THE BIG BOOK OF
PowerShell
Error Handling
Don Jones
The Community Book of PowerShell Practices

by Don Jones
and the Windows PowerShell Community

Copyright ©2013 by PowerShell.org, Inc.

Licensed under Creative Commons Attribution-NoDerivs 3.0 Unported
Foreword

During the 2013 Scripting Games, it became apparent that a great many folks in the PowerShell community have vastly different ideas about what’s “right and wrong” in the world of PowerShell scripting. Some folks down-scored techniques that others praised, and vice-versa.

After the Games, PowerShell.org ran a series of “Great Debate” blog posts, outlining some of the more controversial issues and asking for community input. The catch is the input had to not only state what the person thought was best, but very specifically why they thought that.

This book, which will likely be an ongoing effort, is a compilation of those Debates’ comments, formed as a “best practices” guide that is based on community input. It’s important to realize that practices are not hard-and-fast rules. Best practices are the things you usually do as a starting point, and deviate from when it’s appropriate. You should understand that these practices were formed primarily by people who are writing scripts to manage their own environment, and who are often pressed for time. They are not developing commercial software, they rarely have the luxury of extended test-and-develop cycles, and for the most part are not professional software developers. Their perspective definitely drives what they consider “best” and “worst,” but if you are working from a different perspective then you’ll have to take all of this with a grain of salt.

Copyright This guide is released under the Creative Commons Attribution-NoDerivs 3.0 Unported License. The authors encourage you to redistribute this file 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) so that your suggestions can be incorporated into the master document and published in a future revision.
Contents

Writing Help and Comments ................................................................. 4
Versioning ......................................................................................... 5
Performance .................................................................................... 6
Aesthetics ......................................................................................... 7
Verbose and Debug Output ............................................................... 8
Script or Function? ........................................................................ 9
The Purity Laws ............................................................................ 10
Pipelines vs. Constructs ................................................................. 11
One Liners .................................................................................... 12
Backticks ....................................................................................... 13
“The PowerShell Way” ................................................................ 14
Trapping and Capturing Errors ...................................................... 15
Aesthetics ..................................................................................... 17
“Fixing” Output ............................................................................ 18
Writing Help and Comments

You should always strive to include comment-based help in your scripts and functions. 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. Your help should describe the proper use of each parameter, and should provide examples for each major use case. If you need to explain some of the subtle details about how the tool works, do so in a Notes section.

(As a note, within the context of the Scripting Games, be really careful about adding help that explains that you took a particular approach because such-and-such other approach wouldn’t work. If you write that, you’d better be correct. If you’re not, expect to fail without explanation.)

Comment-based help should be written in simple language. You’re not writing a these for your college Technical Writing class - you’re writing something that describes how a function works. Avoid gradiloquent, multisyllabic verbiage and phrasing that serves best to confound, and least to inform. In other words, put the thesaurus down. You’re not impressing anyone. 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.

Be complete, but be concise.

Inline comments are also appreciated. Keep in mind that these should explain the working of the code, but not how to use the tool itself, since an ordinary user won’t see inline comments. Try to put a small block of inline comment before each functional hunk of your code. Don’t precede each line of code with a comment - doing so breaks up the code and makes it a lot harder to follow. Don’t restate the obvious:

```powershell
# Get the WMI BIOS object from the computer
Get-WmiObject -Class Win32_BIOS -ComputerName $computer
```

Duh. A well-written PowerShell command, with full command and parameter names, can be pretty self-explanatory. Don’t comment-explain it unless it isn’t self-explanatory. Comments should describe the logic of your code, not attempt to document what each command is going to do.
Versioning

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

That said, don’t sacrifice functionality or performance just to stick with an older version. If you *can* safely write for a higher version (meaning you’ve deployed it everywhere the script will need to run), then take advantage of that version. Keep in mind that some newer features that *seem* like window dressing might actually have underlying performance benefits. For example, in PowerShell v3:

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

Will run significantly slower than:

```
Get-Service | Where Status -eq Running
```

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

All that said, make sure you document the version of PowerShell you wrote for by using an appropriate `#requires` statement:

```
#requires -version 3.0
```

(In the context of the Scripting Games, don’t ever down-score someone for writing to a particular version, so long as they’ve documented it, unless the scenario explicitly states that a particular version must be used.)
PowerShell comes equipped with 3.2 million performance quirks. Approximately. For example, the first line below executes a lot faster than the second:

```powershell
[void]Do-Something
Do-Something | Out-Null
```

That said, performance is not the only reason you write a script. If a script is expected to deal with ten 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 - and “easier” is a very subjective term. Of the two commands above, any given person might select either of them as being “easier” to understand or read.

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 will become a cumulative factor), then test the performance using something like Measure-Command.

(In the context of the Scripting Games, don’t down-score someone for using an approach that you personally don’t like to use for aesthetic reasons. If performance is a stated goal, then it’s fine to down-score someone for using a technique that’s demonstrably slower than an alternative, but never down-grade for personal, subjective, or aesthetic reasons.)
Consider this code example:

```powershell
if ($this -gt $that) {
    Do-Something -with $that
}
```

And now consider this one:

```powershell
if ($this -gt $that)
{
    Do-Something -with $that
}
```

Neither of these is better than the other. Ask 100 coders which they prefer and you’ll get roughly half liking either one. Now, when you start dealing with commands that accept script blocks as parameters, things can get trickier because of the way PowerShell parses syntax. “Wrong” is wrong. With scripting constructs, like the two examples above, there’s no functional difference.
Verbose and Debug Output

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 verb explicitly means “show on the screen, with no other possibilities.” Like Show-Command.

Verbose output is generally held to be output that is useful or anyone running the script, providing status information (“now attempting to connect to SERVER1”) or progress information (“10% complete”). Verbose output is off by default, and when you use Write-Verbose, it should be in a script or function that uses the [CmdletBinding()] declaration, which automatically enables the -Verbose switch.

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. Debug output is off by default, and when you use Write-Debug, it should be in a script or function that uses the [CmdletBinding()] declaration, which automatically enables the -Debug switch.
For the sake of discussion, consider a *script* to be any text file that contains an unbroken sequence of commands, loops, and logical constructs. A *function* is a more modularized hunk of commands and constructs. A script may contain one or more functions. A script may also utilizes functions and other commands that are packaged as modules.

Generally, people tend to feel that most *working code* - that is, your 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.

A script - what some call a *controller script* to be more precise - calls many functions or commands in a sequence to accomplish some multi-part business process. A 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 script is only intended to automate that one process, and so it doesn’t need to exhibit reusability concepts. It’s a standalone thing.
The Purity Laws

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.

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.

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.

Ignorance, however, is no excuse. If you’ve written some big wrapper function around Ping.exe simply because you were unaware of Test-Connection, then you’ve wasted a lot of time, and that is not commendable. Before you move on to a less-preferred approach, make sure the shell doesn’t already have a way to do what you’re after.

Pipelines vs. Constructs

Consider this:

```powershell
Get-Content computer-names.txt |
ForEach-Object -Process {  
    Get-WmiObject -Class Win32_BIOS -ComputerName $_  
}
```

And now this alternative:

```powershell
$computers = Get-Content computer-names.txt
foreach ($computer in $computers) {
    Get-WmiObject -Class Win32_BIOS -ComputerName $computer
}
```

The world definitely prefers the latter approach in a script or function. Constructs nearly always exhibit higher performance than pipeline-only approaches. Constructs also 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.

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.
One Liners

If someone asks you to do something as a “one-liner,” that means a single, continuous pipeline. Using semicolons to jam multiple “lines” into a single “line” does not count.

If someone is asking for a one-liner, they’re doing it to test a certain type of PowerShell mettle. Mashing multiple lines together by means of semicolons does not fulfill the intent of the question.
Consider this:

```powershell
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” when possible. They’re simply hard to read, easy to miss, and easy to mis-type - add an extra whitespace after the backtick in the above example, and the command won’t work. The resulting error is a bit hard to correlate to the actual problem, making debugging the issue harder.

Here’s an alternative:

```powershell
$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. You can also line break after almost any comma, pipe character, or semicolon without using a backtick.

The backtick is not universally hated - but it can be inconvenient. If you *have* to use it for line breaks, well then use it. Just try not to have to.
There is no one, true way to do anything in PowerShell. To resurrect a previous pair of examples:

```powershell
[void]Do-Something
Do-Something | Out-Null
```

Same end effect, but some folks will argue that the first command is more “programmer-y” and less “PowerShell-y.” The first example runs faster, however, so there’s more than an aesthetic difference here.

It’s important to note that although PowerShell isn’t a programming language, it - like most good shells - does contain a programming language. It’s okay to use it. It’s fine to use more programming-style approaches. Don’t dismiss a technique or approach because it looks like programming to you, and because you aren’t a programmer and don’t want to be one. You might actually be missing out on something.
Trapping and Capturing Errors

When trapping an error, try to use `-ErrorAction Stop` on cmdlets to generate terminating, trappable exceptions. 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. Consider using a more scripting-construct-style approach, because those approaches are inherently better for automated error handling.

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.

Try to avoid setting flags:

```powershell
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:

```powershell
Try {
    Do-Something -ErrorAction Stop
    Do-This
    Set-That
    Get-Those
} Catch {
    Handle-Error
}
```

It’s a lot easier to follow the logic.

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.

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

```powershell
$user = Get-ADUser -Identity DonJ
if ($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.
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.
Aesthetics

Now, understand that the following are basically guidelines from mass consensus; they’re not hard-and-fast rules. That said, there are arguments in favor of these, and you should consider the arguments before dismissing these ideas.

First, format your code. The convention is to indent within constructs, to make it clearer what “belongs to” the construct.

```powershell
ForEach ($computer in $computers) {
    Do-This
    Get-Those
}
```

You will probably be reviled if you’re not careful with this kind of code formatting.

Second, avoid over-commenting.

```powershell
#########################################
#  # PREPARE VARIABLES
#  #---------------------------------------
$computers = @()
```

Really? Five lines of comment to one line of functional code? That is not okay. It actually does make the code harder to read. It doesn’t add value - anyone looking at the code can tell that a variable is being set up for something. It’s just noise.
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:

- 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.

- Some scripts are controllers, meaning they are intended to utilize one or more tools (functions, commands, etc) to automate a specific business process.

These different types of scripts have a number of identifiable characteristics. Tools, for example, should accept input from parameters and should (in most cases) produce any output to the pipeline; this approach helps maximize reusability. Tools should be consistent with PowerShell native cmdlets in regards to naming, parameter naming, and so on. Controllers, on the other hand, often produce output directly to the screen (when designed for interactive use), or may log to a file (when designed to run unattended).

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.

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.

An intermediate step is useful for tools that are packaged in script modules: views. By building a manifest for the module, you can have the module also include a custom .format.ps1xml view definition file. The view can specify manipulated data values, such as the default view used by PowerShell to display the output of Get-Process. The view does not manipulate the underlying data, leaving the raw data available for any purpose.