_TCPIP_* -ComputerName $env:computername |
Select-Object -property @{Name="Classname";Expression = {$_.CimClassName}},
@{Name="computername";Expression={$_.cimsystemproperties.servername}} |
Get-Ciminstance | Sort-Object {$_.CimClass.CimClassName} |
Format-List -GroupBy @{Name="Class";Expression={$_.CimClass.CimClassName}}
```

Comments

In this solution, there may not be a need to sort on the custom property, but it is a good habit when grouping with `Format-Table` or `Format-List`. Notice too that I renamed the CimClassName and Servername properties so that I could take advantage of parameter binding with `Get-CimInstance`.

As far as creating an object with the counter class, there are several options depending on how you might need to use the data. I should point out that the class name is already a part of the object, but it is buried, and I'd like to see it by default. You could use `Add-Member` to add a custom property.

```
Get-CimClass win32_PerfformattedData_TCPIP_* -ComputerName $env:computername |
Select-Object -property @{Name="Classname";Expression = {$_.CimClassName}},
@{Name="computername";Expression={$_.cimsystemproperties.servername}} |
Get-Ciminstance | Add-Member -MemberType ScriptProperty -Name CounterClass `
-Value {$this.cimclass.CimClassName} -PassThru
```

Or you could create a custom object with nested properties.

```
Get-CimClass win32_PerfFormattedData_TCPIP_* -ComputerName $env:computername |
Select-Object -property @{Name="Classname";Expression = {$_.CimClassName}},
@{Name="computername";Expression={$_.cimsystemproperties.servername}} |
ForEach-Object {

 [pscustomobject]@{
    CounterClass = $_.classname
    Counters = ($_ | Get-Ciminstance |
       Select-Object * -ExcludeProperty PSComputername,CIM*)
    Computername = $_.Computername
    Date = (Get-Date)
  }
}
```

Once you understand how objects move through the PowerShell pipeline and how to leverage that knowledge, PowerShell becomes a very mighty tool in your IT toolbox.

Part 5 - PowerShell 7

Since *The PowerShell Practice Primer* was first published, Microsoft has released the next generation of PowerShell. PowerShell 7.x is a continuation of PowerShell Core. It is an open-source project and runs cross-platform. On Windows, you can install it side-by-side with Windows PowerShell 5.1. When it comes to fundamentals, there isn't much difference between Windows PowerShell and PowerShell. All of the exercises up to now could be solved in either version, at least on a Windows-based computer.

This is a bonus part that offers some PowerShell 7.x specific problems. The exercises were developed on a Windows 10 desktop running PowerShell 7.1.2. Later versions of PowerShell might offer variations on the suggested solutions. Some of the exercises also assume a bit more experience or skill with PowerShell.

To learn more about PowerShell 7 and how to install it, go to https://github.com/powershell/powershell[8].

[8]https://github.com/powershell/powershell

Part 5 Exercises

Exercise 1

Using the ternary operator, write a PowerShell expression that if the day of the week is Monday, Wednesday, or Friday to run `Get-Service`, otherwise run `Get-Process`.

Hint

You might need to read some help.,

Work Area and Notes

Exercise 2

Get the last 1000 entries from the System event log from 10 computers. Use your local computer name ten times.

Hint

You need to do something for each computer as quickly as possible.

Work Area and Notes

Exercise 3

Create a simple pipeline chain that if Notepad is running, kills the Notepad process and then creates a text file with content that includes the current date that you killed the process. Also, suppress the error message if Notepad is not running.

Hint

You will need to chain some commands together in the pipeline.

Work Area and Notes

Exercise 4

Get about help topics grouped by the first letter of the topic name, after `about_`. The grouped output should be sorted alphabetically. Then export to a file and re-import into PowerShell. You should be able to run a command like `$about.P` and see the topics that begin with P. Don't export the help content, just the name.

Hint

You might need to normalize the first letter for proper grouping.

Work Area and Notes

Exercise 5

Create a formatted report that shows PowerShell experimental features. The report should show if the feature is enabled and the description. Create the report so that you can read the entire description.

Work Area and Notes

Exercise 6

Using your solution from the previous problem, revise the output to use ANSI formatting to show the name with a reversed background and if the feature is enabled, to show the enabled value in Green. You might need to enable a feature to test. Or show disabled features in red.

Hint

PSReadline options can point you in the right direction. It might be easier to construct your answer in VSCode and test in the console.

Work Area and Notes

Exercise 7

Get all local user accounts and export to a CSV file. The export should include type information and add quotes to the `Description` column. Do not overwrite existing files. Verify by re-importing the CSV file. Give yourself an extra challenge by using splatting. You should be able to get account information without using WMI/CIM.

Work Area and Notes

Exercise 8

Using the REST API described at https://github.com/chubin/wttr.in[9], get the current weather information for your location in JSON format. You can use a URI like `https://wttr.in/Chicago?format=j1`. Using the JSON data, create a formatted table showing the temperature, feels like temperature, humidity, description, and wind speed. Use a table grouping that will show the location and the time of the weather observation. Convert the time to a datetime value. Give yourself an extra challenge and right-align the wind speed. Select temperature and speed formats appropriate to your region.

Hint

You'll find this easier to break down into steps. Splatting might help.

Work Area and Notes

Exercise 9

First, find all markdown files in any module found in your module path. Then view a random markdown file at the console.

Hint

Markdown files have a `.md` extension.

Work Area and Notes

[9]https://github.com/chubin/wttr.in

Exercise 10

Using your previous solution as a starting point, find files that don't have `DSC` in the name and select them in a console grid view. Show each selected markdown file in your web browser.

Hint

You might need to install the `Microsoft.PowerShell.ConsoleGuiTools` module. Or use the traditional gridview command. There are several ways to show markdown in case one way fails or is buggy.

Work Area and Notes

Suggested solutions are in the next chapter.

Part 5 Solutions

Exercise 1

Using the ternary operator, write a PowerShell expression that if the day of the week is Monday, Wednesday, or Friday to run `Get-Service`*, otherwise run* `Get-Process`*.*

Solution

```
(Get-Date).DayofWeek -match 'Mon|Wed|Fri' ? (Get-Service) : (Get-Process)
```

Comments

Using the ternary operator is an alternative to a traditional `If/Else` statement. The action after the `?` and `:` operators need to be interpreted as a PowerShell command, which is why they are enclosed in parentheses.

Exercise 2

Get the last 1000 entries from the System event log from 10 computers. Use your local computer name ten times.

Solution

```
1..10 | foreach -begin { $c = @()} -process {$c+=$env:computername}
$logs = $c | foreach -parallel {
  Get-Winevent -ComputerName $_ -LogName system -MaxEvents 1000
}
```

Comments

The first line is defining an array with the computer name listed ten times. The second command is using the `Parallel` parameter in `ForEach-Object`. Make a note of how long the command ran, then run it again without `-parallel` to see the difference.

Exercise 3

Create a simple pipeline chain that if Notepad is running, kills the Notepad process and then creates a text file with content that includes the current date that you killed the process. Also, suppress the error message if Notepad is not running.

Solution

```
Get-Process notepad -ea silentlycontinue && stop-process -name notepad && `
 "$(Get-Date) killed Notepad" | Out-file d:\temp\log.txt
```

Comments

Hopefully, you took some time to read `about_pipeline_chain_operators`.

If the first expression is true, then the expression after `&&` runs. Assuming the `Stop-Process` command is successful, then the string is piped to `Out-File`. The suggested solution is a combination of conditional and traditional pipelined commands.

Exercise 4

Get about help topics grouped by the first letter of the topic name, after `about_` *. The grouped output should be sorted alphabetically. Then export to disk and re-import into PowerShell. You should be able to run a command like* `$about.P` *and see the topics that begin with P. Don't export the help content, just the name.*

Solution

```
$helpgroup = Get-Help about_* |
Group-Object { $_.name.split("_", 2)[1][0].toString().toupper() } -AsHashTable `
-AsString | Sort-Object Name

$helpgroup.GetEnumerator() |
ForEach-Object {
    [ordered]@{$_.name = ($_.value | Select-Object Name, Synopsis) }
} | ConvertTo-Json -Depth 3 | Out-File c:\work\about.json

$about = Get-Content C:\work\about.json | ConvertFrom-Json

$about.P
```

Comments

This is a potentially tricky problem because you first need to get the list of help topics and not the content. The grouping is using a custom property to split the name of the help topic, get the second item in the split, get the first letter of this item, treat it as a string and make it upper case. The grouped objects are treated as a hashtable. The next step is to take the hashtable and for each item, create a secondary hashtable with the required data and then export to a JSON file. The last part imports the file and tests the results.

Exercise 5

Create a formatted report that shows PowerShell experimental features. The report should show if the feature is enabled and the description. Create the report so that you can read the entire description.

Solution

```
Get-ExperimentalFeature |
Format-Table -GroupBy Name -Property Enabled,Description -wrap
```

Comments

There is no reason to sort because the `Name` property is unique. Using the `GroupBy` parameter saves some space yet still gives you information you want.

Exercise 6

Using your solution from the previous problem, revise the output to use ANSI formatting to show the name with a reversed background and if the feature is enabled, to show the Enabled value in Green. You might need to enable a feature to test. Or show disabled features in red.

Solution

```
Get-ExperimentalFeature |
Format-Table -GroupBy @{ Name="Feature";
Expression = {"`e[7;38;5;251m$($_.Name)`e[0m"}} `
-Property @{Name='Enabled';Expression = {
  if ($_.enabled) {
    "`e[92m$($_.enabled)`e[0m"
  }
  else {
      $_.Enabled
  }
}},Description -wrap
```

Comments

Using ANSI escape sequences offers some interesting options in your scripting work, although you need to ensure that that the host supports it. For this particular situation, a better long-term answer would be to create a custom format extension with a named view. The custom format file can use ANSI sequences.

Exercise 7

Get all local user accounts and export to a CSV file. The export should include type information and add quotes to the `Description` *column. Do not overwrite existing files. Verify by re-importing the CSV file. Give yourself an extra challenge by using splatting. You should be able to get account information without using WMI/CIM.*

Solution

```
$p = @{
    IncludeTypeInformation = $true
    QuoteFields = "Description"
    Path  = "c:\work\localusers.csv"
    NoClobber = $True
}
Get-LocalUser | Export-Csv @p
Import-CSV $p.path
```

Comments

If you only use what you know of the `Export-CSV` from Windows PowerShell, you would have missed out on changes to that command. You may think you know a command, but things change, which is why reading help is so important.

Exercise 8

Using the REST API described at https://github.com/chubin/wttr.in[10], get the current weather information for your location in JSON format. You can use a URI like `https://wttr.in/Chicago?format=j1`. *Using the JSON data, create a formatted table showing the temperature, feels like temperature, humidity, description, and wind speed. Use a table grouping that will show the location and the time of the weather observation. Convert the time to a datetime value. Give yourself an extra challenge and right-align the wind speed. Select temperature and speed formats appropriate to your region.*

Solution

```
$location = "Syracuse"
$j = Invoke-RestMethod -uri "https://wttr.in/$($location)?format=j1"

$Properties = @{Name ="TempF"; Expression = { $_.current_condition.temp_F }},
@{Name="FeelF"; Expression = { $_.current_condition.FeelsLikeF } },
@{Name="Humidity";Expression={$_.current_condition.humidity}},
@{Name="Description";Expression={$_.current_condition.weatherdesc.value}},
@{Name="WindMph";Expression={$_.current_condition.WindSpeedMiles};Align="right"}

$ft = @{
    GroupBy = @{Name="$location";
    Expression={$_.current_condition.localObsDateTime -as [datetime]}}
    Property = $Properties
}

$j | Format-Table @ft
```

Comments

JSON data is no different than any type of object. Although you often have to resort to re-formatting to make it easier to consume. In this example, I wanted nicer looking property names than the raw JSON values. In a scripting project, you might use a PowerShell class and create functions to abstract the entire process from the user.

Exercise 9

First, find all markdown files in any module found in your module path. Then view a random markdown file at the console.

[10] https://github.com/chubin/wttr.in

Solution

```
$mdfiles = $env:PSModulePath -split ";" | ForEach-Object -Parallel {
    Get-ChildItem -Path $_ -Filter *.md -Recurse
}
$mdfiles | Get-Random | Show-Markdown
```

Comments

You didn't have to use the `-Parallel` parameter, but since you need to go through multiple directories, why not be quick about it?

Exercise 10

Using your previous solution as a starting point, find files that don't have `DSC` *in the name and select them in a console grid view. Show each selected markdown file in your web browser.*

Solution

```
$mdfiles = $env:PSModulePath -split ";" | ForEach-Object -Parallel {
    Get-ChildItem -Path $_ -Filter *.md -Recurse
}
$mdfiles | Where-Object { $_.fullname -notmatch "DSC" } |
Select-Object fullname |
Out-ConsoleGridView -Title "Select files to view" -OutputMode Multiple |
ForEach-Object {
    Get-Content -Path $_.fullname | Out-String | Show-Markdown -UseBrowser
}
```

Comments

You should be able to use `Show-Markdown -path $_.fullname` in the `ForEach-Object` command, but at least for me, this appeared to be buggy. As a work-around, I got the raw content of each selected markdown file, and then piped to `Show-Markdown`.

Afterword

Congratulations! Assuming you worked your way through the book from start to finish, you are now 100 times smarter than you were when you started the first exercise. The purpose of this book was not to provide 100 answers to 100 problems, but rather to focus on the *process* you had to go through to discover, develop and test your solutions. Sure, I trust you can better use cmdlets like `Select-Object`, `Group-Object`, and `Get-CimInstance`. But my larger goal was to make PowerShell feel more comfortable. I hope you noticed that as you worked your way through the book that the process of solving problems got easier.

At this point, you should feel much more at ease in a PowerShell session. It shouldn't feel alien or intimidating. You should feel that you *own* it. I trust you'll continue your PowerShell education. If you found this book helpful, I hope you'll let me know, including what other areas you think would make good primers. In the meantime, there are plenty of PowerShell courses on Pluralsight.com[11]. There is also a great deal of free content on YouTube.

I assumed at the start that you either read or had experience equivalent to *Learn Windows PowerShell in a Month of Lunches*[12]. If so, the next logical book for you is *Learn PowerShell Scripting in a Month of Lunches*[13] also published by Manning. After that ,you should get a copy of The PowerShell Scripting and Toolmaking Book[14] from Leanpub. While you're at it, you'll find quite a few PowerShell-related titles on Leanpub.com[15].

If you like this learning style, I encourage you to visit the Iron Scripter[16] web site. There you will find many PowerShell scripting challenges for all skill levels. The challenges are open-ended with no official solutions. In many instances, other PowerShellers have submitted links to their work and answers.

I hope you'll find me online or at a conference and share *your* PowerShell experiences, especially with this book. This project has been something of a teaching experiment, and I am eager for feedback. Thank you very much for your support in this and all of my PowerShell work.

[11]https://www.pluralsight.com/search?q=powershell
[12]http://bit.ly/PSMoL3
[13]http://bit.ly/PSScriptingMoL
[14]https://leanpub.com/powershell-scripting-toolmaking
[15]https://leanpub.com/bookstore?type=all&search=powershell
[16]https://ironscripter.us

About the Author

Jeff Hicks

Jeff Hicks is an IT veteran with 30 years of experience, much of it spent as an IT infrastructure consultant specializing in Microsoft server technologies with an emphasis in automation and efficiency. He is a multi-year recipient of the Microsoft MVP Award for his PowerShell-related contributions. He works today as an independent author, teacher, and consultant. Jeff has taught and presented on PowerShell and the benefits of automation to IT Pros worldwide for almost 20 years. He has authored and co-authored several books, writes for numerous online sites, is a Pluralsight[17] author, and a frequent speaker at technology conferences and user groups.

You can keep up with Jeff on Twitter as @JeffHicks[18] and his blog at https://jdhitsolutions.com/blog[19].

[17]https://app.pluralsight.com/profile/author/jeff-hicks
[18]https://twitter.com/jeffhicks
[19]https://jdhitsolutions.com/blog

Release Notes

March 2, 2021

- Updated extras files.
- Edited text for formatting errors.
- Updated sample chapters.
- Adjusted markdown formatting in solution chapters.
- Updated author bio.

March 1, 2021

- Added Part 5 with PowerShell 7.x related exercises.
- Updated solutions with PowerShell 7.x related answers or commentary where necessary.
- Updates to author bio.
- Updates to Note on Code Listings.
- Updated Afterword.
- Updated Preface.
- Updated How to Use This Book.
- Updated PowerShell Refresher chapter.
- Minor updates to sample problem chapters.
- Added an extra file with all suggestion solutions.

July 7, 2020

- Updates to the author bio.
- Updated afterword.
- Updated the PowerShell Refresher chapter.
- Updates to exercises with PowerShell 7 compatibility.
- Minor code cleanup for clarity.

December 17, 2018

- fixed filename error in Part 1 Exercise 24 solution.
- Added Confucius epigraph to preface.
- Minor updates to the PowerShell Refresher chapter.
- Minor updates to How to Use This Book chapter.
- Reorganized the order of intro material.

August 15, 2018

- Added additional comments on the solution for Part 4 #7
- Added a foreword from Don Jones
- Renamed my foreword to a preface
- Minor edits to the note on code formatting
- Updates to How to Use This Book
- Minor edits to front matter
- Updates to samples.

August 14, 2018

- Modified Question 15 to more accurately reflect the chosen operator.

June 28, 2018

- Fixed typos throughout the book

June 22, 2018

- Fixed error in sample solution chapter
- General cleanup and code tweaks

June 6, 2018

- Added Part 4 content
- Added afterword

June 1, 2018

- Fixed title for Part 2 Solutions
- Added Part 3 content

May 31, 2018

- Initial release of intro material, Part 1 and Part 2