PowerShell Quick Style Guide

This guide is intended to be a short overview of The PowerShell Best Practices and Style Guide.

Author: David Wettstein
Version: v1.2.0, 2020-12-01

If you rather search a cheat sheet, I recommend the PDF from Warren Frame: PowerShell Cheat Sheet
(Source: PowerShell Cheat Sheet, all credits go to Warren Frame.)

Table of Contents

Style Guide

Code Layout and Formatting

- Capitalization Conventions
  - Use *PascalCase* or *camelCase* for all public identifiers: module names, function or cmdlet names, class, enum, and attribute names, public fields or properties, global variables and constants, parameters etc.

    ```
    [String] $MyVariable = "A typed variable."
    ```

  - PowerShell language keywords are written in *lowercase* (e.g. `if`, `foreach`), as well as operators such as `-eq` and `-match`.

    ```
    if ($MyVariable -eq "AnyValue") {
        # ...
    ```

  - Keywords in comment-based help are written in *UPPERCASE* (e.g. `.SYNOPSIS`).

    ```
    <#
    .SYNOPSIS
        A short description...
    #>
    ```

  - Function names should follow PowerShell's *Verb-Noun* naming conventions, using *PascalCase* within both Verb and Noun (list allowed verbs with the cmdlet `Get-Verb`).

    ```
    function Invoke-HttpRequest {
        # ...
    ```

- Start all scripts or functions with `[CmdletBinding()]`.

  ```
  function Invoke-HttpRequest {
      [CmdletBinding()]
      param (
          # ...
  ```

- Follow either the *Stroustrup* or the *One True Brace Style (1TBS or OTBS)* brace style. Write the opening brace always on the same line and the closing brace on a new line!

  ```
  function Invoke-HttpRequest {
      # ...
      end {
          if ($MyVariable -eq "AnyValue") {
              # either Stroustrup:
          }
          elseif {
              # or 1TBS:
          } else {
              # ...
          }
      }
  }
  ```

- Indentation and line handling
  - Use four spaces per indentation level (and never tabs!).
  - Lines should not have trailing whitespace.
  - Limit lines to 115 characters when possible, but avoid backticks.
  - Surround function and class definitions with two blank lines (similar to Python).
  - Method definitions within a class are surrounded by a single blank line (similar to Python).
  - End each file with a single blank line.
  - I recommend using EditorConfig plugin to automatically set indentation levels and trim trailing whitespaces. Example config (create new file `.editorconfig` in root folder):

    ```
    root = true

    [*]
    indent_style = space
    indent_size = 4
    charset = utf-8
    trim_trailing_whitespace = true
    insert_final_newline = true
    ```

- Put spaces around keywords, operators and special characters

  ```
  # Bad
  if($MyVariable-eq"AnyValue"){
      $Index=$MyVariable.Length+1
  }

  # Good
  if ($MyVariable -eq "AnyValue") {
      $Index = $MyVariable.Length + 1
  }
  ```

- Avoid using semicolons `;` as line terminators
  - PowerShell will not complain about extra semicolons, but they are unnecessary.
  - When declaring hashtables, which extend past one line, put each element on it's own line.

Read the full page Code Layout and Formatting for more information.

Function Structure

- When declaring simple functions leave a space between the function name and the parameters.
- For advanced functions follow PowerShell's *Verb-Noun* naming conventions as mentioned in section Code Layout and Formatting.
- Avoid using the `return` keyword in your functions. Just place the object variable on its own.

  ```
  function MyAddition ($Number1, $Number2) {
      $Result = $Number1 + $Number2
      $Result
  }
  ```

- When using blocks, return objects inside the `process` block and not in `begin` or `end`.
- When using parameters from pipeline use at least a `process` block.
- Specify an `OutputType` attribute if the advanced function returns an object or collection of objects.

  ```
  function Invoke-HttpRequest {
      [CmdletBinding()]
      [OutputType([PSObject])]
      param(
          # ...
  ```

- For validating function or script parameters, use validation attributes (e.g. `ValidateNotNullOrEmpty` or `ValidatePattern`)

  ```
  function Invoke-HttpRequest {
      [CmdletBinding()]
      param (
          [Parameter(Mandatory = $true, ValueFromPipeline = $true, Position = 0)]
          [ValidatePattern("^http(s)?.*")]
          [String] $Url
      )
      # ...
  ```

Read the full page Function Structure for more information.

Documentation and Comments

- Indent comments to the same level as the corresponding code.
- Each comment line should start with a # and a single space.
- Keep comments up-to-date when code changes.

- Write comments in English and as complete sentences.
- Inline comments should be separated with two spaces.

```
$MyVariable = $null  # Declare variable
```

- Comments should serve to your reasoning and decision-making, not attempt to explain what a command does.

```
# Bad
# Decrease length by one
$LastIndex = $MyVariable.Length - 1

# Good
# Indices usually start at 0, thus subtract one to access the last char.
$LastIndex = $MyVariable.Length - 1
```

- When documenting functions, place the comment inside the function at the top, instead of above.
- Provide usage examples when documenting functions (`.EXAMPLE`)

Read the full page Documentation and Comments for more information.

Naming Conventions

- Use the full name of each command instead of aliases or short forms.
- Additionally, use full parameter names.

```
# Bad
gps Explorer

# Good
Get-Process -Name Explorer
```

- Use full, explicit paths when possible (avoid `..` or `.`).

```
# Bad
$Result = & ".\Invoke-AnotherScript.ps1" -Param1 "Value"

# Good
$Result = & "$PSScriptRoot\Invoke-AnotherScript.ps1" -Param1 "Value"
```

- Avoid using ~, instead use `$env:USERPROFILE` or `$HOME`.

Read the full page Naming Conventions for more information.

Tips and Common Pitfalls

- When creating a new file, ensure that the encoding is *UTF-8*. Be aware that PowerShell creates files with encoding *UTF-8 with BOM* by default. See also What's the difference between UTF-8 and UTF-8 without BOM? and Using PowerShell to write a file in UTF-8 without the BOM.

```
# Attention: Out-File writes UTF8 with BOM!
Write-Output $Text | Out-File -Encoding UTF8 -FilePath $Path -Append
# As an alternative use:
[System.IO.File]::AppendAllText($Path, "$Text`n")
```

- See `Get-Verb` to list all approved PowerShell verbs, always use one of them for your functions.
- `$ErrorActionPreference` is set to `Continue` by default, thus the script continues to run even after an error happens. Additionally, `$WarningActionPreference` is set to `Continue` as well, and thus unwanted output might be in the standard output of your script. Add the following code at the beginning of your script to change the default behavior:

```
if (-not $PSCmdlet.MyInvocation.BoundParameters.ErrorAction) { $ErrorActionPreference = "Stop" }
if (-not $PSCmdlet.MyInvocation.BoundParameters.WarningAction) { $WarningPreference = "SilentlyContinue" }
```

- Even when you add `-ErrorAction:SilentlyContinue`, the errors are written to the error stream (but not displayed). To ignore all error output use `-ErrorAction:Ignore` or stream redirection and append `2>$null`.
- The results of each statement are added to the output stream and thus returned as script output, not only the ones containing the `return` keyword.
  - Use `$null = {{statement}}` or `{{statement}} | Out-Null`, to ignore certain results.
- When invoking other scripts within your script, use explicit not relative paths.
  - Since PowerShell 3.0 you can get the directory of the current script with `$PSScriptRoot`.
- Avoid blocking/waiting statements, like `Read-Host` or `Get-Credential`, in scripts that are run as child processes or in background.
- When using `ConvertTo-Json` be aware that the default value for the parameter `-Depth` is only 2.

> *-Depth [<Int32>]*
> *Specifies how many levels of contained objects are included in the JSON representation. The default value is 2.*

- Additionally `ConvertTo-Json` replaces special characters with *Unicode* escape characters. See also ConvertTo-Json problem containing special characters. If you need the unescaped characters, use the following code:

```
$UnescapedJson = [Regex]::Unescape($EscapedJson)
```

- For returning/printing a string with a colon (`:`) or text right after a variable, use `"${MyVariable}"` instead of `"$MyVariable"`.

```
# Error
Write-Verbose "$Result: $env:USERNAME_Will_Be_A_Pro"

# Good
Write-Verbose "${Result}: ${env:USERNAME}_Will_Be_A_Pro"
```

- Since PowerShell is a dynamically typed language, it won't return an empty array or an array with only one item, but either `$null` or the item on its own. See also PowerShell doesn't return an empty array as an array. To return such an array, use the following code:

```
if ($Result.Length -le 1) {
    , $Result  # Force PowerShell to return $Result as array.
} else {
    $Result  # $Result is an array anyway.
}
```

- When using a `exit $ExitCode` statement in a script or function, be aware that if you dot-source and execute it (also when importing as module), the current console will be exited. To set an exit code, but not close the console, use the following code:

```
# Set the script/function exit code. Can be accessed with `$LASTEXITCODE` automatic variable.
& "powershell.exe" "-NoLogo" "-NoProfile" "-NonInteractive" "-Command" "exit $ExitCode"

Write-Host $LASTEXITCODE
```