IT Scripting and Automation

PowerShell Scripting

Lecturer: Art Ó Coileáin

Creating and Using PowerShell profiles

- User profiles are used to set up user customized PowerShell sessions.
- These profiles can be blank, contain aliases, custom functions, load modules, or any other PowerShell tasks.
- When you open a PowerShell session, the contents of the profile are executed the same as executing any other PowerShell script.
- Open the PowerShell console (***not the ISE***) and list your current profile locations by executing:

$PROFILE or **$PROFILE | Format-List * -Force**

```
PS C:\WINDOWS\system32> $PROFILE
C:\Users\Nando\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
PS C:\WINDOWS\system32> $PROFILE | Format-List * -Force


AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost    : C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\Nando\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost : C:\Users\Nando\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Length                 : 75



PS C:\WINDOWS\system32>
```

Creating and Using PowerShell profiles

- If the **CurrentUserCurrentHost** profile file does not already exist, create the folder and file structure:

```
$filePath = $PROFILE.CurrentUserCurrentHost
if(!(Test-Path $filePath))
{
New-Item -Path $filePath -ItemType File
}
```

- (*You might need to use the **-Force** option on the **new-item** cmd.*)
- The $PROFILE automatic variable stores the paths to the PowerShell profiles that are available in the current session.
- Edit the **CurrentUserCurrentHost** profile by opening it in a text editor. Make the necessary changes and save the file:
  - **notepad $profile**

Creating and Using PowerShell profiles

Order of application:

- First the more general profiles, such as **AllUsersAllHosts** are applied, ending with more specific profiles such as **CurrentUserCurrentHost**.
- As the individual profiles are applied, any conflicts that arise are simply overwritten by the more specific profile.

- More information on PowerShell profiles can be found at:

https://msdn.microsoft.com/en-us/library/windows/desktop/bb613488(v=vs.85).aspx

And

https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_profiles?view=powershell-6#how-to-create-a-profile

Passing Variables to Functions

- One of the most powerful features of PowerShell functions is in using variables to pass data into the function.
- By passing data into a function, the function can be more generic, and can perform actions on many types of objects.
- In following example, how to accept variables in functions, (*and how to report errors if a mandatory variable is not included*), will demonstrated.

Example:

```
Function Add-Numbers
{
    Param(
        [int]$FirstNum = $(Throw "You must supply at least 1 number")
        , [int]$SecondNum= $FirstNum
        )
    Write-Host ($FirstNum+ $SecondNum)
}
```

Passing Variables to Functions

Explanation:

- At the beginning of the function we reference the **Param()** keyword which defines the parameters the function will accept.
- The first parameter, **$FirstNum**, we define as being mandatory and of type **[int]** or integer.
  - We did not have to classify the parameter type, and the function would have worked without this, but it's a good practice to validate the input of your functions.
- The second parameter, **$SecondNum**, is also typed as **[int]**, but also has a default value defined.
  - This way if no value is passed for the second parameter, it will default to the **$FirstNum**.

Continues…

Passing Variables to Functions

- When the function runs, it reads in the parameters from the command line and attempts to place them in the variables.
  - The parameters can be assigned based on their position in the command line (*that is, the first number is placed into* **$FirstNum**, *and the second number is placed into* **$SecondNum**).
  - Additionally, we can call the function using named parameters with the **-FirstNum** and **-SecondNum** switches.

- If a parameter has a **Throw** attribute assigned, but the value is not provided, the function will end and return an error.
  - Additionally, if a parameter has a type defined, but the value received is incompatible (*such as a string being placed into an integer*), the function will end and return an error.

```
PS C:\WINDOWS\system32> Add-Numbers 2 3
5
PS C:\WINDOWS\system32> Add-Numbers -FirstNum 2 -SecondNum 3
5
PS C:\WINDOWS\system32>
```

Passing Variables to Functions

- Functions are not only capable of receiving input, but also returning output.
  - This ability can come in very handy when trying to return values into other variables instead of simply returning the values to the screen.
- In our example, we can replace our **Write-Host** command with a Return command:

```
#Write-Host ($FirstNum+ $SecondNum)
Return ($FirstNum+ $SecondNum)
```

- The output of the function is mostly the same, except now we can assign the output to a variable and use that variable at a later time:

```
PS C:\> Add-Numbers 5 22
27

PS C:\> $foo = Add-Numbers 9 8

PS C:\> $foo
17
```

Validating Parameters in Functions

- Whenever a script or program receives data from an unknown source, the general rule is that the data should be validated prior to being used.
- Validation can take many forms, with simple validations such as confirming the value exists, is of the right type, or fits a predefined format.
- Validation can also be complex multi-stage events such as ensuring a username exists in a database before prompting for a password.

Example:

- Start by creating a basic function with no input validation:

```
Function Hello-World
{
        param($foo)
        "Hello $foo"
}
```

```
PS C:\> Hello-World Ed
Hello Ed

PS C:\> Hello-World 55
Hello 55

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

Continues…

Validating Parameters in Functions

- Then update the function to perform basic integer validation:

```
Function Hello-WorldInt
{
        param([int] $foo)
        "Hello $foo"
}
```

```
PS C:\WINDOWS\system32> Hello-WorldInt 3
Hello 3
PS C:\WINDOWS\system32> Hello-WorldInt 3.43
Hello 3
PS C:\WINDOWS\system32> Hello-WorldInt r
Hello-WorldInt : Cannot process argument transformation on parameter 'foo'. Cannot convert value "r" to type
"System.Int32". Error: "Input string was not in a correct format."
At line:1 char:16
+ Hello-WorldInt r
+                ~
    + CategoryInfo          : InvalidData: (:) [Hello-WorldInt], ParameterBindingArgumentTransformationException
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,Hello-WorldInt
```

Validating Parameters in Functions

Another Example:

- Update the function to perform basic array validation.

```
Function Hello-WorldStringArray
{
        param([string[]] $foo)
        "Hello " + $foo[0]
}
```

```
PS C:\> Hello-WorldStringArray ("Ed", "Goad")
Hello Ed

PS C:\> Hello-WorldStringArray Ed
Hello Ed
```

Validating Parameters in Functions

More Examples:

- Create a function to validate the length of a parameter:

```
function Hello-WorldLength{
        param([ValidateLength(4,10)] $foo)
        "Hello $foo"
}
```

- Create a function to validate a number in a range:

```
function Hello-WorldAge{
        param([ValidateRange(13,99)] $age)
        "Hello, you are $age years old"
}
```

Validating Parameters in Functions

More Examples:

- Create a function to validate a set of parameters:

```
function Hello-WorldSize{
        param([ValidateSet("Skinny", "Normal", "Fat")] $size)
        "Hello, you are $size"
}
```

- Create a function that validates against a script:

```
function Hello-WorldAge2{
        param([ValidateScript({$_ -ge13 -and $_ -lt99})] $age)
        "Hello, you are $age years old"
}
```

Validating Parameters in Functions

- **Example**; Create a function to validate the input as a phone number:

```
Function Test-PhoneNumber
{
        param([ValidatePattern("\d{3}-\d{4}")] $phoneNumber)
        Write-Host "$phoneNumber is a valid number"
}
```

```
PS C:\> Test-PhoneNumber 867-5309
867-5309 is a valid number

PS C:\> Test-PhoneNumber 206-867-5309
206-867-5309 is a valid number
```

Validating Parameters in Functions

- The last example function can be updated to use custom validation internal to the script with regular expressions:

```
Function Test-PhoneNumberReg
{
        param($phoneNumber)
        $regString=[regex]"^\d{3}-\d{3}-\d{4}$|^\d{3}-\d{4}$"
        if($phoneNumber -match $regString){
                Write-Host "$phoneNumberis a valid number"
        } else {
                Write-Host "$phoneNumberis not a valid number"
        }
}
```

- More information about using regular expressions for validation can be found at:

https://technet.microsoft.com/en-us/magazine/2007.11.powershell.aspx

Piping Data to Functions

- In addition to passing data to functions via parameters, functions can receive data directly from another object or command via a pipe "|".
- Receiving values by piping helps improve scripting by limiting the use of temporary variables, as well as more easily passing complex object types or descriptors.

Example:

- Start by creating a simple function that accepts a parameter:

```
Function Square-Num
{
        Param([float] $FirstNum)
        Write-Host ($FirstNum * $FirstNum)
}
```

Piping Data to Functions

- Then, use the **ValueFromPipeline** parameter to enable the script to accept input from the pipeline:

```
Function Square-Num
{
        Param([float]
        [Parameter(ValueFromPipeline = $true)]
        $FirstNum)
        Write-Host ($FirstNum * $FirstNum)
}
```

```
PS C:\> Square-Num 9
81

PS C:\> 5 | Square-Num
25
```

- This parameter option allows the function to assign a value to **$FirstNum** from the command line, as well as from the pipeline.
- PowerShell will first look for the value on the command line via name or location, and if it isn't listed, it will look for the value from the pipe.

Recording Sessions with Transcripts

PowerShell transcripts:

- Transcripts are a great way of recording everything you do in a PowerShell session and saving it in a text file for later review.

Method:

- Open the PowerShell console (not the ISE) and ***begin recording*** a transcript in the default location by executing **Start-Transcript**.
- ***Stop the recording*** by executing **Stop-Transcript**.
- Begin recording a transcript into a different location by calling **Start-Transcript** with the **-Path** switch.

```
PS C:\> Start-Transcript -Path C:\temp\foo.txt -Force
Transcript started, output file is C:\temp\foo.txt
PS C:\>
```

Signing PowerShell Scripts

- When creating PowerShell scripts, modules, and profiles, it is considered best practice to digitally sign them.

- Signing scripts performs the following two functions:
  1) Ensures the script is from a trusted source
  2) Ensures the script hasn't been altered since it was signed

- To sign a PowerShell script, a code-signing certificate will be needed.
  - Normally these certificates will be provided by your enterprise **Private Key Infrastructure (PKI)**, and the PKI Administrator should be able to help you with the requesting process.
  - Code-signing certificates can also be purchased from third party **Certificate Authorities (CA)** which can be helpful if your scripts are being distributed outside of your corporate environment.

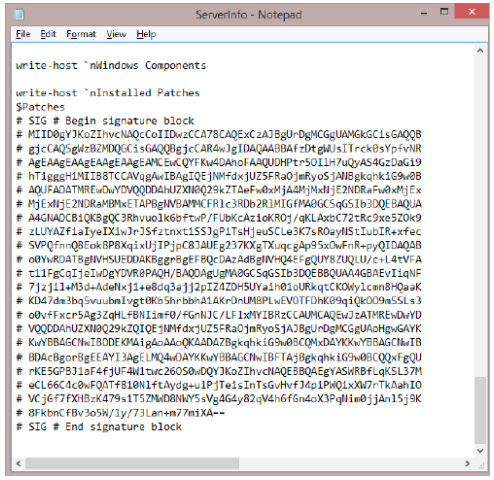
Signing PowerShell Scripts

- Once received, the code-signing cert should be added to your **Current User** | **Personal**| **Certificates** certificate store on your computer.
  - Additionally, the root certificate from the Certificate Authority should be added to the Trusted Publishers store for all computers that are going to execute the signed scripts.

Example:

- Task: Create and test a PowerShell script
- Then Sign the script with **Set-AuthenticodeSignature**.

```
$cert = Get-ChildItem Cert:CurrentUser\My\ -CodeSigningCert
Set-AuthenticodeSignature C:\temp\ServerInfo.ps1 $cert
```



Sending e-mail

- Integration with e-mail is a key capability for automating administration tasks. With e-mail, you can run tasks that automatically let you know when they are complete, or e-mail users when information is needed from them, or even send reports to administrators.

- To send an e-mail using PowerShell, we will need a mail system capable of accepting SMTP mail from your computer.
  - This can be a Microsoft Exchange server, and IIS server, a Linux host, or even a public mail service such as Google Mail.
  - The method we use may change how the e-mail appears to the end recipient and may cause the message to be flagged as spam.

Continues…

Sending e-mail

- To send e-mail using the traditional .NET method:

```
function Send-SMTPmail($to, $from, $subject, $smtpServer, $body)
{
        $mailer = new-object Net.Mail.SMTPclient($smtpServer)
        $msg= new-object Net.Mail.MailMessage($from, $to, $subject, $body)
        $msg.IsBodyHTML = $true
        $mailer.send($msg)
}
```

Continues…

Sending e-mail

- To send the mail message, call the following function:

Send-SMTPmail -to "admin@domain.com" -from "mailer@domain.com" -subject "test email" -smtpserver "mail.domain.com" -body "testing“

- To send e-mail using the included **PowerShell Cmdlet**, use the **Send-MailMessage** command as shown:

Send-MailMessage -To admin@domain.com -Subject "test email" -Body "this is a test" -SmtpServer mail.domain.com -From mailer@domain.com

Sorting and Filtering

- One of the great features of PowerShell is its ability to sort and filter objects.
- This filtering can be used to limit a larger result set and reporting only the information necessary.

Example:

- To explore the filtering capabilities of PowerShell, we look at the running processes on our computer.
- We then use the ***Where*** clause to filter the results.

```
Get-Process | Where-Object {$_.Name -eq "chrome"}
Get-Process | Where-Object Name -eq "chrome"
Get-Process | Where-Object Name -like "*hrom*"
Get-Process | Where-Object Name -ne "chrome"
Get-Process | Where-Object Handles -gt 1000
```

Sorting and Filtering

- To view sorting in PowerShell, we again view the processes on our computer and use **Sort-Object** to change the default sort order.

Example:

Get-Process | Sort-Object Handles

Get-Process | Sort-Object Handles -Descending

Get-Process | Sort-Object Handles, ID –Descending

- To view the grouping capabilities of PowerShell, we use Format-Table with the **-GroupBy** command:

Example:

Get-Process | Format-Table -GroupBy ProcessName

Using Formatting to Export Numbers

Examples:

```
$jenny = 1206867.5309
Write-Host "Original:`t`t`t" $jenny
Write-Host "Whole Number:`t`t" ("{0:N0}" -f $jenny)
Write-Host "3 decimal places:`t" ("{0:N3}" -f $jenny)
Write-Host "Currency:`t`t`t" ("{0:C2}" -f $jenny)
Write-Host "Percentage:`t`t`t" ("{0:P2}" -f $jenny)
Write-Host "Scientific:`t`t`t" ("{0:E2}" -f $jenny)
Write-Host "Fixed Point:`t`t" ("{0:F5}" -f $jenny)
Write-Host "Decimal:`t`t`t" ("{0:D8}" -f [int]$jenny)
Write-Host "HEX:`t`t`t`t" ("{0:X0}" -f [int]$jenny)
```

```
Original:               1206867.5309
Whole Number:           1,206,868
3 decimal places:       1,206,867.531
Currency:               €1,206,867.53
Percentage:             120,686,753.09%
Scientific:             1.21E+006
Fixed Point:            1206867.53090
Decimal:                01206868
HEX:                    126A54
```

Using Formatting to Export Data Views

- Using Get-Process to list all running Chrome processes:

```
Get-Process chrome
```

- To list all available attributes for our processes, execute the following code:

```
Get-Process chrome | Select-Object *
```

- To return a select list of attributes, update the following command:

```
Get-Process chrome | `
Select-Object Name, Handles, Threads, `
NonpagedSystemMemorySize, PagedMemorySize, `
VirtualMemorySize, WorkingSet, `
PrivilegedProcessorTime, UserProcessorTime, `
TotalProcessorTime
```

- Note the use of the backtick (`) character at the end of all but the last line. This tells PowerShell to include the contents of the lines as a single line.
  - This allows us to more easily format the script for readability.

Dealing with Errors in PowerShell

- When creating a script in any language, error handling is needed to ensure proper operations.
  - Error handling is useful when debugging scripts and ensuring scripts work properly, but they can also present alternative methods of accomplishing tasks.

Example:

- Start by creating a simple function that uses no error handling

```
Function Multiply-Numbers
{
        Param($FirstNum, $SecNum)
        Write-Host ($FirstNum * $SecNum)
}
```

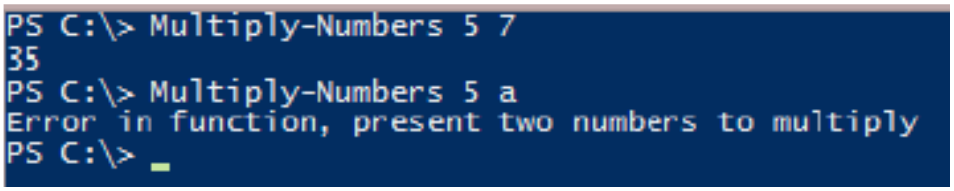
```
PS C:\> Multiply-Numbers 5 7
35
PS C:\> Multiply-Numbers 5 a
Cannot convert value "a" to type "System.Int32". Error: "Input string was not
in a correct format."
At line:4 char:5
+     Write-Host ($FirstNum * $SecNum)
+     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvalidCastFromStringToInteger

PS C:\>
```

Dealing with Errors in PowerShell

- Then update the function using a Try/Catch block:

```
Function Multiply-Numbers
{
        Param($FirstNum, $SecNum)
        Try
        {
        Write-Host ($FirstNum * $SecNum)
        }
        Catch
        {
        Write-Host "Error in function, present two numbers to multiply"
        }
}
```



Dealing with Errors in PowerShell

- The updated script uses a **Try/Catch** block to find errors and return a more friendly error.
  - The **Try** block attempts to perform the multiplication, and if an error is returned then processing ***exits***.
  - When the **Try** block fails for any reason, it then executes the **Catch** block instead.

- In this case, we are returning a command specific error message, in other scenarios we could initiate an alternative task or command that was based on the error.

- The **$Error** variable is an in-built array that automatically captures and stores errors as they happen.
  - You can view the variable to report all errors listed, or you can use indexing such as **$Error[1]** to return specific errors.

Dealing with Errors in PowerShell

Task:

- In the PowerShell console, execute a command to generate an error such as **Get-Item foo**.
- View the **$Error** variable to return the error code history.

```
PS C:\> Get-Item foo
Get-Item : Cannot find path 'C:\foo' because it does not exist.
At line:1 char:1
+ Get-Item foo
+ ~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (C:\foo:String) [Get-Item], Item
   NotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetIt
   emCommand


PS C:\> $Error
Get-Item : Cannot find path 'C:\foo' because it does not exist.
At line:1 char:1
+ Get-Item foo
+ ~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (C:\foo:String) [Get-Item], Item
   NotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetIt
   emCommand
```

Dealing with Errors in PowerShell

Clearing error codes:

- By default, the **$Error** array will retain a number of error codes. These errors are only removed from the array when it reaches its maximum size, or when the user session is ended.

- It is possible to clear out the array before doing a task, so that you can then review the **$Error** array after and know that all the alerts are relevant.

Example:

```
$Error.Count
$Error.Clear()
$Error.Count
```

- This example starts by returning the number of items in the array.
- Then **$Error. Clear()** is called to empty the array.
- Lastly, the number of array items is returned to confirm that it has been cleared.