**Powershell Coding Standard**

**Last Modified: May 13, 2016 by Kris Sherrerd**

**5-13-2016 Initial creation**

***Word to the wise:*** *Windows in general is NOT case sensitive, so neither is PowerShell. With that in mind the format are for readability only.*

**Names**:

**Class Names**

* Use a Specific name that makes sense for a Class Name
* Use Pascal Case for Class Names
* Shall not use all caps words
* Shall have no underscores or dashes ('\_')

Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough.

Suffixes are usually helpful. For example, if your system uses agents then naming something DownloadAgent conveys real information.

Example:

class GetLinkList

{

}

**Cmdlet & Function Name**

***A quick note:*** *A True cmdlet is written and compiled, usually in C#, however we will simulate Cmdlets through the use of functions and for the purposes of this document they are one and the same.*

* Use a Specific Verb-Noun that makes sense for a function Name
* Use useful prefixes for function

Verb-Nouns used in function naming need to be very specific so that the user can discover your functions. Prefix generic nouns such as "server" with a shortened version of the product name. For example, if a noun refers to a server that is running an instance of Microsoft SQL Server, use a noun such as "SQLServer". The combination of specific nouns and the short list of approved verbs enable the user to quickly discover and anticipate functionality while avoiding duplication among function names.

* function name should be singular
* Use Pascal Case for function Names
* Use '-' as a Verb-Noun Separator

For example, use the name **Get-Process** instead of **Get-Processes**. It is best to follow this rule for all function names, even when a function is likely to act upon more than one item. Use Pascal case for parameter names. In other words, capitalize the first letter of verb and all terms used in the noun.

Example:

function Clear-ItemProperty(){  
}

Variable Names

* Use lower case for variable names
* Make use of Microsoft Standard Parameter Names when it makes sense
* Must use 'f\_' at the beginning of the name if it contains a function
* Use names that can be associated with what the variable will hold.

These should be descriptive, a noun if at all possible to describe what the variable is.

Please see <https://msdn.microsoft.com/en-us/library/dd878352%28v=vs.85%29.aspx> for the Microsoft standard names or see <https://github.com/stimepy/smallprgs/blob/master/Documents/Standard%20Cmdlet%20Parameter%20Names%20and%20Types.pdf> (Temporary Holding area!)

Example:

$alarmtime = ’12:00’

$f\_sedclock = cat DATA.TXT | % { $\_ -replace “6:00”, $alarmtime }

**Class Variable Names**

* 'my\_' should be in front of any variables.
* Use lowercase for variable names
* Use names that can be associated with what the variable will hold

These names should have the word 'my\_' in front of them, in order to distinguish that they are a part of the actual class when it is created. They also need to describe what information they will be holding if any. These are NOT to be used as a temp variable in any function.

Example:

$my\_alarmtime = ’12:00’

**Cmdlet & Function Argument Names**

* Use lower case for argument names
* Use Standard Parameter Names when possible (See list above)
* Use names that can be associated with what the variable will hold

Should no standard parameter name be available, use the above to differentiate between a function argument variable and other variables.

**Global Variables**

* Should be avoided.
* Must follow variable naming formats
* Global variables should be pre-pended with a 'g\_'

**Braces *{}* Policy**

Of the three major brace placement strategies one of the two are acceptable methods must be used, with the first one listed being preferable:

If ($condition)

{

while ($condition)

{

...

}

}

• Traditional Unix policy of placing the initial brace on the same line as the keyword and the trailing

brace inline on its own line with the keyword:

if ($condition) {

while ($condition){

...

}

}

Either form is acceptable, many people, however, find the first form more pleasant. Why is the topic of many psychological studies. As long as the beginning and ending braces can be identified, any form listed here will work.

**Indentation/Tabs/Space Policy**

* Indent using 4 spaces for each level. (Standard TAB on Most IDEs!)
* Do not use a tradition tab of 5 spaces.
* Indent as much as needed, but no more. There are no arbitrary rules as to the maximum indenting level.
* If the indenting level is more than 4 levels you may need to think about factoring out code.

When people using different tab settings the code is impossible to read or print, which is why spaces are preferable to tabs however as stated most IDE applications use 4 spaces as the default for tabs.

As much as people would like to limit the maximum indentation levels it never seems to work in general. We'll trust that programmers will choose wisely how deep to nest code allowing the code to still be readable.

**Constructors**

Do Not do Real Work in Object Constructors! Must not do any real work in an object's constructor. Inside a constructor initialize variables only and/or do only actions that can't fail. If needed create another method(ie Open()) for an object which completes construction. This method should be called after object instantiation.

***Example***

class Device

{

function Device() { <# initialize and other stuff #> }

function Open() { return ‘FAIL’ }

}

$dev =[Device]::New()

if ( FAIL == $dev.Open() ) {

return false

}

**Switch Formatting**

* Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
* The default case should always be present and trigger an error if it should not be reached, yet is reached.
* If you need to create variables put all the code in a block

**Example**

switch -regex ($a)

{

{($\_ -eq "[a-d]") -or ($\_ -eq "[e-g]")}// fall through because…

{"The color is blue."}

"[h-k]" {"The color is green."}

"[l-o]" {"The color is yellow."}

"[p-s]" {"The color is orange."}

"[t-v]" {"The color is purple."}

"[w-y]" {"The color is pink."}

"[z]" {"The color is brown."}

default {"The color could not be determined."}

}

**Use of *continue, and break***

Continue and break like goto should be used sparingly(or never) as they are magic in code. With a simple spell the reader is beamed to god knows where for some usually undocumented reason. The two main problems with continue are:

* It may bypass the test condition
* It may bypass the increment/decrement expression

Consider the following example where both problems occur:

while (TRUE)

{

...

// A lot of code

...

if (/\* some condition \*/) {

$continue;

}

...

// A lot of code

...

if ( $i++ > STOP\_VALUE) $break;

}

***Note:*** *"A lot of code" is necessary in order that the problem cannot be caught easily by the programmer.*

*From the above example, a further rule may be given: Mixing continue with break in the same loop is a sure way to disaster.*

**One Statement Per Line**

There should be only one statement per line unless the statements are very closely related.

**Short Functions**

Functions must limit themselves to a single page of code.

The idea is that the each function represents a technique for achieving a single objective. Most arguments of inefficiency turn out to be false in the long run. True function calls are slower than not, but there needs to a thought out decision. The longer the function the more likely that it's doing more than needed

**Document Null Statements**

Always document a null body statement so that it is clear that the null body is intentional and not missing code.

**Usually Avoid Embedded Assignments**

There is a time and a place for embedded assignment statements. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable.

while ($a -nq ($c = getchar ) )

{

process the character

}

The ++ and -- operators count as assignment statements. So, for many purposes, do functions with side

effects. Using embedded assignment statements to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example:

$a = $b + $c;

$d = $a + $r;

should not be replaced by

$d = ($a = $b + $c) + $r;

even though the latter may save one cycle. In the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade.

**Reusing Your Hard Work and the Hard Work of Others**

Reuse across projects is almost impossible without a common framework in place. Objects conform to the

services available to them. Different projects have different service environments making object reuse

difficult.

Developing a common framework takes a lot of up front design effort. When this effort is not made, for

whatever reasons, there are several techniques one can use to encourage reuse:

* Small Libraries
* Keep a repository of Code

**Comments on Comments:**

**Comments Should Tell a Story**

Consider your comments a story describing the system. Expect your comments to be extracted by a robot and formed into a man page. Class comments are one part of the story, function signature comments are another part of the story, function arguments another part, and function implementation yet another part. All these parts should weave together and inform someone else at another point of time just exactly what you did and why.

**Use Headers for functions**

Use something to describe the function and what it does. This way people reading over your code can quickly look and see exactly what it should be doing these headers are structured in such a way as they can be parsed and extracted. They are not useless like

normal headers. So take time to fill them out. If you do it right once no more documentation may be necessary.

**Example:**

<#

.SYNOPSIS

<A brief description of the script>

.DESCRIPTION

<A detailed description of the script>

.PARAMETER <paramName>

<Description of script parameter>

.EXAMPLE

<An example of using the script>

#>

**Document Decisions**

Comments should document decisions. At every point where you had a choice of what to do place a comment describing which choice you made and why. Archeologists will find this the most useful information.

**Make Gotchas Explicit**

Explicitly comment variables changed out of the normal control flow or other code likely to break during

maintenance. Embedded keywords are used to point out issues and potential problems. Consider a robot will parse your comments looking for keywords, stripping them out, and making a report so people can make a special effort where needed.

* Make the gotcha keyword the first symbol in the comment. - Comments may consist of multiple lines, but the first line should be a self-containing, meaningful summary.
* The writer's name and the date of the remark should be part of the comment. - As so often gotchas stick around longer than they should the Embedding date and author information allows other programmer to make this decision as to what to do and whom to ask information of.

**Gotcha Keywords**

* **:TODO: topic -** Means there's more to do here, don't forget.
* **:BUG: [bugid] topic -** means there's a Known bug here, explain it and optionally give a bug ID.
* **:KLUDGE: -** When you've done something ugly say so and explain how you would do it differently next time if you had more time.
* **:TRICKY: -** Tells somebody that the following code is very tricky so don't go changing it without thinking.
* **:WARNING: -** Beware of something.
* **:ATTRIBUTE: value -** The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.
* **:DEPRECATED: -** Code that may be active but is going away at some point.

***Word to the wise:*** *Comments are nice, they can point out bugs, todos and complexity issues, but there is a point where they could be too much or worse, the comments could be wrong! This could be due to refactoring or overzealous commenting or code being updated. In a perfect world people would update comments but they don't. Keep comments simple, and keep your code clear so comments can stick to only complex items!*

**File Documentation**

Every file should have a overall header comment that covers:

* the purpose of the file and what it contains
* direct people to related resources:
  + directories of source
  + online documentation
  + paper documentation
  + design documentation
* anything else that might help someone

In addition depending on the project size it may also be useful to have one or more ReadMe.txt files(the more would mean that each directory should have one) that cover the following:

* the purpose of the directory and what it contains
* direct people to related resources:
  + directories of source
  + online documentation
  + paper documentation
  + design documentation
* anything else that might help someone

Consider a new person coming in 6 months after every original person on a project has gone. That lone scared explorer should be able to piece together a picture of the whole project by traversing a source directory tree and reading README files, and source file headers.

**Test Till you break it, then FIX IT!**

Testing is important, by making it is part of the standard coding it will ensure we have better code.

* Write tests before you write code
* Run the tests whenever you have modified the code.
* Write 1 test package per class, 1 test per function and keep it with the class so that when you code is modified the test can be run to ensure everything still works!

**Code Reviews**

Code reviews can be very useful. Unfortunately they often degrade into nit picking sessions and endless arguments about silly things. They also tend to take a lot of people's time for a questionable payback.

My god he's questioning code reviews, he's not an engineer!

Not really, it's the form of code reviews and how they fit into normally late chaotic projects is what is being questioned.

First, code reviews are **way too late** to do much of anything useful. What needs reviewing are requirements and design. This is where you will get more bang for the buck.

There a few methods of code review:

* First get all relevant people in a room. Lock them in. Go over the class design and requirements until the former is good and the latter is being met. Having all the relevant people in the room makes this process a deep fruitful one as questions can be immediately answered and issues immediately explored. Usually only a couple of such meetings are necessary.

If the above process is done well coding will take care of itself. If you find problems in the code review the best you can usually do is a rewrite after someone has sunk a ton of time and effort into making the code "work."

* Second is to do the review with just a couple people you trust. Read the code in question and simply make comments to the programmer. Then the programmer and reviewers can discuss issues and work them out.

Each approach has its advantages and disadvantages, though I would recommend the second approach as it can be less time consuming and is less of a problem to align every ones schedules.

***One thing of note***: Even with this document as your ultimate standard it is possible that some things just won't be seen the same way. Keep in the following in mind when doing a code review for someone (or getting one done by someone else) ***“Don't sweat the small stuff”***. If the code works, is readable, and is to the best of your (and their) knowledge correct, be happy!

**Create a Source Code Control System Early and Not Often**

A common build system and source code control system should be put in place as early as possible in a

project's lifecycle, preferably before anyone starts coding. Source code control is the structural glue binding a project together. If programmers can't easily use each other's products then you'll never be able to make a good reproducible build and peoples time will be wasted. It's also hard converting rogue build

environments to a standard system. With that in mind:

* Make simple things simple. It should be dead simple and well documented on how to
  + check out modules to build
  + how to change files
  + how to add new modules into the system
  + how to delete modules and files
  + how to check in changes
  + what are the available libraries and include files
  + how to get the build environment including all compilers and other tools
* On check ins’ log comments should be useful.

**Create a Bug Tracking System Early and Not Often**

The earlier people get used to using a bug tracking system the better. If you are 3/4 through a project and

then install a bug tracking system it won't be used. You need to install a bug tracking system early so people will use it.

Bug tracking, when used correctly it can really help a project:

* Problems aren't dropped on the floor.
* Problems are automatically routed to responsible individuals.
* The lifecycle of a problem is tracked so people can argue back and forth with good information.
* Managers can make the big schedule and staffing decisions based on the number of and types of bugs in the system.
* Configuration management has a hope of matching patches back to the problems they fix.
* QA and technical support have a communication medium with developers.

Source code control should be linked to the bug tracking system. During the part of a project where source is frozen before a release only checkins accompanied by a valid bug ID should be accepted. And when code is changed to fix a bug the bug ID should be included in the checkin comments.

**No Magic Numbers**

A magic number is a bare-naked number used in source code. It's magic because no-one has a clue what it

means including the author inside 3 months. For example:

if (22 -eq $foo) { start\_thermo\_nuclear\_war –Process 42 }

elseif (19 -eq $foo) { refund\_lotso\_money –Path $bankfolder }

elseif (16 -eq $foo) { infinite\_loop }

else { Get-DontKnowWhat }

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know?

Instead of magic numbers use a real name that means something. You should use a variable. For example:

$presidentwentcrazy = 22

$wegoofed = 19

$theydidntpay = 16

$universe = 42

if ($presidentwentcrazy -eq $foo) { start\_thermo\_nuclear\_war –process $universe }

else if ($wegoofed -eq $foo) { refund\_lotso\_money –Path $bankfolder }

else if ($theydidntpay -eq $foo) { infinite\_loop }

else { Get-KnowWhat }

Now isn't that better?

**Refactor Refactor Refactor!!!**

Ok, now on to reality, If you have used this standard 100% and you know your code is the best in the

universe, there may still be “bad smells” in it. If you have not refactored at all in your project chances are

you have lots of “bad smells”, some may be an issue of taste (nothing wrong with them, just not pretty code), some of them may actually be things that can be made better. The personal taste we can do nothing about, but the other, we can refactor it till it shines!