Microsoft

Tag 4

AZ-040 Automating Administration with PowerShell

Guten Morgen!

Learning Path 7: Windows PowerShell scripting

Learning objectives

- Create and run scripts by using Windows PowerShell
- Work with scripting constructs in Windows PowerShell
- Import data in different formats for use in scripts by using Windows PowerShell cmdlets
- Use methods to accept user inputs in Windows PowerShell scripts
- Troubleshoot scripts and handle errors in Windows PowerShell
- Use functions and modules in Windows PowerShell scripts

Agenda

LP 1 Get started with Windows PowerShell

LP 2 Maintain system administration tasks in Windows PowerShell

LP 3 Work with the Windows PowerShell pipeline

LP 4 Work with PowerShell providers and PowerShell drives in Windows PowerShell

LP 5 Querying management information by using CIM and WMI

LP 6 Use variables, arrays, and hash tables in Windows PowerShell scripts

LP 7 Create and modify scripts by using Windows PowerShell

LP 8 Administer remote computers by using Windows PowerShell

LP 9 Manage cloud resources by using Windows PowerShell

LP 10 Manage Microsoft 365 services by using Windows PowerShell

LP 11 Create and manage background jobs and scheduled jobs in Windows PowerShell

Overview

You can use Windows PowerShell to create scripts to accomplish more complex tasks that aren't possible by using a single command. You can reuse scripts multiple times to accomplish repetitive tasks. After the necessary commands are in a script, there's a lower risk of errors when you run it.

Modules:

- Create and run scripts by using Windows PowerShell
- Work with scripting constructs in Windows PowerShell
- Work with scripting constructs in Windows PowerShell
- Use methods to accept user inputs in Windows PowerShell scripts
- Troubleshoot scripts and handle errors in Windows PowerShell
- Use functions and modules in Windows PowerShell scripts

Create and run scripts by using Windows PowerShell

Module overview

Most administrators start working with scripts by either downloading or modifying scripts that others created. After you've obtained a script, you must run it. There are some important differences between running scripts in Windows PowerShell versus running them at a traditional Windows Command Prompt. These differences are covered in this Module along with other concepts that are related to scripting.

Units:

- Overview of Windows PowerShell scripts
- Modifying scripts
- Creating scripts
- What is the PowerShellGet module?
- Running scripts
- The script execution policy
- Demonstration: Setting the script execution policy
- Windows PowerShell and AppLocker
- Digitally signing scripts
- Demonstration: Digitally signing a script

Overview of Windows PowerShell scripts

Windows PowerShell scripts can be used for:

- Repetitive tasks.
- Complex tasks.
- Reporting.

Windows PowerShell scripts:

- Use constructs such as **ForEach, If,** and **Switch**.
- Have a .ps1 file extension.

Modifying scripts

- Modifying an existing script is easier and faster than creating your own.
- You should:
  - Understand how a downloaded script works.
  - Test a downloaded script in a non-production environment.
- You can get scripts from:
  - The PowerShell Gallery.
  - Blogs and websites.

Creating scripts

- Create a script if you can’t find one that meets your needs.
- Use code snippets from other sources when building your script.
- Develop scripts in an isolated environment that can’t affect production.
- Build scripts incrementally for easier testing during development.

What is the PowerShellGet module?

- Windows PowerShellGet:
  - Has cmdlets for accessing and publishing items in the PowerShell Gallery.
  - Is included in Windows Management Framework 5.0.
  - Can be downloaded for Windows PowerShell 4.0.
  - Uses the NuGet provider.
- You must enable TLS 1.2 to access the PowerShell Gallery
  - **[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12**
- You can implement a private PowerShell Gallery.
- Cmdlets for finding items are:
  - **Find-Module**
  - **Find-Script**

Running scripts

- To enhance security, the .ps1 file extension is associated with Notepad.
- Integration with File Explorer:
  - Open
  - Run with PowerShell
  - Edit
- To run scripts at the Windows PowerShell prompt:
  - Enter the full path - **C:\Scripts\MyScript.ps1**
  - Enter a relative path - **\Scripts\MyScript.ps1**
  - Reference the current directory - **.\MyScript.ps1**

The script execution policy

- The options for the execution policy are:
  - **Restricted**
  - **AllSigned**
  - **RemoteSigned**
  - **Unrestricted**
  - **ByPass**
- Modify the execution policy by using:
  - **Set-ExecutionPolicy**
  - The **Turn on Script Execution** Group Policy setting
- Override the execution policy for a single instance:
  - **Powershell.exe -ExecutionPolicy Bypass**
- Remove downloaded status from a script by using **Unblock-File**

Set-AuthenticodeSignature

Demonstration: Setting the script execution policy

In this demonstration, you will:

1. Double-click a script or select it and then press the Enter key.
2. Run a script using a full path.
3. Run a script from the current directory.
4. Set the execution policy to restricted.
5. Set the execution policy to unrestricted.

Windows PowerShell and AppLocker

- You can use AppLocker to control the running of Windows PowerShell scripts.
- AppLocker can limit scripts based on:
  - Name.
  - Location.
  - Publisher (with digital signature).
- In Windows PowerShell 5.0 and newer, interactive prompts are limited to **ConstrainedLanguage** mode.

Digitally signing scripts

- Use digital signatures on scripts with the **AllSigned** execution policy.
- Use digitally signed scripts to:
  - Formalize the script approval process.
  - Prevent accidental changes.
- A trusted code-signing certificate is required to add a digital signature to a script.
- To set a digital signature:
  - **$cert = Get-ChildItem -Path "Cert:\CurrentUser\My" -CodeSigningCert**
  - **Set-AuthenticodeSignature -FilePath "C:\Scripts\MyScript.ps1" -Certificate $cert**

Demonstration: Digitally signing a script

In this demonstration, you will learn how to:

1. Install a code-signing certificate.
2. Digitally sign a certificate.
3. Review the digital signature.

Work with scripting constructs in Windows PowerShell

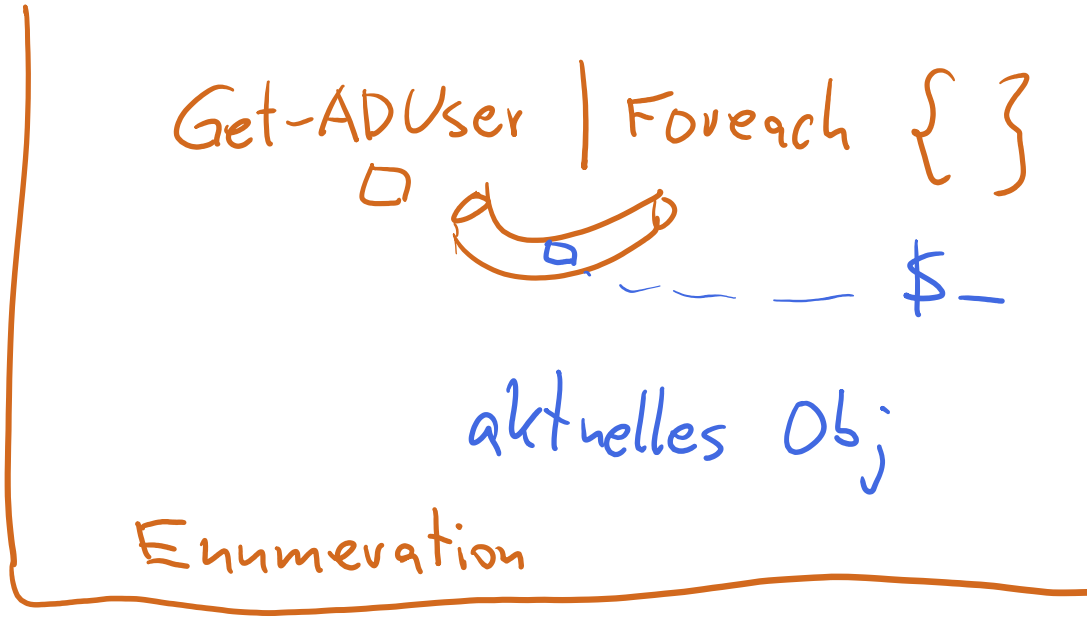
Module overview

While there are some simple scripts that use only Windows PowerShell commands, most scripts use scripting constructs to perform more complex actions. You can use the **ForEach** construct to process all the objects in an array. You can use **If..Else** and **Switch** constructs to make decisions in your scripts. Finally, there are less common looping constructs such as **`For`, `While`,** and **`Do..While`** loops that you can use when appropriate.

Units:

- Understanding **ForEach** loops
- Demonstration: Using a **ForEach** loop
- Understanding the **If** construct
- Demonstration: Using the **If** construct
- Understanding the **Switch** construct
- Demonstration: Using the **Switch** construct
- Understanding the **For** construct
- Understanding other loop constructs
- Understanding **Break** and **Continue**

Understanding ForEach loops



- Example:

```
ForEach ($user in $users) {
    Set-ADUser $user -Department “Marketing”
}
```

- Commands between braces are processed once for each item in the array.
- You don’t need to know how many items are in the array.
- **$user** contains each array item during the loop.
- The indent is to make it easier to review.
- Variable names should be meaningful.
- The **ForEach-Object** cmdlet has the *-Parallel* parameter.

Demonstration: Using a ForEach loop

In this demonstration, you will:

1. Review a script with a **ForEach** loop.
2. Run the script.

Understanding the If construct

- Use the **If** construct to make decisions.
- There can be zero or more **ElseIf** statements.
- **Else** is optional.
- Example:

```
If ($freeSpace -le 5GB) {
    Write-Host “Free disk space is less than 5 GB”
} ElseIf ($freeSpace -le 10GB) {
    Write-Host “Free disk space is less than 10 GB”
} Else {
    Write-Host “Free disk space is more than 10 GB”
}
```

ISE

VS Code

Ctrl-J

Demonstration: Using the If construct

In this demonstration, you will:

1. Review a script with the **If** construct.
2. Test the script functionality.

Understanding the Switch construct

- The **Switch** construct compares a variable to a list of values.
- Example:

```
Switch ($choice) {
    1 { Write-Host “You selected menu item 1” }
    2 { Write-Host “You selected menu item 2” }
    3 { Write-Host “You selected menu item 3” }
    Default { Write-Host “You did not select a valid option” }
}
```

- You can also use wildcards and regular expressions:
  - Multiple matches are possible.

Demonstration: Using the Switch construct

In this demonstration, you will:

1. Review a script with the **Switch** construct.
2. Test the script functionality.

Understanding the For construct

- The **For** construct is used to run a script block a specific number of times based on the:
  - Initial state.
  - Condition.
  - Action.
- Example:

```
For($i=1; $i -le 10; $i++) {
    Write-Host “Creating User $i”
}
```

Understanding other loop constructs

- **Do..While**
  - Loops until the condition is false
  - Guaranteed to process the script block once.
  - Example:

```
Do {
   Write-Host “Code block to process”
} While ($answer -eq "go")
```

- **Do..Until**
  - Loops until the condition is true.
  - Guaranteed to process the script block once.
  - Example:

```
Do {
    Write-Host “Code block to process”
} Until ($answer -eq “stop”)
```

Understanding other loop constructs (Slide 2)

- **While**
  - Processes the script block until the condition is false.
  - Execution on the script block is not guaranteed.
  - Example:

```
While ($answer -eq “go”) {
    Write-Host “Script block to process”
}
```

Understanding Break and Continue

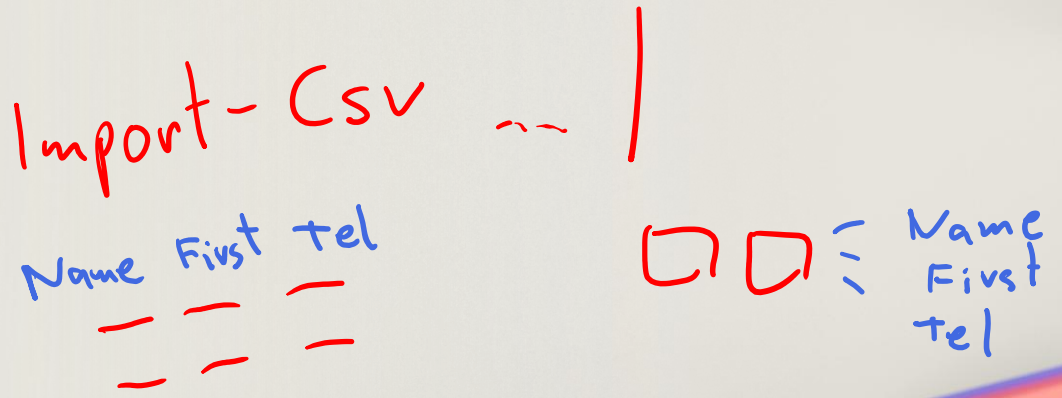
- **Continue** stops processing the current iteration of a loop:

```
ForEach ($user in $users) {
    If ($user.Name -eq “Administrator”) {Continue}
    Write-Host “Modify user object”
}
```

- **Break** completely stops loop processing:

```
ForEach ($user in $users) {
    $number++
    Write-Host “Modify User object $number”
    If ($number -ge $max) {Break}
}
```

Import data in different formats for use in scripts by using Windows PowerShell cmdlets



Module overview

When you create a script, reusing data from other sources is useful. Most applications can export the data you want in various formats such as a CSV file or an XML file. You can use Windows PowerShell cmdlets to import data in different formats for use in your scripts. In this Module, you'll learn how to import data from a text file, CSV file, XML file, and JavaScript Object Notation (JSON) file.

Units:

- Using **Get-Content**
- Using **Import-Csv**
- Using **Import-Clixml**
- Using **ConvertFrom-Json**
- Demonstration: Importing data

Using Get-Content

- **Get-Content** retrieves content from a text file.
- Each line in the file becomes an item in an array:

  $computers = Get-Content “C:\Scripts\computers.txt”
- Import multiple files by using wildcards:

  Get-Content -Path “C:\Scripts*” -Include "*.txt”,”*.log”
- Limit the data retrieved by using the -*TotalCount* and -*Tail* parameters.

Using Import-Csv

- The first row in the CSV file is a header row.
- Each line in the CSV file becomes an array item.
- The header row defines the property names for the items:

  $users = Import-Csv C:\Scripts\Users.csv
- You can specify a custom delimiter by using the *-Delimiter* parameter.
- You can specify a missing header row by using the *-Header* parameter.

Using Import-Clixml

- XML can store more complex data than CSV files.
- **Import-Clixml** creates an array of objects:

  $users = Import-Clixml C:\Scripts\Users.xml
- Use **Get-Member** to review object properties.
- Limit the data retrieved by using the *-First* and *-Skip* parameters.

Using ConvertFrom-Json

- JSON is:
  - A lightweight data format similar to XML.
  - Commonly used by web services.
- You can convert from JSON, but not import directly.

  $users = Get-Content C:\Scripts\Users.json | ConvertFrom-Json
- Retrieve JSON data directly from web services by using **Invoke-RestMethod.**

  $users = Invoke-RestMethod "https://hr.adatum.com/api/staff"

ConvertTO-Json
ConvertTo-Yaml nur 7

Demonstration: Importing data

In this demonstration, you will learn how to:

1. Use **Get-Content** to retrieve text data.
2. Use **Import-Csv** to retrieve CSV data.
3. Use **Import-Clixml** to retrieve XML data.

Use methods to accept user inputs in Windows PowerShell scripts

Module overview

To enhance the usability of your scripts, you must learn how to accept user input. This skill allows you to create scripts that you can use for multiple purposes. In addition, accepting user input allows you to create scripts that are easier for others to use. In this Module, you learn about multiple methods for accepting user input in a script.

Units:

- Identifying values that might change
- Using **Read-Host**
- Using **Get-Credential**
- Using **Out-GridView**
- Demonstration: Obtaining user input
- Passing parameters to a script
- Demonstration: Obtaining user input by using parameters

Identifying values that might change

- Scripts might initially have static values:
  - Find users that haven't signed in for 30 days.
  - Find specific events on domain controllers.
- When scripts are reused, some of those values might need to change.
- To simplify changing values in scripts:
  - Use variables defined at the beginning of the script.
- To avoid changing scripts:
  - Accept user input.

Using Read-Host

- Use **Read-Host** to request user input while a script is running:

  $answer = Read-Host "How many days"

- To avoid displaying a colon, combine **Write-Host** and **Read-Host:**

  Write-Host "How many days?" –NoNewline
  $answer = Read-Host

- The *-MaskInput* and *-AsSecureString* parameters hide content as it's entered.

Using Out-Gridview

- **Out-GridView** can be used as a simple menu system

  $selection = $users | Out-GridView -PassThru

- For more control over the items selected, you can use the -*OutputMode* parameter

Value	Description
None	Doesn't allow any rows to be selected
Single	Allows zero rows or one row to be selected
Multiple	Allows zero rows, one row, or multiple rows to be selected

Demonstration: Obtaining user input

In this demonstration, you will learn how to:

1. Use **Read-Host.**
2. Use **Get-Credential.**
3. Use **Out-GridView.**

Passing parameters to a script

- Use a **Param()** block to identify the variables that will store parameter values:

```
Param(
    [string]$ComputerName
    [int]$EventID
)
```

- The variable names in the **Param()** block define the parameter names:

```
.\GetEvent.ps1 -ComputerName LON-DC1 -EventID 5772
```

- It's a best practice to define variable types:
  - Errors are generated when data can't be converted.
  - [switch] defines parameters that are on or off.
- You can:
  - Define default values for parameters: **[string]$ComputerName** = **"LON-DC1"**
  - Request user input for parameters: **[int]$EventID** = **Read-Host "Enter event ID"**

Demonstration: Obtaining user input by using parameters

In this demonstration, you will learn how to:

1. Review a script with a **param()** block.
2. Pass parameters to the script.
3. Request user input when a parameter isn't supplied.
4. Set a default value for a parameter.

Troubleshoot scripts and handle errors in Windows PowerShell

Module overview

As you develop more scripts, you'll find that efficient troubleshooting makes your script development much faster. A big part of efficient troubleshooting is understanding error messages. For some difficult problems, you can use breakpoints to stop a script partially through execution to query values for variables.

Units:

- Understanding error messages
- Adding script output
- Using breakpoints
- Demonstration: Troubleshooting a script
- Understanding error actions

Understanding error messages

- Error messages are useful for troubleshooting.
- You might encounter error messages because:
  - You made a mistake while entering text.
  - You queried an object that doesn't exist.
  - You attempted to communicate with a computer that's offline.
- Errors are stored in the **$Error** array
  - The most recent error is stored in **$Error[0].**

Adding script output

- Use **Write-Host** to display additional information when the script is running:
  - Variable values.
  - Location in the script.
- To slow down the data onscreen for easier reviewing, use:
  - **Start-Sleep**
  - **Read-Host**
- Comment out the **Write-Host** commands when not required.
- When you use **CmdletBinding(),** you can also use:
  - **Write-Verbose**
  - **Write-Debug**

Using breakpoints

- Use breakpoints to pause a script and query or modify variable values.
- **Set-PSBreakPoint** examples:
  - **Set-PSBreakPoint -Line 23 -Script "MyScript.ps1"**
  - **Set-PSBreakPoint -Command "Set-ADUser" -Script "MyScript.ps1"**
  - **Set-PSBreakPoint -Variable "computer" -Mode ReadWrite -Script "MyScript.ps1"**
- You use the *-Action* parameter to specify code to perform.
- The Windows PowerShell ISE has line-based breakpoints.
- Visual Studio Code:
  - Has conditional breakpoints.
  - Displays variable values in a dedicated area.

Demonstration: Troubleshooting a script

1. Run a script the generates an error.
2. Review the **$Error[0]** variable.
3. Clear the **$Error** variable.
4. Configure and use and break point.
5. Remove all breakpoints.

Understanding error actions

- **$ErrorActionPreference** defines what happens for non-terminating errors:
  - **Continue**
  - **SilentlyContinue**
  - **Inquire**
  - **Stop**
- It's preferred to modify the error action for individual commands rather than globally:
  - **Get-WmiObject -Class Win32_BIOS -ComputerName LON-SVR1,LON-DC1 -ErrorAction Stop**
- Set the error action globally when it can't be done at the command:
  - For example, using methods on an object

Use functions and modules in Windows PowerShell scripts

Module overview

When you create many scripts, you'll have snippets of code that you want to reuse. You might also have snippets of code that you want to reuse within the same script. Rather than having the same lines of code appearing multiple times in a script, you can create a function that appears once in the script but is called multiple times. If you need to use the same code across multiple scripts, then you can place the function in a module that subsequently can be shared by multiple scripts. In this Module, you'll learn to create functions and modules.

Units:

- What are functions?
- Using variable scopes
- Demonstration: Creating a function in a script
- Creating a module
- Demonstration: Creating a module from a function
- Using dot sourcing

What are functions?

- Use functions to perform the same set of tasks multiple times within a script without having to add the same lines
of code in each of the corresponding sections of the script.
- Pass data to a function by using a **Param()** block:
  - **Function Get-SecurityEvent {**
    Param (
    [string]$ComputerName
) #end Param
    Get-EventLog -LogName Security -ComputerName -$ComputerName -Newest 10
    }
- To call a function:
  - **Get-SecurityEvent -ComputerName LON-DC1**
- To store the results of a function:
  - **$securityEvents=Get-SecurityEvent -ComputerName LON-DC1**

Using variable scopes

- There are three scopes for variables:
  - Global
  - Script
  - Function
- Avoid using the same variable name in multiple scopes.
- You can modify a variable in a higher scope by specifically referencing the scope:
  - **$script:var="Modified from function"**
- Set a script scope variable equal to the function output instead.
  - Use **Return()** to pass a variable value back

Demonstration: Creating a function in a script

In this demonstration, you will learn how to:

1. Review a script that accepts parameters.
2. Add code to create a function.
3. Call the function within the script.

Creating a module

- Functions in the module should use standard cmdlet naming conventions.
- Module files use the **.psm1** file extension.
- Module locations are stored in **$PSModulePath** environmental variable.
- For Windows PowerShell 5.0, the paths are:
  - **C:\Users*UserID*\Document\WindowsPowerShell\Modules**
  - **C:\Program Files\WindowsPowerShell\Modules**
  - **C:\Windows\system32\WindowsPowerShell\1.0\Modules**
- PowerShell 7 also includes:
  - **C:\Program Files\PowerShell\Modules**
  - **C:\Program Files\PowerShell\7\Modules**
- Module files are placed in a subfolder of the same name:
  - **C:\Program Files\WindowsPowerShell\Modules\AdatumFunctions\AdatumFunctions.psm1**

Demonstration: Creating a module from a function

In this demonstration, you will learn how to:

1. Rename a script with a .ps1 file extension.
2. Remove all code except the function.
3. Verify the module is available.
4. Use the function.
5. Verify the module loaded automatically.

Using dot sourcing

- Dot sourcing:
  - Loads the contents of a script into the current scope.
  - Can load from a local file or over the network.
- Syntax:
  - **. C:\Scripts\Functions.ps1**
- It’s preferred to use modules instead of dot sourcing when possible.

Lab – Using scripts with PowerShell

Lab: Using scripts with PowerShell

- **Exercise 1:** Signing a script
- **Exercise 2:** Processing an array with a ForEach loop
- **Exercise 3:** Processing items by using If statements

Sign-in information for the exercises:

Virtual machines: **AZ-040T00A-LON-DC1** and **AZ-040T00A-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

Lab: Using scripts with PowerShell (Slide 2)

- **Exercise 4:** Creating users based on a CSV file
- **Exercise 5:** Querying disk information from remote computers
- **Exercise 6:** Updating the script to use alternate credentials

Sign-in information for the exercises:

Virtual machines: **AZ-040T00A-LON-DC1** and **AZ-040T00A-LON-CL1**

User name: **Adatum\Administrator**

Password: **Pa55w.rd**

Lab scenario

You've started to develop Windows PowerShell scripts to simplify administration in your organization. There are multiple tasks to accomplish, and you'll create a Windows PowerShell script for each one.

Lab-review questions

- You configured a switch parameter to indicate whether alternate credentials are required. What are the possible values for a switch parameter?
- Is a code-signing certificate from an internet certification authority automatically trusted by computers outside your AD DS forest?
- In Exercise 2, you configured the **ipPhone** attribute for a group of test users. How would you update that script for a larger set of users as the solution is deployed to the rest of the organization?

Lab-review answers

- You configured a switch parameter to indicate whether alternate credentials are required. What are the possible values for a switch parameter?
  A switch parameter has a value of **$true** or **$false**.
- Is a code-signing certificate from an internet certification authority automatically trusted by computers outside your AD DS forest?
  No. You should use a public certification authority for compatibility outside your AD DS forest.
- In Exercise 2, you configured the ipPhone attribute for a group of test users. How would you update that script for a larger set of users as the solution is deployed to the rest of the organization?
  The script in Exercise 2 modified the **ipPhone** attribute for members of the **IPPhoneTest** group. When this functionality is being deployed to the remainder of the organization, the query for users will need to be expanded. For example, the script could be modified to work for individual organizational units as the new system is deployed to each department.

References

PowerShell Gallery

Hosting your own NuGet feeds

About Language Modes

about_Functions_Advanced_Parameters

Learning Path Recap

In this learning path, we:

- Learned how to create and modify scripts
- Explored useful scripting techniques such as:
  - Importing data from a file
  - Accepting user input
  - Error handling

End of presentation