**Windows PowerShell $\_. Variable**

**Introduction to the PowerShell Dollar Underscore Variable**

Constructions such as ... | Where {**$\_.**name -Match "win"} are incredibly useful in PowerShell.  Definitions such as: '$\_ means in this pipeline', are a bit stuffy.  The best way to understand $\_ is to work through examples.

**Examples of the PowerShell $\_ Variable**

* [**PowerShell's $\_ Variable**](http://www.computerperformance.co.uk/powershell/powershell_dollar_variable.htm#PowerShells_$__Variable)
* [**The Significance of the Dot in PowerShell's $\_.**](http://www.computerperformance.co.uk/powershell/powershell_dollar_variable.htm#The_Significance_of_the_Dot_in_PowerShells_$_.)
* [**$\_ Example to Filter WmiObjects**](http://www.computerperformance.co.uk/powershell/powershell_dollar_variable.htm#$__Example_to_Filter_WmiObjects)
* [**PowerShell $\_ with ForEach**](http://www.computerperformance.co.uk/powershell/powershell_dollar_variable.htm#PowerShell_$__with_Foreach)  **$\_.**
* [**More PowerShell Variables**](http://www.computerperformance.co.uk/powershell/powershell_variables.htm)

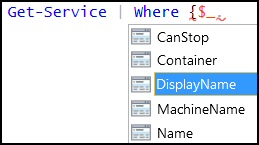
**PowerShell's $\_ Variable**

The first point to remember is that $\_ is a variable or placeholder.

# PowerShell $\_ Variable Example  
Get-Service | Where-Object {**$\_**.name -Match "win"}

What we want to say here is  
Get the services, where the service name matches 'Win'.  Did the repetition of 'service' seem a little verbose?  Well, script writing is brilliant at eliminating extraneous words?

In a nutshell $\_ saves us repeating Get-Service, or whatever else may be to the left of ... | Where{$\_

**The Significance of the Dot in PowerShell's $\_.[](http://www.computerperformance.co.uk/powershell/index_syntax.htm)**

Here is a similar dollar underscore example but featuring .DisplayName instead of .Name.  My point is I want to illustrate how the .dot command introduces a property.

If you use PowerShell's ISE GUI, then as soon as you type that period (.) you get a drop-down list of properties.

# PowerShell $\_ Variable Example  
Get-Service | Where {**$\_**.DisplayName -Match "win"}

**Challenge:**  Research more properties with Get-Service | Get-Member.

**$\_ Example to Filter WmiObjects**

The key to remembering the syntax is to breakdown the construction in to:  $dollar / \_underscore / dot.property.  The commonest example would be: $\_.name.

# PowerShell script to find Network WMI Objects  
Get-WmiObject -List | Where-Object {$\_.name -Match 'Network'}

**Note 1:** The real-life task is to research for network type WMI objects.  Without the where clause it would be like looking for a needle in a haystack.

**The PowerShell Variable $\_**

Another way of looking at PowerShell's $\_ is purely as a variable.  After all, the dollar sign is PowerShell's way of introducing any variable, not just this special 'In this pipeline' item.

#PowerShell Script to List Dll files  
$Path = "C:\Windows\System32\"   
Get-ChildItem $Path | Where {**$\_**.extension -eq '.DLL'}

**Note 2:** The point of this example is to compare the special $\_ variable with an ordinary variable called $Path.

**Significance of the Where {Evaluation}**

Most of the $\_ examples feature the 'where' filter.  While many scripters also like the alias ?, the underlying cmdlet is: Where-Object.  Let us take the time to research its properties.

Clear-Host  
Get-Help Where-Object -full

**Note 3:** Remember that Where-Object is a filter; therefore to perform its job Where needs a ScriptBlock to evaluate the test.  Actually, the two characters '$ and \_' play a small but important role; $\_'s job is to shorten the evaluation by saying, 'In this pipeline', rather than explicitly mentioning the test for a second time.

[**See more on measuring the speed of Where-Object**](http://www.computerperformance.co.uk/powershell/powershell_measure_object_filter.htm)

**PowerShell $\_ with ForEach**

Most of my $\_ examples are found in Where-Object clauses, but here is a different use for this special pipeline variable: ForEach.  Once again, observe that $\_ is the first item inside the curly brackets, but this time the underscore is followed by the -replace parameter rather than a .property.

The purpose of this script is to remove and duplicate "the the" in documents stored in the ProofRead folder.

Clear-Host  
$file = gci "D:\ProofRead\\*.doc"  
$file  
ForEach ($str in $file)   
{  
$cont = Get-Content -path $str  
$cont  
$cont | ForEach {**$\_** -replace "the , "the"} | Set-Content $str  
}

**Note 4:** In the ForEach example the syntax is purely $\_ there is no (.) property.

**More of PowerShell's Built-In Variables**

You can enumerate PowerShell's variables with this command:

Get-Variable | Format-Table name, value -auto

|  |  |
| --- | --- |
| **Variable Name** | **Description** |
| **$\_** | **The current pipeline object; used in script blocks, filters, the process clause of functions, where-object, ForEach-object and switch** |
| $Args | Used in creating functions that require parameters |
| $Env:Path | Environmental Path to files. |
| $Error | If an error occurred, the object is saved in the $error PowerShell variable |
| $ForEach | Refers to the enumerator in a ForEach loop. |
| $HOME | The user's home directory; set to %HOMEDRIVE%\%HOMEPATH% |
| $Input | Input piped to a function or code block |
| $Match | A hash table consisting of items found by the -Match operator. |
| $Host | Information about the currently executing host |
| $LastExitCode | The exit code of the last native application to run |
| $true | Boolean TRUE |
| $false | Boolean FALSE |
| $null | A null object |
| $ShellID | The identifier for the shell.  This value is used by the shell to determine the ExecutionPolicy and what profiles are run at startup. |
| $StackTrace | contains detailed stack trace information about the last error |

**Windows PowerShell Variables**

**Introduction to Windows PowerShell Variables**

All scripting languages use placeholders or variables to hold data.  Furthermore, each language has its own rules and symbols.  I have found that using PowerShell variables is straightforward, just remember to introduce your variable with a dollar sign, for example: $Memory.

**Topics for PowerShell's Variables**

* [**$Dollar Variables**](http://www.computerperformance.co.uk/powershell/powershell_variables.htm#PowerShells_$Dollar_Variables_)
* [**Set-Variable, Scope and Option**](http://www.computerperformance.co.uk/powershell/powershell_variables.htm#Set-Variable,_Scope_and_Option._)
* [**PowerShell's Dot Properties**](http://www.computerperformance.co.uk/powershell/powershell_variables.htm#PowerShells_Dot_Properties)
* [**Special Pipeline Variable:**](http://www.computerperformance.co.uk/powershell/powershell_dollar_variable.htm)  **$\_.**
* [**More PowerShell Variables**](http://www.computerperformance.co.uk/powershell/powershell_variables.htm#More_Variables_)
* [**Summary of PowerShell Variables**](http://www.computerperformance.co.uk/powershell/powershell_variables.htm#Summary_of_Monad_Variables_)

**PowerShell's $Dollar Variables**

Variables are handy for storing data which can be used later in the script; for example, storing a file path.  Creating a PowerShell variable could not be more straightforward; just declare the variable by putting the dollar sign in front of the name you wish to call the variable.  Let us create, then set, a variable called $Mem:

$Mem= WmiObject Win32\_ComputerSystem

Once we have created $Mem, then we can put the variable to work and calculate the RAM memory in Mega bytes.

# PowerShell $ Variable Example  
$Mem= WmiObject Win32\_ComputerSystem  
$Mbyte =1048576 # Another variable  
"Memory Mbyte " + [int]($Mem.TotalPhysicalMemory/$Mbyte)

PowerShell has no built-in mechanism for enforcing variable types, for example, string variables can begin with letters or numbers.  Yet numeric variables can also begin with letters (or numbers).  However, you can restrict the values a PowerShell variable accepts by preceding it with [int] or [string], here is an example:

**Example 1a: Declaring a PowerShell Variable as an Integer**

# Declaring PowerShell Integer Variable  
[int]$a =7  
$a +3   
$a  
# PS>10

**Example 1b: "Twenty" is not an [int]**

# Declaring PowerShell Integer Variable  
[int]$a =7  
$a ="Twenty"  
$a

# PS> Error "Cannot Convert value.

**Note 1:** I cannot resist pointing out the significance of [Square brackets].  The reason is that PowerShell only ever uses square brackets for optional items, and declaring the type of a variable is just that - [optional[.

**Example 2: Declaring the Variable Without Specifying the Type.**

$b = 7  
$b = "Twenty"  
$b

# PS> Twenty

There is no error here because $b was not declared as number or a string.

Do you think that PowerShell variables are case sensitive, or case insensitive?  The answer is insensitive, just as with most other PowerShell commands, upper or lower case work equally.

When Windows PowerShell evaluates a potential variable name, it carries on from the $Dollar until it meets a word breaking character such as a space or punctuation mark.  This does not give me a problem because I only use snappy OneWord names, but if you use variables with strange characters - watch out!  If you insist on using variables with names such as a\*?,v\*\*, then you could enclose them in braces - thus {a\*?,v\*\*}.  Clever stuff, but best to keep it simple, and don't ask for trouble I say.

**Reserved Words - Not To Be Used For Variables**

Avoid using these reserved keywords for your variables: Break, continue, do, else, ElseIf, for, foreach, function, filter, in, if, return, switch, until, where and while.

Incidentally, you can join string variables simply by using a plus (+) sign.  The reason that I mention this is because I spent ages searching fruitlessly for a special text concatenator, only to discover that the plain plus sign was all I needed.

**Declaring Multiple Variables**

In scripting we are always seeking ways of writing tighter code.  In the case of variables you could make multiple declarations on the same line; for example:

$DriveA, $DriveB, $DriveC, $DriveD = 250, 175, 330, 200

**PowerShell Variables - Text Examples**

What you find when researching WMI objects is that there are so many of them.  The purpose of this script is filter, or home in on, a keyword such as 'Win32', or 'network'.  To aid the research, and to recycle the code I introduce a variable $Type.

To begin with I set the value of $Type to "Win32", later it would be a trivial task to amend to "Network".  Once again, note how this script uses the built in $\_. variable, and see how it uses .name in this pipeline.

# PowerShell Variables Including $\_.   
$i=0   
$Type = "Win32"   
$WMI = Get-WmiObject -List | Where-Object {$\_.name -Match $Type}  
Foreach ($CIM in $WMI) {$i++}  
Write-Host 'There are '$i' types of '$Type

**Note 2:** The benefit of introducing the variable $Type is that you can change its value to say "Network" easily.  I love variables, they help me understand the structure of my scripts.

[**See more on my Get-WmiClass function »**](http://www.computerperformance.co.uk/powershell/powershell_function_get_wmiclass.htm)

**Set-Variable, Scope and Option**

You can control, or restrict, PowerShell variables with the set-Variable command.  These extra properties of 'Option' and 'Scope' are not really necessary for beginners, nevertheless as you grow in ambition, you may like to revisit these additional features.

**Option:** You can set the variable to be read-only, or alternatively, it could be a constant.  Constant variables sound strange, and I rarely use them, but their killer feature is they cannot be deleted.

# Example: PowerShell Set-Variable constant  
Set-Variable Thermometer 32 -option constant.

**Note 3:** When initializing with Set-Variable, $Thermometer would be wrong, plain Thermometer is what you need here.  Once the value is set to 32 it cannot be changed.

**Scope** can be local, global or script.  The default value for the scope of a variable is local.

# Example: PowerShell Set-Variable Global  
Set-Variable AllOverPlace 99 -scope global

**Note 4:** For more information about Set-Variable, try Help Set-Variable.

**Note 5:** The value would be 99, again you don't add the $dollar sign when you execute set-variable.  Actually, there is an alternative method for setting and creating Scope:

# Set PowerShell Global Variable  
$global:runners = 8

**PowerShell's Dot Properties**

Before we look at the pipeline variable $\_.  please remember that PowerShell variables support the dot (.) properties.  For example:

$alert = Get-Service NetLogon  
$alert.status

# PS> Started

Not only is $variable.property a neat technique, but also realize that Get-Service NetLogon.status does not work; all that you get is an error saying: 'Cannot find any service called NetLogon.status'.

**Special Pipeline Variable:  $\_.**

$\_ or $\_. takes the dot notation one stage further.  It acts a placeholder for the current object.  The official definition of $\_. is the current pipeline object; used in script blocks, filters, the process clause of functions, where-object, foreach-object and switches.  However, I believe that special PowerShell variable $\_. is best explained by examples.

**PowerShell Example to find all services that are Running (not stopped)**

# PowerShell Pipeline $\_. example  
Get-Service | Where-Object {$\_.status -eq "Running" }

Remember that this Get-Service command lists all those services on the machine.  Status is a property of the service.  One of the possible values for status is "Running", another value is "Stopped".  Should you wish to employ a 'where' clause, then you need the $\_. variable to introduce or link to the property 'status', hence $\_.status.

**Example to find all WmiObjects containing 'CIM'**

Get-WmiObject -List | Where-Object {$\_.name -Like "CIM\*"}

**Note 6:** The point is that the list is too long when you try:   
Get-WmiObject -List

By pipelining $\_.name, we can filter just names containing "CIM".  Incidentally it does not work without the wildcard \* -Like "CIM\*".  However you could experiment with -Match, or -Contains.

See more on [**PowerShell's $\_.**](http://www.computerperformance.co.uk/powershell/powershell_pipeline.htm)  meaning In the Current Pipeline

**More Built-In PowerShell Variables**

You can enumerate PowerShell's variables with this command:

Get-Variable

Another example showing more information and more control:

Get-Variable | Format-Table name, value -auto

|  |  |
| --- | --- |
| **Variable Name** | **Description** |
| $\_ | The current pipeline object; used in script blocks, filters, the process clause of functions, where-object, foreach-object and switch |
| $^ | contains the first token of the last line input into the shell |
| $$ | contains the last token of last line input into the shell |
| $? | Contains the success/fail status of the last statement |
| $Args | Used in creating functions that require parameters |
| $Env:Path | Environmental Path to files. |
| $Error | If an error occurred, the object is saved in the $error PowerShell variable |
| $foreach | Refers to the enumerator in a foreach loop. |
| $HOME | The user's home directory; set to %HOMEDRIVE%\%HOMEPATH% |
| $Input | Input piped to a function or code block |
| $Match | A hashtable consisting of items found by the -Match operator. |
| $MyInvocation | Information about the currently script or command-line |
| $Host | Information about the currently executing host |
| $LastExitCode | The exit code of the last native application to run |
| [**$PSVersionTable**](http://www.computerperformance.co.uk/powershell/powershell-version-check.htm) | Chech the version of PowerShell |
| $true | Boolean TRUE |
| $false | Boolean FALSE |
| $null | A null object |
| $PsUnsupported ConsoleApplications | [**List unsupported commands**](http://www.computerperformance.co.uk/powershell/powershell3_ise.htm) |
| $OFS | Output Field Separator, used when converting an array to a string. By default, this is set to the space character. |
| $ShellID | The identifier for the shell.  This value is used by the shell to determine the ExecutionPolicy and what profiles are run at startup. |
| $StackTrace | contains detailed stack trace information about the last error |

**PowerShell Variables in Action**

**Example 1)**  
To discover which version of PowerShell you are running, go to the PowerShell command line and type:

$Host

**Example 2)**  
You could modify the Environment Path value thus:

Clear-Host  
$Env:Path = $Env:Path + ";C:\Wizzo\Stuff"

**Note 7:** The plus (+) means that you keep the existing path locations and append C:\Wizzo\Stuff

**Note 8:** Talking of Env variables, you can list them with gci thus:

Get-ChildItem Env:\

**String and Int32 Variables**

PowerShell takes care of strings and numbers automatically, thus you can produce excellent scripts without worrying about this aspecto of PowerShell theory.

$Files = "C:\Windows" | Get-Member

Result (At the very top of the list)   
**TypeName System.String**

PowerShell takes care of numbers automatically, but assigns the variable to a different class.

$Bit = 64 | Get-Member

Result (At the very top of the list)   
**TypeName System.Int32**

[**Review of PowerShell Math »**](http://www.computerperformance.co.uk/powershell/index_math.htm)

**Further Research on PowerShell Variables**

There is a wealth of information in PowerShell' built-in help, access the examples and detailed examples via the Help About files.

Get-Help about\_Variable  
#Help about\_Automatic\_Variables  
#Help about\_Environment\_Variables  
#Help about\_Preference\_Variables

**PowerShell Variable Theory**

PowerShell variables are mapped to classes in the Microsoft .NET Framework. One benefit is that variables are objects and thus can be manipulated in many ways.  There is also a family of variable cmdlets which you can see with this command:

What impresses programmers is the ability to assign not just text to the variable, but also to assign an object to a variable.  While most proper scripting languages are able to handle objects through variables, CMD lacks this ability.

Get-Command -Noun variable

Although there is cmdlet called New-Variable, I have never seen anyone use it for real because you can just wade in with a declaration.

$Files = "C:\Windows"  # Is so much easier than New-Variable  
Get-ChildItem $Files

**Declaring Variables with New-Variable**

Remove-Variable $Files  
New-Variable files -value "C:\windows"   
Get-ChildItem $Files

[**See more on Get-PSProvider »**](http://www.computerperformance.co.uk/powershell/powershell_get_psprovider.htm)

**Summary of PowerShell Variables**

In PowerShell, variables are easy to create, just precede the name with a dollar sign, for example $Disk.  For more ambitious scripting you can restrict their type for example [int]$Memory, you can also prescribe the variable's scope, local or global.

One variable worth mastering is the special pipeline variable controlled by $\_.