

CNT 4603: System Administration Fall 2013

Scripting – Windows PowerShell – Part 4

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 4078-823-2790
<http://www.cs.ucf.edu/courses/cnt4603/fall2013>

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida



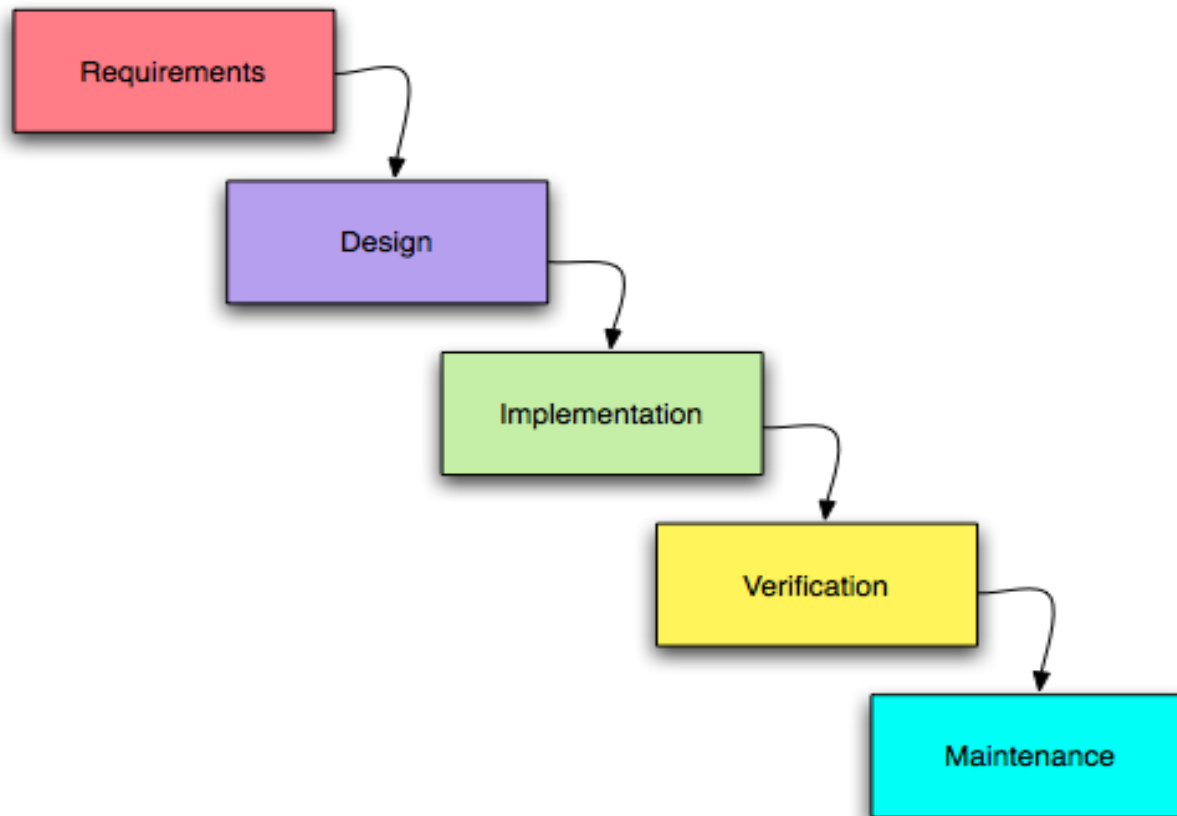
Script Development In PowerShell

- While we are currently focusing on PowerShell, much of what is covered in this section of notes pertains to script development in any language or environment.
- A structured approach to scripting can be applied to a wide variety of coding and development projects. Although not every scripting project requires the same level of attention and detail, any scripts that are developed for use in a production environment will benefit tremendously from applying a clear, well-documented, and supportable scripting methodology.
- As with any software development project, you should choose a development life cycle model that fits the needs of your project.



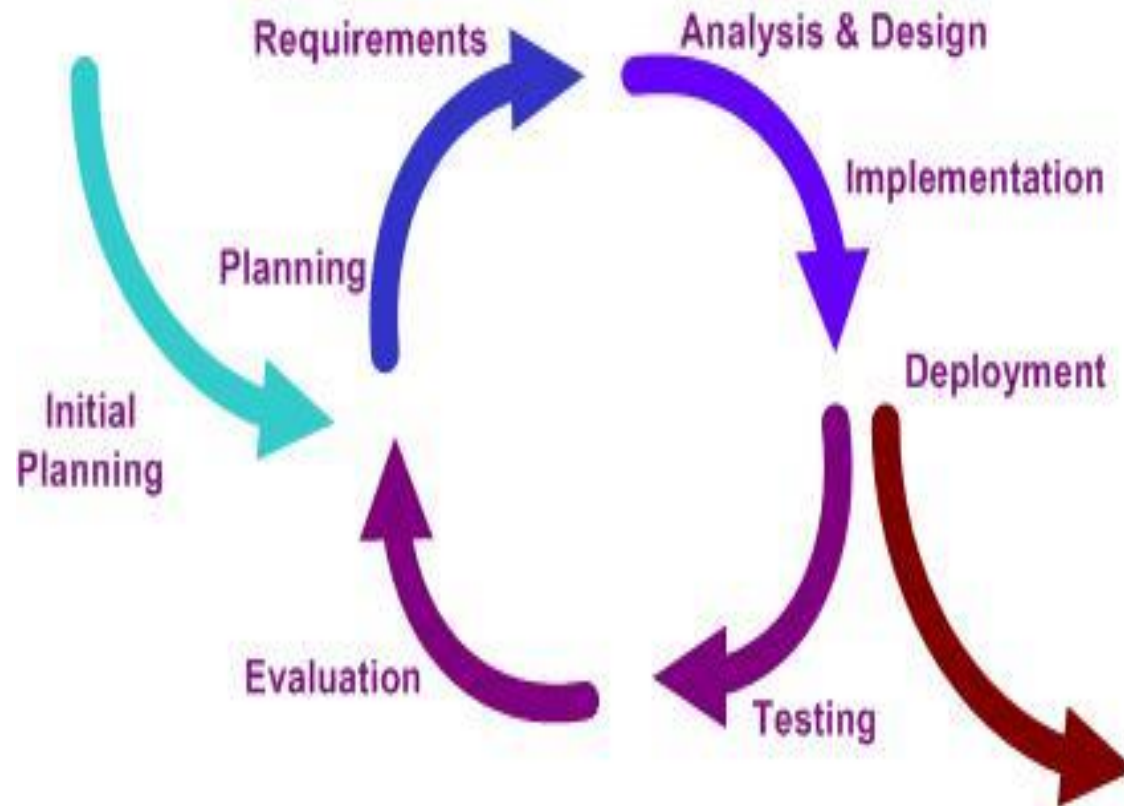
Script Development In PowerShell

- The Waterfall model is a traditional approach to the software development life cycle.



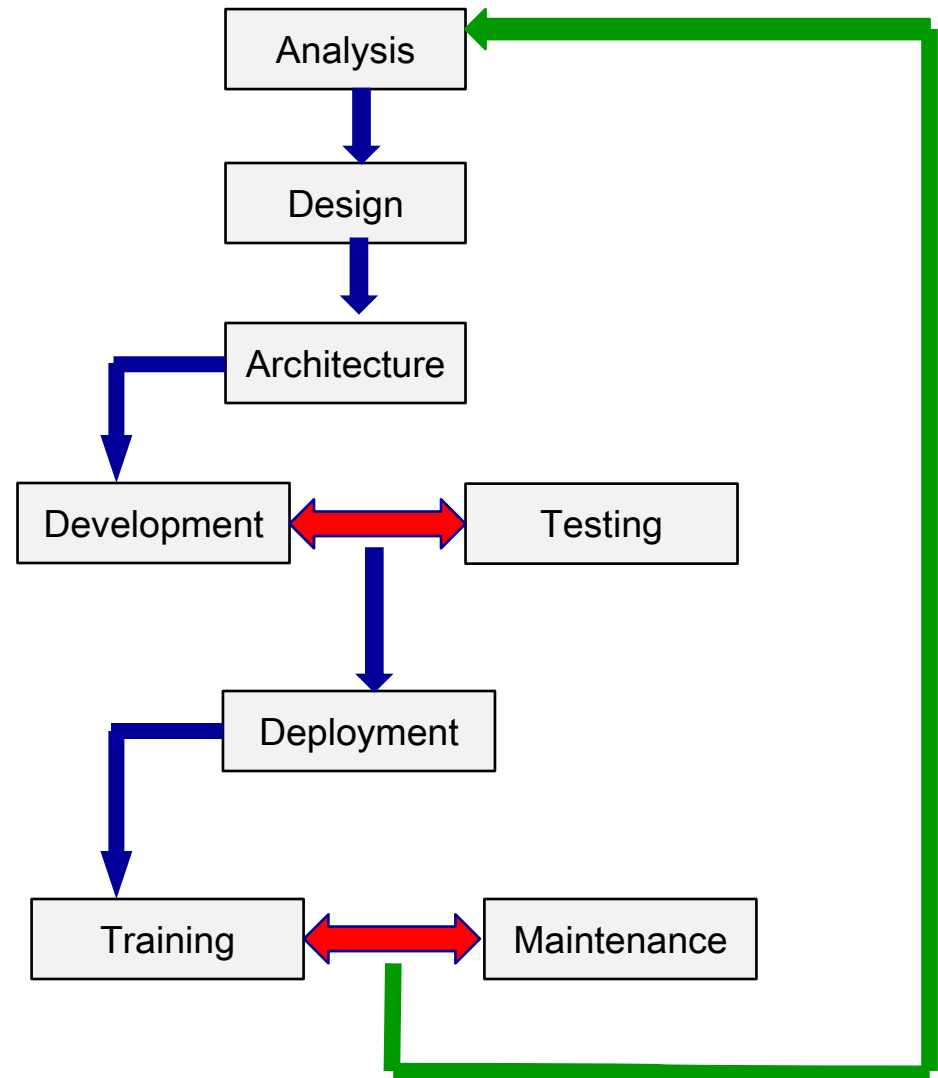
Script Development In PowerShell

- The Iterative model is a more current model used for the software development life cycle.



Script Development In PowerShell

- A more general process map for a generic scripting project. Although similar to a full development life cycle model, the steps are simply pointers to tasks that need to be completed in a typical scripting project.
- You can choose to follow this model or develop your own, but choose some method for managing your project.



Design And Prototype Scripts Using Pseudocode

- Often scripts are designed and prototyped using pseudocode. This technique allows you to develop the structure and logic of a script before writing any code.
- Working out the structure and logic beforehand helps you ensure that your script will meet its requirements and allows you to detect possible logic flaws early in the process.
- Pseudocode is language independent and can be written so that other people, especially those who need to provide input on the script design, can read and understand it easily.
- The example on the next page illustrates a pseudocode mock-up for a script.



Script Development In PowerShell

Param domain

Param resource account CSV file

Bind to domain

Open and read CSV file

For each resource account in CSV file:

- Create a new account in the specified OU.
- Set the password (randomly generate complex 14-character password).
- Log password to administrative password archive.
- Set the user account attributes based on CSV file information.
- Mail-enable the new account.
- Add the user to the appropriate groups based on CSV file information.

Next

Pseudocode Representation Of A Script



Gather Script Requirements Effectively

- As with any project, you need to define the problem your script will be addressing to determine what is required of the script.
- Sometimes a script is solving a simple automation requirement, so the requirements will be fairly easy to determine.
- When a script is solving a more complex business/enterprise automation need, you might need to learn more about the business/enterprise processes being automated to determine the requirements that the script must meet.
- In any case, to ensure the success of your scripting project, you must identify the requirements for the script and have all parties sign off on those requirements.
- Overlooking these steps in the development process might mean that your final script fails to meet its requirements and is then rejected as a solution for the original business/enterprise need.



Don't Develop Scripts In A Production Environment

- Most scripts are designed to make changes to a system, so there is always a chance that running a script in a production environment could have unwanted or potentially damaging results.
- Even if a script makes no changes, it may still have an undesirable effect, or you might not fully understand its impact.
- Even worse, when the script is run to test its functionality, you might accidentally run the script outside your designated testing scope and affect the production systems.
- Scripts can be very powerful and fast-acting tools that can be exceptionally helpful to a system administrator, but with this power must come knowledge and responsibility. Until you fully understand exactly what a script will do, it should only be executed in an isolated environment. This is another application in which virtualization plays a major role.



Test, Test, and More Testing

- Scripts are typically written to perform some automation task. An example might be to modify an attribute on every user in an Active Directory domain.
- The automation task might carry a high or low level of impact, but some form of quality assurance testing should be conducted on the script before it is ever run in a production environment.
- Scripts in particular should be thoroughly tested because of their potential effect on an environment.
- Scripts which only read information are generally quite safe to run. For example, a script that creates a list of Active Directory users attributes but does not write any information to AD has a very low probability of causing any problems.



Test, Test, and More Testing

- On the other hand, whenever a script makes modifications to objects, it is important to understand what changes are being made, and to provide a rollback strategy whenever possible.
- For example, if you are writing a script to modify the description field of every user object in Active Directory, you could use the CSVDE utility (Comma Separated Value Directory Exchange, see <http://social.technet.microsoft.com/wiki/contents/articles/comma-separated-value-directory-exchange-csvde-utility.aspx>) to export the existing user description fields to a file prior to running the script. This way, if the script produced some unexpected results, you could review the export file for the differences, or even re-import the original description fields if necessary.



Develop Professional Level Scripts

- Many scripters tend to view scripting as a quick and easy way to complete tasks and don't see the need for professional considerations, such as planning, documentation, and standards.
- This mindset may be a holdover from the days when scripting was considered a clandestine task reserved for Unix and Linux gurus. Scripting languages, like PowerShell, are changing this mindset and are leading the way for how system administrators manage their environments. This is changing rapidly how scripting is viewed to where it is now considered an essential solution to business/enterprise automation needs and therefore, a task must be done with professionalism.



Develop Professional Level Scripts

- To be professional when developing scripts, you should make sure that your work meets a certain level of quality by developing standards for all your scripts to meet: writing clear and concise documentation, following best practices in planning and layout, testing thoroughly, and so forth.
- Adhering to professional standards can also ensure that others accept your work more readily and consider it more valuable.
- The next few pages will examine more closely some best practices that should be applied to script design.



Script Design: Configuration Information

- When setting variables, parameters, and so on that control script configuration, you should always place them near the beginning of the script to make locating these items easy for anyone using, reading, or editing the script.
- This practice will help to reduce the number of errors introduced when editing the script configuration. If configuration information is spread throughout the script, it is much more likely to be misconfigured, declared multiple times, or simply forgotten.
- The template on the next page illustrates a best practice format for script configuration information.



Script Design: Configuration Information

```
#-----  
# Set variables  
#-----  
$Owner = "Administrators"  
$Targets = import-csv $ImportFile  
...  
  
#-----  
# Script Body  
#-----  
...
```

Template For Script Configuration Information Location



Script Design: Use Comments

- Most programmers and scripters groan when they think about commenting. It is however, an extremely important and often neglected aspect of professional coding.
- You can't assume that other users, reviewers, editors, etc., of your scripts will understand the logic you've employed in a script or be familiar with the methods you used to perform various tasks. Therefore, using comments to assist users in understanding your script is not only good practice, but often vital.
- Comments don't always have to be extensive, but should provide enough information to help users see how the logic flows.
- If your script contains a complex method, class, or function, adding a comment to explain what it does is very useful.



Script Design: Use Comments

- While commenting is often viewed from the point of others who use your scripts/code, it is also very helpful from your own viewpoint.
- Often you will be the one to update or review your own scripts and the comments will be most helpful to you when you review a script that you created 6 months or a year ago.
- The following page illustrates a common commenting style applied to scripts that most readers/reviewers find helpful and informative. This one illustrates the comments for a method and the variables defined within the method.



Script Design: Use Comments

```
#-----  
# ADD – DACL method  
#-----  
# Usage: Grants right to a folder or file.  
#  
# $Object: The directory or file path. (Ex. "c:\myfolder" or "c:\myfile.txt")  
#  
# $Identity: User or Group name. (Ex. "Administrators" or "mydomain\user1")  
#  
# $AccessMask: The access right to use when creating the access rule.  
#               (Ex. "FullControl", "ReadandExecute", "Write", etc.)  
#  
# $Type: Allow or deny access. (Ex. "Allow" or "Deny")  
#
```

Commenting Style For Scripting



Script Design: Avoid Hard-Coding Configuration Info.

- Hard-coding configuration information is a commonly made mistake. Instead of asking the user to supply the required information, the configuration information is hard-coded into variables or randomly scattered throughout the script.
- Hard-coding requires users to manually edit scripts to set the configuration information, which increases the risk of mistakes that will result in errors when the script executes.
- Part of your goal as a script designer is to provide usable scripts; hard-coding information makes using a script in a different environment difficult.
- Instead, use parameters or configuration files, as shown on the next page.





PS-Part4-p19.ps1

```
1 #-----
2 # PS-Part4-p19
3 # test script - PowerShell Notes - Part 4 Page 19
4 #
5 #-----
6
7 #-----
8 # Configuration Information
9 #-----
10 param([string] $SearchPath=$(throw "Please specify the search path:"))
11
12 #-----
13 # Script Body
14 #-----
15 get-childitem $SearchPath
16 #-----
17 # END SCRIPT
18 #-----
19
```

```
PS C:\users\Administrator\MyScripts> .\PS-Part4-p19.ps1
Please specify the search path:
At C:\users\Administrator\MyScripts\PS-Part4-p19.ps1:10 char:35
+ param([string] $SearchPath=$(throw <<<< "Please specify the search path:"))
+ CategoryInfo          : OperationStopped: (Please specify the search path::String) [], Runtime
+ FullyQualifiedErrorId : Please specify the search path:
```

```
PS C:\users\Administrator\MyScripts> .\PS-Part4-p19.ps1 .

Directory: C:\users\Administrator\MyScripts
```

Mode	LastWriteTime	Length	Name
-a---	10/26/2011 10:08 AM	138	ArrayScript.ps1
-a---	10/26/2011 11:23 AM	138	ArrayScript2.ps1
-a---	10/26/2011 11:42 AM	150	ArrayScript3.ps1
-a---	10/26/2011 11:23 AM	138	ArrayScript4.ps1
-a---	10/31/2011 12:59 PM	408	ListStoppedServices.ps1
-a---	11/2/2011 9:45 AM	683	PS-Part4-p19.ps1
-a---	10/31/2011 3:09 PM	96	server.csv

```
PS C:\users\Administrator\MyScripts>
```

User forgets to enter the search path. Error message generated informing them to supply the search path.

User supplies the search path...all is good!



Script Design: Avoid Hard-Coding Configuration Info.

- Occasionally, it is necessary to hard-code configuration information into a script.
- When this is necessary, use the convention of representing the information in the form of a variable.
- Define the configuration information in a variable in one place (as before, near the top of the script), rather than actually hard-coding it into several different places scattered throughout the script. This will decrease the chances of introducing errors when/if the information needs to be changed.
- Representing the information in a variable at the top of the script also decreases the time required to reconfigure the script for a different environment.



Script Design: Provide Instructions

- Most scripts you develop will be written for use by others, such as other system administrators, or customers. Customers in particular, will not be comfortable with code or CLI.
- Remember that your scripts must be usable and useful. If you don't provide instructions to make sure a novice can run the script and understand what it does, you haven't succeeded as a scripter.
- It's pretty common to see scripts without any instructions, or incorrect instructions, or provide little if any explanation of what the script does. For a user, this type of script can be quite frustrating. Even worse, the user may have no clue as to the impact the script might have on their environment and running it could lead to disaster.
- The example on the next page illustrates instructions that might be in a readme file or at the top of the script to explain a script's purpose and how it works.



Script Design: Provide Instructions

```
#-----  
# Script Information  
#-----  
# Name: AddProxyAddress.ps1  
# Author: Mark Llewellyn  
# Date: October 15, 2013  
#  
# Description:  
# Use this script to add secondary proxy addresses to users based on a CSV import file. When trying to  
# add the additional proxy addresses, this script checks the following conditions:  
#  
#         Does the user exist?  
#         Is the user mail-enabled?  
#         Does the proxy address currently exist?  
#  
# This script creates a log file each time it is run.  
# CSV file format:  
# [sAMAccountName],[ProxyAddresses]  
# mark, mark@savn.local;support@savn.local  
# kristy, kristy@savn.local  
#  
# The ProxyAddress column is ; delimited for more than one proxy address.  
#-----
```



Script Design: Perform Validity Checking

- Failing to perform validity checks on required parameters is another common mistake. If your script requires input from the user, neglecting these validity checks might mean users enter the wrong input and the script halts with an error.
- This sort of oversight might not be a major issue with small scripts, but with large, complex scripts, it can seriously affect their usability.
- To illustrate this consider the scenario on the following page.
- To prevent this problem, make sure you perform validity checking on required parameters as shown in the example on page 27.

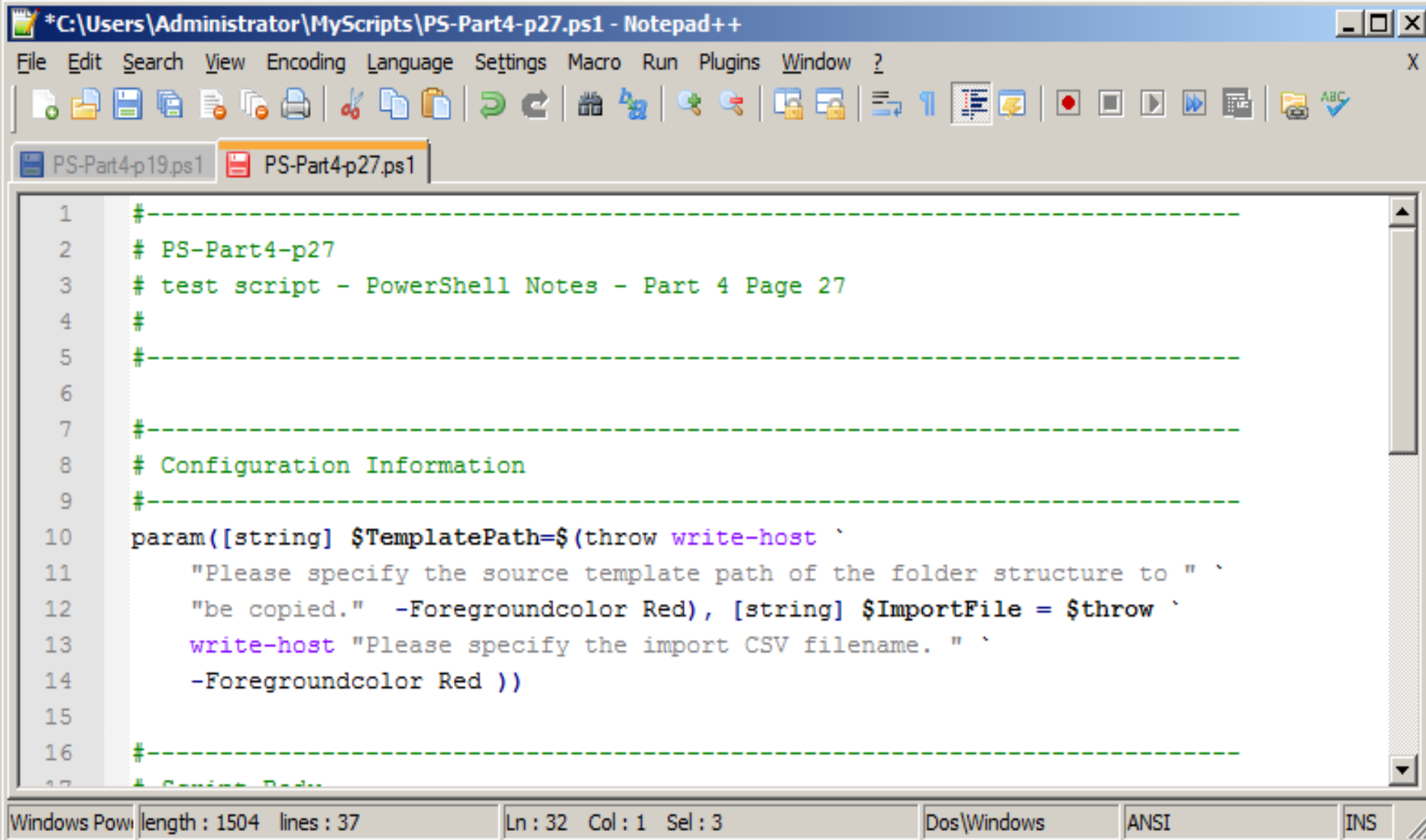


Script Design: Perform Validity Checking

- Let's say you wrote a script that performs a software inventory. In your development environment consisting of a few machines, you run the script, but fail to provide the correct information for a required parameter. The script runs, and a couple of seconds later, it fails. You realize that you mistyped a parameter, so you correct your mistake and rerun the script.
- Now the system administration runs your script in the production environment against thousands of machines; it runs for six hours and then fails. Reviewing the error information, the SA discovers the script failed because of a mistyped parameter. At that point the SA has already invested six hours only to encounter an error and might conclude that your script isn't usable. In other words, you wrote the script that worked in your developmental environment but not in the production environment.



Script Design: Perform Validity Checking



The image shows a Notepad++ window titled '*C:\Users\Administrator\MyScripts\PS-Part4-p27.ps1 - Notepad++'. The window contains a PowerShell script with the following content:

```
1 #-----
2 # PS-Part4-p27
3 # test script - PowerShell Notes - Part 4 Page 27
4 #
5 #-----
6
7 #-----
8 # Configuration Information
9 #-----
10 param([string] $TemplatePath=$(throw write-host `
11     "Please specify the source template path of the folder structure to " `
12     "be copied." -ForegroundColor Red), [string] $ImportFile = $throw `
13     write-host "Please specify the import CSV filename. " `
14     -ForegroundColor Red ))
15
16 #-----
17 # Script Body
```

The status bar at the bottom of the window displays: Windows PowerShell, length : 1504 lines : 37, Ln : 32 Col : 1 Sel : 3, Dos\Windows, ANSI, and INS.





```
15
16 #-----
17 # Script Body
18 #-----
19 write-host "Checking Template Path" -NoNewLine
20 if (!(test-path $TemplatePath)) {
21     throw write-host `t "$TemplatePath is not a valid directory!" `
22         -ForegroundColor Red
23 }
24 else {
25     write-host `t "[OK]" -ForegroundColor Green
26 }
27
28 write-host "Checking Import File" -NoNewLine
29 if (!(test-path $ImportFile)) {
30     throw write-host `t "$ImportFile is not a valid file!" -ForegroundColor Red
31 }
32 else {
33     write-host `t "[OK]" -ForegroundColor Green
34 }
35 #-----
36 # END SCRIPT
37 #-----
```

User forgets to enter the either the source folder or the file name

```
PS C:\users\Administrator\MyScripts>
PS C:\users\Administrator\MyScripts> .\PS-Part4-p27.ps1
```

Please specify the source template path of the folder structure to be copied.

```
ScriptHalted
At C:\users\Administrator\MyScripts\PS-Part4-p27.ps1:10 char:37
+ param([string] $TemplatePath=${throw <<< write-host `
+ CategoryInfo          : OperationStopped: (:) [], RuntimeException
+ FullyQualifiedErrorId : ScriptHalted
```

User enters source folder but forgets to enter the file name

```
PS C:\users\Administrator\MyScripts> .\PS-Part4-p27.ps1 .
```

Please specify the import CSV filename.

```
ScriptHalted
At C:\users\Administrator\MyScripts\PS-Part4-p27.ps1:12 char:69
+ "be copied." -ForegroundColor Red), [string] $ImportFile = ${throw <<< `
+ CategoryInfo          : OperationStopped: (:) [], RuntimeException
+ FullyQualifiedErrorId : ScriptHalted
```

User enters both required parameters – script executes successfully.

```
PS C:\users\Administrator\MyScripts> .\PS-Part4-p27.ps1 . textfile.csv
```

```
Checking Template Path [OK]
Checking Import File [OK]
```

```
PS C:\users\Administrator\MyScripts> .
```



Script Design: Provide Status Information To Users

- Providing status information in an automation script is essential so that the user understands how the script is progressing during execution and can confirm whether script tasks have been completed successfully.
- Status information also lets users know whether any errors have occurred, and it can even indicate how much longer until the script will finish running.
- In PowerShell you can provide status information to users in the form of console displays by using the `write-host` and `write-progress` cmdlets, write status information to a log file, or leverage Windows Forms to report on the status of your script.
- The script execution on the following page illustrates some of these features.



Script Design: Provide Status Information To Users

```
Administrator: Windows PowerShell
PS C:\users\Administrator\MyScripts> c:\users\Administrator\myscripts\StatusInformation\ProvisionWeb
\administrator\myscripts\StatusInformation\template c:\users\administrator\myscripts\StatusInformat

-----
ProvisionWebFolders
-----

Checking Template Path [OK]
Checking Import File [OK]

Provision Web Folders:
C:\StatusInformation\WWWs\Mark
Folder [EXISTS]

C:\StatusInformation\WWWs\Kristy
Folder [COPIED]
SetOwner for Administrators [OK]
AddACE for Administrators [OK]
ClearInherit [OK]
AddACE for SYSTEM [OK]
AddACE for Kristy [ERROR] Failed to add rights!

Done Provisioning Web Folders:
```

This script is providing various types of status information to the user as it performs its tasks.



Script Design: Provide Status Information To Users

- Regardless of the method you use, the idea is to provide enough status information without overloading the user with useless details.
- If you need to write different levels of detail when displaying information to the user, you can use the `write-verbose` and `write-debug` cmdlets, the `verbose` and `debug` parameters, or create entirely custom output.



Script Design: Use The `WhatIf` and `Confirm` Parameters

- PowerShell includes two cmdlet parameters that are designed to help prevent scripters and system administrators from making unintended changes.
- The `WhatIf` parameter is designed to return information about changes that would occur if the cmdlet runs but doesn't actually make those changes.
- The `Confirm` parameter prevents unwanted modifications by forcing PowerShell to prompt users before making any changes.
- The example on page 35 illustrates the `WhatIf` parameter. The example on page 36 illustrates the `Confirm` parameter.



```
Administrator: Windows PowerShell
PS C:\users\Administrator\MyScripts> get-process expl* | stop-process -WhatIf
What if: Performing operation "Stop-Process" on Target "explorer (2584)".
PS C:\users\Administrator\MyScripts> get-process | stop-process -WhatIf
What if: Performing operation "Stop-Process" on Target "ApacheMonitor (3044)".
What if: Performing operation "Stop-Process" on Target "csrss (496)".
What if: Performing operation "Stop-Process" on Target "csrss (540)".
What if: Performing operation "Stop-Process" on Target "dllhost (2236)".
What if: Performing operation "Stop-Process" on Target "dwm (2536)".
What if: Performing operation "Stop-Process" on Target "explorer (2584)".
What if: Performing operation "Stop-Process" on Target "httpd (1568)".
What if: Performing operation "Stop-Process" on Target "httpd (3540)".
What if: Performing operation "Stop-Process" on Target "Idle (0)".
What if: Performing operation "Stop-Process" on Target "jucheck (3332)".
What if: Performing operation "Stop-Process" on Target "jusched (3028)".
What if: Performing operation "Stop-Process" on Target "lsass (640)".
What if: Performing operation "Stop-Process" on Target "lsm (648)".
What if: Performing operation "Stop-Process" on Target "msdtc (2348)".
What if: Performing operation "Stop-Process" on Target "mysqld (1680)".
What if: Performing operation "Stop-Process" on Target "notepad++ (2068)".
What if: Performing operation "Stop-Process" on Target "powershell (1664)".
What if: Performing operation "Stop-Process" on Target "services (628)".
What if: Performing operation "Stop-Process" on Target "SLsvc (1028)".
What if: Performing operation "Stop-Process" on Target "smss (428)".
What if: Performing operation "Stop-Process" on Target "spoolsv (1524)".
What if: Performing operation "Stop-Process" on Target "svchost (816)".
What if: Performing operation "Stop-Process" on Target "svchost (876)".
What if: Performing operation "Stop-Process" on Target "svchost (952)".
What if: Performing operation "Stop-Process" on Target "svchost (996)".
What if: Performing operation "Stop-Process" on Target "svchost (1012)".
What if: Performing operation "Stop-Process" on Target "svchost (1088)".
What if: Performing operation "Stop-Process" on Target "svchost (1144)".
What if: Performing operation "Stop-Process" on Target "svchost (1168)".
What if: Performing operation "Stop-Process" on Target "svchost (1332)".
What if: Performing operation "Stop-Process" on Target "svchost (1468)".
What if: Performing operation "Stop-Process" on Target "svchost (1736)".
What if: Performing operation "Stop-Process" on Target "svchost (1748)".
```



```
Administrator: Windows PowerShell
PS C:\users\Administrator\MyScripts>
PS C:\users\Administrator\MyScripts> get-process expl* | stop-process -Confirm

Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "explorer (2584)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): n
PS C:\users\Administrator\MyScripts> .
```



Script Design: Use The `WhatIf` and `Confirm` Parameters

- The `WhatIf` and `Confirm` parameters are a programmatic way to provide a safety net for the users of your scripts. These parameters are unique to PowerShell and are often overlooked by scripters.
- Although you need to include the support for the `WhatIf` and `Confirm` parameters in your script, doing so makes it far less likely that your script will accidentally perform an unintentional damaging action.
- The `WhatIf` parameter specifically allows you to see the results of a script without actually executing it, making it easy to fine-tune your script before actually running it live. Note that these two parameters only apply to cmdlets that make modifications.



PowerShell Community Scripting Standards

- One of the defining characteristics of PowerShell as a language is its adoption and advancement by a diverse, worldwide group of scripters and IT professionals.
- One example of this type of adoption is the PowerShell Community web site at <http://powershellcommunity.org>. The members of this community collaborate on a wide variety of PowerShell topics, including scripts, blog entries, and the latest PowerShell developments from technology vendors.
- The community maintains a library of generally agreed-upon best practices for scripting and related topics. I urge you to check out the site.
- Several of the PowerShellCommunity.org scripting standards are summarized in the remaining pages of this set of notes.



Summary Of PowerShellCommunity.org Best Practices

Name Your Scripts And Functions Using The Verb-Noun Syntax.

- Although you can use any naming convention, the Verb-Noun syntax makes it easier for others to understand the purpose of your script, and it also provides consistency with the native PowerShell cmdlets.
- The Verb-Noun syntax is also used by several other programming languages, such as VAX VMS/DCL and AS/400 CL, so following this syntax will help to ensure that your script names are understandable to users with experience in these languages as well.



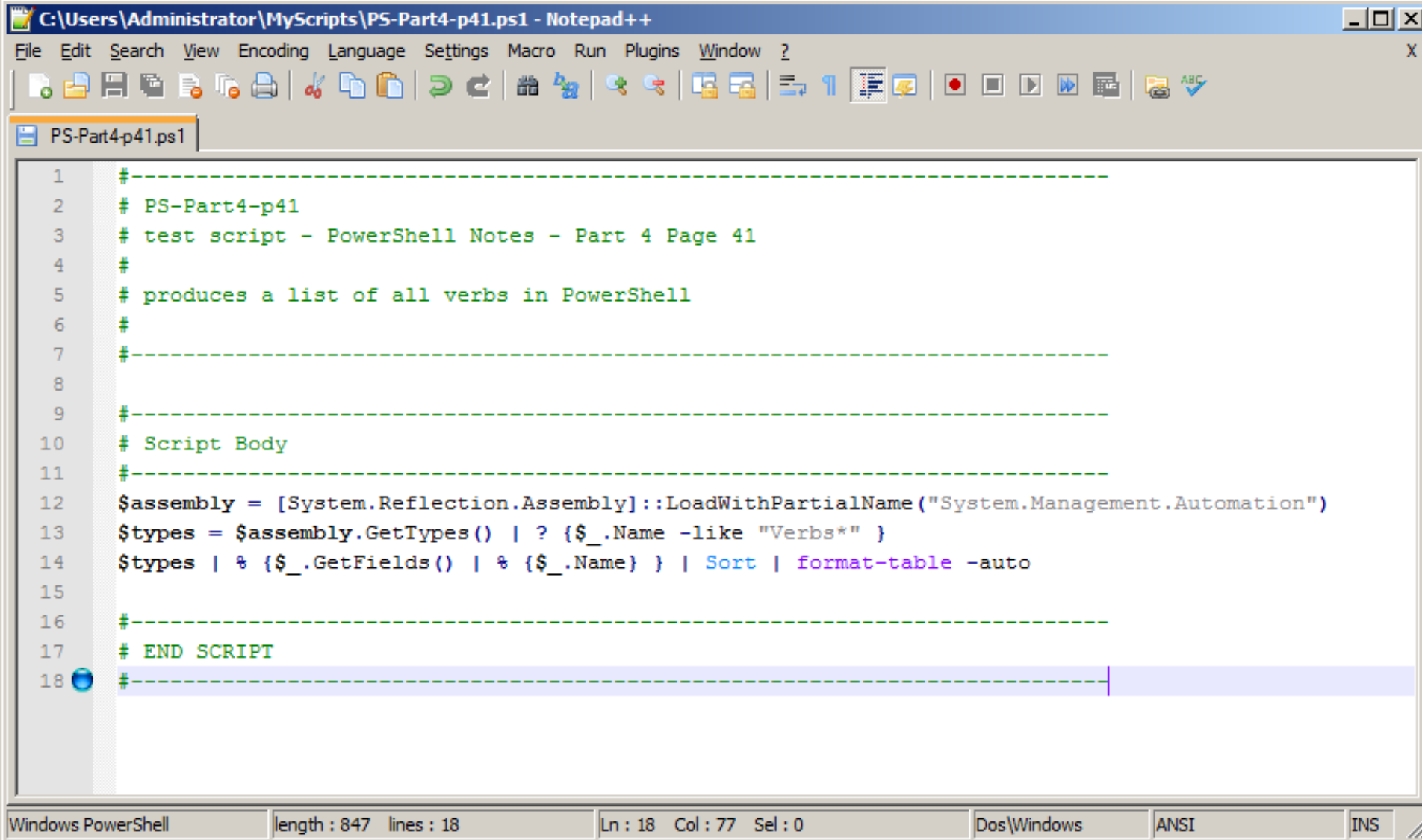
Summary Of PowerShellCommunity.org Best Practices

Whenever Possible, Use Verbs From The Standard PowerShell Verb List.

- One of the goals of the PowerShell design team was to enable system administrators to accomplish most of their tasks using just the standard set of PowerShell verbs.
- This makes it easier for system administrators to learn about the properties of new PowerShell nouns, because they will already be familiar with the common verbs that are used to work with new nouns.
- By running the script shown on the next page, PowerShell will return a list of all the standard PowerShell verb names.



Summary Of PowerShellCommunity.org Best Practices



```
C:\Users\Administrator\MyScripts\PS-Part4-p41.ps1 - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
PS-Part4-p41.ps1
1 #-----
2 # PS-Part4-p41
3 # test script - PowerShell Notes - Part 4 Page 41
4 #
5 # produces alist of all verbs in PowerShell
6 #
7 #-----
8
9 #-----
10 # Script Body
11 #-----
12 $assembly = [System.Reflection.Assembly]::LoadWithPartialName("System.Management.Automation")
13 $types = $assembly.GetTypes() | ? { $_.Name -like "Verbs*" }
14 $types | % { $_.GetFields() | % { $_.Name } } | Sort | format-table -auto
15
16 #-----
17 # END SCRIPT
18 #-----
```

Windows PowerShell | length : 847 lines : 18 | Ln : 18 Col : 77 Sel : 0 | Dos\Windows | ANSI | INS



Administrator: Windows PowerShell

```
PS C:\users\Administrator\MyScripts> .\PS-Part4-p41.ps1
```

Add	Hide	Search
Approve	Import	Select
Assert	Initialize	Send
Backup	Install	Set
Block	Invoke	Show
Checkpoint	Join	Skip
Clear	Limit	Split
Close	Lock	Start
Compare	Measure	Step
Complete	Merge	Stop
Compress	Mount	Submit
Confirm	Move	Suspend
Connect	New	Switch
Convert	Open	Sync
ConvertFrom	Out	Test
ConvertTo	Ping	Trace
Copy	Pop	Unblock
Debug	Protect	Undo
Deny	Publish	Uninstall
Disable	Push	Unlock
Disconnect	Read	Unprotect
Dismount	Receive	Unpublish
Edit	Redo	Unregister
Enable	Register	Update
Enter	Remove	Use
Exit	Rename	Wait
Expand	Repair	Watch
Export	Request	Write
Find	Reset	PS C:\users\Administrator\MyScripts> _
Format	Resolve	
Get	Restart	
Grant	Restore	
Group	Resume	
	Revoke	
	Save	



Summary Of PowerShellCommunity.org Best Practices

Use Unique Noun Names.

- Many times, using generic nouns, such as Computer or User, in a script or function causes a name collision with an existing noun of the same name.
- If a name collision does occur, users need to explicitly reference the full path and filename of the script in order for it to run successfully.
- PowerShell addresses this issue by applying a PS prefix to make its nouns unique (as in PSDrive or PSBreakPoint), and it is recommended that you use a similar method for any nouns that you reference in your scripts (as in MyTable or MyPrinter).



Summary Of PowerShellCommunity.org Best Practices

Make Noun Names Singular.

- Because the English language handles pluralization for different nouns in different ways (as in single potato or several potatoes, a single mouse or several mice), using pluralization in PowerShell noun names can be confusing, especially for those PowerShell users whose first language is not English.
- This issue is made more complicated by the fact that not all cmdlets provide support for multiple return values. For example, the `get-date` cmdlet can return only a single value, but the `get-process` cmdlet can return multiple values.
- If PowerShell users had to guess at the capabilities of a cmdlet based on whether the noun portion of the cmdlet name was singular or plural, ease of use would quickly grind to a halt.



Summary Of PowerShellCommunity.org Best Practices

Using Pascal Casing For Script And Function Names – Camel Casing For Variable Names.

- Pascal casing capitalizes the first letter of each word, such as `Get-Process` and `ConvertFrom-StringData`.
- Camel casing has the first letter of a word in lowercase and capitalizes subsequent first letters, such as `remoteComputerName`.
- Using Pascal casing for script names follows the standard used by other PowerShell cmdlets, whereas using camel casing for variable names follows the standards established in the .NET style guidelines. Because of PowerShell's frequent interaction with .NET objects, methods, and properties, it makes sense to adhere to the .NET naming standards for variable names.



Summary Of PowerShellCommunity.org Best Practices

Use Descriptive Variable Names.

- When you are writing a script, the variable names that you choose can go a long way toward clarifying the actions that your script performs.
- Applying some thought to your choice of variable names can make your scripts almost self-documenting. Although the variable names `$comp` and `$remoteComputerDNSName` function the same way when used in a script, the second variable clearly illustrates the data that the variable contains making any code which contains this variable much easier to understand.
- This is especially important in environments where your scripts will be used/modified by multiple administrators with different levels of scripting knowledge.



Summary Of PowerShellCommunity.org Best Practices

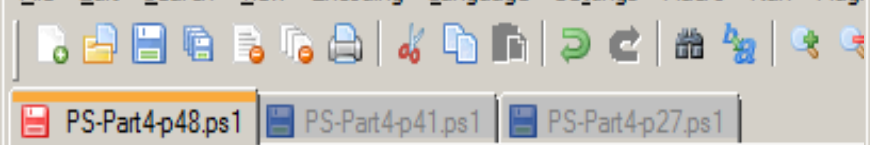
Add A Unique Namespace Prefix To Global Variables.

- As with PowerShell verbs and nouns, it is possible to create naming collisions with existing environment variables and produce unexpected results.
- One common way to reduce the possibility of variable name collisions is to prefix any global variables that you define with a unique identifier for the namespace. This identifier could be your company name, the name of the software application, or similarly unique data.
- A global variable is declared in this manner in PowerShell according to the following syntax:

```
 ${GLOBAL:uniqueIdentifier.variableName} =
```

- The example script on the next page further illustrates this convention.





```
1 #-----
2 # PS-Part4-p48
3 # test script - PowerShell Notes - Part 4 Page 48
4 #
5 # Illustration of prefixing unique namespace to global variables in PowerShell
6 #
7 #-----
8
9 #-----
10 # Script Body
11 #-----
12 ${global:savn_PI} = 3.14159
13 write-host "The value of PI is:"
14 write-host $savn_PI -ForegroundColor Red
15 write-host "The circumference of a circle of radius 1m is: "
16 $circumference = 2 * $global:savn_PI
17 write-host $circumference -ForegroundColor Green
18
19 #-----
20 # END SCRIPT
21 #-----
```

```
Administrator: Windows PowerShell
PS C:\users\Administrator\MyScripts> .\PS-Part4-p48.ps1
The value of PI is:
3.14159
The circumference of a circle of radius 1m is:
6.28318
PS C:\users\Administrator\MyScripts>
```

Summary Of PowerShellCommunity.org Best Practices

Ensure That Your Scripts Run Successfully In Strict Mode.

- Using strict mode in PowerShell is an optional setting that causes PowerShell to raise a terminating error when a script or expression does not follow best practice coding standards (such as referencing a variable that does not exist).
- Strict mode in PowerShell can be turned on or off by using the `Set-StrictMode` cmdlet, and it is enforced only in the scope where you set it.
- Because some users always run PowerShell with strict mode turned on, it is to your advantage to verify that your scripts will run in strict mode without errors. Otherwise, users have to disable strict mode to run the script, and it is possible that a script that raises an error in strict mode might be written in a way that causes the script to produce unreliable results. A script that runs successfully in strict mode meets at least a baseline set of programming standards, which helps to validate that the script is suitable for use in a production environment.





```

1  #-----
2  # PS-Part4-p50
3  # test script - PowerShell Notes - Part 4 Page 50
4  #
5  # Illustration of strict mode in
6  #
7  #-----
8  #-----
9  # Script Body
10 #-----
11 #turn strict mode off - default value is on
12 #set-strictmode -off
13 #turn strict mode on - set to version 1 level of enforcement
14 set-strictmode -version 1
15
16 write-host "The value of PI is:"
17 write-host $savn_PI -ForegroundColor Red
18 write-host "The circumference of a circle of radius 1m is: "
19 $circumference = 2 * $global:savn_PI
20 write-host $circumference -ForegroundColor Green
21 #-----
22 # END SCRIPT
23 #-----

```

```
PS C:\users\Administrator\MyScripts>  
PS C:\users\Administrator\MyScripts> .\PS-Part4-p50.ps1  
The value of PI is:
```

With strictmode = off, uninitialized variables are assumed to have a value of 0 or \$null depending on type.

```
The circumference of a circle of radius 1m is:
```

```
PS C:\users\Administrator\MyScripts>  
PS C:\users\Administrator\MyScripts> .\PS-Part4-p50.ps1  
The value of PI is:
```

With strictmode = on, uninitialized variable access generates errors.

The variable '\$savn_PI' cannot be retrieved because it has not been set.

```
At C:\users\Administrator\MyScripts\PS-Part4-p50.ps1:19 char:20  
+ write-host $savn_PI <<<< -ForegroundColor Red  
+ CategoryInfo          : InvalidOperation: (savn_PI:Token) [], RuntimeException  
+ FullyQualifiedErrorId : VariableIsUndefined
```

```
The circumference of a circle of radius 1m is:
```

The variable '\$global:savn_PI' cannot be retrieved because it has not been set.

```
At C:\users\Administrator\MyScripts\PS-Part4-p50.ps1:21 char:37  
+ $circumference = 2 * $global:savn_PI <<<<  
+ CategoryInfo          : InvalidOperation: (global:savn_PI:Token) [], RuntimeException  
+ FullyQualifiedErrorId : VariableIsUndefined
```

The variable '\$circumference' cannot be retrieved because it has not been set.

```
At C:\users\Administrator\MyScripts\PS-Part4-p50.ps1:22 char:26  
+ write-host $circumference <<<< -ForegroundColor Green  
+ CategoryInfo          : InvalidOperation: (circumference:Token) [], RuntimeException  
+ FullyQualifiedErrorId : VariableIsUndefined
```

```
PS C:\users\Administrator\MyScripts>
```



Summary Of PowerShellCommunity.org Best Practices

Avoid Using Aliases In Scripts Whenever Possible.

- Although there is no technical obstacle to including aliases in a script, aliases make the script harder to read and understand for those scripters who are not familiar with all of PowerShell's command aliases or the ones you create yourself.
- The reason behind PowerShell's aliases is to reduce the number of keystrokes needed when working interactively from the command line. Thus, there is only a minimal time saving achieved when aliasing commands in a script, which is quickly erased the next time you need to review/modify the script and must spend time recalling the commands that you aliased to once again understand the script.
- In general, readability and ease of understanding should override speed of data entry when creating a script that will be reused.



PS C:\users\Administrator\MyScripts> gps | fw

ApacheMonitor	
csrss	csrss
dwm	dllhost
httpd	explorer
Idle	httpd
jusched	jucheck
lsn	lsass
mysqld	msdtc
powershell	notepad++
SLsvc	services
spoolsv	smss
svchost	svchost
svchost	svchost
svchost	svchost
svchost	svchost
svchost	svchost
svchost	svchost
svchost	svchost
System	taskeng
taskeng	Tomcat7.0.22w
TPAutoConnect	TPAutoConnSvc
vmtoolsd	VMUpgradeHelper
VMwareTray	VMwareUser
wininit	winlogon
wuauc lt	

Aliased Version
Readable?



PS C:\users\Administrator\MyScripts> get-process | format-wide

Non-Aliased Version
Much More Readable

ApacheMonitor	csrss
csrss	dllhost
dwm	explorer
httpd	httpd
Idle	jucheck
jusched	lsass
lsn	msdtc
mysqld	notepad++
powershell	services
SLsvc	smss
spoolsv	svchost
svchost	svchost
svchost	svchost
svchost	svchost
svchost	svchost
svchost	svchost
svchost	svchost
System	taskeng
taskeng	Tomcat7.0.22w
TPAutoConnect	TPAutoConnSvc
vmtoolsd	VMUpgradeHelper
VMwareTray	VMwareUser
wininit	winlogon
wuaclt	

