IT Scripting and Automation

Introduction to PowerShell

Lecturer: Art Ó Coileáin

PowerShell

Versions:

- PowerShell V5 –Windows 10 and Server 2016
- PowerShell V4 –Windows 8.1 and Server 2012 R2
- PowerShell V3 –Windows 8 and Server 2012
- PowerShell V2 –Windows 7 and Server 2008
- Windows XP and Server 2003 can run V2

- Windows Management Framework 4.0
  - http://www.microsoft.com/en-ie/download/details.aspx?id=40855
  - Systems supported: Windows 7, Windows Embedded Standard 7, Windows Server 2008 R2, Windows Server 2012

- Windows Management Framework 5.0
  - https://www.microsoft.com/en-us/download/details.aspx?id=45883
  - Windows Server 2012, 2012 R2, 8.1 Pro, 8.1 Enterprise

What is PowerShell ?

General definition:

PowerShell is an object based management engine based on the .NET Framework. This engine is exposed through a series of commands called cmdlets and providers.

- PowerShell can be used through an:
  a) **interactive console** or by
  b) a **batch-oriented scripting language**
- Can be used as a task automation framework
- It is a scripting language based on .NET framework

Continues

What is PowerShell ?

- Allows one access to administrative tasks:
  - both locally and
  - remotely
- PowerShell is sitting on top of CMD window, with all the benefits and drawbacks that entails.
- PowerShell can also be hosted in a GUI (PowerShell ISE).

What PowerShell is not?

- PowerShell is not another scripting language like VBScript.
  - Although It allows you to create powerful scripts using PowerShell's simple scripting language (***but you do not have to use a script to use PowerShell***).
- PowerShell is not a programming language although it is built on the .NET Framework.
- Using PowerShell does not mean that you will be spending all day typing commands at a command prompt.
  - There are graphical tools, but they run on top of PowerShell. There are limitations in respect to the GUI (*related to the nature of its design*).

Cmdlets

- Pronounced “command-let”
- The heart of PowerShell
- Verb-Noun
  - Get-ChildItem
  - Set-Date
- The verbs is from a list of standard .NET verbs, like Set, Remove or Get.
- Installed via PowerShell modules
- Cmlets are compiled commands written in a .NET Framework language.

Cmdlets

- The Noun is a singular version of the “thing” you want to work with like Service, Process or EventLog.
- Cmlets are small and single purpose.
- Usage: Load module to use cmdlets
- The available modules are based on Server role and features;

For example:

- File Services
- Print and Document Services
- Web Server (IIS)

Parameter

- Cmdlet behaviour can be customised by the use of parameters.
- The syntax when using a parameter is a dash (-), immediately followed by the parameter name a space and finally the value.

```
PS C:\> Get-Service -Name browser

Status   Name               DisplayName
------   ----               -----------
Running  browser            Computer Browser
```

- Alternatively, you can use the parameter shortcuts and save some typing. Because **Get-Service** has no other parameter that starts with 'N', it know you mean 'Name'.

```
PS C:\> Get-Service -N browser

Status   Name               DisplayName
------   ----               -----------
Running  browser            Computer Browser
```

Alias

- An alias is an alternative command name.

For example, in Linux systems the ***ps*** command for listing processes.

- The same command is an alias for the **Get-Process** cmdlet.

```
PS C:\> get-process
PS C:\> ps
```

More examples:

Set-location c:\,	Get-childitem,	Clear-host
cd \,	dir,	cls
cd \,	ls,	clear

Alias (2)

- Many aliases are built-in to PowerShell. (Use **Get-alias** for listing)
- You can create your own aliases for any command, even non-PowerShell commands.

Example:

PS C:\> Set-Alias -Name np -Value Notepad.exe

- Aliases will only exist for as long as your PowerShell session is open. Or you can use a PowerShell profile script

Exercise: Read: **PS C:\> help about_profiles** or

- https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_profiles?view=powershell-6

- Windows native commands: notepad, calc, mspaint, ping

Help

- Dir command -> Get-ChildItem

Get-Help Get-ChildItem

- You may need to … Update-Help

PowerShell ISE

Power Shell Integrated Scripting Environment (ISE):

- Command shell
- Command Builder
- Integrated help
- Debugger

Execution Policy

- Controls what the commands shell allows
- Prevent accidental execution of malicious scripts
- By default is restricted
- Type of Policies:
  - **Restricted**. No scripts are executed. This is the default setting.
  - **AllSigned**. This policy allows scripts signed by a trusted publisher to run.
  - **RemoteSigned**. This policy requires remote scripts to be signed by a trusted publisher.
  - **Unrestricted**. This policy allows all scripts to run. It will still prompt for confirmation for files downloaded from the internet.

Continues…

Execution Policy (2)

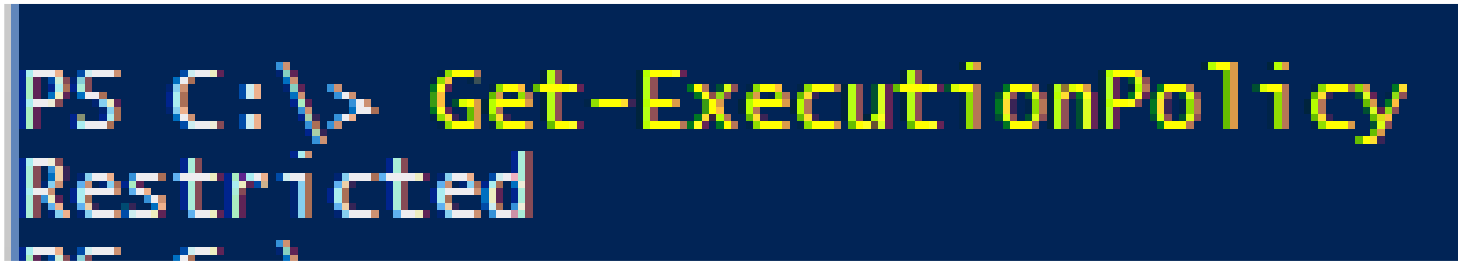
- **Bypass**. This policy allows all scripts to run and will not prompt.
- **Undefined**. This policy resets the policy to the default.

Execution Policy (3)

- Process
- CurrentUser
- LocalMachine (default)

- We will view the system's current execution policy and change it to suit various needs. To do this, carry out the following steps:

1. To find the system's current execution policy, open PowerShell and execute **Get-ExecutionPolicy**.



Execution Policy (4)

2. To change the system's execution policy,

   run **Set-ExecutionPolicy <policy name>** command.

```
PS C:\WINDOWS\system32> Set-ExecutionPolicy AllSigned

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose
you to the security risks described in the about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"): Y
PS C:\WINDOWS\system32> Get-ExecutionPolicy
AllSigned
```

3. To reset the execution policy to the system default, set the policy to **Undefined**.

```
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose
you to the security risks described in the about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"): yes
PS C:\WINDOWS\system32> Get-ExecutionPolicy
Restricted
```

Execution Policy (5)

- To change the execution policy for a specific session, go to ***Start | Run*** and enter

  PowerShell.exe -ExecutionPolicy <policy name>.

- When a script is executed, the first thing PowerShell does is, determine the system's execution policy.
- ***By default***, this is set to **Restricted**, which blocks all the PowerShell scripts from running.
- If the policy allows signed scripts, it analyses the script to confirm it is signed and that the signature is from a trusted publisher.
- If the policy is set to unrestricted, then all the scripts run without performing checking.

Creating and Using Functions

- Functions allow for individual commands or groups of commands and variables to be packaged into a single unit.
- These units are **reusable** and can then be accessed similar to native commands and Cmdlets, and are used to perform larger and more specific tasks.
- Unlike Cmdlets, which are precompiled, **functions** are interpreted at runtime.
- This increases the runtime by a small amount (*due to the code being interpreted by the runtime when executed*), but its performance impact is often outweighed by the flexibility that the scripted language provides.

Creating and Using Functions (2)

Example:

Being PowerShell learners, we have decided to use PowerShell to tell us how many days there are until Christmas.

- We start by identifying the necessary PowerShell commands to determine the number of days until Christmas.

```
PS C:\> $Christmas=Get-Date("12/25/" + (Get-Date).Year.ToString() + " 7:00 AM")

PS C:\> $Today = (Get-Date)

PS C:\> $TimeTilChristmas = $Christmas - $Today

PS C:\> Write-Host $TimeTilChristmas.Days " Days 'til Christmas"
94  Days 'til Christmas

PS C:\>
```

Creating and Using Functions (3)

Example continued:

- Next, we combine the commands into a function:

```
Function Get-DaysTilChristmas
{
<#          .Synopsis
            This function calculates the number of days until Christmas
            .Description
            This function calculates the number of days until Christmas
            .Example
            DaysTilChristmas
#>
$Christmas=Get-Date("25 Dec " + (Get-Date).Year.ToString() + " 7:00 AM")
$Today = (Get-Date)
$TimeTilChristmas= $Christmas -$Today
Write-Host $TimeTilChristmas.Days"Days 'til Christmas"
}
```

Creating and Using Functions (4)

- Once the function is created, we either type it or copy/paste it into a PowerShell console.

- Finally, we simply call the function by the name,

```
Get-DaysTilChristmas
```

- At its simplest, a function is composed of the **Function** keyword, a function name, and commands encapsulated in curly braces:

```
Function FunctionName{
        # commands go here
}
```

Creating and Using Functions (5)

Function scope:

- Custom functions are traditionally limited to the currently active user session.
- If you create a function such as **Get-DaysTilChristmas**, and then open a new PowerShell window, the function will not be available in the new session, even though it is still available in the original session.
- Additionally, if you close your original session, the function will be removed from the memory and won't be available until it is re-entered.

Creating and Using Functions (6)

Variable types:

- It may be interesting to note that the variables *$Christmas* and *$Today* are of different types than *$TimeTilChristmas*.
  - The first two are ***date*** and ***time*** variables which refer to a specific point in history (year, month, day, hour, minute, second, millisecond, ticks).
- *$TimeTilChristmas* however is a ***time span***; which refers to a ***length*** of time (day, hour, minute, second, millisecond, ticks), relative to a specific time.
- The type of a variable can be viewed by typing **$<variableName>.GetType()** as shown in the following screenshot:

```
PS C:\temp> $Today.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DateTime                                 System.ValueType
```

Creating and Using Functions (7)

- **Returning content**:
  - The *example function* in its current form returns the number of days until Christmas, but that is all. Because the function uses date and time variables, it can easily include the number of hours, minutes, and seconds as well.
  - See **Get-Date | Get-Member** for a list of properties that can be accessed.

- **Naming of functions and commands in PowerShell**:
  - Commands in PowerShell are traditionally named in a verb-noun pair, and for ease of use, a similar process should be used when naming custom functions.

Creating and Using Modules

Modules:

- **Modules** are a way of grouping functions for similar types of tasks or components into a common place.
- These modules can then be loaded, used, and unloaded together as needed.
- Modules are *similar in concept to libraries* in the Windows world - they are used to contain and organise tasks, while allowing them to be added and removed dynamically

Creating and Using Modules (2)

An **example** of a module is working with the DNS client:

- When working with the DNS client, you will have various tasks to perform: *get configuration, set configuration, resolve hostname, register client*, and so on.
- All of these tasks have to do with a common component, the DNS client, they can be logically grouped together into the **DNSClient** module. We can then view the commands included in the module using:

Get-Command –Module DnsClient

```
PS C:\> Get-Command -Module DnsClient

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Add-DnsClientNrptRule                              DnsClient
Function        Clear-DnsClientCache                               DnsClient
Function        Get-DnsClient                                      DnsClient
Function        Get-DnsClientCache                                 DnsClient
Function        Get-DnsClientGlobalSetting                         DnsClient
```

Creating and Using Modules (3)

Module Example:

1. we will be creating a module named ***Hello***.
2. Create several functions that can be logically grouped together:

```
Function Get-Hello
{
        Write-Host "Hello World!"
}
Function Get-Hello2
{
        Param($name)
Write-Host "Hello $name"
}
```

Creating and Using Modules (4)

3. Using the PowerShell ISE or a text editor, save the functions into a single file name **Hello.PSM1**
4. If the folder for the module doesn't exist yet, create the folder

```
PS C:\Users\Administrator\Desktop\ITSA> $modulePath= "$env:USERPROFILE\Documents\WindowsPowerShell\Modules\Hello"
PS C:\Users\Administrator\Desktop\ITSA> if(!(Test-Path $modulePath))
>> {
>> New-Item -Path $modulePath -ItemType Directory
>> }


    Directory: C:\Users\Administrator\Documents\WindowsPowerShell\Modules


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        03/12/2020     17:25                Hello
```

```
$modulePath= "$env:USERPROFILE\Documents\WindowsPowerShell\Modules\Hello"
if(!(Test-Path $modulePath))
    { New-Item -Path $modulePath -ItemType Directory }
```

5. Copy **Hello.PSM1** to the new module folder.

$modulePath= "$env:USERPROFILE\Documents\WindowsPowerShell\Modules\Hello"

Copy-Item -Path Hello.PSM1 -Destination $modulePath

Creating and Using Modules (5)

6. In a PowerShell console, execute **Get-Module -ListAvailable** to list all the available modules:

```
PS C:\> Get-Module -ListAvailable

    Directory: C:\Users\Ed\Documents\WindowsPowerShell\Modules

ModuleType Name                                ExportedCommands
---------- ----                                ----------------
Script     Hello                               {Get-Hello, Get-Hello2}
```

7. Run **Import-Module Hello** to import our new module.
8. Run **Get-Command –Module Hello** to list the functions included in the module.
9. Execute the functions in the module as normal:

```
PS C:\> Get-Hello
Hello World!

PS C:\> Get-Hello Ed
Hello World!

PS C:\> Get-Hello2 Ed
Hello Ed
```

Creating and Using Modules (6)

Auto-loading of modules:

- From PowerShell 3.0 automatically imports modules as they are needed.
- While it is best practice to load and unload modules, you do not necessarily have to use Import-Module prior to accessing the functions contained within.

Example:

- As can be seen in the following screenshot, the currently loaded modules are listed using Get-Module.
- Once it is confirmed the new Hello module was not loaded; the Get-Hello2 function is executed in the *module*, which completed successfully.

Continues….

Creating and Using Modules (6)

Example continued:

- Executing Get-Module again shows the module has been automatically loaded.

```
PS C:\> Get-Module

ModuleType Name                                     ExportedCommands
---------- ----                                     ----------------
Script     ISE                                      {Get-IseSnippet, Import-IseSnippet...
Manifest   Microsoft.PowerShell.Management          {Add-Computer, Add-Content, Checkp...
Manifest   Microsoft.PowerShell.Utility             {Add-Member, Add-Type, Clear-Varia...



PS C:\> Get-Hello2 Ed
Hello Ed

PS C:\> Get-Module

ModuleType Name                                     ExportedCommands
---------- ----                                     ----------------
Script     Hello                                    {Get-Hello, Get-Hello2}
Script     ISE                                      {Get-IseSnippet, Import-IseSnippet...
Manifest   Microsoft.PowerShell.Management          {Add-Computer, Add-Content, Checkp...
Manifest   Microsoft.PowerShell.Utility             {Add-Member, Add-Type, Clear-Varia...
```

Creating and Using Modules (7)

Module manifest:

- In addition to the modules themselves, you can also create a module manifest. A module manifest is a file with a **.PSD1** extension that **describes the contents of the module**.
- Manifests can be useful because they allow for defining the environment in which a module can be used, its dependencies, additional help information, and even which set of commands to make available.
- The following code is a basic example of creating a manifest for our Hello World module:

```
New-ModuleManifest-Path "$env:USERPROFILE\Documents\
WindowsPowerShell\Modules\Hello\Hello.PSD1" -Author “John Smith" -
Description "Hello World examples" -HelpInfoUri"http://blog.com" -
NestedModules'Hello.PSM1'
```

- Once the manifest is created, we can view the manifest properties using the following code:

```
Get-Module Hello -ListAvailable| Format-List -Property *
```

IF statements in PS

Syntax:

```
If (condition) {Do stuff}

# Another explanation would be
If (test) {
"Execute when true“
}
```

Example:

```
# PowerShell If Statement Simple Example
$Number = 10
If ($Number -gt0) {
        Write-Host "Bigger than zero“
}
```