# Foreword

The ‘Community Book of PowerShell Practices’ came out of the 2013 Scripting Games and subsequent series of PowerShell.org ‘Great Debate’ blog posts.

The Practices aren’t intended as hard-and-fast rules. They are, we hope, a useful starting point when you’re creating and maintaining PowerShell scripts.

Copyright This guide is released under the Creative Commons Attribution-NoDerivs 3.0 Unported License. The authors encourage you to redistribute this document as widely as possible, but ask that you do not modify the document. However, you are encouraged to submit changes to the authors directly ([via the forums at PowerShell.org](http://ConcentratedTech.com/contact)) so that your suggestions can be incorporated into the master document and published in a future revision.

**KNOW OF A BEST PRACTICE THAT ISN’T WELL-COVERED HERE?  
POST IN THE FORUMS AT POWERSHELL.ORG AND LET US KNOW!**

# Contents

Foreword 1

Contents 2

Help and Comments 5

DOC-01 Write comment-based help 5

DOC-03 Provide usage examples 6

DOC-04 Use the Notes section for detail on how the tool works 6

DOC-05 Keep your language simple 6

DOC-06 Comment your code… 6

DOC-07 …but don’t over-comment 6

PowerShell Version 7

VER-01 Write for the lowest version of PowerShell that you can 7

VER-02 Document the version of PowerShell the script was written for 7

Performance 8

PERF-01 If performance matters, test it 8

PERF-02 Consider trade-offs between performance and readability 8

Aesthetics 10

READ-01 Indent your code 10

READ-02 Avoid backticks 10

OUT-01 Don’t use write-host unless writing to the host is all you want to do 12

OUT-02 Use write-verbose to give information to someone running your script 12

OUT-03 Use write-debug to give information to someone maintaining your script 12

OUT-04 Use [CmdletBinding()] if you are using write-debug or write-verbose 12

Tools vs. Controller? 14

TOOL-01 Decide whether you’re coding a ‘tool’ or a ‘controller script’ 14

TOOL-02 Make your code modular 14

TOOL-03 Make tools as re-usable as possible 15

TOOL-04 Use PowerShell standard cmdlet naming 15

TOOL-05 Use PowerShell standard parameter naming 15

TOOL-06 Tools should output raw data 15

TOOL-07 Controllers should typically output formatted data 15

The Purity Laws 16

PURE-01 Use native PowerShell where possible 16

PURE-02 If you can’t use just PowerShell, use .net, external commands or COM objects, in that order of preference 16

PURE-03 Document why you haven’t used PowerShell 16

PURE-04 Wrap other tools in an advanced function of cmdlet 17

Pipelines vs. Constructs 18

PIPE-01 Avoid using pipelines in scripts 18

Error-handling 19

ERR-01 Use -ErrorAction Stop when calling cmdlets 19

ERR-02 Use $ErrorActionPreference='Stop'/’Continue’ when calling non-cmdlets 19

ERR-03 Avoid using flags to handle errors 19

ERR-04 Avoid using $? 20

ERR-05 Avoid testing for a null variable as an error condition 20

ERR-06 Copy $Error[0] to your own variable 20

Wasted Effort 21

WAST-01 Don’t re-invent the wheel 21

WAST-02 Report bugs to Microsoft 21

# Help and Comments

### DOC-01 Write comment-based help

You should always write comment-based help in your scripts and functions.

Comment-based help is formatted as follows:

function your-function {

<#

.SYNOPSIS

A brief description of the function or script.

.DESCRIPTION

A longer desciption

.PARAMETER FirstParameter

Description of each of the parameters

.INPUTS

Description of objects that can be piped to the script

.OUTPUTS

Description of objects that are output by the script

.EXAMPLE

Example of how to run the script

.LINK

Links to further documentation

.NOTES

Detail on what the script does, if this is needed

#>

Comment-based help is displayed when the user types ‘help <your-function>’

Your help should be helpful. That is, if you’ve written a tool called “Get-LOBAppUser”, don’t write help that merely says, “Gets LOB App Users.” Duh.

**Further information:** You can get more on the use of comment-based help by typing ‘help about\_Comment\_Based\_Help’ within Powershell.

DOC-02 Describe each parameter

Your help should include a description of each of the script’s parameters/

### DOC-03 Provide examples

Your help should provide usage examples for each major use case.

A ‘usage example’ is just an example of how you run the script – exactly like when you run help <Powershell command> -example. You can cut and paste usage examples from the command line.

### DOC-04 Use the Notes section for detail on how the tool works

If you need to explain some of the detail about how the tool works, do so in the Notes section.

### DOC-05 Keep your language simple

Comment-based help should be written in simple language. Avoid grandiloquent, multisyllabic verbiage and phrasing.

In other words, put the thesaurus down.

If you’re writing in what is, for you, a foreign language, simpler words and simpler sentence structures are better, and more likely to make sense to a native reader.

### DOC-06 Comment your code…

Inline comments are useful.

Keep in mind that these should explain the working of the code, but not how to use the tool, since an ordinary user won’t see inline comments.

### DOC-07 …but don’t over-comment

Don’t precede each line of code with a comment - doing so breaks up the code and makes it harder to follow.

# PowerShell Version

### VER-01 Write for the lowest version of PowerShell that you can

As a rule, write for the lowest version of PowerShell that you can, especially with scripts that you plan to share with others. Doing so provides the broadest compatibility for other people.

That said, don’t sacrifice functionality or performance just to stick with an older version.

Keep in mind that some newer features may have performance benefits. For example, in PowerShell v3:

Get-Service | Where-Object -FilterScript { $\_.Status -eq 'Running' }

…will run significantly more slowly than:

Get-Service | Where Status -eq Running

…because of the way the two different syntaxes have to be processed under the hood.

**Further information:** You can get details on the differences between PowerShell versions by typing ‘help about\_Windows\_PowerShell\_2.0’ (or \_3.0 or 4.0) in PowerShell

### VER-02 Document the version of PowerShell the script was written for

Specify the version of PowerShell you wrote for by using an appropriate '#requires' statement:

#requires -version 3.0

The #requires statement will prevent the script from running on the wrong version of PowerShell.

# Performance

### PERF-01 If performance matters, test it

PowerShell comes equipped with 3.2 million performance quirks. Approximately.

For example, the first line below executes a lot faster than the second:

[void]Do-Something

Do-Something | Out-Null

If you’re aware of multiple techniques to accomplish something, and you’re writing a production script that will be dealing with large data sets (meaning performance could be significant), then test the performance using Measure-Command, or some other tool.

### PERF-02 Consider trade-offs between performance and readability

If a script is expected to deal with 10 pieces of data, a 30% performance improvement will not add up to a lot of actual time. It’s okay to use a slower-performing technique that is easier to read, understand, and maintain.

On the other hand if a script is dealing with 10,000 pieces of data, then a 30% performance improvement could be significant.

Most people would see this code:

$content = Get-Content file.txt

ForEach ($line in $content) {

Do-Something -input $line

}

…as aesthetically better than this code:

Get-Content file.txt |

ForEach-Object -Process {

Do-Something -input $\_

}

The first snippet is probably more easy to follow, to debug, and to expand if necessary.

However the second snippet will, all things being equal, run faster and use less resource.

# Aesthetics

### READ-01 Indent your code

Indent within constructs, to make it clearer what “belongs to” the construct. For example

ForEach ($computer in $computers) {

Do-This

Get-Those

}

Indentation does not affect the functionality of a script – it just makes it more readable.

There is no strong consensus on styles of indentation. This style:

if ($this -gt $that) {

Do-Something -with $that

}

…is no better or worse than this one:

if ($this -gt $that)

{

Do-Something -with $that

}

…but you should decide to use one or the other, not both.

### READ-02 Avoid backticks

Consider this:

Get-WmiObject -Class Win32\_LogicalDisk `

-Filter "DriveType=3" `

-ComputerName SERVER2

In general, the community feels you should avoid using those backticks as “line continuation characters” whenever possible.

Backticks are hard to read, easy to miss, and easy to mis-type. Also, if you add an extra whitespace after the backtick in the above example, then the command won’t work.

Here’s an alternative:

$params = @{Class=Win32\_LogicalDisk;

Filter='DriveType=3';

ComputerName=SERVER2}

Get-WmiObject @params

The technique is called splatting. It lets you get the same nice, broken-out formatting without using the backtick.

Output

### OUT-01 Don’t use write-host unless writing to the host is all you want to do

It is generally accepted that you should never use Write-Host to create any script output whatsoever, unless your script (or function, or whatever) uses the Show verb (as in, Show-Performance).

That word ‘show’ in this context explicitly means “show on the screen, with no other possibilities.” Like Show-Command.

### OUT-02 Use write-verbose to give information to someone running your script

Verbose output is generally held to be output that is useful to anyone running the script, providing status information (“now attempting to connect to SERVER1”) or progress information (“10% complete”).

### OUT-03 Use write-debug to give information to someone maintaining your script

Debug output is generally held to be output that is useful for script debugging (“Now entering main loop,” “Result was null, skipping to end of loop”), since it also creates a breakpoint prompt.

### OUT-04 Use [CmdletBinding()] if you are using write-debug or write-verbose

Both Verbose and Debug output are off by default, and when you use Write-Verbose or Write-Debug, it should be in a script or function that uses the [CmdletBinding(] declaration, which automatically enables the switch.

The CmdletBinding attribute is specified on the first line of the script or function. After the name and inline help, but before the parameter definition:

function your-function {

<#

<Comment-based help>

#>

[CmdletBinding()]

Param( [String] $Parameter1)

# Tools vs. Controller?

### TOOL-01 Decide whether you’re coding a ‘tool’ or a ‘controller script’

For this discussion, it’s important to have some agreed-upon terminology. While the terminology here isn’t used universally, the community generally agrees that several types of “script” exist:

1. Some scripts contain tools, when are meant to be reusable. These are typically functions or advanced functions, and they are typically contained in a script module or in a function library of some kind. These tools are designed for a high level of re-use.
2. Some scripts are controllers, meaning they are intended to utilize one or more tools (functions, commands, etc) to automate a specific business process. This sort of script is not intended to be reusable; it is intended to make use of reuse by leveraging functions and other commands

For example, you might write a “New-CorpUser” script, which provisions new users. In it, you might call numerous commands and functions to create a user account, mailbox-enable them, provision a home folder, and so on. Those discrete tasks might also be used in other processes, so you build them as functions. The New-CorpUser script is only intended to automate that one process, and so it doesn’t need to exhibit reusability concepts. It’s a standalone thing.

### TOOL-02 Make your code modular

Generally, people tend to feel that most working code - that is code which does things - should be modularized into functions and ideally stored in script modules.

That makes those functions more easily re-used. Those functions should exhibit a high level of reusability, such as accepting input only via parameters and producing output only as objects to the pipeline

### TOOL-03 Make tools as re-usable as possible

Tools should accept input from parameters and should (in most cases) produce any output to the pipeline; this approach helps maximize reusability.

### TOOL-04 Use PowerShell standard cmdlet naming

Use the verb-noun convention, and use the PowerShell standard verbs.

You can get a list of the verbs by typing ‘get-verb’ at the command line.

### TOOL-05 Use PowerShell standard parameter naming

Tools should be consistent with PowerShell native cmdlets in regards parameter naming.

For example, use $ComputerName and $ServerInstance rather than something like $Param\_Computer and $InstanceName

### TOOL-06 Tools should output raw data

The community generally agrees that tools should output raw data. That is, their output should be manipulated as little as possible. If a tool retrieves information represented in bytes, it should output bytes, rather than converting that value to another unit of measure. Having a tool output less-manipulated data helps the tool remain reusable in a larger number of situations.

### TOOL-07 Controllers should typically output formatted data

Controllers, on the other hand, may reformat or manipulate data because controllers do not aim to be reusable; they instead aim to do as good a job as possible at a particular task.

For example, a function named Get-DiskInfo would return disk-sizing information in bytes, because that’s the most-granular unit of measurement the operating system offers.

A controller that was creating an inventory of free disk space might translate that into gigabytes, because that unit of measurement is the most convenient for the people who will view the inventory report.

# The Purity Laws

### PURE-01 Use native PowerShell where possible

### PURE-02 If you can’t use just PowerShell, use .net, external commands or COM objects, in that order of preference

First, at the end of the day, get the job done the best way you can. Utilize whatever means you have at your disposal, and focus on the techniques you already know, because you’ll spend less time coding that way.

That said, there are advantages to sticking with “PowerShell native.” In general, folks tend to prefer that you accomplish tasks using the following, in order of preference:

1. **PowerShell** cmdlets, functions, and other “native” elements. These are (or can be) very well documented right within the shell itself, can (and should) use consistent naming and operation, and are generally more discoverable and easier to understand by someone else.
2. **.NET** Framework classes, methods, properties, and so on. While not documented in-shell, they at least stay “inside the boundaries” of .NET, and .NET Framework classes are typically well-documented online.
3. **External commands**, like Cacls.exe or PathPing.exe. While not documented in-shell, most tools do offer help displays, and most (especially ones that ship with the OS or server product) have numerous online examples.
4. **COM objects**. These are rarely well-documented, making them harder for someone else to research and understand. They do not always work flawlessly in PowerShell, as they must be used through .NET’s Interop layer, which isn’t 100% perfect.

### PURE-03 Document why you haven’t used PowerShell

So when is it okay to move from one item on this list to another?

Obviously, if a task can’t be accomplished with a more-preferred way, you move on to a less-preferred way.

If a less-preferred approach offers far superior performance, and performance is a potential issue, then go for the better-performing approach.

For example, Robocopy is superior in nearly every way to Copy-Item, and there are probably numerous circumstances where Robocopy would do a much better job.

Document the reason for using tools other than PowerShell in your comments.

### PURE-04 Wrap other tools in an advanced function of cmdlet

That said, you truly become a better PowerShell person if you take the time to wrap a less-preferred way in an advanced function or cmdlet. Then you get the best of both worlds: the ability to reach outside the shell itself for functionality, while keeping the advantages of native commands.

# Pipelines vs. Constructs

### PIPE-01 Avoid using pipelines in scripts

Consider this:

Get-Content computer-names.txt |

ForEach-Object -Process {

Get-WmiObject -Class Win32\_BIOS -ComputerName $\_

}

And now this alternative:

$computers = Get-Content computer-names.txt

foreach ($computer in $computers) {

Get-WmiObject -Class Win32\_BIOS -ComputerName $computer

}

The PowerShell world definitely prefers the latter approach in a script or function.

Constructs offer far more flexibility, especially when you’re looping through multiple commands. Error handling becomes far easier and more structured with the latter approach, as does debugging.

Constructs often exhibit higher performance than pipeline-only approaches, but not always – it’s worth testing if you are dealing with large data-sets and if you’re in a context where performance matters. See [PERF-02](#_PERF-02_Consider_trade-offs) for more detail on this.

On the command-line, by yourself, do whichever you prefer. It’s as you move into a script - an inherently more structured environment - that you want to start shifting to a more programming-style approach.

# Error-handling

### ERR-01 Use -ErrorAction Stop when calling cmdlets

When trapping an error, try to use -ErrorAction Stop on cmdlets to generate terminating, trappable exceptions.

### ERR-02 Use $ErrorActionPreference='Stop'/’Continue’ when calling non-cmdlets

When executing something other than a cmdlet, set $ErrorActionPreference = 'Stop' before executing, and re-set to Continue afterwards. If you’re concerned about using -ErrorAction because it will bail on the entire pipeline, then you’ve probably over-constructed the pipeline.

Ideally, whatever command or code you think might bomb should be dealing with one thing: querying one computer, deleting one file, updating one user. That way, if an error occurs, you can handle it and then get on with the next thing.

### ERR-03 Avoid using flags to handle errors

Try to avoid setting flags:

Try {

$continue = $true

Do-Something -ErrorAction Stop

} Catch {

$continue = $false

}

if ($continue) {

Do-This

Set-That

Get-Those

}

Instead, put the entire “transaction” into the Try block:

Try {

Do-Something -ErrorAction Stop

Do-This

Set-That

Get-Those

} Catch {

Handle-Error

}

It’s a lot easier to follow the logic.

### ERR-04 Avoid using $?

When you need to examine the error that occurred, try to avoid using $?. It actually doesn’t mean an error did or did not occur; it’s reporting whether or not the last-run command considered itself to have completed successfully. You get no details on what happened.

### ERR-05 Avoid testing for a null variable as an error condition

Also try to avoid testing for a null variable as an error condition:

$user = Get-ADUser -Identity DonJ

if ($user) {

$user | Do-Something

} else [

Write-Warning "Could not get user $user"

}

There are times and technologies where that’s the only approach that will work, especially if the command you’re running won’t produce a terminating, trappable exception. But it’s a logically contorted approach, and it can make debugging trickier.

### ERR-06 Copy $Error[0] to your own variable

Within a Catch block, $\_ will contain the last error that occurred, as will $Error[0].

Use either - but immediately copy them into your own variable, as executing additional commands can cause $\_ to get “hijacked” or $Error[0] to contain a different error.

It isn’t necessary to clear $Error in most cases. $Error[0] will be the last error, and PowerShell will maintain the rest of the $Error collection automatically.

# Wasted Effort

### WAST-01 Don’t re-invent the wheel

There are a number of approaches in PowerShell that will “get the job done.” However, wasted effort on your part is never commendable. When your wasted effort further involves poor aesthetics or less-preferred approaches, then you should expect members of the community to be less-than-welcoming of your product.

For example:

Function Ping-Computer ($computername) {

$ping = Get-WmiObject Win32\_PingStatus -filter "Address='$computername'"

if ($ping.StatusCode -eq 0) {

return $true

} else {

return $false

}

}

This function has a few problems, but most significantly in this context, there’s no reason to write it in PowerShell v2 or later. Simply use:

Test-Connection $computername -Quiet

This built-in command accomplishes the exact same task with less work on your part.

### WAST-02 Report bugs to Microsoft

An exception: if you know that a built-in technique doesn’t work properly (e.g., it is buggy or doesn’t accomplish the exact task), then obviously it’s fine to re-invent the wheel.

However, in cases where you’re doing so to avoid a bug or design flaw, then you should - as an upstanding member of the community - report the bug on Connect.Microsoft.com also.