The PowerShell Best Practices and Style Guide

What are Best Practices

PowerShell Best Practices are what you should usually do as a starting point. They are ways of writing, thinking, and designing which make it *harder* to get into trouble. The point of a *Best Practice* is to help the reader to fall into the pit of success:

> **The Pit of Success:** in stark contrast to a summit, a peak, or a journey across a desert to find victory through many trials and surprises, we want our customers to simply fall into winning practices by using our platform and frameworks. To the extent that we make it easy to get into trouble we fail.

> -- Rico Mariani, MS Research MindSwap Oct 2003.

Like English spelling and grammar rules, PowerShell programming best practices and style rules nearly always have exceptions, but we are documenting a baseline for code structure, command design, programming, formatting, and even style which will help you to avoid common problems, and help you write more reusable, readable code -- because reusable code doesn't have to be rewritten, and readable code can be maintained.

Having said that, remember: the points in the Best Practices documents and the Style Guide are referred to as *practices* and *guidelines*, not rules. If you're having trouble getting something done because you're trying to avoid *breaking* a style or best practice rule, you've misunderstood the point: this document is pragmatic, rather than dogmatic. We'll leave dogmatism to teams and projects that require you to meet their specific guidelines.

Table of Contents

The guidelines are divided into these sections:

- Style Guide (Introduction)
  - Code Layout and Formatting
  - Function Structure
  - Documentation and Comments
  - Readability
  - Naming Conventions
- Best Practices (Introduction)
  - Naming Conventions
  - Building Reusable Tools
  - Output and Formatting
  - Error Handling
  - Performance
  - Security
  - Language, Interop and .Net
  - Metadata, Versioning, and Packaging

Current State:

Remember what we mean by *Best Practices*

The *PowerShell Best Practices* are always evolving, and continue to be edited and updated as the language and tools (and our community understanding of them) evolve. We encourage you to check back for new editions at least twice a year, by visiting https://github.com/PoshCode/PowerShellPracticeAndStyle

The *PowerShell Style Guide* in particular is in PREVIEW, and we are still actively working out our disagreements about the rules in the guide through the GitHub issues system.

Contributing

Please use the issues system or GitHub pull requests to make corrections, contributions, and other changes to the text - we welcome your contributions!

For more information, see CONTRIBUTING

Credits

The Community Book of PowerShell Practices was originally compiled and edited by Don Jones and Matt Penny with input from the Windows PowerShell community on PowerShell.org

The PowerShell Style Guide was originally created by Carlos Perez, for his students, and all the good parts were written by him.

Any mistakes in either of these documents are there because Joel Bennett got involved. Please submit issues and help us correct them.

The PowerShell Style Guide

Introduction

In the Python community, developers have a great programming style reference provided as part of the language enhancement process specifications (PEP-8), but in the PowerShell world there has been no official documentation of community preferences.

This document is an attempt to come to an agreement on a style-guide because we know that the more people follow the same set of code-style habits, the more readable the community's code will be. In other words, although the recommendations of this guide are *just recommendations*, if you follow them, you will write PowerShell code that is more easily read, understood, and maintained.

Code Layout & Formatting

These guidelines are about readability. Some of them are arbitrary rules, but they are based on decades of traditions in programming, so while you may disagree with some rules (and should always follow the rules of individual projects), when we ask you to leave an empty line after a closing function brace, or two lines before functions, we're not being capricious, we're doing so because it makes it easier for experienced developers to scan your code.

Maintain Consistency in Layout

Rules about indentation, line length, and capitalization are about consistency across code bases. Long practice has shown that it's easier to read and understand code when it looks familiar and you're not being distracted by details, which means that it's better for everyone in the community to follow a single set of rules.

We don't expect everyone to follow these guidelines, and rules for individual projects always trump these. Whether for legacy reasons, or to match guidelines for multiple languages in a

single project, different projects may have different style guidelines. Since the goal is consistency, you should always abide by any style rules that are in place on the project you are contributing to.

If you do have a legacy project that is in source control and you decide to reformat code to adopt these rules, try to make all of your whitespace changes in a single commit that does *nothing* but edit the whitespace. You should never reformat the whitespace on a file as *part* of a content change because it makes the changes hard to spot.

Capitalization Conventions

PowerShell is **not** case sensitive, but we follow capitalization conventions to make code easy to read. They are based on the capitalization conventions Microsoft created for the .NET framework, since PowerShell is a .NET scripting language, and PowerShell cmdlets are primarily written in .NET languages following those guidelines.

Terminology

- lowercase - all lowercase, no word separation
- UPPERCASE - all capitals, no word separation
- PascalCase - capitalize the first letter of each word
- camelCase - capitalize the first letter of each word *except* the first.

PowerShell uses PascalCase for *all* public identifiers: module names, function or cmdlet names, class, enum, and attribute names, public fields or properties, global variables and constants, etc. In fact, since the *parameters* to PowerShell commands are actually *properties* of .Net classes, even parameters use PascalCase rather than camelCase.

PowerShell language keywords are written in lower case (yes, even `foreach` and `dynamicparam`), as well as operators such as `-eq` and `-match`. The keywords in comment-based help are written in UPPERCASE to make it easy to spot them among the dense prose of documentation.

```
function Write-Host {
    <#
    .SYNOPSIS
        Writes customized output to a host.
    .DESCRIPTION
        The Write-Host cmdlet customizes output. You can specify the color of tex
        the ForegroundColor parameter, and you can specify the background color b
        BackgroundColor parameter. The Separator parameter lets you specify a str
        separate displayed objects. The particular result depends on the program
        hosting Windows PowerShell.
```

```
#>
[CmdletBinding()]
param(
    [Parameter(Position = 0, ValueFromPipeline = $true, ValueFromRemainingArg
    [psobject]$Object,

    [switch]$NoNewline,

    [psobject]$Separator,

    [System.ConsoleColor]$ForegroundColor,

    [System.ConsoleColor]$BackgroundColor
)
begin {
...
```

As stated previously, PowerShell uses PascalCase for *all* public identifiers. Function names should follow PowerShell's `Verb-Noun` naming conventions, using PascalCase within both Verb and Noun.

A special case is made for two-letter acronyms in which both letters are capitalized, as in the variable `$PSBoundParameters` or the command `Get-PSDrive`. Note that (as specified in the .NET guidelines) this does not affect the commonly capitalized (but not acronym) words "OK" and "ID". You should also not extend it to compound acronyms, such as when Azure's Resource Manager (RM) meets a Virtual Machine (VM) in `Start-AzureRmVM` ...

> We are aware that there are **many** places where these conventions have not been followed properly for various reasons -- you should consider these *exceptions* (such as for COM interop) or *mistakes* (such as `System.Data.SqlClient.SQLDebugging`), but not a reason for you to disregard the conventions.

If you wish, you may use camelCase for variables within your functions (or modules) to distinguish *private* variables from parameters, but this is a matter of taste. Shared variables should be distinguished by using their scope name, such as `$Script:PSBoundParameters` or `$Global:DebugPreference`. If you are using camelCase for a variable that starts with a two-letter acronym (where both letters are capitalized), both letters should be set to lowercase (such as `adComputer`).

One True Brace Style

This guide recommends the so-called "One True Brace Style" variant to K&R, which requires that every braceable *statement* should have the opening brace on the *end of a line*, and the closing brace at the *beginning of a line*.

There is one notable exception when passing small scriptblocks to parameters (where K&R would allow leaving off the braces entirely), we allow putting the entire statement on a single line.

```
enum Color {
    Black,
    White
}

function Test-Code {
    [CmdletBinding()]
    param (
        [int]$ParameterOne
    )
    end {
        if (10 -gt $ParameterOne) {
            "Greater"
        } else {
            "Lesser"
        }
    }
}

# An Exception case:
Get-ChildItem | Where-Object { $_.Length -gt 10mb }
```

The primary reason for this recommendation is practical: there are no exceptions necessary when following this rule, and when code is written following this style, *new lines* of code can be inserted between any two lines with no risk of accidentally breaking the code by separating braces from their statement blocks. Thus, it's easier to follow, and makes errors less likely.

Because this choice was somewhat contentious in the community (about 1/3 of voters opposed), it's worth adding some additional reasoning here: First: in some historical consoles, it was necessary to write this way, so much of the early PowerShell code follows this style anyway. Second: PowerShell functions which accept scriptblocks (such as `ForEach-Object` and `Where-Object`) are common, and an *inherent* part of the syntax of important PowerShell-based domain-specific languages such as DSC. Since it's **required** to place the opening brace on the end of the line in those cases, the only *consistent* option is to follow OTBS.

Always Start With CmdletBinding

All of your scripts or functions should start life as something like this snippet:

```
[CmdletBinding()]
```

```
param ()
process {
}
end {
}
```

You can always delete or ignore one of the blocks (or add the `begin` block), add parameters and necessary validation and so on, but you should **avoid** writing scripts or functions without `[CmdletBinding()]`, and you should always at least *consider* making it take pipeline input.

Prefer: param (), begin, process, end

Having a script written in the order of execution makes the intent clearer. Since there is no functional reason to have these blocks out of order (they *will* still be executed in the normal order), writing them out of order can be confusing, and makes code more difficult to maintain and debug.

More explicit code is more maintainable. While PowerShell allows leaving off the explicit name of the `end` block (and even has a `filter` keyword that converts the anonymous block to a `process` block), we recommend against using these features as it results in less explicit code.

Indentation

Use four *spaces* per indentation level

Usually you will press the `[Tab]` key to indent, but most editors can be configured to insert spaces instead of actual tab characters. For most programming languages and editors (including PowerShell ISE) the default is four spaces, and that's what we recommend. Different teams and projects may have different standards, and when contributing to a project, you should abide by the predominant style, of course.

```
function Test-Code {
    foreach ($exponent in 1..10) {
        [Math]::Pow(2, $exponent)
    }
}
```

Indenting more than 4-spaces is acceptable for continuation lines (when you're wrapping a line which was too long). In such cases you might indent more than one level, or even indent indent an odd number of spaces to line up with a method call or parameter block on the line before.

```
function Test-Code {
    foreach ($base in 1,2,4,8,16) {
```

```
        foreach ($exponent in 1..10) {
            [System.Math]::Pow($base,
                               $exponent)
    }
}
```

Maximum Line Length

Limit lines to 115 characters when possible.

Keeping lines to a small width allows scripts to be read in *one* direction (top to bottom) without scrolling back-and-forth horizontally. What, exactly, this width should be is a one of the favorite arguing points among developers on the internet (more splintered than emacs vs vi or gnu GPL vs MIT).

In this guide we use two particular sources for the maximum line width:

The PowerShell console is, by default, 120 characters wide, but it allows only 119 characters on output lines, and when entering multi-line text, PowerShell uses a line continuation prompt: `>>>` and thus limits your line length to 116 anyway.

Github's current maximum line width varies between 121 and 126 depending on your browser and OS (and thus, font). However, the 115 line length suggested by PowerShell would be enough to even allow side-by-side diffs to be displayed without scrolling or wrapping on the current "standard" 1080p monitor.

Again, this is a particularly flexible rule, and you should always follow the guidelines of projects when you're contributing to other people's projects. Although most of us work on widescreen monitors, not everyone can see well without magnification or extremely large fonts.

The preferred way to avoid long lines is to use splatting (see Get-Help about_Splatting) and PowerShell's implied line continuation inside parentheses, brackets, and braces -- these should **always** be used in preference to the backtick for line continuation when applicable, even for strings:

```
Write-Host -Object ("This is an incredibly important, and extremely long message. " +
                    "We cannot afford to leave any part of it out, " +
                    "nor do we want line-breaks in the output. " +
                    "Using string concatenation lets us use short lines here " +
                    "and still get a long line in the output")
```

Blank Lines and Whitespace

Surround function and class definitions with *two* blank lines.

Method definitions within a class are surrounded by a single blank line.

Blank lines may be omitted between a bunch of related one-liners (e.g. empty functions)

Additional blank lines may be used sparingly to separate groups of related functions, or within functions to indicate logical sections (e.g. before a block comment).

End each file with a single blank line.

Trailing spaces

Lines should not have trailing whitespace. Extra spaces result in future edits where the only change is a space being added or removed, making the analysis of the changes more difficult for no reason.

Spaces around parameters and operators

You should use a single space around parameter names and operators, including comparison operators and math and assignment operators, even when the spaces are not necessary for PowerShell to correctly parse the code.

One notable exception is when using colons to pass values to switch parameters:

```
# Do not write:
$variable=Get-Content $FilePath -Wait:($ReadCount-gt0) -First($ReadCount*5)

# Instead write:
$variable = Get-Content -Path $FilePath -Wait:($ReadCount -gt 0) -TotalCount ($Re
```

Another exception is when using Unary Operators:

```
# Do not write:
$yesterdaysDate = (Get-Date).AddDays( - 1)

$i = 0
$i ++

# Instead write:
$yesterdaysDate = (Get-Date).AddDays(-1)

$i = 0
$i++
```

```
# Same principle should be applied when using a variable.
$yesterdaysDate = (Get-Date).AddDays(-$i)
```

Spaces around special characters

White-space is (mostly) irrelevant to PowerShell, but its proper use is key to writing easily readable code.

Use a single space after commas and semicolons, and around pairs of curly braces.

Subexpressions `$( ... )` and scriptblocks `{ ... }` should have a single space on the *inside* of the braces or parentheses to improve readability by making code blocks stand out -- and to further distinguish scriptblocks from variable delimiter braces `${...}`

Avoid unnecessary spaces inside parenthesis or square braces.

```
$Var = 1
"This is a string with one (${Var}) delimited variable."

"There are $( (Get-ChildItem).Count ) files."
```

Obviously, these rules should not be applied in such a way as to affect output.

Avoid Using Semicolons (;) as Line Terminators

PowerShell will not complain about extra semicolons, but they are unnecessary, and can get in the way when code is being edited or copy-pasted. They also result in extra do-nothing edits in source control when someone finally decides to delete them.

They are also unnecessary when declaring hashtables if you are already putting each element on its own line:

```
# This is the preferred way to declare a hashtable if it extends past one line:
$Options = @{
    Margin   = 2
    Padding  = 2
    FontSize = 24
}
```

Functions

Avoid using the `return` keyword in your functions. Just place the object variable on its own.

When declaring simple functions leave a space between the function name and the parameters.

```
function MyFunction ($param1, $param2) {
    ...
}
```

Advanced Functions

For Advanced Functions and scripts use the format of **<verb>-<noun>** for naming. For a list of approved verbs the cmdlet `Get-Verb` will list them. On the noun side it can be composed of more than one joined word using Pascal Case and only singular nouns.

In Advanced Functions do not use the keyword `return` to return an object.

In Advanced Functions you return objects inside the `Process {}` block and not in `Begin {}` or `End {}` since it defeats the advantage of the pipeline.

```
# Bad
function Get-USCitizenCapability {
    [CmdletBinding()]
    [OutputType([psobject])]
    param (
        [Parameter(Mandatory = $true,
                   ValueFromPipelineByPropertyName = $true,
                   Position = 0)]
        [int16]
        $Age
    )
    process {
        $Capabilities = @{
            MilitaryService = $false
            DrinkAlcohol = $false
            Vote = $false
        }
        if ($Age -ge 18) {
            $Capabilities['MilitaryService'] = $true
            $Capabilities['Vote'] = $true
        }

        $Obj = New-Object -Property $Capabilities -TypeName psobject
    }
    end { return $Obj }
}

# Good
```

```
function Get-USCitizenCapability {
    [CmdletBinding()]
    [OutputType([psobject])]
    param (
        [Parameter(Mandatory = $true,
                   ValueFromPipelineByPropertyName = $true,
                   Position = 0)]
        [int16]
        $Age
    )
    process {
        $Capabilities = @{
            MilitaryService = $false
            DrinkAlcohol = $false
            Vote = $false
        }

        if ($Age -ge 18) {
            $Capabilities['MilitaryService'] = $true
            $Capabilities['Vote'] = $true
        }

        New-Object -Property $Capabilities -TypeName psobject
    }
}
```

Always use CmdletBinding attribute.

Always have at least a `process {}` code block if any parameters takes values from the Pipeline.

Specify an OutputType attribute if the advanced function returns an object or collection of objects.

If the function returns different object types depending on the parameter set provide one per parameter set.

```
[OutputType([<TypeLiteral>], ParameterSetName = "<Name>")]
[OutputType("<TypeNameString>", ParameterSetName = "<Name>")]
```

When a ParameterSetName is used in any of the parameters, always provide a DefaultParameterSetName in the CmdletBinding attribute.

```
function Get-User {
    [CmdletBinding(DefaultParameterSetName = "ID")]
```

```
    [OutputType("System.Int32", ParameterSetName = "ID")]
    [OutputType([String], ParameterSetName = "Name")]
    param (
        [parameter(Mandatory = $true, ParameterSetName = "ID")]
        [Int[]]
        $UserID,

        [parameter(Mandatory = $true, ParameterSetName = "Name")]
        [String[]]
        $UserName
    )
    <# function body #>
}
```

When using advanced functions or scripts with CmdletBinding attribute avoid validating parameters in the body of the script when possible and use parameter validation attributes instead.

- **AllowNull** Validation Attribute

  The AllowNull attribute allows the value of a mandatory parameter to be null ($null).

```
param (
    [Parameter(Mandatory = $true)]
    [AllowNull()]
    [String]
    $ComputerName
)
```

- **AllowEmptyString** Validation Attribute

  The AllowEmptyString attribute allows the value of a mandatory parameter to be an empty string ("").

```
param (
    [Parameter(Mandatory = $true)]
    [AllowEmptyString()]
    [String]
    $ComputerName
)
```

- **AllowEmptyCollection** Validation Attribute

  The AllowEmptyCollection attribute allows the value of a mandatory parameter to be an

empty collection (@()).

```
param (
    [Parameter(Mandatory = $true)]
    [AllowEmptyCollection()]
    [String[]]
    $ComputerName
)
```

- **ValidateCount** Validation Attribute

  The ValidateCount attribute specifies the minimum and maximum number of parameter values that a parameter accepts. Windows PowerShell generates an error if the number of parameter values in the command that calls the function is outside that range.

```
param (
    [Parameter(Mandatory = $true)]
    [ValidateCount(1,5)]
    [String[]]
    $ComputerName
)
```

- **ValidateLength** Validation Attribute

  The ValidateLength attribute specifies the minimum and maximum number of characters in a parameter or variable value. Windows PowerShell generates an error if the length of a value specified for a parameter or a variable is outside of the range.

```
param (
    [Parameter(Mandatory = $true)]
    [ValidateLength(1,10)]
    [String[]]
    $ComputerName
)
```

- **ValidatePattern** Validation Attribute

  The ValidatePattern attribute specifies a regular expression that is compared to the parameter or variable value. Windows PowerShell generates an error if the value does not match the regular expression pattern.

```
param (
```

```
    [Parameter(Mandatory = $true)]
    [ValidatePattern("[0-9][0-9][0-9][0-9]")]
    [String[]]
    $ComputerName
)
```

- **ValidateRange** Validation Attribute

  The ValidateRange attribute specifies a numeric range for each parameter or variable value. Windows PowerShell generates an error if any value is outside that range.

```
param (
    [Parameter(Mandatory = $true)]
    [ValidateRange(0,10)]
    [Int]
    $Attempts
)
```

- **ValidateScript** Validation Attribute

  The ValidateScript attribute specifies a script that is used to validate a parameter or variable value. Windows PowerShell pipes the value to the script, and generates an error if the script returns "false" or if the script throws an exception.

  When you use the ValidateScript attribute, the value that is being validated is mapped to the $_ variable. You can use the $_ variable to refer to the value in the script.

```
param (
    [Parameter()]
    [ValidateScript({$_ -ge (get-date)})]
    [DateTime]
    $EventDate
)
```

- **ValidateSet** Attribute

  The ValidateSet attribute specifies a set of valid values for a parameter or variable. Windows PowerShell generates an error if a parameter or variable value does not match a value in the set. In the following example, the value of the Detail parameter can only be "Low," "Average," or "High."

```
param (
    [Parameter(Mandatory = $true)]
```

```
    [ValidateSet("Low", "Average", "High")]
    [String[]]
    $Detail
)
```

- **ValidateNotNull** Validation Attribute

  The ValidateNotNull attribute specifies that the parameter value cannot be null ($null). Windows PowerShell generates an error if the parameter value is null.

  The ValidateNotNull attribute is designed to be used when the type of the parameter value is not specified or when the specified type will accept a value of Null. (If you specify a type that will not accept a null value, such as a string, the null value will be rejected without the ValidateNotNull attribute, because it does not match the specified type.)

```
param (
    [Parameter(Mandatory = $true)]
    [ValidateNotNull()]
    $ID
)
```

- **ValidateNotNullOrEmpty** Validation Attribute

  The ValidateNotNullOrEmpty attribute specifies that the parameter value cannot be null ($null) and cannot be an empty string (""). Windows PowerShell generates an error if the parameter is used in a function call, but its value is null, an empty string, or an empty array.

```
param (
    [Parameter(Mandatory = $true)]
    [ValidateNotNullOrEmpty()]
    [String[]]
    $UserName
)
```

Documenting and Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be in English, and should be complete sentences. If the comment is short, the period at the end can be omitted.

Remember that comments should serve to your reasoning and decision-making, not attempt to

explain what a command does. With the exception of regular expressions, well-written PowerShell can be pretty self-explanatory.

```
# Do not write:
# Increment Margin by 2
$Margin = $Margin + 2

# Maybe write:
# The rendering box obscures a couple of pixels.
$Margin = $Margin + 2
```

Block comments

Don't go overboard with comments. Unless your code is particularly obscure, don't precede each line with a comment -- doing so breaks up the code and makes it harder to read. Instead, write a single block comment.

Block comments generally apply to some or all of the code which follows them, and are indented to the same level as that code. Each line should start with a # and a single space.

If the block is particularly long (as in the case of documentation text) it is recommended to use the `<# ... #>` block comment syntax, but you should place the comment characters on their own lines, and indent the comment:

```
# Requiring a space makes things legible and prevents confusion.
# Writing comments one-per line makes them stand out more in the console.

<#
    .SYNOPSIS
        Really long comment blocks are tedious to keep commented in single-line m
    .DESCRIPTION
        Particularly when the comment must be frequently edited,
        as with the help and documentation for a function or script.
#>
```

Inline comments

Comments on the same line as a statement can be distracting, but when they don't state the obvious, and particularly when you have several short lines of code which need explaining, they can be useful.

They should be separated from the code statement by at least two spaces, and ideally, they should line up with any other inline comments in the same block of code.

```
$Options = @{
    Margin = 2          # The rendering box obscures a couple of pixels.
    Padding = 2         # We need space between the border and the text.
    FontSize = 24       # Keep this above 16 so it's readable in presentations.
}
```

Documentation comments

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 unnecessarily large words, and keep your explanations short. You're not trying to impress anyone, and the only people who will ever read this are just trying to figure out how to use the function.

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.

Location

In order to ensure that the documentation stays with the function, documentation comments should be placed INSIDE the function, rather than above. To make it harder to forget to update them when changing a function, you should keep them at the top of the function, rather than at the bottom.

Of course, that's not to say that putting them elsewhere is wrong -- but this is easier to do, and harder to forget to update.

Put Details in the Notes

If you want to provide detailed explanations about how your tool works, use the `Notes` section for that.

Describe The Function

Every script function command should have at least a short statement describing it's function. That is the `Synopsis`.

Document Each Parameter

Each parameter should be documented. To make it easier to keep the comments synchronized with changes to the parameters, the preferred location for parameter documentation comments

is *within* the `param` block, directly above each parameter. Examples can be found in the ISE snippets:

```
param (
    # Param1 help description
    [Parameter(Mandatory = $true,
               ValueFromPipelineByPropertyName = $true,
               Position = 0)]
    $Param1,

    # Param2 help description
    [int]
    $Param2
)
```

It is also possible to write `.PARAMETER` statements with the rest of the documentation comments, but experience shows they are more likely to be kept up-to-date if you put them closer to the code they document.

Provide Usage Examples

Your help should always provide an example for each major use case. A 'usage example' is just an example of what you would type in to Powershell to run the script - you can even cut and paste one from the command line while you're testing your function.

```
function Test-Help {
    <#
        .SYNOPSIS
            An example function to display how help should be written.

        .EXAMPLE
            Get-Help -Name Test-Help

            This shows the help for the example function.
    #>
    [CmdletBinding()]
    param (
        # This parameter doesn't do anything.
        # Aliases: MP
        [Parameter(Mandatory = $true)]
        [Alias("MP")]
        [String]$MandatoryParameter
    )

    <# code here ... #>
```

```
}
```

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 Get-Example {
    <#
    .SYNOPSIS
        A brief description of the function or script.

    .DESCRIPTION
        A longer description.

    .PARAMETER FirstParameter
        Description of each of the parameters.
        Note:
        To make it easier to keep the comments synchronized with changes to the p
        the preferred location for parameter documentation comments is not here,
        but within the param block, directly above each parameter.

    .PARAMETER SecondParameter
        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 Get-Example` or `Get-Example -?` , etc.

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. TODO: This section should probably be merged to Code Layout and Formatting, and based on the #15, we should remove or rewrite the backticks section.

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 properly. 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 don't format carefully.

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 mistype. 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:

```
$GetWmiObjectParams = @{
    Class = "Win32_LogicalDisk"
    Filter = "DriveType=3"
    ComputerName = "SERVER2"
}
Get-WmiObject @GetWmiObjectParams
```

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.

Naming Conventions

In general, prefer the use of full explicit names for commands and parameters rather than aliases or short forms. There are tools like PSScriptAnalyzer's `Invoke-Formatter` and scripts like Expand-Alias for fixing many, but not all of these issues.

Use the full name of each command.

Every PowerShell scripter learns the actual command names, but different people learn and use

different aliases (e.g.: ls for Linux users, dir for DOS users, gci ...). In your shared scripts you should use the more universally known full command name. As a bonus, sites like GitHub will highlight commands properly when you use the full Verb-Noun name:

```
# Do not write:
gps –Name Explorer

# Instead write:
Get–Process –Name Explorer
```

Use full parameter names.

Because there are so many commands in PowerShell, it's impossible for every scripter to know every command. Therefore it's useful to be explicit about your parameter names for the sake of readers who may be unfamiliar with the command you're using. This will also help you avoid bugs if a future change to the command alters the parameter sets.

```
# Do not write:
Get–Process Explorer

# Instead write:
Get–Process –Name Explorer
```

Use full, explicit paths when possible.

When writing scripts, it is only safe to use `..` or `.` in a path if you have previously set the location explicitly (within the current function or script). Even if you *have* explictly set the path, you must beware of using relative paths when calling .Net methods or legacy/native applications, because they will use `[Environment]::CurrentDirectory` which is not automatically updated to PowerShell's present working directory (`$PWD`).

Because troubleshooting these types of errors is tedious (and they are easy to over-look) it's best to avoid using relative paths altogether, and instead, base your paths off of $PSScriptRoot (the folder your script is in) when necessary.

```
# Do not write:
Get–Content .\README.md

# Especially do not write:
[System.IO.File]::ReadAllText(".\README.md")
```

```
# Although you can write:
Push-Location $PSScriptRoot
Get-Content README.md

# It would be better to write:
Get-Content -Path (Join-Path -Path $PSScriptRoot -ChildPath README.md)
# Or to use string concatenation:
Get-Content "$PSScriptRoot\README.md"

# For calling .net methods, pass full paths:
[System.IO.File]::ReadAllText("$PSScriptRoot\README.md")

# Optionally by calling Convert-Path
Push-Location $PSScriptRoot
[System.IO.File]::ReadAllText((Convert-Path README.md))
```

Avoid the use of `~` to represent the home folder.

The meaning of ~ is unfortunately dependent on the "current" provider at the time of execution. This isn't really a style issue, but it's an important rule for code you intend to share anyway. Instead, use `${Env:UserProfile}` or `(Get-PSProvider -PSProvider FileSystem).Home` ...

```
PS C:\Windows\system32> cd ~
PS C:\Users\Name> cd HKCU:\Software
PS HKCU:\Software> cd ~
cd : Home location for this provider is not set. To set the home location, call "
At line:1 char:1
+ cd ~
+ ~~~~
    + CategoryInfo          : InvalidOperation: (:) [Set-Location], PSInvalidOper
    + FullyQualifiedErrorId : InvalidOperation,Microsoft.PowerShell.Commands.SetL
```

See also the Capitalization Conventions

In the Code Layout and Formatting chapter, there is a section on capitalization conventions.

PowerShell Best Practices

Foreword

If you scan through code projects on PoshCode or the Technet ScriptCenter Gallery, it will be

immediately apparent that people in the PowerShell community have vastly different ideas about what's "right and wrong" in the world of PowerShell scripting.

Over the years several attempts have been made to arrive at a consensus, most notably the "Great Debate" series of blog posts on PowerShell.org following the 2013 Scripting Games, which outlined some of the more controversial issues and asked for community discussions.

This project has been created, in part, as a public place to continue those debates as well as to document the consensus of the community when we *do* arrive at a consensus.

Remember that best practices are not hard-and-fast rules. We don't even consider them as solid as the style guide rules. They are the things you should *usually* do as a starting point, and should deviate from when it's appropriate.

One final note about these Best Practices: the perspective of these guidelines has so-far been strongly influenced by system administrator practitioners, less-so by language geeks and developers. We're trying to balance that and provide perspective, but when it comes to best practices, we definitely allow the experience of administrators to drive what the PowerShell community considers best and worst practices -- so if you are working from a different perspective, you'll have to take all of this with a grain of salt.

Table of Contents

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).

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.

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 or $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.

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.

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.

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 github.com/powershell also.

TODO: The "PURE" section is dubious at best. We need to discuss it.

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.

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 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.

Writing Parameter Blocks

Always write Help

For every script and function you should have a comment-based help block (we recommend using a block comment). The best place for these is *inside* the `function` above the `param` block, but they can also be placed *above* the function, or at the bottom just before closing.

In order for it to register as help, you must provide a `.SYNOPSIS` and/or `.DESCRIPTION`

Always Write Examples

You should also always provide at least one example (even if your function doesn't take parameters), where you show the output (or the need to capture it), and explain what happens when the command is run. Note that examples should have the *code* first, and the *documentation* of what it does after an empty line or two.

```
function Test-Help {
    <#
        .SYNOPSIS
            An example function to display how help should be written.

        .EXAMPLE
            Test-Help -Name Test-Help

            This tests the help for the Test-Help function.
    #>
    [CmdletBinding()]
    param (
        # This parameter doesn't do anything, but you must provide a value
        # Aliases: MP
        [Parameter(Mandatory = $true)]
        [Alias("MP")]
        [String]$MandatoryParameter
    )

    <# code here ... #>
}
```

Always Document Every Parameter

You should always provide at least a brief explanation of each parameter, what it's expected or

allowed values are, etc.

The best place for this is a simple comment directly above the parameter (inside the param block) so you don't forget to update it if you remove, rename, or otherwise change the parameter, but you can also place them in the comment help block by using `.PARAMETER ParameterName` and writing the help on the next line.

You should specify `[CmdletBinding()]`

CmdletBinding makes functions and scripts behave like the built-in commands, adding support for the "common" output parameters like `-Verbose` and `-ErrorAction` and supporting `-?` for help. If you don't support it, you risk someone *accidentally* running your code when they were just trying to `Get-Help` on it.

There are a few specific advanced cases where you might want to write an old-fashioned script that doesn't use CmdletBinding() -- they are very rare, and are all exceptions to the best practices, so we won't go into them further.

You should pick a default ParameterSet

If you have more than one ParameterSetName on your parameters, you should specify one of them as the `DefaultParameterSetName` in the CmdletBinding.

You should support --whatif

If you write a command that changes state, you should probably add `SupportsShouldProcess` to your CmdletBinding. This allows users to specify `-WhatIf` and `-Confirm` when calling your command, so you'll need to actually support those by using `$PSCmdlet.ShouldProcess(...)` or `$PSCmdlet.ShouldContinue(...)` or by passing the preference variable on to other commands you're calling (e.g. `-Whatif:$WhatIfPreference`).

Here's an example of what that might look like:

```
# NOTE: ConfirmImpact defaults to Medium
# But I recommend setting ConfirmImpact explicitly as a reminder :)
[CmdletBinding(SupportsShouldProcess, ConfirmImpact = "Medium")]
param([switch]$Force)

# You need to pre-define these (because they're passed by [ref])
$RejectAll = $false;
$ConfirmAll = $false;
```

```
# Note: please don't actually do this with services, restarting them in non-depen
foreach ($service in Get-Service | Where Status -eq "Running") {
    # This will normally automatically be TRUE. It will only query if the user:
    # 1. Has their $ConfirmPreference (default High) set LOWER or equal to the Co
    # 2. Passes -Confirm, which sets the $ConfirmPreference in the function's sco
    if ($PSCmdlet.ShouldProcess( "Restarted the service '$($service.Name)'",
                                 "Restart the '$($service.DisplayName)' service (
                                 "Restarting Services" )) {

        # If you use ShouldContinue, have a -Force parameter that bypasses it
        # And if you know there may be multiple prompts, you should use this over
        # In this example, we're only prompting when there are dependent services
        if ($Force -Or $service.DependentServices.Count -eq 0 -or $PSCmdlet.Shoul
                                             "$($service.Name) has $($service.Depe
                                             "Restarting the '$($service.DisplayNa
                                             [ref]$ConfirmAll,
                                             [ref]$RejectAll)) {
            "(Not actually) restarting $($service.DisplayName)"
        }
    }
}
```

You should strongly type parameters

Although PowerShell is a dynamic language, we can specify types, and in parameters, it's particularly useful.

First, because it hints to users what sort of values they can pass to your command. Is it numeric? Text? An object?

Second, because using types on parameters helps validate the input, which is crucial because parameters are where you get your user input. Strong types can help you avoid code injection and other problems with user inputs, and will allow failures to happen as early as possible (even before your command is called).

Additionally, when passing on parameters to another command, you should be *at least* as strongly typed as the other command, to avoid casting exceptions within your script.

Be careful with `[string]` or `[object]` (and `[PSObject]`)

Obviously [string] is one of the most common parameter types, and [object] is the default type. However, because anything can be cast to these types, you should avoid combining these types with parameters that are designed to differentiate parameter sets, or that accept `ValueFromPipeline` because PowerShell will coerce *everything* to that.

Obviously if you want to accept more than one type of object on the same parameter, you have to use `[object]` or `[PSObject]` as the universal base types. You have to be very careful when doing this, and normally should use a `ValidateScript` to ensure the objects are one of your *actual* supported types.

Using [pscredential]

When you need to accept credentials, you almost always want to name the parameter `$Credential` and accept a System.Management.Automation.PSCredential, which has special support in PowerShell for automatically coercing user names to credentials and more.

In old versions of PowerShell, you needed to manually decorate these PSCredential parameters with `[System.Management.Automation.CmdletAttribute()]` in order to automatically coerce user names to credentials -- that is, to support the use case where someone writes `–Credential Jaykul` and is automatically prompted for the password using the secure credential prompt. In current versions, this is automatically added when you use the PSCredential type.

Using [switch]

Parameters of type `[switch]` support passing as switches (without a value), and by default cannot be passed by position.

- Switch parameters should not be given default values. They should always default to false.
- Switch parameters should be designed so that setting them moves a command from its default functionality to a less common or more complicated mode.
- Switch parameters should be treated as boolean values in your scripts. Corrolary: you should not write logic that depends on whether or not the user explicitly passed a value to a switch -- do not attempt to treat a switch as having three states!
- When you need to pass the value of a switch on to another command, you can either splat it, or specify it using the colon syntax for parameters, as in `–TheirSwitch:$MySwitch`

Be generous with accept ValueFromPipelineByPropertyName

For the most flexibility, whenever it's practical, you should write your commands to accept their parameters from the pipeline *by property name*. To enhance your ability to match objects, you can add aliases for the parameter name using the `[Alias()]` attribute.

Don't forget that values set from the pipeline are only available in the `process` block.

Formatting Output

TODO: This whole document is STILL ROUGH DRAFT

Don't use Write-Host unless you really mean it

Previous to PowerShell 5, Write-Host has no functionality at all in non-interactive scripts. It cannot be captured or redirected, and therefore should only be used in functions which are "Show"ing or "Format"ing output, or to display something as part of an interactive prompt to the user.

That is: you should not use Write-Host to create script output unless your script (or function, or whatever) uses the Show verb (as in, Show-Performance) or the Format verb (as in, Format-Hex), or has a `-Formatted` switch parameter. You may also use it to build a interactions with the user in other cases (e.g. to write extra information to the screen before prompting the user for a choice or input).

Generally, you should consider the other Write-* commands first when trying to give information to the user.

Use Write-Progress to give progress information to someone running your script

When you're letting the user know how far through the script they are, or just making sure they know that *something* is happening, Write-Progress is the right command to use. In the case of graphical hosts or remote jobs, this output can be shown to the user in real time, even when verbose and other streams are being collected and logged.

Progress output is ephemeral, however, in that it doesn't stick around. You should not put anything exclusively in the progress stream that the user *needs* to see, or might want to review after the script finishes.

Use Write-Verbose to give details to someone running your script

You should use verbose output for information that contains details about the value of computation, or the reason why a certain execution path was chosen. It should be information that is useful *but not necessary* for anyone running the script, providing status information such as "Server1 processed", or logic information such as "Server2 up-to-date, skipped", but shouldn't be used for actual results or important information.

Use Write-Debug to give information to someone maintaining

your script

You should use the debug output stream for output that is useful for script debugging (ie: "Now entering main loop" or "Result was null, skipping to end of loop"), or to display the value of a variable before a conditional statement, so the maintainer can break into the debugger if necessary.

> TIP: When debugging you should be aware that you can set `$DebugPreference = "Continue"` to see this information on screen without entering a breakpoint prompt.

Use CmdletBinding if you are using output streams

As we've already written elsewhere, you should probably always use CmdletBinding.

Using CmdletBinding is particularly important, however, when you're using Write-Verbose and Write-Debug, as the Verbose and Debug output streams are off by default, and the `[CmdletBinding()]` attribute enables the common `-Verbose` and `-Debug` switches which turr those streams on.

CmdletBinding also enables the switches for the Warning and Error streams, as well as ways of collecting those streams into variables. You should read the always use CmdletBinding topic for more information.

Use Format Files for your custom objects

You should not use format commands inside functions. Instead you should include a `modulename.format.ps1xml` file in the FormatsToProcess field of your module's PSD1 manifest, and define a `PSTypeName` on your objects, so that PowerShell will format your output the way you want, automatically.

Only output one "kind" of thing at a time

You should avoid mixing different types of objects in the output of a single command, because you may get empty rows in your output or cause table output to break into list output, etc.

For the sake of tools and command-search, you should indicate with the `[OutputType()]` attribute the output type(s) of your scripts, functions or cmdlets (see about_Functions_OutputTypeAttribute for more information).

When you combine the output of multiple types objects, they should generally be derived from a common basetype (like FileInfo and DirectoryInfo come from System.IO.FileSystemInfo), or

should have format or type files which cause them to output the same columns. In particular, you must avoid outputting strings interspersed in your output.

Two important exceptions to the single-type rule

For internal functions it's ok to return multiple different types because they won't be "output" to the user/host, and can offer significant savings (e.g. one database call with three table joins, instead of three database calls with two or three joins each). You can then call these functions and assign the output to multiple variables, like so:

```
$user, $group, $org = Get-UserGroupOrg
```

When you call Out-Default. If you must return multiple object types from an external command, you should name your function in such a way that it's obvious to users that you're returning multiple things, and you *must* call `Out-Default` separately for each type of object to ensure that the outputs don't ever get mixed up by the formatter.

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' 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.

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.

PERF-01 If performance matters, test it

PowerShell comes equipped with 3.2 million performance quirks. Approximately.

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 Measure-Command or the Profiler module, or some other tool.

For example:

```
foreach($result in Do-Something) { $result.PropertyOne + $result.PropertyTwo }
Do-Something | ForEach-Object { $_.PropertyOne + $_.PropertyTwo }
```

In this case, the `foreach` language construct is faster than piping to the `ForEach-Object` cmdlet -- but the point is that you should measure, and do so on the hardware and PowerShell version where the performance matters to you.

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 - although "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 -Path 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 -Path 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 -TypeName System.IO.StreamReader -ArgumentList 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 -Path 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.

PERF-03 Language > Framework > Script > Pipeline

This is just a rough guideline, but as a general rule:

1. Language features are faster than features of the .net framework
2. Compiled methods on objects and .net classes are still faster than script
3. Simple PowerShell script is still faster than calling functions or cmdlets

It's counter-intuitive that script is faster than calling cmdlets that are compiled, but it's frequently true, unless there is a lot of work being done by each cmdlet. The overhead of calling cmdlets and passing data around is significant. Of course, this is just a guideline, and you should always **measure**.

Security

Always use PSCredential for credentials/passwords

You must avoid storing the password in a plain string object, or allowing the user to type them in as a parameter (where it might end up in the history or exposed to screen-scraper malware). The best method for this is to always deal with PSCredential objects (which store the Password in a SecureString).

More specifically, you should always take PSCredentials as a parameter (and never call Get-Credential within your function) to allow the user the opportunity to reuse credentials stored in a variable.

Furthermore, you should use the Credential attribute as the built-in commands do, so if the user passes their user name (instead of a PSCredential object), they will be prompted for their password in a Windows secure dialog.

```
param (
    [System.Management.Automation.PSCredential]
    [System.Management.Automation.Credential()]
    $Credentials
)
```

If you absolutely must pass a password in a plain string to a .Net API call or a third party library it is better to decrypt the credential as it is being passed instead of saving it in a variable.

```
# Get the cleartext password for a method call:
$Insecure.SetPassword( $Credentials.GetNetworkCredential().Password )
```

Other Secure Strings

For other strings that may be sensitive, use the SecureString type to protect the value of the string. Be sure to always provide an example for the user of passing the value using `Read-Host -AsSecureString`.

Note, if you ever need to turn a SecureString into a string, you can use this method, but make sure to call ZeroFreeBSTR to avoid a memory leak:

```
# Decrypt a secure string.
$BSTR = [System.Runtime.InteropServices.marshal]::SecureStringToBSTR($this);
$plaintext = [System.Runtime.InteropServices.marshal]::PtrToStringAuto($BSTR)
[System.Runtime.InteropServices.Marshal]::ZeroFreeBSTR($BSTR);
return $plaintext
```

- For credentials that need to be saved to disk, serialize the credential object using Export-CliXml to protect the password value. The password will be protected as a secure string and will only be accessible to the user who generated the file on the same computer where it was generated.

```
# Save a credential to disk
Get-Credential | Export-CliXml -Path c:\creds\credential.xml

# Import the previously saved credential
$Credential = Import-CliXml -Path c:\creds\credential.xml
```

- For strings that may be sensitive and need to be saved to disk, use ConvertFrom-SecureString to encrypt it into a standard string that can be saved to disk. You can then use ConvertTo-SecureString to convert the encrypted standard string back into a SecureString. NOTE: These commands use the Windows Data Protection API (DPAPI) to encrypt the data, so the encrypted strings can only be decrypted by the same user on the same machine, but there is an option to use AES with a shared key.

```
# Prompt for a Secure String (in automation, just accept it as a parameter)
$Secure = Read-Host -Prompt "Enter the Secure String" -AsSecureString
```

```
# Encrypt to Standard String and store on disk
ConvertFrom-SecureString -SecureString $Secure | Out-File -Path "${Env:AppDat

# Read the Standard String from disk and convert to a SecureString
$Secure = Get-Content -Path "${Env:AppData}\Sec.bin" | ConvertTo-SecureString
```

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.

Further information: You can get some detail on the differences between versions of Powershell 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

All that said, make sure you 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.

PowerShell Supported Version

When working in an environment where there are multiple versions of PowerShell make sure to specify the lowest version your script will support by providing a Requires statement at the top of the script.

```
#Requires -Version 2.0
```

When a *module* uses specific cmdlets or syntax that is only present on a specific minimum version of PowerShell in the module manifest ps1d file.

```
PowerShellVersion = '3.0'
```