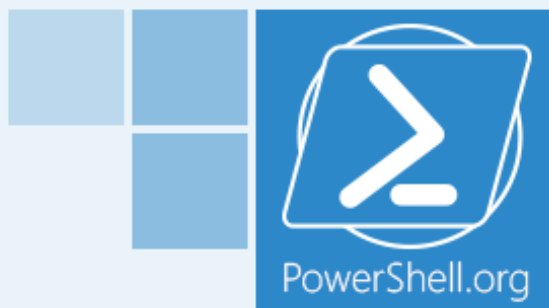


The Community Book of PowerShell Practices

compiled by
Don Jones
and Matt Penny





Copyright ©PowerShell.org, Inc.
All Rights Reserved.

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 (<http://ConcentratedTech.com/contact>) so that your suggestions can be incorporated into the master document and published in a future revision.

PowerShell.org ebooks are works-in-progress, and many are curated by members of the community. We encourage you to check back for new editions at least twice a year, by visiting PowerShell.org. You may also subscribe to our monthly e-mail TechLetter for notifications of updated ebook editions. Visit PowerShell.org for more information on the newsletter.

Feedback and corrections, as well as questions about this ebook's content, can be posted in the PowerShell Q&A forum on PowerShell.org. Moderators will make every attempt to either address your concern, or to engage the appropriate ebook author.

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.

Contents

Foreword	3
Contents	4
Writing Help and Comments	8
DOC-01 Write comment-based help	8
DOC-02 Describe each parameter	8
DOC-03 Provide usage examples	8
DOC-04 Use the Notes section for detail on how the tool works	8
DOC-05 Keep your language simple	8
DOC-06 Comment your code	8
DOC-07 Don't over-comment	8
Versioning	10
VER-01 Write for the lowest version of PowerShell that you can	10
VER-02 Document the version of PowerShell the script was written for	10
Performance	12
PERF-01 If performance matters, test it	12
PERF-02 Consider trade-offs between performance and readability	12
Aesthetics	14
READ-01 Indent your code	14
Verbose and Debug Output	15
OUT-01 Don't use write-host unless writing to the host is the only goal	15
OUT-02 Use write-verbose to give information to someone running your script	15
OUT-03 Use write-debug to give information to someone maintaining your script	15
OUT-04 Use [CmdletBinding()] if you are using write-debug or write-verbose	15
Tools vs. Controller?	16

TOOL-01 Decide whether you're coding a 'tool' or a 'controller script'	16
TOOL-02 Make your code modular	16
TOOL-03 Make tools as re-usable as possible	16
TOOL-04 Use PowerShell standard cmdlet naming	16
TOOL-05 Use PowerShell standard parameter naming	16
TOOL-06 Tools should output raw data	16
TOOL-07 Controllers should typically output formatted data	17
The Purity Laws	18
PURE-01 Use native PowerShell where possible	18
PURE-02 If you can't use just PowerShell, use .net, external commands or COM objects, in that order of preference	18
PURE-03 Document why you haven't used PowerShell	18
PURE-04 Wrap other tools in an advanced function or cmdlet	18
Pipelines vs. Constructs	19
PIPE-01 Avoid using pipelines in scripts	19
One Liners	20
Backticks	21
READ-02 Avoid backticks	21
"The PowerShell Way"	22
Trapping and Capturing Errors	23
ERR-01 Use -ErrorAction Stop when calling cmdlets	23
ERR-02 Use \$ErrorActionPreference='Stop'/'Continue' when calling non-cmdlets	23
ERR-03 Avoid using flags to handle errors	23
ERR-04 Avoid using \$?	23
ERR-05 Avoid testing for a null variable as an error condition	23
ERR-06 Copy \$Error[0] to your own variable	24
Wasted Effort	25
WAST-01 Don't re-invent the wheel	25



Upgrade your brain

With advanced Microsoft® IT training from CBT Nuggets

Are you an Exchange admin, Active Directory architect, or SharePoint manager? Do you use Hyper-V? Do you struggle with storage and security, or worry about compliance and business continuity? Visit us at cbtnuggets.com for the latest advanced Microsoft IT training, and upgrade YOUR brain.

- Active Directory
- SharePoint
- SQL Server
- Hyper-V
- Programming
- Security
- System Center Config.
- Network Infrastructure
- Database
- Exchange Server
- PowerShell
- Lync Server
- Windows
- Windows Server
- Office
- Cloud
- Messaging
- Systems Management

Our nationally renowned trainers are the authority on Microsoft IT:



James Conrad



Don Jones
Microsoft MVP



Greg Shields
Microsoft MVP



Tim Warner

CBTNUGGETS.COM

Writing Help and Comments

DOC-01 Write comment-based help

You should always write 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.

DOC-02 Describe each parameter

Your help should describe the proper use of each parameter

DOC-03 Provide usage examples

Your help should provide examples for each major use case.

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

If you need to explain some of the subtle details 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. You're not writing a thesis for your college Technical Writing class - you're writing something that describes how a function works. Avoid grandiloquent, 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.

DOC-06 Comment your code

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.

DOC-07 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. *Don't* restate the obvious:

```
# 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.

Another example:

```
#####
#
# 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.



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.

Versioning

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

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.



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.

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

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

```
#requires -version 3.0
```



The advertisement features a blue and white color scheme with a grid pattern on the left side. At the top left is a logo consisting of a blue square with a white grid of dots. At the top right, a red banner reads "DESIGNED FOR WINDOWS 8 SUPPORTS & POWERSHELL V3". The main title "SAPIEN PowerShell STUDIO 2012" is prominently displayed in the center. Below the title, a descriptive sentence states: "PowerShell Studio 2012 is the premier PowerShell Integrated Scripting and Toolmaking Environment available." A list of features follows, including a full featured PowerShell editor, GUI tool creation, V3 and V2 support, script modules, remote machine interaction, integrated consoles, a script debugger, remote debugging, multi-file debugging, 32 and 64 bit integration, script-to-executable conversion, built-in help, source control, and more. A price tag of \$349.00 is shown with a "Buy It Now" button. To the right is an image of the software box and a QR code. At the bottom, there is a download link for a 45-day trial, the SAPIEN logo, and contact information for SAPIEN Technologies, Inc.

DESIGNED FOR WINDOWS 8
SUPPORTS & POWERSHELL V3

SAPIEN PowerShell STUDIO 2012


PowerShell Studio 2012 is the premier PowerShell Integrated Scripting and Toolmaking Environment available.

- ❖ Full Featured **PowerShell Editor**
- ❖ Visually create **PowerShell GUI** tools
- ❖ **PowerShell V3** and **V2** support
- ❖ **Create Script Modules**
- ❖ Script against a remote machine's **Installed Module Set (IMS)**
- ❖ Integrated **PowerShell Consoles** (32 & 64 bit)
- ❖ Comprehensive **Script Debugger**
- ❖ **Remote Debugging**
- ❖ Multi-file and **module debugging**
- ❖ **32 bit** and **64 bit PowerShell** integration
- ❖ Convert Scripts into **Executables**
- ❖ Built in **PowerShell Help**
- ❖ Integrated **Source Control**
- ❖ Plus much more

\$349.00
Take advantage of this great deal
Buy It Now

Download a fully functional 45 day trial from:
www.sapien.com

SAPIEN Technologies, Inc.
www.sapien.com
707.252.8700



PrimalScript, SAPIEN PowerShell Studio, SAPIEN Software Suite (S3), iPowerShell Pro, ChangeVue, PrimalSQL, and PrimalXML are trademarks of SAPIEN Technologies, Inc. All other logos, trademarks, and service marks are the property of their respective owners. © 2002-2013 SAPIEN Technologies, Inc. All Rights Reserved.

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



In the context of the Scripting Games, 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.

PERF-02 Consider trade-offs between performance and readability

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.

This is an important area for people in the PowerShell community. While everyone agrees that aesthetics are important - they help make scripts more readable, more maintainable, and so on - performance can also be important. However, the advantages of a *really tiny performance gain* do not always outweigh the "soft" advantages of nice aesthetics.

For example:

```
$content = Get-Content file.txt
ForEach ($line in $content) {
    Do-Something -input $line
}
```

Most folks will agree that the basic aesthetics of that example are good. This snippet uses a native PowerShell approach, is easy to follow, and because it uses a structural programming approach, is easy to expand (say, if you needed to execute several commands again each line of content). However, this approach *could* offer extremely poor performance. If file.txt was a few hundred kilobytes, no problem; if it was several hundred *megabytes*, potential problem. Get-Content is forced to read the entire file into memory at once, storing it in memory (in the \$content variable).

Now consider this alternate approach:

```
Get-Content file.txt |
ForEach-Object -Process {
    Do-Something -input $_
}
```

As described elsewhere in this guide, many folks in the community would dislike this approach for aesthetic reasons. However, this approach has the advantage of utilizing PowerShell's pipeline to "stream" the content in file.txt. Provided that the fictional "Do-Something" command isn't blocking the pipeline (a la Sort-Object), the shell can send lines of content (String objects, technically) through the pipeline in a continuous stream, rather than having to buffer them all into memory.

Some would argue that this second approach is always a poor one, and that if performance is an issue then you should devolve from a PowerShell-native approach into a lower-level .NET Framework approach:

```
$sr = New-Object -Type System.IO.StreamReader -Arg file.txt
while ($sr.Peek() -ge 0) {
    $line = $sr.ReadLine()
    Do-Something -input $line
}
```

There are myriad variations to this approach, of course, but it solves the performance problem by reading one line at a time, instead of buffering the entire file into memory. It maintains the structured programming approach of the first example, at the expense of using a potentially harder-to-follow .NET Framework model instead of native PowerShell commands. Many regard this third example as an intermediate step, and suggest that a truly beneficial approach would be to write PowerShell commands as "wrappers" around the .NET code. For example (noting that this fourth example uses fictional commands by way of illustration):

```
$handle = Open-TextFile file.txt
while (-not Test-TextFile -handle $handle) {
    Do-Something -input (Read-TextFile -handle $handle)
}
```

This example reverts back to a native PowerShell approach, using commands and parameters. The proposed commands (Open-TextFile, Test-TextFile, and Read-TextFile) are just wrappers around .NET Framework classes, such as the StreamReader class shown in the third example.

You will generally find that it *is* possible to conform with the community's general aesthetic preferences while still maintaining a good level of performance. Doing so may require more work - such as writing PowerShell wrapper commands around underlying .NET Framework classes. Most would argue that, for a tool that is intended for long-term use, the additional work is a worthwhile investment.

The moral here is that both aesthetic and performance are important considerations, and without some work context, neither is inherently more important than the other. It is often possible, with the right technique, to satisfy *both*. As a *general* practice, you should avoid giving up on aesthetics *solely* because of performance concerns - when possible, make the effort to satisfy both performance and aesthetics.



In the context of the Scripting Games, *do not* penalize entries for using a particular aesthetic approach on the grounds of poor performance, unless the Games assignment explicitly makes performance a goal and sets up a scenario where performance would be a problem. In other words, don't penalize someone because "if such-and-such were true, performance would be bad," unless it's made clear that such-and-such is in fact true.

Aesthetics

READ-01 Indent your code

Consider this code example:

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

And now consider this one:

```
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*.

Continuing in that vein, 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.

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

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



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.

Verbose and Debug Output

OUT-01 Don't use write-host unless writing to the host is the only goal

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.

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

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”).

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.

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:

- 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. 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).

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

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.

TOOL-05 Use PowerShell standard parameter naming

Tools should be consistent with PowerShell native cmdlets in regards to naming, parameter naming, and so on.

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.

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.

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 Cacs.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.

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.

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 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



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.

Backticks

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 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:

```
$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.

“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.

Trapping and Capturing Errors

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. 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.

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 ($computename) {
    $ping = Get-WmiObject Win32_PingStatus -filter "Address='$computename'"
    if ($ping.StatusCode -eq 0) {
        return $true
    } else {
        return $false
    }
}
```

This function has a few problems. One, it uses the **Return** keyword, which some people find problematic (not a huge majority, but they do exist) because it's really a kind of “syntax sugar.” They would argue that you should use Write-Output instead. Second, the parameter block is not explicitly declared; the shortcut declaration here is less preferred. Third, the command verb (“Ping”) isn't an approved PowerShell command verb.

Fourth, and more to the point, there's no reason whatsoever to write this function in PowerShell v2 or later. Simply use:

```
Test-Connection $computename -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, thus rendering your work “wasted effort.”

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.