# Introduction to Powershell Scripts - Variables and Loops

**(For more information see:** [**http://www.powershellpro.com/powershell-tutorial-introduction/variables-arrays-hashes/**](http://www.powershellpro.com/powershell-tutorial-introduction/variables-arrays-hashes/)

**)**

**Start Powershell ISE as Admininstrator in your Virtual Machine for the exercizes here.**

**To run powershell scripts on your machine you may need to change the execution policy on your machine to unrestricted using the Set-ExecutionPolicy commandlet**

Set-ExecutionPolicy unrestricted

## PowerShell Variables

Variables can contain text strings, integers, and even objects (complete with properties and methods).

There are also “Special” variables exist, which are pre-defined within PowerShell.

**Some of the Special Variables**

* **$\_** – Contains the current pipeline object, used in script blocks, filters, and the where statement.
* **$Args** – Contains an array of the parameters passed to a function.
* **$Error** – Contains objects for which an error occurred while being processed in a cmdlet.
* **$Home** – Specifies the user’s home directory.
* **$PsHome** – The directory where the Windows PowerShell is installed.

To view the complete list of Special Variables in PowerShell, type the following:

Get-Help about\_automatic\_variables<enter>

**ALL** variable names must start with the “**$**” character. Once data is assigned to a PowerShell variable, it’s automatically declared. The “=” operator is used to assign data to a variable.

$strComputer = “Computer1″

There is a “Set-Variable” cmdlet that can also be used:

Set-Variable -Name strUser -Value “John Doe”<enter>

The characters such as; **! @ # % & , .** and **spaces cannot be used in variable names**

The following words are also reserved and cannot be used

* Continue
* break
* do
* else
* elseif
* filter
* foreach
* function
* if
* in
* return
* switch
* until
* where
* while

Let’s verify that the $strComputer variable is holding the data we assigned to it.

Write-Output $strComputer<enter>

-or just type-

$strComputer<enter>

The output verifies the data in the **$strComputer** variable is the “Computer1″ text string. Go ahead and verify the data value for the “strUser” variable.

$strUser<enter>

Did you get **John Doe** for the output? Yes… great! No?… I knew you were going to skip the long version of using the “Set-Variable” cmdlet.

**Note:**If you use the “Set-Variable” cmdlet and specify the -Name parameter, you do not use the “**$**” character when defining your variable.

**Working with Strings**

A [sting] is a “text string” data type. There are many data types that we will be working with in PowerShell:

|  |  |
| --- | --- |
| **Type** | **Description** |
| [int] | 32-bit signed integer |
| [long] | 64-bit signed integer |
| [string] | Fixed-length string of Unicode characters |
| [char] | A Unicode 16-bit character |
| [byte] | An 8-bit unsigned character |
| [bool] | Boolean True/False value |
| [decimal] | An 128-bit decimal value |
| [single] | Single-precision 32-bit floating point number |
| [double] | Double-precision 64-bit floating point number |
| [xml] | Xml object |
| [array] | An array of values |
| [hashtable] | Hashtable object |

**String Concatenation**

Concatenation is the process of joining together two strings.

$strB = “World!”<enter>  
$strC = $strA += $strB<enter>  
$strC<enter>

**Working with Numbers**

Powershell will do its best to figure out the data type for a variable based on what is being assigned.

$x = 1<enter> - Integer  
$y = 1.2<enter> - Decimal  
$x<enter>  
$y<enter>

You can explicitly state the data type ..

[string]$strComputer = “MyFileServer01″<enter>  
[int]$x = 9<enter>  
[decimal]$y = 9.9<enter>

**PowerShell Operators**

* **=***Assigns a value to a variable.*
* **+** or **+=***Addition.*
* **-** or **-=***Subtraction.*
* **\*** or **\*=***Multiplication.*
* **/** or **/=***Division.*
* **%** or **%=***Modulus (retrieves the remainder of a division operation).*

Equations are read from left-to-right, multiplication and division are preformed then addition and subtraction.

$x = 10 \* 2 / 5 \* 2 + 5 \* 5<enter>  
$x<enter>

**PowerShell Arrays**

Variables can also store arrays which by definition are an indexed list of values.

To create an array, we create a variable and assign the array. Arrays are noted by the “@” symbol. Let’s take the discussion above and use an array to connect to multiple remote computers:

$strComputers = @(“Server1″, “Server2″, “Server3″)<enter>

To list the values in the array:

$strComputers<enter>

List the count of items

$strComputers.Count<enter>

$strComputers[0]<enter>  
$strComputers[1]<enter>  
$strComputers[2]<enter>

We can also modify elements in an array. Here I’m modifying an element in an existing array by providing the index number and the new string value. This changes the item Server3 to Server4.

$strComputers[2] = “Server4″<enter>

Using the **+**operator, we can combine arrays:

$v = @(1, 2, 3, 4, 5)<enter>  
$y = @(6, 7, 8, 9, 10)<enter>  
$z = $v + $y<enter>  
$z<enter>

Here is a typical script taken from the Microsoft Script Repository that enumerates the BIOS information on the (strComputer = “.”) local computer.

$strComputer = “.”

$colItems = get-wmiobject -class Win32\_BIOS -namespace root\CIMV2 -comp $strComputer

foreach ($objItem in $colItems) {  
write-host “BIOS Characteristics: ” $objItem.BiosCharacteristics  
write-host “BIOS Version: ” $objItem.BIOSVersion  
write-host “Build Number: ” $objItem.BuildNumber  
write-host “Caption: ” $objItem.Caption  
write-host “Code Set: ” $objItem.CodeSet  
write-host “Current Language: ” $objItem.CurrentLanguage  
write-host “Description: ” $objItem.Description  
write-host “Identification Code: ” $objItem.IdentificationCode  
write-host “Installable Languages: ” $objItem.InstallableLanguages  
write-host “Installation Date: ” $objItem.InstallDate  
write-host “Language Edition: ” $objItem.LanguageEdition  
write-host “List Of Languages: ” $objItem.ListOfLanguages  
write-host “Manufacturer: ” $objItem.Manufacturer  
write-host “Name: ” $objItem.Name  
write-host “Other Target Operating System: ” $objItem.OtherTargetOS  
write-host “Primary BIOS: ” $objItem.PrimaryBIOS  
write-host “Release Date: ” $objItem.ReleaseDate  
write-host “Serial Number: ” $objItem.SerialNumber  
write-host “SMBIOS BIOS Version: ” $objItem.SMBIOSBIOSVersion  
write-host “SMBIOS Major Version: ” $objItem.SMBIOSMajorVersion  
write-host “SMBIOS Minor Version: ” $objItem.SMBIOSMinorVersion  
write-host “SMBIOS Present: ” $objItem.SMBIOSPresent  
write-host “Software Element ID: ” $objItem.SoftwareElementID  
write-host “Software Element State: ” $objItem.SoftwareElementState  
write-host “Status: ” $objItem.Status  
write-host “Target Operating System: ” $objItem.TargetOperatingSystem  
write-host “Version: ” $objItem.Version  
write-host  
}

Now we want the same information from multiple remote computers. This time you choose which ones. Create an array with two (or more) computers, FYI – you must have local admin rights on each computer to run the script. Use notepad and change the computer names in the array. Copy and paste your new code into PowerShell.

$strComputer = @(“computer1″, “computer2″, “computer3″)

$colItems = get-wmiobject -class Win32\_BIOS -namespace root\CIMV2 -comp $strComputer

foreach ($objItem in $colItems) {  
write-host “BIOS Characteristics: ” $objItem.BiosCharacteristics  
write-host “BIOS Version: ” $objItem.BIOSVersion  
write-host “Build Number: ” $objItem.BuildNumber  
write-host “Caption: ” $objItem.Caption  
write-host “Code Set: ” $objItem.CodeSet  
write-host “Current Language: ” $objItem.CurrentLanguage  
write-host “Description: ” $objItem.Description  
write-host “Identification Code: ” $objItem.IdentificationCode  
write-host “Installable Languages: ” $objItem.InstallableLanguages  
write-host “Installation Date: ” $objItem.InstallDate  
write-host “Language Edition: ” $objItem.LanguageEdition  
write-host “List Of Languages: ” $objItem.ListOfLanguages  
write-host “Manufacturer: ” $objItem.Manufacturer  
write-host “Name: ” $objItem.Name  
write-host “Other Target Operating System: ” $objItem.OtherTargetOS  
write-host “Primary BIOS: ” $objItem.PrimaryBIOS  
write-host “Release Date: ” $objItem.ReleaseDate  
write-host “Serial Number: ” $objItem.SerialNumber  
write-host “SMBIOS BIOS Version: ” $objItem.SMBIOSBIOSVersion  
write-host “SMBIOS Major Version: ” $objItem.SMBIOSMajorVersion  
write-host “SMBIOS Minor Version: ” $objItem.SMBIOSMinorVersion  
write-host “SMBIOS Present: ” $objItem.SMBIOSPresent  
write-host “Software Element ID: ” $objItem.SoftwareElementID  
write-host “Software Element State: ” $objItem.SoftwareElementState  
write-host “Status: ” $objItem.Status  
write-host “Target Operating System: ” $objItem.TargetOperatingSystem  
write-host “Version: ” $objItem.Version  
write-host  
}

**Hash Tables**

A Hash table is also known as a dictionary. It is an array that allows you to store data in a “key-value” pair association. The “key” and “value” entries can be any data type and length. The elements must be quoted if they contain a space.

Just like an array we designate a hash table with the @ symbol. There are differences, array elements are enclosed in parenthesis () where hash “key-value” pairs are enclosed in curly brackets {}. Hash elements are separated by the semi-colon.

Creating a hash of users associated with their employee number:

$EmpNumbers = @{“John Doe” = 112233; “Dave Davis” = 223344; “Justine Smith” = 334455}<enter>

Output the hash table

$EmpNumbers<enter>

To **Remove** all records in the hash table, we use the **clear**method:

$EmpNumbers.Clear()<enter>

## LOOPS

In this section we will examine some loops in Powershell

* **do while**
* **while**
* **do until**
* **for**
* **foreach**

**do while and while**

**do while**and **while**Loops continue to execute while the condition is true

The Do-While runs the code block at least once. The While loop checks the condition before entering even once.

**do while**  
do

{code block}  
while (condition)

**while**while (condition) {code block}

**Enter the following in your Powershell ISE**

**do while – Count to 5**

$i = 1

do {Write-Host $i; $i++}  
while ($i -le 5)

$i = 1

do {  
Write-Host $i  
$i++  
}  
while ($i -le 5)

**Using the Do While as a VERY primitive process monitor…**

$processes = get-process

write-output $processes

do {

Start-Sleep -s 5;

$rTime = get-date

Write-Host “The Application Notepad is working at: $rTime”

}

While (get-process notepad++ | select -Property Responding)

write-output "Notepad++ is stopped....."

$strTime = get-date

Write-Host “The Application Notepad failed to respond on: $strTime”

Try out the above script in Powershell ISE. Add a comment to each line to explain what is going on.

Before running the script Start Notepad++ on your machine and then start the script

While the script is running close down NotePad++

One issue with the above script is that the process Notepad is initially reported as running at least once even if its not running.

Re-write the script to use a while loop instead

$processes = get-process

write-output $processes

While (get-process notepad++ | select -Property Responding)

{

Start-Sleep -s 5;

$rTime = get-date

Write-Host “The Application Notepad is working at: $rTime”

}

write-output "Notepad++ is stopped....."

$strTime = get-date

Write-Host “The Application Notepad failed to respond on: $strTime”

**for**

The standard use of the **for** statement it to run the code block a specified number of times.

Syntax:  
for (initialization; condition; repeat)  
{code block}

**Note:** each of the three parameters are optional.

* **initialization** -  one or more commands, separated by commas, that run before the loop begins. Commonly used to create and initialize a variable with a starting value. This variable is the basis for the condition to be tested in the next portion of the **for**statement.
* **Condition** – Resolves to a Boolean **True**or **False**value. PowerShell evaluates the condition each time the loop runs. If the condition is **True** the code block is executed, then the condition is tested again.
* **Repeat** – one or more commands, separated by commas, that run each time the loop repeats. Commonly used to modify a variable that is tested inside the “condition” portion of the **for**statement.

**Example:**

for ($i=1; $i -le 5; $i++)  
{Write-Host $i}

**foreach**

The **foreach**loop description from the PowerShell Help file:

The foreach statement (also known as a foreach loop) is a language construct for stepping through (iterating) a series of values in a collection of items.

The simplest and most typical type of collection to traverse is an array. Within a foreach loop it’s common to run one or more commands against each item in an array.

**Note:**Those writing VBScript code will have to get use to **not** using **“Next”**as PowerShell doesn’t require the keyword to move to the next item in the collection.

**Syntax:**foreach ($<item> in $<collection>)  
{code block}

Sticking with our number example, let’s see how the **foreach** loop works

$ints = @(1, 2, 3, 4, 5)

foreach ($i in $ints)  
{Write-Host $i}

**Output:  
1  
2  
3  
4  
5**

Looks a little like the **for**loop example presented earlier. However, notice that we are not testing a condition. The **foreach**loop runs the script block against each element within the array. It doesn’t required testing a condition and it doesn’t care how many elements exist in the array. You point it and it runs. As stated in the help file, **foreach**loop is the most typical statement used when working will arrays and collections.

I have presented many examples of **foreach** loops on this site. Most of the Microsoft PowerShell Scripts in the Scripting Center use this loop.

**Real-World Examples:**

How about listing processor information.

$strComputer = “.”

$colItems = get-wmiobject -class “Win32\_Processor” -namespace “root\CIMV2″ `  
-computername $strComputer

foreach ($objItem in $colItems) {  
write-host “Caption: ” $objItem.Caption  
write-host “CPU Status: ” $objItem.CpuStatus  
write-host “Current Clock Speed: ” $objItem.CurrentClockSpeed  
write-host “Device ID: ” $objItem.DeviceID  
write-host “L2 Cache Size: ” $objItem.L2CacheSize  
write-host “L2 Cache Speed: ” $objItem.L2CacheSpeed  
write-host “Name: ” $objItem.Name  
write-host  
}

Listing disk information

$strComputer = “.”

$colItems = get-wmiobject -class “Win32\_DiskDrive” -namespace “root\CIMV2″ `  
-computername $strComputer

foreach ($objItem in $colItems) {  
write-host “Description: ” $objItem.Description  
write-host “Device ID: ” $objItem.DeviceID  
write-host “Interface Type: ” $objItem.InterfaceType  
write-host “Media Type: ” $objItem.MediaType  
write-host “Model: ” $objItem.Model  
write-host “Partitions: ” $objItem.Partitions  
write-host “Size: ” $objItem.Size  
write-host “Status: ” $objItem.Status  
write-host  
}

Let’s say we want to determine which processes are running on our computer.

foreach ($item in Get-Process)  
{if ($item.Responding -eq “True”){Write-Host $Item.Name}}

Just showing that we can add the conditional logic**if**statement within the foreach loop to give us more control over the data we wish to retrieve.

Notice the code block – {{}} – what I’ve done is called nesting. Perfectly legal and a concept that we will discuss as you progress through these PowerShell Training sessions.

**Additional Reading:**

Get-Content $PSHome\about\_foreach.help.txt