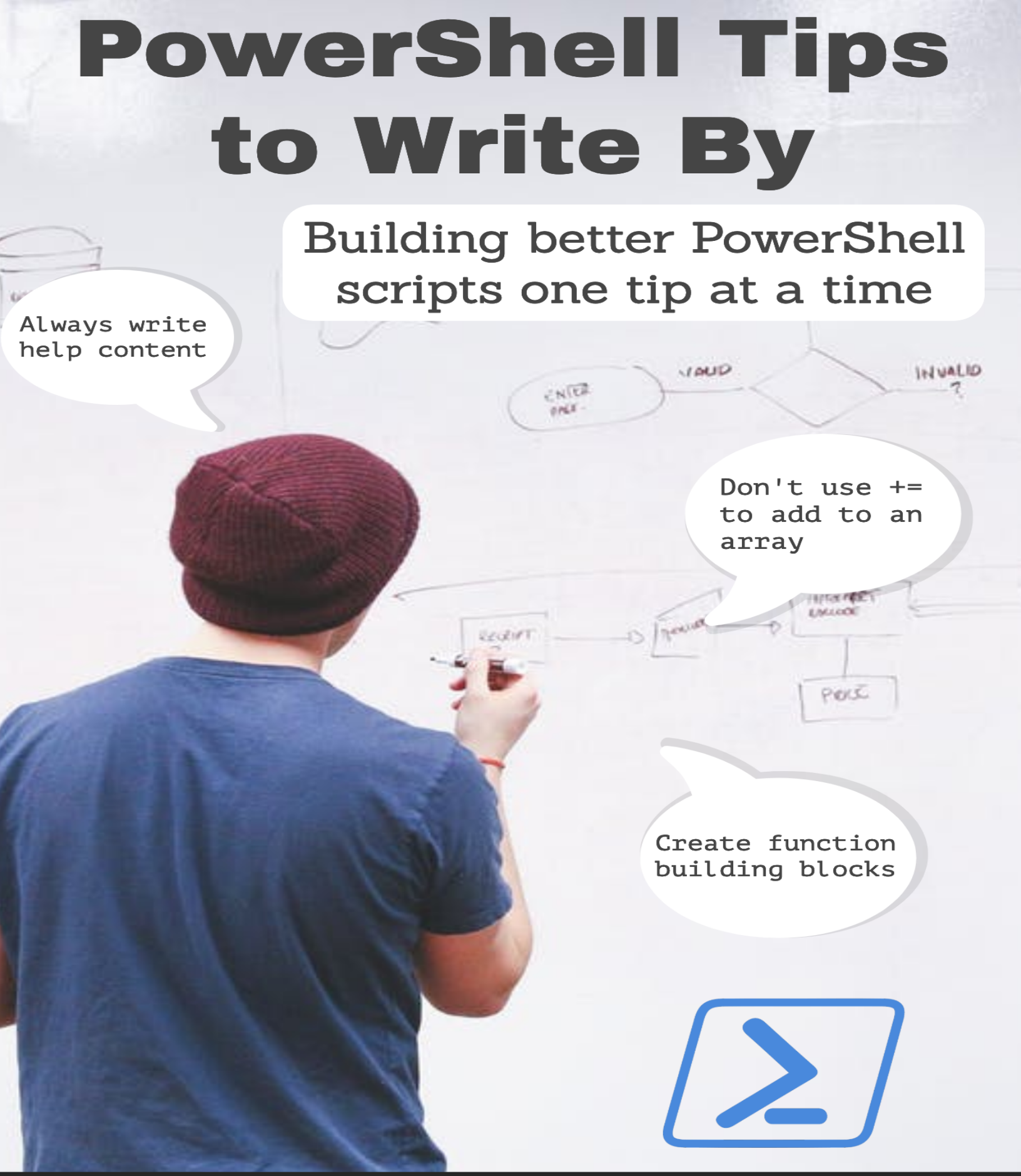
PowerShell Tips to Write By

Building better PowerShell scripts one tip at a time



Adam Bertram, Bryce McDonald

PowerShell Tips to Write By

Building better PowerShell scripts one tip at a time

Adam Bertram

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

This version was published on 2020-06-04

Tweet This Book!

Please help Adam Bertram by spreading the word about this book on Twitter!

The suggested hashtag for this book is #PowerShellTips.

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

#PowerShellTips

Contents

About This Book

This is the 'Forever Edition' on LeanPub[1]. That means when the book is published as it's written and may see periodic updates.

If you purchased this book, thank you. Know that writing a book like this takes hundreds of hours of effort, during which I'm not making any other income. Your purchase price is important to keeping a roof over my family's heads and food on my table. I'm not rich–this income is important to us. 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've made sure to provide a DRM-free file (excepting any DRM added by a bookseller other than LeanPub) so that you can use your copy any way that's convenient for you. I appreciate your respecting my rights and not making unauthorized copies of this work.

—

If you got this book for free from someplace, 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 encouraged to stop writing them. If you find this book useful, I would greatly appreciate you purchasing a copy from LeanPub.com or another bookseller. When you do, you'll be letting me know that books like this are useful to you, and that you want people like me to continue creating them.

-Adam

[1] http://leanpub.com/

About the Authors

Adam Bertram is a 20-year veteran of IT and experienced online business professional. He's an entrepreneur, IT influencer, Microsoft MVP, blogger at adamtheautomator.com[2], trainer and content marketing writer for multiple technology companies. Adam is also the founder of the popular IT career development platform TechSnips. Catch up on Adam's articles at adamtheautomator.com, connect on LinkedIn[3] or follow him on Twitter[4].

Bryce McDonald is a 15 year veteran of IT and in the process of building his online business. He's an entrepreneur, blogger at brycemcdonald.net[5], and founder of NorthCode Solutions[6] which specializes in bringing cloud services and automation to small to medium sized businesses. Catch up with Bryce at brycemcdonald.net[7], on LinkedIn[8], or follow him on Twitter[9].

[2] https://adamtheautomator.com
[3] https://linkedin.com/in/AdamBertram
[4] https://twitter.com/adbertram
[5] https://www.brycemcdonald.net
[6] https://www.northcodesolutions.com
[7] https://www.brycemcdonald.net
[8] https://linkedin.com/in/mcdonaldbm
[9] https://twitter.com/_brycemcdonald

Feedback

We'd love your feedback. Found a typo? Discovered a code bug? Have a content suggestion? Wish we'd answered a particular question? Let me know.

1. Please have a chapter name, heading reference, and a brief snippet of text for us to reference. We can't easily use page numbers, because source documents don't have any.
2. Email adam@adamtheautomator.com and let us know.

Thank you for taking the time to help make this book better!

Introduction

This book was created out of necessity. There are many books out there on how to learn PowerShell. You'll also find thousands of articles and blog posts on PowerShell best practices. But there wasn't an entire collection of PowerShell learning and best practices brought together before.

Each chapter in this book is broken down by chapter with multiple "tips" inside. Each chapter is a bucket for the kinds of tips you can expect to read about. Each tip is a best practice. Tips are short, actionable steps you can take today to help you improve your PowerShell scripts.

Tips do not go into major detail. There are other resources out there for that. The tips in this book are not meant to be exhaustive how-tos but to rather act as a checklist for actions to take.

All tips in this book were written by the authors but many were contributed by the PowerShell community. If a tip did come from the community, the community member will be referenced.

Tip Format

Each tip will be in a specific, standardized format as shown below. Every tip will contain a short summary, the community member that submitted it (if applicable), along with a link for more information and the type of resource the link points to.

—

SUMMARY OF TIP HERE

Tip Source: XXXXXXXX

Further learning:

Link: –Title–
Source: –URL–
Format: Article|Book|eBook|Course

—

Do The Basics

When it comes to code, there are *a lot* of opinions out there about "best practices". What one developer thinks is a must another will refute it. But these disagreements typically happen around specific, nuanced situations like tabs vs. spaces and if a curly brace should go on a new line.

There are larger categories of tips and best practices that are universal. Everyone can agree on broad tips like "write tests", "make code reusable" and "don't store passwords in clear text".

In this chapter, we're going to hit those broad strokes. We're going to cover the basic truths that *almost* everyone can agree on.

In the later chapters, we'll dive deeper into each of these areas to provide more specfic tips that we (the authors) and the community have come up with.

Without further ado, let's get to the tips!

Plan Before you Code

Don't automatically jump into coding. Instead, take a little bit of time to do a "back of the napkin" plan on what your code will do. Scaffold out code in comments briefly outlining what the code will do in those spots.

Write pseudocode. The practice of writing pseudocode will take your brain through the mental steps of what you need to do.

Further learning:

Link: How to write a Pseudo Code?[10]
Source: geeksforgeeks.org
Format: Article

Don't Reinvent the Wheel

Leverage the hard work of others. Don't completely write a new solution if one already exists. Issue pull requests to existing open source projects if an existing PowerShell module doesn't *quite* fit your needs. Don't fork it and build your own.

Look to the PowerShell Gallery first before embarking on a new script. Someone may have already solved that problem.

[10] https://www.geeksforgeeks.org/how-to-write-a-pseudo-code/

Further learning:

Link: Get Started with the PowerShell Gallery[11]
Source: docs.microsoft.com
Format: Article

Build Functions as Building Blocks

As you begin to build more complex PowerShell code, begin to think in functions; not lines of code. As you write code, consider if what you're writing could stand on it's own. Consider what the default commands in PowerShell do already. `Get-Content` reads a text file. `Test-Connection` checks the status of a network connection. `Copy-Item` copies a file.

If a script does more than one "thing", consider breaking it up into one or more functions. If you begin to collect a library of functions, create a module.

Further learning:

Link: Building Advanced PowerShell Functions and Modules[12]
Source: pluralsight.com
Format: Online Training Course

Build Re-usable Tools

Similar to the *Build functions as building blocks* tip, build PowerShell *tools*, not code. Focus on building re-usable scripts, functions or modules that you can re-use consistently. You should strive to build a library of tools that can be called from other code.

Instead of rewriting code over and over again, you should naturally begin to write code that calls tools. Eventually, you'll find the majority of your code will be making calls to your tools rather than re-creating the wheel.

Further learning:

Link: Learn PowerShell Toolmaking in a Month of Lunches[13]
Source: manning.com
Format: Book

Don't Focus Purely on Peformance

Jeffrey Snover, the father of PowerShell, once said that (paraphrasing) PowerShell was meant for humans, not computers. It was built not for blazing performance but to be human-readable. It was

[11] https://docs.microsoft.com/en-us/powershell/gallery/getting-started
[12] http://bit.ly/2Za0MBQ
[13] http://bit.ly/33HS1xA

built to be approachable for non-developers; the IT admins that need to automate tasks but can't and would rather not develop software applications.

If you're trying to eek out every last bit of speed from a PowerShell script for no reason other than to satisfy your own OCD tendencies, you're doing it wrong.

Further learning:

Link: Writing PowerShell Code for Performance[14]
Source: mcpmag.com
Format: Article

Build Pester tests

If you build scripts that make their way into production, always include Pester tests with it. Pester tests:

- ensure that your code (at all angles) "works"
- allows you to make changes and confirm new bugs weren't introduced
- helps you trust your code instead of being afraid you're going to break it by introducing changes

Build Pester unit tests to test your code. Build Pester integration/acceptance/infrastructure tests to confirm the code you wrote changed what you expected.

Further learning:

Link: The Pester Book[15]
Source: pesterbook.com
Format: eBook

Implement Error handling

Never let your code throw an exception you didn't account for. Use `$ErrorActionPreference = 'Stop'` religiously to ensure all errors are hard-terminating errors. Wrap all code in try/catch blocks and handle thrown exceptions. Don't leave any possibility for your code to exit without you already expecting it.

Further learning:

Link: The Big Book of PowerShell Error Handling[16]
Source: devops-collective-inc.gitbook.io
Format: eBook

[14] http://bit.ly/33SuCtl
[15] http://bit.ly/30n040F
[16] http://bit.ly/2TP1Tkw

Build Manageable Code

Build code for the future. Build PowerShell code that won't leave you wondering WTF this thing does a year from now. Ensure your code can be managed in the long-term. Practice the DRY (don't repeat yourself) principle. Write code once and refer to it rather than copying/pasting.

The fewer place code duplication exists, the simpler the code and the easier that code is to manage.

Further learning:

Link: Clean Code: A Handbook of Agile Software Craftsmanship[17]
Source: amazon.com
Format: Book

Don't Skimp on Security

Many PowerShell developers disregard security implications. It's not because they don't care (well, some don't), it's because they don't know any better. Infosec professionals and IT have always been separate wheelhouses. You may not need to know the ins and outs of vulnerability assessments, root kits, ransomware, encryption or trojans. But you do need to practice common security sense.

Don't put plain-text passwords in your scripts. Sign your scripts. Don't set your execution policy to unrestricted. We'll cover these and more in the security chapter.

Further learning:

Link: PowerShell Security at Enterprise Customers[18]
Source: blogs.msdn.microsoft.com
Format: Article

Log Script Activity

Record script activity across your organization. PowerShell is a powerful scripting language. PowerShell can automate tasks in no time but it doesn't discriminate between test and production. It can also be run by nefarious individuals.

Log script activity. Log it to a text file, a database or some other data source; just record and audit the activity somehow. Logging is like backups. You might not need them now but when you do, you'll be glad you did.

Further learning:

[17] https://amzn.to/32MzP4I
[18] http://bit.ly/2KLmO5k

Link: Greater Visibility Through PowerShell Logging[19]
Source: fireeye.com
Format: Article

Parameterize Everything

This tip is related to the functions and tools tip. When building scripts and functions, add as many parameters as necessary. Always add a parameter that might one day hold a different value. Never add "static" values to your scripts. If you need to define a value in code, create a parameter and assign it a default value. Don't assign that value in the code.

Creating parameters allows you to pass different values to your functions or scripts at run-time instead of changing the code.

Further learning:

Link: The PowerShell parameter demystified[20]
Source: adamtheautomator.com
Format: Article

Limit Script and Function Input

When you open up your scripts to input, limit what can be input as tightly as possible. Be sure to account for as many different ways as possible values may be passed into your code at run-time.

Validate as much input as possible with parameter validation attributes, conditional checks and so on. Try to never allow a value or type of input into your code you didn't expect.

Further learning:

Link: The PowerShell parameter demystified[21]
Source: adamtheautomator.coom
Format: Article

Maintain Coding Standards

Come up with a standard and stick to that for everythng. Don't name variables `$Var1` in one script and `$var1` in another. Don't include a bracket on the same line in one script and the bracket on a new line in the next.

Maintain a consistent coding methodology for everything. If your code is consistent, others (and you) will be able to understand your code much better.

[19] http://bit.ly/2TRoYmN
[20] http://bit.ly/2k2cETr
[21] https://adamtheautomator.com/the-powershell-parameter/#parameter-validation-attributes

Further learning:

Link: How to use the PowerShell Script Analyzer to Clean Up Your Code[22]
Source: interfacett.com
Format: Video

Code in Context

Write a solution specific to context in which it will run. Don't write a script that expects 10 parameters and give it to your helpdesk staff. Write a GUI instead. Don't quickly bang out a script without much thought if it's going to be used in production.

Worry about performance if the action your script is taking in time-sensitive. How you code always depends on the context in which it will run. Don't assume the code you write and test on your workstation is going to run fine in another environment. Code in the context that the script will run in.

Further learning:

Link: Clean Code: A Handbook of Agile Software Craftsmanship[23]
Source: amazon.com
Format: Book

Return Informative Output

Even though you can return anything you want to the output stream, do it wisely. Define what's verbose, informational and error output and only show that output in the various streams. Don't show unecessary object properties. Instead, use PowerShel formatting rules to hide properties from being seen by default.

Further learning:

Link: PowerShell Format Output View[24]
Source: sconstantinou.com
Format: Article

Understand Your Code

If you don't know what a line of code does, remove it or write a Pester test. Understand what each command or function you call is capable of at all times. Don't simply run a command under a single situation and expect it to run the same way every time.

[22] https://www.interfacett.com/videos/use-powershell-script-analyzer-clean-code/
[23] https://amzn.to/32MzP4I
[24] https://www.sconstantinou.com/powershell-format-output-view/

Test as many scenarios as possible to understand what your code is capable of until various circumstances.

Further learning:

Link: 10 Steps to Plan Better so you can Write Less Code[25]
Source: freecodecamp.org
Format: Article

Use Version Control

File names like *myscript.ps1.bak* and *myscript.ps1.bak2* shouldn't exist. Instead, use tools like Git and GitHub to put your scripts under version control. Version control allows you to audit and roll back changes to your code if necessary.

Version control becomes even more important in a team environment. If you're serious at all about PowerShell scripting, you must use version control.

Tip Source: https://twitter.com/Dus10

Further learning:

Link: Git Basics for IT Pros: Using Git with your PowerShell Scripts[26]
Source: techgenix.com
Format: Article

Write for Cross Platform

PowerShell isn't just on Windows anymore. PowerShell Core is cross-platform and so should be your scripts. If you ever see a time when your scripts need to run on other operating systems, account and test for that now.

If you're sharing scripts via the PowerShell Gallery or some other community repository, cross platform is especially important. Don't let others find out the hard way your script only runs on Windows.

Further learning:

Link: Tips for Writing Cross-Platform PowerShell Code[27]
Source: powershell.org
Format: Article

[25] https://www.freecodecamp.org/news/10-steps-to-plan-better-so-you-can-write-less-code-ece655e03608/
[26] http://techgenix.com/git-powershell/
[27] https://powershell.org/2019/02/tips-for-writing-cross-platform-powershell-code/

Write for the Next Person

Be sure other people understand your code. Write your code (and comments) in a clear, concise manner. A layperson should be able to look at your code and understand what it's doing.

Don't get fancy just because you can. Don't use aliases. Instead, write understandable code, include detailed help content and comment code heavily. Ensure the next person can easily digest your code. You never know. That next person might be you!

Further learning:

Link: Getting Fancy with Code Just Makes You Look Stupid[28]
Source: adamtheautomator.com
Format: Article

Use a Code Editor

You can't get away with a text editor anymore. Your code is too important to sloppily throw together and hope that it runs in a terminal. Using an integrated scripting environment (ISE) is a *must* if you want your code to be taken seriously. Our recommendation? Visual Studio Code with the PowerShell extension. You'll get all the benefits of a full ISE, linting functionality, autocompletion and more.

Further learning:

Link: PowerShell Tip: Use a Code Editor[29]
Source: brycemcdonald.net
Format: Article

[28] https://adamtheautomator.com/getting-fancy-code-just-makes-look-stupid/
[29] http://bit.ly/2Hkx2aC

Don’t Reinvent the Wheel

You are surrounded by a community of coding professionals that has, for a very long time, given a lot of their work away for free. By searching back through all of the community work that has been written, you can find a treasure trove of quality code that has been used by thousands of people. Although it gives us warm fuzzies to think that our code is so unique and so special that no one has ever thought to write what we’ve written, a simple search will show how many people in the community have already accomplished (albeit, with varying degrees of success) the same thing you’re trying to accomplish.

I don’t say this to try and put you down or shame you for not having ideas that are good enough, all I’m trying to do is get you to reuse the code that was written before you and to leverage others work. This will save you time, brainpower, and all of those creative juices that you need to write your best code possible.

Use Community Modules

The PowerShell Gallery is a great place to find community modules that are available to the public. If you’re not familiar, the Gallery will allow you to discover and download modules, read public documentation, and even update the modules from the PowerShell terminal! Once you’re comfortable, you can even contribute back to the community by uploading your own modules and scripts for public consumption.

Link: PowerShell Gallery: Getting Started[30]
Source: docs.microsoft.com
Format: Article

Leverage Others’ work

Code reuse is becoming just as much of an artform as it is a skill. Thankfully, as more time goes on, this is becoming more of the norm than the exception. By practicing and developing your code re-use skills, you’ll be able to integrate and operate more code than you would otherwise. In turn, this will help you create code with a higher degree of complexity or solve more high-level problems than you’d be able to do if you were trying to code everything yourself.

Link: 10 Tips on Writing Reusable Code[31]
Source: hoskinator.blogspot.com
Format: Article

[30] https://docs.microsoft.com/en-us/powershell/gallery/getting-started
[31] http://hoskinator.blogspot.com/2006/06/10-tips-on-writing-reusable-code.html

Plan Before you Code

As a coder and avid scripter, it's hard to not just get down to coding right away. Planning and documentation is boring. You just want to get to the fun stuff!

Although a laissez faire attitude may work for simple scripts, you'll soon find yourself in a world of hurt if no planning is done on larger projects. Putting together a plan, whether it be a simple back-of-the-napkin sketch or an entire project outline is critical for ensuring success on larger projects.

If you don't plan ahead, you'll find yourself having to put fixes in code to cover up previous problems, introduce unecessary performance degradation and end up with a plate of spaghetti code. Poor planning will force you to accrue technical debt and will always result in a management nightmare.

Write Comments Before Coding

Large software projects require extensive planning before diving into code. Why shouldn't your important PowerShell scripts get the same treatment albeit at a much smaller scale?

It may be more fun to start coding away immediately, refrain. Ask yourself what the end goal of the script is and document it with comments. Think through what the script will do before writing a single line of code. Instead, break down each task in your head and document it with simple comments in the script.

Regions[32] are a great commenting feature to use when planning. Regions allow you to easily collapse and expand parts allowing you to pay attention to the task at hand.

Don't worry about using a particular comment structure. Use whatever structure is most comfortable for you to guide you through the coding process when the time comes.

Tip Source: https://twitter.com/duckgoop

Use your Code as a Todo List

We've all had those times when you're in the middle of a script and get interrupted somehow. You may be in just the right frame of mind and on the cusp of solving a problem that's been plaguing you for days. Or, you might be building a script as fast as possible but don't want to forget to come back to a certain area. Either way, using your code as a todo list will help.

Perhaps you know code will break under certain circumstances, see a way to improve performance or perhaps you're working on a team and need to assign a junior developer a piece of code, try adding `## TODO` to the code at that particular line.

[32] https://devblogs.microsoft.com/scripting/use-regions-in-powershell-ise-2/

The comment doesn't have to specifically be `## TODO`. The point here is to create a "comment flag" that points to areas that need to be addressed at some point. At some point in the future, you can then perform a search on the codebase for that flag and find all of the instances to address.

Tip Source: https://twitter.com/guyrleech

Further learning:

Link: 5 ways using TODO comments will make you a better programmer[33]
Source: medium.com
Format: Article

[33] https://medium.com/imdoneio/5-ways-using-todo-comments-will-make-you-a-better-programmer-240abd00d9e4

Create Building Blocks with Functions

Once you create a few PowerShell scripts, you're bound to start feeling like you're recreating the wheel. You will inevitably begin seeing patterns in what solutions you build with PowerShell.

One of the most important concepts when building great PowerShell code is treating code like building blocks. Don't build unique solutions for all problems. Instead, build blocks in the form of functions and modules and then use those modules.

Over time, you'll find that you're saving tons of time by not rewriting the same code you wrote months ago and you only have one piece of code to maintain rather than ten copies.

Write Functions with One, Single Goal

Functions are supposed to be small, bite-sized code snippets that perform a single action. Build functions to not boil the ocean but to increase the temperate one degree at a time.

Functions should serve one, primary purpose and should be easily describable with a quick glance of the code. If you find it hard to describe a function without saying the word "and", it probably needs to be split into multiple functions.

Make functions small and easily callable from other functions. Functions are your building blocks. Build solutions with blocks not by pouring a solid building of concrete at once.

Tip Source: https://twitter.com/pgroene

Further learning:

Link: PowerShell Functions Introduction[34]
Source: adamtheautomator.com
Format: Article

Build Functions with Pipeline Support

PowerShell wouldn't be PowerShell without the pipeline. Be a good PowerShell citizen and include pipeline support in your functions. Passing command output to other commands via the pipeline is an intuitive way to run functions. It's also easier for newcomers to understand.

Further learning:

[34] https://adamtheautomator.com/powershell-functions/

Link: The PowerShell Parameter Demystified[35]
Source: adamtheautomator.com
Format: Article

Save Commonly-Used, Interactive Functions to Your Profile

If you're working at the PowerShell console interactively, it's always handy to have a function created and available as a shortcut to performing a certain task. Think about all of the common tasks you perform while working in the PowerShell console.

To ensure these "console" functions are available to you across all sessions, add them to your PowerShell profile. This will allow you to always have them available.

Tip Source: https://www.reddit.com/user/TheDinosaurSmuggler/

Further learning:

Link: Understanding the Six PowerShell Profiles[36]
Source: microsoft.com
Format: Article

[35] https://adamtheautomator.com/the-powershell-parameter/#pipeline-input
[36] https://devblogs.microsoft.com/scripting/understanding-the-six-powershell-profiles/

Parameterize Everything

One of the key differences between a simple script and a PowerShell tool are parameters. Parameters allow developers to write scripts that are reusable. Parameters don't force developers to edit their scripts or functions every time they need to run them. They allow users to modify how the script or function works without modifying the code.

Parameters are an integral component of building a reusable PowerShell tool that turns ad-hoc scripts into building blocks.

In this chapter, you'll learn many different tips on how to properly use parameters in your daily life.

Don't Hardcode. Always Use Parameters

You should make it your mission to reuse as many scripts and functions as possible. There's no need to recreate the wheel. One of the easiest ways to do that is to define parameters for *everything*.

Before you finish up a script, think about how it could be reused for other similar purposes. Which components may need to be changed for next time?Is it a script to run against a remote computer? Make the computer name a parameter. How about referencing a file that could be anywhere. Create a `FilePath` parameter.

If, most of the time, the value is the same, set a default parameter and override it as necessary.

Don't hardcode values that may change in scripts or functions.

Further learning:

Link: PowerShell Parameters: Everything You Ever Wanted to Know[37]
Source: adamtheautomator.com
Format: Article

Use Parameter Sets When All Parameters Should Not be Used at Once

If you have a script or function with parameters that cannot be used at the same time, create parameter sets. For example, if a function takes input via an object *or* a simple name, for example, create two parameter sets.

[37] https://adamtheautomator.com/the-powershell-parameter/

Perhaps you have a function called `Get-Server` and `Reboot-Server`. On the `Reboot-Server` function, you have a parameter to accept pipeline input like `Get-Server | Reboot-Server`. You also have a `Name` parameter used like `Reboot-Server -Name FOO`. Each of these instances cannot be used at the same time; create parameter sets for each of them.

Further learning:

Link: PowerShell Parameters: Everything You Ever Wanted to Know[38]
Source: adamtheautomator.com
Format: Article

Use a PSCredential Object Rather than a Separate Username and Password

PowerShell has a type of object called PSCredential. This object stores a username and password with the password securely encrypted.

When you write a new script or function, use a `[pscredential]$Credential` parameter rather than a `UserName` and `Password`. It's cleaner, ubiquitously common in the PowerShell world and a more secure way to pass sensitive information to a function.

Tip Source: https://www.reddit.com/user/thedean_801/

Further learning:

Link: Using the PowerShell Get-Credential Cmdlet and all things credentials[39])
Source: adamtheautomator.com
Format: Article

[38] https://adamtheautomator.com/the-powershell-parameter/#parameter-sets
[39] https://adamtheautomator.com/powershell-get-credential/

Log Script Activity

If you don't know what your code is doing, how are you supposed to troubleshoot it? How are you supposed to optimize it? To show what it changed in an environment? Log everything!

Especially on long-running scripts or scripts executed in production, you must bake logging routines into your code. It's not only helpful to monitor activity when things go well, it's especially helpful when problems arise that need investigation.

In this chapter, you'll learn some tips on how best implement logging in your PowerShell scripts.

Use a Logging Function

Logging to a text file is an extremely common need. Great scripts log everything and do it a lot. This seems like a good opportunity to build a little tool to do that for us!

You should have or be building up an arsenal of small helper functions. A `Write-Log` function needs to be in that arsenal somewhere. By creating a function with a few parameters like `Message`, `Severity`, etc which then records information to a structured text file, is a must-have.

Tip Source: https://twitter.com/brentblawat

Further learning:

Link: How to Build a Logging Function in PowerShell[40]
Source: adamtheautomator.com
Format: Article

Clean up Verbose Messages

Think of a PowerShell script as a piece of software. Software has three rough stages of creation; development, testing and production. It's a shame so many scripts out there are "in production" but still have development code throughout.

One example of "development code" in a script is all of those `Write-Verbose` lines. Developers use `Write-Verbose` to troubleshoot and debug code while it's being developed. That code isn't necessary when the code is solid. You don't need to return minute, granular details you required when debugging in the final product. Remember to clean all of those debugging messages up before signing off.

[40] https://adamtheautomator.com/how-to-build-a-logging-function-in-powershell/

Instead, consider taking the next step in “professional” development and using the PowerShell debugger.

Tip Source: https://twitter.com/JimMoyle

Further learning:

Link: Quick and Efficient PowerShell Script Debugging with Breakpoints[41]
Source: informit.com
Format: Article

[41] http://www.informit.com/articles/article.aspx?p=2421573

Build with Manageability in Mind

You can write code all day to solve all the things but if you can't manage it over time, you're sunk. It's important to not only solve the problems of today but think about how those solutions will be maintained over time.

DRY: Don't Repeat Yourself

Notice when you're repeating the same code snippets. Be cognizant you're following the same patterns over and over again. Being great at coding is about pattern recognition and improving efficiency.

Don't type out the same command ten times to process ten different parameter values. Use a loop. Write "helper" functions that can be called from other functions to eliminate writing that same code again.

Further learning:

Link: The DRY Principle: How to Write Better PowerShell Code[42]
Source: adamtheautomator.com
Format: Article

Don't Store Configuration Items in Code

Always treat configuration items that are required in your code as separate entities. Items like usernames, passwords, API keys, IP addresses, hostnames, etc are all considered configuration items. Configuration items should then be pointed to in your code. These artifacts are static values that should be injected in your code; not stored in the code.

Separating out configuration items from your code allows you the code to be more flexible. It allows you to make a change at a global level and that change be immediately consumed by the code.

Further learning:

Link: Configuration PowerShell Module[43]
Source: github.com
Format: GitHub project

[42] https://adamtheautomator.com/dry-principle-powershell/
[43] https://github.com/PoshCode/Configuration

Always Remove Dead Code

Although not critical, leaving code in scripts that will never be executed is bad practice. It clutters up the important code and makes it harder to understand and troubleshoot scripts. Remove it.

Look into using the code coverage options in Pester to discover all of the unused code in your scripts.

Tip Source: https://twitter.com/JimMoyle

Further learning:

Link: Testing Pester Code Coverage[44]
Source: sapien.com
Format: Article

[44] https://www.sapien.com/blog/2016/06/24/testing-pester-code-coverage/

Be Specific

One of the most important rules to remember in coding, not just PowerShell, is never to assume. Never assume your user will run your code as you'd expect. Never assume that PowerShell will surface any information you need to see. Be specific about everything.

In this chapter, you'll learn a few tips on how you can practice specific coding and get away from those assumptions.

Use Strict Mode

PowerShell has a feature called strict mode. This mode imposes various restrictions when your code is run. This mode ensures you don't use variables that have not been defined, cannot reference non-existent object properties and more.

If your code does not meet these requirements, it will return an error. Use this mode for all coding you do. It will ensure you are explicit as possible with your code. It forces you to account for situations that may end up hurting you in the long run.

Further learning:

Link: Enforce Better Script Practices by Using Set-StrictMode[45]
Source: devblogs.microsoft.com
Format: Article

Don't Ignore Errors

PowerShell has a concept called terminating and non-terminating errors. Terminating errors typically are treated as more severe than non-terminating errors. When a terminating error is surfaced, PowerShell will terminate code execution. This behavior is contradictory to a non-terminating error which will return that ugly red text to the user but will not stop code execution.

PowerShell developers can ignore non-terminating errors completely, if they wish by using the `$ErrorActionPreference` automatic variable or the `ErrorAction` parameter on cmdlets and advanced functions. If you don't want to see a non-terminating error ever again, simply run `$ErrorActionPreference = 'Ignore'` or `$ErrorActionPreference = 'SilentlyContinue'`.

If the nuclear approach doesn't suit your fancy, you could also use the `ErrorAction` parameter on cmdlets and advanced functions. Setting this value to `Ignore` or `SilentlyContinue` will perform the same behavior not globally just for that particular command.

[45] http://bit.ly/2L3Y7jf

Don't do this!

Instead, use `$ErrorActionPreference = 'Stop'` or `-ErrorAction 'Stop'` on individual commands to "convert" non-terminating errors to terminating errors and catch them with try/catch blocks!

Further learning:

Link: About Preference Variables[46]
Source: docs.microsoft.com
Format: Article

Validate Input Parameters

If you have a function or script that require input of any kind, always validate parameters as much as possible. Leverage PowerShell's parameter validation attributes religiously.

If you know ahead of time what specific values, patterns a string value should match, a number in a particular range, etc., always use parameter validation attributes. Doing so will prevent unnecessary error handling in the function/script and limit the ways it can blow up.

Further learning:

Link: Parameter Validation Attributes[47]
Source: adamtheautomator.com
Format: Article

Explicitly Define Parameter Types

When setting up parameters, always define object types that are allowed with that parameter. For example, you can define parameters with no type at all like below:

```
function Do-Thing {
    [CmdletBinding()]
    param(
        [Parameter()]
        $ParamHere
    )
}
```

The `Do-Thing` function though now accepts anything as a value for `ParamHere`. You can pass it a string, array, hashtable, integer...anything! Chances are, you depend on the value of that parameter to be a specific type inside of the function.

Rather than learning the hard way, limit the type to what you need. For example, if you know the value of `ParamHere` should be a string, define it as such as shown below.

[46] http://bit.ly/2kCTzYh
[47] https://adamtheautomator.com/the-powershell-parameter/#parameter-validation-attributes

```
function Do-Thing {
    [CmdletBinding()]
    param(
        [Parameter()]
        [string]$ParamHere
    )
}
```

By explicitly defining the type of parameter it's value should be will save you a lot of headache down the road.

Further learning:

Link: PowerShell Data Types[48]
Source: ss64.com
Format: Article

Enforce Mandatory Parameters

If you're depending on a parameter value being passed to a function, always make it mandatory with the `Mandatory` parameter attribute. The function will work as expected if the function is called as you'd expect, but that doesn't always happen.

Lock it down and be 100% sure a needed parameter value is used by using the `Mandatory` parameter attribute.

Further learning:

Link: The Mandatory Parameter Attribute[49]
Source: adamtheautomator.com
Format: Article

Use the #requires Statement

Shared by: https://twitter.com/MerlinFromBE

If you know your script needs to run with a specific version of PowerShell, on a system with individual modules installed, run as administrator or with Windows PowerShell or PowerShell Core, use the `#requires` statement.

The `#requires` statement is a simple one-line defined at the top of your script, ensuring it will only execute under certain conditions.

[48] https://ss64.com/ps/syntax-datatypes.html
[49] https://adamtheautomator.com/the-powershell-parameter/?nocache#the-mandatory-parameter-attribute

Using the `#requires` statement limits the script from running until it's run in a specific, pre-defined environment.

Further learning:

Link: about_Requires[50]
Source: docs.microsoft.com
Format: Article

[50] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_requires

Write for the Next Person

Write your code so the next person has an easy time and you'll end up helping yourself. More often than not it seems that after not touching a piece of code for six months or more, *you* are the "next person" who ends up having to come back through and clean up what you previously wrote. By taking the precautions below, you'll be able to save yourself and the next coder countless hours on your projects.

Shared by: John Eijgensteijn Twitter[51]

Give your Variables Meaningful Names

Variable names should be meaningful. When you go back through your code, you shouldn't be wondering what `$x` is or how you are supposed to use it. You should be able to pick up exactly what it's supposed to do the first time through. With PowerShell, there's no practical limit to how long your variable names should be, so feel free to be as descriptive as possible.

Tip Source: - make variable names meaningful. "self-documenting code" - https://twitter.com/lh_-aranda

Link: The PowerShell variable - Naming, value, data type[52]
Source: 4sysops.com
Format: Article

String Substitution

By utilizing a technique called "String Substitution" we're able to make our code more concise and more readable. When we combine placeholders with the `-f` format operator and assign variables we make something that can be long or complex and boil it down to a single line.

Tip Source: https://twitter.com/harringg

Further learning:

Link: Keep Your Hands Clean: Use PowerShell to Glue Strings Together[53]
Source: devblogs.microsoft.com
Format: Article

[51] https://twitter.com/JohnEijg
[52] https://4sysops.com/archives/the-powershell-variable-naming-value-data-type/
[53] https://devblogs.microsoft.com/scripting/keep-your-hands-clean-use-powershell-to-glue-strings-together/

Don’t use Aliases in a Script

Aliases can help us quickly type a PowerShell one-liner much more quickly than if we were to use the “long form” command. There are a few problems with aliases in a script, though. For starters, aliases can vary system by system, so you don’t know if the alias will always be the same on every machine. Aliases can be less readable than the long form counterparts, and because of this they can make your scripts more difficult to understand. Best practice regarding aliases? Just don’t use them.

Tip Source: https://twitter.com/TheTomLilly

Further learning:

Link: When You Should Use PowerShell Aliases[54]
Source: devblogs.microsoft.com
Format: Article

Put functions in Alphabetical Order in a Module

If you have many functions in a module script, put them in alphabetical order. Doing so will allow you or the next person to quickly scan down the file and find the function they’re looking for. Many editors will read the module and provide a function list too allowing easy navigation.

Tip Source: https://twitter.com/raychatt

Explain Regular Expressions with Comments

If you have to write a complicated regular expression in a script, be sure to provide a comment above it indicating exactly what kind of string it matches. There aren’t many regular expression gurus out there that can read a regex string like reading command name. Make the code as easy to understand as possible.

Include some example strings the regex expression matches too. Examples are always appreciated.

Tip Source: https://twitter.com/guyrleech

Write Comment-Based Help

Always include comment-based help especially if you’re sharing scripts with other people. Comment-based help is the easiest kind of help to write. It shows up when a user runs `Get-Help` and acts as comments in the code too. Comment-based help should be standard across all production-ready scripts you write.

[54] https://devblogs.microsoft.com/scripting/when-you-should-use-powershell-aliases/

Further learning:

Link: Building Advanced PowerShell Functions and Modules[55]
Source: pluralsight.com
Format: Course

Weigh the Difference Between Performance and Readability

When writing code, you're forced to weigh decisions based on many factors. Two factors that sometimes collide are performance and readability. It's important for code to accomplish a task as quickly as possible but not at the cost of readability. Even though a computer doesn't need white space, comments and long command names doesn't mean humans don't.

Further learning:

Link: Consider trade-offs between performance and readability[56]
Source: github.com
Format: Article

[55] https://www.pluralsight.com/courses/powershell-modules-advanced-functions-building

[56] https://github.com/PoshCode/PowerShellPracticeAndStyle/blob/master/Best-Practices/Performance.md#perf-02-consider-trade-offs-between-performance-and-readability

Handle Errors Gracefully

Believe it or not, your PowerShell script isn't going to work right all the time. It will fail and fail hard sometimes. A novice developer typically doesn't worry much about error handling. Error handling is one of those topics that separates the novices from the professionals.

To create a production-ready, robust PowerShell solution, you must ensure all the ways your scripts can fail are properly handled. Inevitably, they will throw an error when you least expect it and it's important to catch that error to either fix the problem mid-run or exit gracefully.

Force Hard-Terminating Errors

Unlike many other languages, PowerShell has two types of exceptions/errors - soft and hard-terminating errors. When encountered, a soft-terminating error does not stop script execution. It does not terminate code execution. A hard-terminating error, on the other hand, does.

Hard-terminating errors or exceptions change the "flow" of code thus they allow you more control using try/catch blocks.

When in doubt, use the common `ErrorAction` parameter on commands and set it to `Stop`. If you'd like to ensure all errors will be thrown as exceptions, set the automatic variable `$ErrorActionPreference` to `Stop`.

Ensuring all errors are thrown as exceptions allows you to expect and better control any errors that come up in your code.

Further learning:

Link: Error Handling: Two Types of Errors[57]
Source: devblogs.microsoft.com
Format: Article

Avoid Using $?

When you write code, it's important to make your code and what your code does as simple as possible. Regardless if you can save a few characters here and there, you should always focus on writing code that can easily be read and understood.

PowerShell has a set of variables that come by default - one of those variables is `$?`. This variable indicates if the last command executed was successful or not by returning `True` or `False`. Don't use it.

[57] https://devblogs.microsoft.com/scripting/error-handling-two-types-of-errors/

First of all, `$?` isn't the most intuitive naming convention. If someone is skimming your code, they may not know what this means. Also, it simply returns a boolean `True` or `False`. There are no other indicators why that error happened.

Further learning:

Link: About Automatic Variables[58]
Source: docs.microsoft.com
Format: Article

Copy `$Error[0]` to your Own Variable

PowerShell keeps a "log" of all errors returned in an array called `$Error`. This array contains a time-sorted list of all error records returned from code in a session.

It's sometimes necessary to find the latest error returned by referencing `$Error[0]`. What some might not realize is that the `$Error` array is always changing. What `$Error[0]` represents now might be completely different a second from now.

If you do intend to use the value of `$Error[0]` in your scripts, be sure to assign that value to your own variable to ensure the value is what you expect when you reference it.

Further learning:

Link: Using PowerShell $Error Variable[59]
Source: maxtblog.com
Format: Article

[58] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-7
[59] http://www.maxtblog.com/2012/07/using-powershell-error-variable/

Don’t Skimp on Security

As developers and system administrators come together to form DevOps, it’s important to not exclude security. Security is an extremely important topic especially in today’s day and age and is one that we can begin to include in our everyday PowerShell code.

Injecting security in PowerShell code is a deep topic and one that could not be completed in a single chapter let alone a single book. But, in this chapter, you’ll learn some tips to take care of some of the low-hanging fruit easily obtained by using some best practices.

Sign Scripts

PowerShell has a built-in method to crytographically sign all scripts to ensure they are not tampered with. Signing a script adds a cryptographic hash at the bottom of the script. When an execution policy is set to RemoteSigned or AllSigned, PowerShell will not allow the script to run if it detects the code has been modified since it was last signed.

Tip Source: https://twitter.com/brentblawat

Further learning:

Link: PowerShell Basics - Execution Policy and Code Signing[60]
Source: darkoperator.com
Format: Article

Use Scriptblock Logging

It’s critical to know what code is being executed in your environment. Unfortunately, if you download a script from the Internet, click a malicious link or a user gets a phishing email, that code may be malicious.

By enabling scriptblock logging in your environment, you can see, down to the scriptblock level, exactly what that code is doing and developing a proper audit trail.

Further learning:

Link: Greater Visibility Through PowerShell Logging[61]
Source: fireeye.com
Format: Article

[60] https://www.darkoperator.com/blog/2013/3/5/powershell-basics-execution-policy-part-1.html
[61] https://www.fireeye.com/blog/threat-research/2016/02/greater_visibilityt.html

Never Store Sensitive Information in Clear Text in Code

This should go without saying but saving passwords, private keys, API keys or any other sensitive information in code is a bad idea. PowerShell allows you many different ways to encrypt this information and decrypt it, when necessary.

Use the `Export` and `Import-CliXml` commands to encrypt and decrypt objects with sensitive information like credentials. Use secure strings rather than plain-text strings, generate your own cryptographic keys and more with PowerShell. There are lots of ways to encrypt and decrypt sensitive information with PowerShell.

Further learning:

Link: How to Encrypt Passwords in PowerShell[62]
Source: altaro.com
Format: Article

Don't use Invoke-Expression

Using the `Invoke-Expression` command can open up your code to code injection attacks. The `Invoke-Expression` command allows you to treat any type of string as executable code. This means that whatever expression you pass to `Invoke-Expression`, PowerShell will gladly execute it under whatever context it's running in. Although executing the expression you intend works great what happens when you open up the code to others?

Perhaps you're accepting input from a webpage or from another script that others' use and their or your system is compromised. By using `Invoke-Expression`, you're essentially giving malicious users an easy way to run PowerShell on your system.

Further learning:

Link: Invoke-Expression considered harmful[63]
Source: devblogs.microsoft.com
Format: Article

Use PowerShell Constrained Language Mode

If you need to allow junior users, employees in your company or service applications the ability to run PowerShell commands, you need to ensure that access is least privilege. There's no need to allow running commands others do not need and may accidentally or purposefully introduce security issues.

[62] https://www.altaro.com/msp-dojo/encrypt-password-powershell/
[63] https://devblogs.microsoft.com/powershell/invoke-expression-considered-harmful/

PowerShell has a mode called constrained language mode that allows you to provide access to a PowerShell environment but not allow access to all commands and modules. Constrained language mode allows you to granularly define activities as allowed and disallowed giving you tight control over what can be done.

Further learning:

Link: PowerShell Constrained Language Mode[64]
Source: devblogs.microsoft.com
Format: Article

[64] https://devblogs.microsoft.com/powershell/powershell-constrained-language-mode/

Stick to PowerShell

PowerShell is a very forgiving scripting language, and that's part of its appeal. When you're working with PowerShell, you can borrow code from C#, .NET, or even COM objects and integrate them seamlessly into your code. This sounds great in theory, but when it comes to working on your code in a team of PowerShell developers, it's best to stick to PowerShell. By straying from it, you could end up leaving your team confused, making your code less understandable, and really giving the next person who needs to edit your scripts a hard time.

Use Native PowerShell Where Possible

With PowerShell, you have a nearly limitless toolset. If you need to accomplish something outside of what you can find PowerShell commands for, you have the option to use COM objects, .NET classes, and so on. That being said, don't immediately jump to using these "non-PowerShell" methods. When there's an existing PowerShell way of doing something, use it! There's a strong chance that it'll process better, faster, or give you an output that's easier to use in the end.

Tip Source: If there is a PowerShell cmdlet and you decide to use C# instead, comment why you didn't use the posh cmdlet and what it would have been. Not everyone who maintains a PoSh script can do C# - https://twitter.com/JimMoyle

Further learning:

Link: Comparing WMI and Native PowerShell[65]
Source: devblogs.microsoft.com
Format: Article

Use PowerShell standard cmdlet naming

Once you dive into PowerShell for a little while, you'll realize that there is a standard naming convention for the cmdlets you write. Each of the verbs in the verb-noun nomenclature do something very specific. In order to make PowerShell as universal as possible, your cmdlets should use the right verbs appropriately.

Further learning:

Link: Approved Verbs for PowerShell Commands[66]
Source: docs.microsoft.com
Format: Article

[65] https://devblogs.microsoft.com/scripting/comparing-wmi-and-native-powershell/
[66] https://docs.microsoft.com/en-us/powershell/developer/cmdlet/approved-verbs-for-windows-powershell-commands

Build Tools

As you begin to write more PowerShell, you'll probably find that you keep reinventing the wheel. This is natural. The reason you're doing this is because you're not building upon the scripts you previously created. You're not building script libraries, modules and tools.

Code for Portablity

Even if not required, always code for portability. Write your code that's decoupled from the current environment. Think about how the code would react if you moved it to the cloud, to another test environment or to just another computer. This kind of methodology is accomplished by building PowerShell modules instead of scripts, creating functions to interface with environmental components and using variables representing things like domain names, user accounts, server names, etc.

Further learning:

Link: Building Advanced PowerShell Functions and Modules[67]
Source: pluralsight.com
Format: Online Course

Wrap Command-Line Utilities in Functions

If you have to use a command-line utility, wrap it in a PowerShell function. If it returns output, parse the output and make it return a pscustomobject. This allows a CLI utility to act like any other PowerShell command. Once you've abstracted away all of the "CLIness", the command can be easily integrated with other PowerShell tools.

Further learning:

Link: Solve Problems with External Command Lines in PowerShell[68]
Source: microsoft.com
Format: Article

[67] https://app.pluralsight.com/library/courses/powershell-modules-advanced-functions-building/table-of-contents
[68] https://devblogs.microsoft.com/scripting/solve-problems-with-external-command-lines-in-powershell/

Force Functions to Return Common Object Types

If you have different PowerShell functions that work together, be sure they always return the same type of object. This object type is usually a pscustomobject because it's generic and easy to create. If functions are all returning a pscustomobject type, you'll know what to expect when it returns information. It also makes troubleshooting easier.

Further learning:

Link: Powershell: Everything you wanted to know about PSCustomObject[69]
Source: powershellexplained
Format: Article

Ensure Module Functions Cover all the Verbs

If you have an immediate need to accomplish a task like creating a user account, removing a file or modifying a database record, don't just create a single function. Instead, create four functions that cover the complete lifecycle of that object - New, Set, Get and Remove.

For example, perhaps you're creating a module for a monitoring appliance that has an API. You decide on an "object" noun of Monitor. If you need to create a new monitor in an automation script, don't just create the `New-Monitor` function. Instead, create `Get-Monitor`, `Set-Monitor` and `Remove-Monitor` to ensure you have support for the monitor's full lifecycle.

Tip Source: https://twitter.com/JimMoyle

Further learning:

Link: How To Design a PowerShell Module[70]
Source: mcpmag.com
Format: Article

[69] https://powershellexplained.com/2016-10-28-powershell-everything-you-wanted-to-know-about-pscustomobject/
[70] https://mcpmag.com/articles/2015/09/17/design-a-powershell-module.aspx

Return Stndardized, Informative Output

Have you ever run a script or function you received from someone else and wondered if it worked? It ran without showing an error but then again, it returned nothing at all! You don't have a clue what it did nor could see it's progress as it was executing.

Because it didn't return any object to the pipeline, you also can't use it's outcome in other commands. You are forced to write more code to check whether it did it's job which results in wasted time and added complexity.

In this chapter, you're going to learn some tips on what to return to your user, how often and how to return output in many different ways.

Use Progress Bars Wisely

PowerShell has a handy cmdlet called `Write-Progress`. This cmdlet displays a progress bar in the PowerShell console. It's never a good idea to leave the user of your script staring at a blinking cursor. The user has no idea if the task should take 10 second or 10 minutes. They also might think the script has halted or crashed in some manner. It's important you provide some visual cues as to what's going on.

Use `Write-Progress` for any task that takes more than 10 seconds or so. The importance increases with the time. For more complicated scripts with many transctions, be sure to use `Write-Progress` at a higher level. Use the progress bar as the main indicator of progress and leave the smaller steps to a verbose message, for example.

Tip Source: https://twitter.com/brentblawat and https://twitter.com/danielclasson

Further learning:

Link: Add a Progress Bar to Your PowerShell Script[71]
Source: microsoft.com
Format: Article

Leave the Format Cmdlets to the Console

For any script that returns some kind of output, a well-developed script contains two "layers" - processing and presentation. The processing "layer" contains all of the code necessary to perform

[71] https://devblogs.microsoft.com/scripting/add-a-progress-bar-to-your-powershell-script/

whatever task is at hand. The "presentation" layer displays what the scripts outputs to the console. Never combine the two.

If you need to change up what the output looks like in the console, do it outside the script. For example, never use a `Format-*` cmdlet inside of a script. You should treat scripts like reusable tools. Unless you are 100% certain that script will never need to send output to another script or function, don't attempt to format the output in the script. Instead, pipe objects from the script into a formatting command at the console or perhaps another "formatting" script.

Further learning:

Link: Using Format Commands to Change Output View[72]
Source: microsoft.com
Format: Article

Use Write-Verbose

Verbose messaging comes in handy in many different scenarios from troubleshooting, script progess indication and debugging. Use the `Write-Verbose` cmdlet as much as possible to return granular information about what's happening in a script. Use verbose messages to display variable values at runtime, indicate what path code takes in a condition statement like if/then or to indicate when a function starts and stops.

There are no defined rules to indicate when to return a verbose message. Since verbose messaging is off by default, you don't need to worry about spewing text to the console. It's better to have more information than less when it comes to verbose messaging.

Tip Source: https://twitter.com/UTBlizzard

Further learning:

Link: Write-Verbose Help Cmdlet[73]
Source: microsoft.com
Format: Article

Use Write-Information

PowerShell has six streams[74]. You can think about three of those steams in terms of verbosity - Debug, Verbose and Information. The debug stream is at the bottom and should return lots of granular information about code activity while the information stream should contain high-level messages.

Use the `Write-Information` cmdlet to display top-level information similar to what would be down in a progress bar. The user doesn't need to see variables, if/then logic or any of that. Use `Write-Information` to display basic, high-level status messages about script activity.

[72] https://docs.microsoft.com/en-us/powershell/scripting/samples/using-format-commands-to-change-output-view?view=powershell-7
[73] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-verbose
[74] https://pwsh.pro/understanding-streams-in-powershell

Further learning:

Link: Welcome to the PowerShell Information Stream[75]
Source: microsoft.com
Format: Article

Ensure a Command Returns One Type of Object

Nothing will confuse a PowerShell developer more than when a script or function returns different types of objects. Keep it simple and ensure regardless of the circumstances, the command only returns one type.

When a script or function returns different types of objects based on various scenarios, it becomes hard to write code that takes input from that command.

Further learning:

Link: About Functions OutputTypeAttribute[76]
Source: microsoft.com
Format: Article

Only Return Necessary Information to the Pipeline

If a command returns information you have no use for, don't allow it to return objects to the pipeline. Instead, assign the output to `$null` or pipe the output to the `Out-Null` cmdlet.

If the command should return the output **sometimes** but not **all** the time, create a `PassThru` parameter. By creating a `PassThru` parameter, you give the user the power to decide to return information or not.

Tip Source: https://twitter.com/JimMoyle

Further learning:

Link: The PassThru Parameter: Gimme Output[77]
Source: sapien.com
Format: Article

[75] https://devblogs.microsoft.com/scripting/weekend-scripter-welcome-to-the-powershell-information-stream/
[76] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_outputtypeattribute?view=powershell-7
[77] https://www.sapien.com/blog/2014/10/06/the-passthru-parameter-gimme-output/

Build Scripts for Speed

Although this chapter conflicts with a tip on not purely focusing on performance, there's a fine line to follow. On the one hand, you don't need to get bogged down shaving off microseconds of runtime. On the other hand, though, you shouldn't completely disregard script performance.

There is a gray area that you need to stay within to ensure a well-built PowerShell script.

Use an ArrayList or GenericList .NET Class when Elements Need to be Added to an Array

A common action nearly every PowerShell newcomer does is to use += to append a new item to an array. This method works just fine most of the time. But I wouldn't recommend it. Why? Because if you're adding an element to an array with ten items, += will work great and you won't notice any performance degradation. But, try it with an array with 1,000,000 elements in it. You'll get quite a different picture vs. using a .NET ArrayList[78] or GenericList class[79].

Further learning:

Link: PowerShell scripting performance considerations[80]
Source: docs.microsoft.com
Format: Article

Use a Regular Expression to Search Multiple String Values

If you need to search a string or a collection of strings for a group of values, don't do this: `$array | Where-Object { $_.Name -eq 'foo' -or $_.Name -eq 'bar' -or $_.Name -eq 'baz' }`. Instead, do this: `$array | Where-Object { $_.Name -match 'foo|bar|baz' }`. It's more concise and performs the same task.

Tip Source: https://twitter.com/brentblawat

Further learning:

Link: Alternation with The Vertical Bar or Pipe Symbol[81]
Source: regular-expressions.info
Format: Article

[78] https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist?view=netframework-4.8
[79] https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1
[80] https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/powershell/script-authoring-considerations
[81] http://bit.ly/2TTWNDI

Don't use `Write-Host` in Bulk

Although some would tell you never to use the `Write-Host` cmdlet, it still has it's place. But, with the functionality, it brings also brings a small performance hit. `Write-Host` does nothing "functional." The cmdlet outputs text to the PowerShell console.

Don't add `Write-Host` references in your scripts without thought. For example, don't put `Write-Host` references in a loop with a million items in it. You'll never read all of that information, and you're slowing down the script unnecessarily.

If you must write information to the PowerShell console, use `[Console]::WriteLine()` instead.

Tip Source: https://twitter.com/brentblawat

Further learning:

Link: PowerShell Performance: Write-Host[82]
Source: blogs.technet.com
Format: Article

Don't use the Pipeline

The PowerShell pipeline, although a wonderful feature, is slow. The pipeline must perform the magic behind the scenes to bind the output of one command to the input of another command. All of that magic is overhead that takes time process.

Further learning:

Link: Quick Hits: Speed Up Some of your Commands by Avoiding the Pipeline[83]
Source: learn-powershell.net
Format: Article

Use the `.foreach()` and `.where()` Methods

As of Windows PowerShell v4, collections have a `foreach()` and a `where()` method. To process large collections, use these methods instead of other methods like a `foreach` loop, the `ForEach-Object` cmdlet or the `Where-Object` cmdlet.

Further learning:

Link: PowerShell V4: Where() and ForEach() Methods[84]
Source: manning.com
Format: Article

[82] https://blogs.technet.microsoft.com/mitchk/2014/03/31/powershell-performance-write-host/
[83] https://learn-powershell.net/2013/01/13/quick-hits-speed-up-some-of-your-commands-by-avoiding-the-pipeline/
[84] https://freecontent.manning.com/powershell-v4-where-and-foreach-methods/

Use Parallel Processing

Leveraging PowerShell background jobs and .NET runspaces, you can significantly speed up processing through parallelization. Backgrounds jobs are a native feature in PowerShell that allows you to run code in the background in a job. A runspace is a .NET concept that's similar but requires a deeper understanding of the .NET language. Luckily, you have the PoshRSJob PowerShell module[85] to make it easier.

Further learning:

Link: Parallel processing with PowerShell[86]
Source: microsoft.com
Format: Article

Use the .NET StreamReader Class When Reading Large Text Files

The `Get-Content` PowerShell cmdlet works well for most files but if you've got a large multi-hundred megabyte or multi-gigabyte file, drop down to .NET. One way to do that is to use the System.IO.StreamReader .NET class. Using this class, you can then invoke the `Peek()` and `ReadLine()` methods to read files much faster.

```
$sr = New-Object -Type System.IO.StreamReader -Arg file.txt

while ($sr.Peek() -ge 0) {
   $line = $sr.ReadLine()
   Do-Something -input $line
}
```

Further learning:

Link: PERF-02 Consider trade-offs between performance and readability[87]
Source: github.com
Format: Article

---

[85] https://github.com/proxb/PoshRSJob
[86] https://blogs.technet.microsoft.com/uktechnet/2016/06/20/parallel-processing-with-powershell/
[87] https://github.com/PoshCode/PowerShellPracticeAndStyle/blob/master/Best-Practices/Performance.md#perf-02-consider-trade-offs-between-performance-and-readability

Build Tests

If you're writing PowerShell for personal, random reasons to save yourself some time, writing tests for your scripts probably isn't worth it. But, if you're writing PowerShell for a business in a production environment, this is a requirement.

Tests, especially automated tests, are quality control for your code. Tests not only ensure you publish code that won't nuke your production environment, but they also help you trust your code. They open up an entirely new, more professional way of PowerShell development.

Learn the Pester Basics

The only dominant testing framework for PowerShell is Pester. If you're developing PowerShell scripts for an organization and don't have the first clue about Pester, stop right now and learn it.

Pester is a PowerShell module that allows you to build unit and integration tests for your code. You can also create infrastructure tests from it to run after your code runs to ensure it made the appropriate changes.

Pester is a necessary tool for any PowerShell developer.

Further learning:

Link: The Pester Book[88]
Source: leanpub.com
Format: eBook

Leverage Infrastructure Tests

Pester is a unit-testing framework but has been adapted to allow you to write infrastructure tests. Infrastructure tests give you two significant benefits:

- Confirming the environment is how you'd expect
- Ensuring the scripts you write made the changes you expect

By writing infrastructure tests with Pester, you can get a bird's eye view of your entire environment boiled down to simple red/green checks. Once the tests are written, they can be executed at any time.

[88] http://pesterbook.com

Pester infrastructure tests also allow you to verify the changes your scripts are supposed to make to your environment actually do. When's the last time you ran a script, it didn't return any errors, but it didn't change what you wanted it to? It happens all the time.

If you have a set of tests to run before and after your script runs, you can easily see if the changes you expected to happen did.

You can either build your own infrastructure tests or use existing projects. Two open-source projects I can recommend are PoshSpec[89] and Operational Validation Framework[90].

Further learning:

Link: How I Learned Pester by Infrastructure Testing a Domain Controller Build[91]
Source: adamtheautomator.com
Format: Article

Automate Pester Tests

Once you've learned about Pester, created some tests, and can now run them on demand, it's time to automate them. Automating Pester tests consists of building some way to automatically run tests once a change is made to code. This step is most commonly used in a continuous integration/continue delivery (CI/CD) pipeline.

To automate tests upon code change, you'll, at a minimum, need some version control like Git. You'll then need an automation engine to detect when a change has been detected in version control and kick off a test. There are many automated build/release pipelines like this. Some popular ones include Jenkins[92], AppVeyor[93] and Azure DevOps[94].

Further learning:

Link: Hitchhikers Guide to the PowerShell Module Pipeline[95]
Source: xainey.github.io
Format: Article

Use PSScriptAnalyzer

PSScriptAnalyzer is a free tool by Microsoft that code linting[96]. Code linting runs checks against your code to ensure you're meeting best practices and building efficient code.

[89] https://github.com/ticketmaster/poshspec
[90] https://github.com/PowerShell/Operation-Validation-Framework
[91] https://adamtheautomator.com/infrastructure-testing-pester/
[92] https://jenkins.io/doc/pipeline/tour/getting-started/
[93] https://www.appveyor.com/docs/
[94] https://docs.microsoft.com/en-us/azure/devops/user-guide/?view=azure-devops
[95] https://xainey.github.io/2017/powershell-module-pipeline/
[96] https://guide.freecodecamp.org/javascript/code-linting-in-javascript/

Download the PSScriptAnalyzer module from the PowerShell Gallery and run it against your scripts and modules to see what errors it finds.

Tip Source: https://twitter.com/DavePinkawa

Further learning:

Link: How to use the PowerShell Script Analyzer to Clean Up Your Code[97]
Source: interfacett.com
Format: Article

[97] https://www.interfacett.com/videos/use-powershell-script-analyzer-clean-code/

Miscellaneous Tips

There will inevitably be tips that don't fit the mold. Tips that don't necessarily fit in a chapter and not enough of them were found to create a chapter will be here.

In this chapter, you will find a smorgasbord of tips ranging from string and array best practices, interactive prompting, creating PowerShell script shortcuts and more.

Write for Cross Platform

At one time, PowerShell was called Windows PowerShell. PowerShell only existed on Windows. Not anymore. Nowadays, PowerShell is available on just about every platform out there. As a result, scripts are being run on Windows, Linux, MacOS, and other platforms every day.

If you're writing PowerShell for the community or others in your organization, it's a good idea to ensure those scripts are cross-platform compatible. IT workloads are constantly moving around on-prem and to/from the cloud on different operating systems. A web server can just as easily run on nGinx as it can on IIS. You should write scripts to account for this potential change even though there are no plans to run on another platform.

Even though the Microsoft PowerShell Team has taken every effort to maintain complete parity between Windows PowerShell and PowerShell, there will be differences especially across different operating sytems. If you believe your script may run on both versions, be sure to write and test for both.

Tip Source: https://twitter.com/migreene

Further learning:

Link: Tips for Writing Cross-Platform PowerShell Code[98]
Source: powershell.org
Format: Article

Don't Query the Win32_Product CIM Class

One common use for PowerShell is to find installed software on a Windows machine. Installed software can be found in various places in both WMI, the registry and the file system.

One place that installed software is located is within the Win32_Product CIM class. It may be tempting to use `Get-CimInstance` to query this class but don't! This class is special and forces

[98] https://powershell.org/2019/02/tips-for-writing-cross-platform-powershell-code/

Windows to run external processes behind the scenes. It's slower to respond and will instruct Windows to run msiexec for *each* instance when you don't expect it.

Instead, use the registry. Use the PSSoftware community module[99] or any other that queries the registry instead.

Further learning:

Link: Why Win32_Product is Bad News![100]
Source: sdmsoftware.com
Format: Article

Create a Shortcut to run PowerShell as Administrator

If you need to quickly open up a PowerShell console as administrator, create a Windows shortcut using the command:

```
PowerShell -NoProfile -ExecutionPolicy Bypass -Command "& {Start-Process PowerShell}"
```

Tip Source: https://www.reddit.com/user/Wugz/

Further learning:

Link: 9 Ways to Launch PowerShell in Windows[101]
Source: digiticalcitizen.com
Format: Article

Store 'Formatable' Strings for Use Later

You can store 'formatable' strings without actually using them until you need to. For example, if you need to store a common SQL query in code, you could create the string with placeholders and invoke the code later.

```
$FString = "SELECT * FROM some.dbo WHERE Hostname = '{0}'"
"comp1","comp2" | ForEach-Object { Invoke-SQLcmd -Query ($FString -f $_) }
```

Tip Source: https://www.reddit.com/user/Vortex100/

Further learning:

Link: Learn the PowerShell string format and expanding strings[102]
Source: adamtheautomator.com
Format: Article

[99] https://github.com/adbertram/PSSoftware
[100] https://sdmsoftware.com/group-policy-blog/wmi/why-win32_product-is-bad-news/
[101] https://www.digitalcitizen.life/ways-launch-powershell-windows-admin
[102] https://adamtheautomator.com/powershell-string-format/

Use Out-GridView for GUI-based Sorting and Filtering

If you or your users would like a grid layout to sort, select and manage a set of data, you can pipe any object to the `Out-GridView` cmdlet. For example, `Get-Service | Out-GridView` displays all Windows services in a GUI grid.

You can then select items in the grid and pipe those results to another cmdlet to perform an action on each object selected like below:

```
Get-Service | Out-GridView -PassThru | Restart-Service
```

Tip Source: https://www.reddit.com/user/alphanimal/

Further learning:

Link: Fun with PowerShell's Out-GridView[103]
Source: mcpmag.com
Format: Article

Don't Make Automation Scripts Interactive

When you're building scripts to *automatically* perform tasks, don't introduce any kind of interactivity. Don't use commands like `Read-Host` or don't not provide mandatory parameters that prompt you for input.

Every time a script prompts for input, it's paused and waits for a human to intervene. Think through what you need to pass to your scripts ahead of time and pass tha tinformation in rather than forcing a human to type on a keyboard.

Further learning:

Link: Working With Interactive Prompts In PowerShell[104]
Source: ipswitch.com
Format: Article

[103] https://mcpmag.com/articles/2013/01/08/pshell-gridview.aspx
[104] https://blog.ipswitch.com/working-with-interactive-prompts-in-powershell

Summary

I hope you learned a few tips and tricks in this eBook. This eBook was created to bring together all of the little gotchas and tips collected by myself and the PowerShell community over the years. It was written not necessary as a best practices guide but more of a source to confirm you're designing scripts according to community guidelines.

PowerShell is a forgiving language. That forgiveness is a double-edged sword. In one sense, you can write PowerShell code that gets a job done quickly many different ways. However, with that kind of flexibility comes design decisions. Even though you **can** do something doesn't mean you should. I hope this ebook has taught you some better ways to write scripts or to notify you of some behaviors to work on.

Release Notes

03/22/20

- removed sponsor

02/13/20
- finished up the Plan Before you Code chapter
- finished the Return Stndardized, Informative Output chapter
- finished Summary
- merged Write for Cross Platform chapter into Misc Tips

01/26/20

- finished up misc-tips chapter
- moved some misc tips to other chapters
- finished the parameterize everything chapter

01/19/20

- finished up the handle errors gracefully chapter
- finished up log script activity chapter

11/04/19

- added dont skimp on security chapter tips
- added create building blocks with functions tips
- aded build with manageability in mind tips

10/07/19

- Added a tip to build-scripts-for-speed chapter
- Finished up the build-tools chapter

09/18/19

- Finished up initial build-scripts-for-speed chapter
- Finished up initial build-tests chapter

09/17/19

- Initial book release

08/13/19

- Initial book setup

01/04/20

- Finished of all chapter intros