**Basic PowerShell info to share with the team, and help with training.**

**Usefull URL’s:**

* Basic PS info
  + <https://technet.microsoft.com/en-us/library/bb963733.aspx>
* Strings
  + <https://technet.microsoft.com/en-us/library/ee692804.aspx>
* Arrays
  + <https://technet.microsoft.com/en-us/library/ee692791.aspx>
  + <https://technet.microsoft.com/en-us/library/ee692797.aspx>
* Hash Tables / Dictionaries
  + <https://technet.microsoft.com/en-us/library/ee692803.aspx>
* PS Modules
  + <https://www.simple-talk.com/sysadmin/powershell/an-introduction-to-powershell-modules/>
* Hey, Scripting Guy!
  + <https://blogs.technet.microsoft.com/heyscriptingguy/?m=20103>

**Basic stuff:**

PowerShell (PS) has an IDE as part of the basic install (Start 🡪 All Programs 🡪 Accessories 🡪 Windows PowerShell 🡪 Windows PowerShell ISE). I use Notepad++ (https://notepad-plus-plus.org/) and a regular PS window (Start 🡪 All Programs 🡪 Accessories 🡪 Windows PowerShell 🡪 Windows PowerShell). Visual Studio (VS) 2012+ has support for PS, so on the HPE side I tend to use it, especially for a project that has a GUI.

PS by default is configured with a fairly tight security posture. So the first thing to do is enable the ability to run scripts we write, with out having to jump through extra hoops. Open a PS window as admin (Shift + Right Click 🡪 Run as Admin). Once the window is open enter the following command: “Set-ExecutionPolicy RemoteSigned -Force”.

Options for the Set-ExecutionPolicy are as follows:

* Restricted: Does not load configuration files or run scripts. -- Default--
* AllSigned: Requires that all scripts and configuration files be signed by a trusted publisher, including scripts that you write on the local computer.
* RemoteSigned: Requires that all scripts and configuration files downloaded from the Internet be signed by a trusted publisher.
* Unrestricted: Loads all configuration files and runs all scripts. If you run an unsigned script that was downloaded from the Internet, you are prompted for permission before it runs.
* Bypass: Nothing is blocked and there are no warnings or prompts.
* Undefined: Removes the currently assigned execution policy from the current scope. This parameter will not remove an execution policy that is set in a Group Policy scope.

PS, simular to a \*unix environment, has built in help. If you need help with any command preface it with “Get-Help”, i.e. “Get-Help Set-ExecutionPolicy”. There are some switches you can use with this command, and when you use it, the interface usually always will let you know what they are.

All most all of the DOS commands you can run from a command window can be run in a PS window, I have even found a few \*unix commands that work too. In pure PS to list the contents of a folder one would do “Get-ChildItems c:\users”, but “dir c:\users\” or “ls c:\users” work the same way too. Arrow keys work, like in a DOS window, to recall previous commands entered.

A lot, if not all, of the pure PS commands have “aliases” for each command, and that is what allows the above to work. Both “dir” and “ls” are aliases for the “Get-ChildItems” PS command, as is “gci”.

PS can also take advantage of .NET code and methods i.e. “[System.DateTime]::Now”. With a wrapper PS can also take code directly from C# and process it too. PS commands typically are not written with a semicolon “;” at the end, but PS allows it. In any C# code used, in a wrapper, the semicolons are required. For this reason I have made it a habit to use semicolons at the end of all my code. Also this is usually the standard for most “real” languages anyway, so it is a good habit to be in.

**Some initial commands:**

To output to the screen typically one will use “Write-Host” i.e. [Write-Host “Hello World.”]. The following will provide the same results [Write-Host ‘Hello World.’]. The difference between the quote [“] and a single tick [‘], is that PS will interpret/translate variables in quotes, but NOT in ticks. Ticks are used when you want a “literal” string to display exactly as you typed it. We can also display the current date and time using a .NET method. { Write-Host ([System.DateTime]::Now);}

Variables in PS MUST begin with a dollar sign “$”, i.e. “$strMessage” or “$sComputer” or “$FirstName”.

Lets make our initial command a little more variable, by entering the following commands:

$strMess = “World”; Write-Host “Hello $strMess”;

So we should get the same results as the first time we ran it. Now try the Write-Host with ticks i.e. [Write-Host ‘Hello $strMess’;]. The output should be different than what we have been getting.

Lets try adding the date and time to the “Hello World” output, on your own....

Did you figure it out? Write-Host “Hello $strMess” ([System.DateTime]::Now);

That date time is a mess to type, so lets put it in a variable. “$strTime = [System.DateTime]::Now;”.

Now output the date time with “Hello World” using the variable. Write-Host “Hello $strMess $strTime”;

**Array basics:**

To define a new, blank, array: $arrTest1 = @(); (those are parens)

To add an element to the array: $arrTest1 += “Value”;

To display a certain element of the array: $arrTest1[#]; ~~ i.e. $arrTest1[0];

To display the last element of an array: $arrTest1[-#]; ~~ i.e. $arrTest1[-1];

**Types of “blocks” that are available:**

If, if else, do while, do until, for, foreach, switch.

* if ($intX -lt 5){Write-Host “Less”;}else{Write-Host “More”;}
* for ($intX = 0; $intX -lt 5; $intX++){Write-Host $intX;}
* do{$intX++;} while ($intX -lt 10);
* foreach ($strEntry in $arrTest1){Write-Host $strEntry;}

**Hash / Dictionary basics:**

To define a new, blank, hash: $hashTest2 = @{}; (those are curly brackets)

To define a new hash: $hashTest2 = @{"Key1"="Val1"; "Key2"="Val2"};

To add an element to the hash: $hashTest2.Add("Key3", "Val3");

To display a certain element of the hash: $hashTest2.Key1;

**Misc Info:**

To see the methods and properties available on a variable/object use: $strVariable | Get-Member;

To get an array size do the following: $arrTest1.Count;

Comments are started with a pound sign “#”.

The exclamation “!” is used for “NOT”. if (!($intX -lt 5)){Write-Host “More”;}else{Write-Host “Less”;}

Comparisons in PS 1 & 2 are:

* -eq
* -ne
* -gt
* -lt
* -ge
* -le

Comparisons in PS 3+ are:

* -eq or =
* -ne or !=
* -gt or >
* -lt or <
* -ge or =>
* -le or <=

**NMCI Path to the ITSS-Tools location:**

[\\nawesdnifs101v.nadsuswe.nads.navy.mil\NMCIISF02$\ITSS-Tools\](file:///\\nawesdnifs101v.nadsuswe.nads.navy.mil\NMCIISF02$\ITSS-Tools\)