The Community Book of PowerShell Practices

>

PowerShell.org

The Community Book of PowerShell Practices

PowerShell.org

This project can be followed at:

https://www.penflip.com/powershellorg/the-community-book-of-powershell-practices

Contents

1 The Community Book of PowerShell Practices

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.

Those discussions got people thinking, and provided the basis for a "Community Book of PowerShell Practices." Over time, however, some of those practices started to show their age, and the fact that they'd come from a discussion involving a specific audience - systems administrators participating in a scripting competition. As the PowerShell community began to grow and become more engaged, a broader discussion ensued. Automated script analyzers joined the mix, offering the ability to more formally codify certain practices, especially with regard to style. And, an audience involving more than just systems administrators began to take an interest.

As a result, the "Community Book of PowerShell Practices" has been discontinued, in favor of the "PowerShell Practice and Style Guide," available as an open-source Markdown project on GitHub. We encourage you to visit that project, review what it offers, and make your own contributions to the continuing evolution of PowerShell Practices.

2 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" when possible. They're 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. The resulting error is hard to correlate to the actual problem, making debugging the issue harder.

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. 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.

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

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

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

5 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.

6 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)
```

7 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. 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.

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).

8 TOOL-02 Make your code modular

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

9 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.

10 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.

11 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

12 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.

13 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.

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.

14 PURE-01 Use native PowerShell where possible

This means not using COM, .NET Framework classes, and so on when there is a native Windows PowerShell command or technique that gets the job done.

15 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.

16 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.

17 PURE-04 Wrap other tools in an advanced function or 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.

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.

18 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 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 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.

One-Liners

| | In the context of the Scripting Games, if you are asked 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. |
| — | — |

“The PowerShell Way”

There is no one, true way to do anything in PowerShell. To resurrect a previous pair of examples:
““

```
[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.

19 ERR-01 Use -ErrorAction Stop when calling cmdlets

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

20 ERR-02 Use $ErrorActionPreference='Stop' or '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. 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.

21 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.

22 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.

23 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.

24 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.

25 WAST-01 Don't re-invent the wheel

There are a number of approaches in PowerShell that will "get the job done." In some cases, other community members may have already written the code to achieve your objectives. If that code meets your needs, then you might save yourself some time by leveraging it, instead of writing it yourself.

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 opportunities for enhancement. First of all, the parameter block is declared as a basic function; using advanced functions is generally preferred. Secondly, the command verb ("Ping") isn't an approved PowerShell command verb. This will cause warnings if the function is exported as part of a PowerShell module, unless they are otherwise suppressed on import.

Thirdly, there's little reason to write this function in PowerShell v2 or later. PowerShell v2 has a built-in command that will allow you to perform a similar function. Simply use:

```
Test-Connection $computername -Quiet
```

This built-in command accomplishes the exact same task with less work on your part - e.g., you don't have to write your own function.

It has been argued by some that, "I didn't know such-and-such existed, so I wrote my own." That argument typically fails with the community, which tends to feel that ignorance is no excuse. Before making the effort to write some function or other unit of code, find out if the shell can already do that. Ask around. And, if you end up writing your own, be open to someone pointing out a built-in way to accomplish it.

On the flip side, it's important to note that writing your own code from the ground up can be useful if you are trying to learn a particular concept, or if you have specific needs that are not offered by another existing solution.

26 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.