POWER SHELL SCRIPTING

Introduction to PowerShell

- Windows PowerShell is a command-line interface and task-based scripting technology that is built into the Windows Server 2012 operating system.
- Windows PowerShell simplifies the automation of common systems administration tasks.
- With Windows PowerShell, you can automate tasks, leaving you more time for more difficult systems administration tasks.

What Is Windows PowerShell?

- Windows PowerShell is a scripting language and command-line interface that is designed to assist you in performing day-to-day administrative tasks.
- Windows PowerShell is made up of cmdlets that you execute at a Windows PowerShell command prompt, or combine into Windows PowerShell scripts.
- Unlike other scripting languages that were designed initially for another purpose, but have been adapted for system administration tasks, Windows PowerShell is designed with system administration tasks in mind.

CONTD.....

- You can extend Windows PowerShell functionality by adding modules.
- For example, the Active Directory module includes Windows PowerShell cmdlets that are specifically useful for performing Active Directory–related management tasks.
- The DNS Server module includes Windows PowerShell cmdlets that are specifically useful for performing DNS server-related management tasks.
- Windows PowerShell includes features such as tab completion, which allows administrators to complete commands by pressing the tab key rather than having to type the complete command.
- You can learn about the functionality of any Windows PowerShell cmdlet by using the **Get-Help cmdlet.**

Windows PowerShell Cmdlet Syntax

- Get
- New
- Set
- Restart
- Resume
- Stop
- Suspend

- Clear
- Limit
- Remove
- Add
- Show
- Write

- Windows PowerShell parameters start with a dash.
- Each Windows PowerShell cmdlet has its own associated set of parameters.
- You can learn what the parameters are for a particular Windows PowerShell cmdlet by executing the following command:

  Get-Help *CmdletName*

Common Cmdlets for Server Administration

Service Cmdlets

- **Get-Service -** View the properties of a service.
- **New-Service -** Creates a new service.
- **Restart-Service -** Restarts an existing service.
- **Resume-Service -**Resumes a suspended service.
- **Set-Service -** Configures the properties of a service.
- **Start-Service -** Starts a stopped service.
- **Stop-Service -** Stops a running service**.**
- **Suspend-Service -**Suspends a service.

Event Log Cmdlets

- **Get-EventLog -***Displays events in the specified event log.*
- **Clear-EventLog -** Deletes all entries from the specified event log.
- **Limit-EventLog -** Sets event log age and size limits.
- **New-EventLog -** Creates a new event log and a new event source on a computer running Windows Server 2012.
- **Remove-EventLog -** Removes a custom event log and unregisters all event sources for the log.
- **Show-EventLog -** Shows the event logs of a computer.
- **Write-EventLog -** Allows you to write events to an event log.

Common Cmdlets for Server Administration

Process Cmdlets

- **Get-Process**- Provides information on a process.
- **Start-Process**-Starts a process.
- Stop-Process-Stops a process.
- **Wait-Process**-Waits for the process to stop before accepting input.
- **Debug-Process**-Attaches a debugger to one or more running processes.

ServerManager Module

- **Get-WindowsFeature** - View a list of available roles and features
- **Install-WindowsFeature** - Installs a particular Windows Server role or feature..
- **Remove-WindowsFeature** - Removes a particular Windows Server role or feature.

What Is Windows PowerShell ISE?

- Windows PowerShell ISE is an **Integrated scripting environment** that provides you with assistance when using Windows PowerShell.
- It provides command completion functionality, and allows you to see all available commands and the parameters that you can use with those commands.

To create a new script file

- On the toolbar, click **New** , or on the **File** menu, click **New**.
- The created file appears in a new file tab under the current PowerShell tab.
- Remember that the PowerShell tabs are only visible when there are more than one.
- By default a file of type script (.ps1) is created, but it can be saved with a new name and extension.
- Multiple script files can be created in the same PowerShell tab.

10 fundamental concepts for PowerShell scripting

1: PS1 files

- A PowerShell script is really nothing more than a simple text file.
- The file contains a series of PowerShell commands, with each command appearing on a separate line.
- For the text file to be treated as a PowerShell script, its filename needs to use the .PS1 extension.

2: Execution permissions

- By default, the execution policy is set to Restricted, which means that PowerShell scripts will not run.
- You can determine the current execution policy by using the following cmdlet:

Get-ExecutionPolicy

- **Restricted** - Scripts won't run.
- **RemoteSigned** - Scripts created locally will run, but those downloaded from the Internet will not (unless they are digitally signed by a trusted publisher).
- **AllSigned** - Scripts will run only if they have been signed by a trusted publisher.
- **Unrestricted** - Scripts will run regardless of where they have come from and whether they are signed.
- You can set PowerShell's execution policy by using the following cmdlet:

Set-ExecutionPolicy <policy name>

3: Running a script

- If you want to execute a PowerShell script, you will usually have to type the full path along with the filename. For example, to run a script named SCRIPT.PS1, you might type:

  C:\Scripts\Script.ps1

- Instead of typing the script's full path in such a situation, you can enter .\ and the script's name. For example, you might type:

  .\Script.ps1

4: Pipelining

- Pipelining is the term for feeding one command's output into another command. This allows the second command to act on the input it has received. To pipeline two commands (or cmdlets), simply separate them with the pipe symbol (|).
- Imagine that you want to create a list of processes that are running on a server and sort that list by process ID number. used looks like this:

Get-Process | Sort-Object ID

5: Variables

- Although you can use pipelining to feed one command's output into another command, sometimes pipelining alone won't get the job done.
- When you pipeline a command's output into another command, that output is used immediately. Occasionally, you may need to store the output for a while so that you can use (or reuse) it later.
- you want to store the list of processes running on a server as a variable. To do so, you could use this line of code:

```
$a = Get-Process
```

- Here, the variable is named $a. If you want to use the variable, simply call it by name. For example, typing *$a* prints the variable's contents on the screen.
- You can assign a variable to the final output of multiple commands that have been pipelined together

```
$a = (Get-Process | Sort-Object ID)
```

6: The @ symbol

- By using the @ symbol, you can turn the contents of a list into an array. For example, take the following line of code, which creates a variable named $Procs that contains multiple lines of text (an array):

```
$procs = @{name="explorer","svchost"}
```

- You can also use the @ symbol when the variable is used, to ensure that it is treated as an array rather than a single value.
- For instance, the line of code below will run the Get-Process cmdlet against the variable I defined a moment ago. In doing so, Windows will display all the processes used by Windows Explorer and Svchost.
- Notice how the @ symbol is being used in front of the variable name rather than the dollar sign that we usually see used:

```
Get-Process @procs
```

7: Split

- The split operator splits a text string based on a character you designate.
- You could do so by using a command like this one:

"This is a test" -split " "

The result would look like this:

This

is

a

test

8: Join

- Just as split can split a text string into multiple pieces, the join operator can combine multiple blocks of text into one. For example, this line will create a text string consisting of my first name and last name:
- "Brien","Posey" -join " "The space between the quotation marks at the end of the command tells Windows to insert a space between the two text strings.

9: Breakpoints

- Running a newly created PowerShell script can have unintended consequences if the script contains bugs.
- One way to protect yourself is to insert breakpoints at strategic locations within your script. That way, you can make sure that the script is working as intended before you process the entire thing.
- The easiest way to insert a breakpoint is by line number. For instance, to insert a break point on the 10th line of a script, you could use a command like this:

  New-PSBreakpoint -Script C:\Scripts\Script.ps1 -Line 10You can also bind a breakpoint to a variable. So if you wanted your script to break any time the contents of a$ changed, you could use a command like this one:
- There are a number of verbs you can use with PSBreakpoint including New, Get, Enable, Disable, and Remove.

10: Step

- When debugging a script, it may sometimes be necessary to run the script line by line.
- To do so, you can use the Step-Into cmdlet. This will cause the script to pause after each line regardless of whether a breakpoint exists.
- When you are done, you can use the Step-Out cmdlet to stop Windows from stepping through the script.
- It is worth noting, however, that breakpoints are still processed even after the Step-Out cmdlet has been used.
- Step-Over works just like Step-Into, except that if a function is called, Windows won't step through it. The entire function will run without stopping.

Operators

$( ) @( ) :: &

Operators

1. ( ) Expression operator.
2. $( ) SubExpression operator.
3. @( ) Array SubExpression operator.
4. :: Static member operator
5. , Comma operator
6. & Call operator
7. . Dot sourcing operator
8. -f Format operator
9. .. Range operator

( ) Expression operator

- Parenthesis/Brackets work just as they do in mathematics, each pair will return the result of the expression within.
  PS C:\> (2 + 3) * 5

$( ) SubExpression operator

- Use a subexpression to return specific properties of an object. Unlike simple parenthesis, a subexpression can contain multiple ; semicolon ; separated ; statements.
- The output of each statement contributes to the output of the subexpression. For a single result, it will return a scalar. For multiple results, it will return an array.
- Subexpressions allow you to evaluate and act on the results of an expression in a single line; with no need for an intermediate variable:

  if(**$(***code that returns a value/object***)** -eq "*somevalue*") { *do_something* }

  PS C:\> $city="Copenhagen"
  PS C:\> $strLength = "**$(**$city.length**)**" #n.b. not "$city.length" that would return "Copenhagen.Length"

  PS C:\> "The result of 2 + 3 = **$(**2+3**)**"
  PS C:\> **$(**Get-WMIObject win32_Directory**)**

@( ) Array SubExpression operator.

- An array subexpression behaves just like a subexpression except that it guarantees that the output will be an array.

- This works even if there is no output at all

  PS C:\> **@(**Get-WMIObject win32_logicalDisk**)**

  "$user.department" ==> JDOE.department

  "$($user.department)" ==>"HumanResources"

:: Static member operator

- Call the static properties operator and methods of a .NET Framework class.
  To find the static properties and methods of an object, use the -Static parameter of Get-Member:

```
[datetime] | gm -static
[datetime]::now
[datetime]::Utcnow
```

, Comma operator

- As a binary operator, the comma creates an array.
  As a unary operator, the comma creates an array with one member. Place the comma before the member.

& Call operator

- Run a command, script, or script block. The call operator, also known as the "invocation operator," lets you run commands that are stored in variables and represented by strings. Because the call operator does not parse the command, it cannot interpret command parameters.

```
C:\PS> $c = "get-executionpolicy"
C:\PS> $c
get-executionpolicy
C:\PS> & $c
AllSigned
```

. Dot sourcing operator

- Run a script in the current scope so that any functions, aliases, and variables that the script creates are added to the current scope.

```
. C:\sample1.ps1
. .\sample2.ps1
```

-f Format operator

- Place {0} {1} etc. into the string as place markers where you want the variables to appear, immediately follow the string with the -f operator and then lastly, the list of comma separated variables which will be used to populate the place markers.

  Get-ChildItem c:\ | ForEach-Object {'File {0} Created {1}' **-f** $_.fullname,$_.creationtime}
- Optional format string(s) can be included to add padding/alignment and display dates/times/percentages/hex etc correctly.

..Range operator

Produce a sequence of numbers:

10..20

5..25

Special character Meaning

- " The beginning (or end) of quoted text
- # The beginning of a comment
- $ The beginning of a variable
- & Reserved for future use
- ( ) Parentheses used for subexpressions
- ; Statement separator
- { } Script block
- | Pipeline separator
- ` Escape character