PowerShell Scripting Fundamentals

▼ PowerShell Variables and Scopes

Variables

A variable is a ***named storage location*** in a computer's memory that holds a value that can change.

> 📣 The ***$*** sign at the beginning indicates a variable.

```
$i = 1
$string = "Hello World!"
$this_is_a_variable = "test"
```

> 💡 Variables are for storing simple values, strings, and also the output of commands.

```
$date = Get-Date
Write-Host "Today is" $date
```

Data Types

PowerShell automatically assigns a data type to a variable based on the type that best suits its content.

- **Using `Get-Type` command to find out the data type of a variable:**

```
$x = 4
$string = "Hello World!"
$date = Get-Date

$x.GetType().Name
$string.GetType().Name
$date.GetType().Name
```

Overview of data types

[string]	System.String. A simple string
[char]	Unicode 16-bit character
[byte]	8-bit unsigned character
[int], [int32]	32-bit signed integer
[long]	64-bit signed integer
[bool]	Boolean: Can be True or False
[decimal]	128-bit decimal
[single],[float]	Single-precision 32-bit floating point number
[double]	Double-precision 64-bit floating point number
[datetime]	Date and time
[array]	Array of values
[hashtable]	Hashtable object
[guid]	Globally Unique Identifier(GUID)
[psobject], [PSCustomobject]	PowerShell object
[scriptblock]	PowerShell script block
[regex]	Regular Expression
[timespan]	Timespan object

Casting Variables

Convert the data type of variable by ***casting*** it to another type.

```
$number = "4"
$number.GetType().Name      # String data type

#Casting string into integers
$int_number = [int]$number
$int_number.GetType().Name   # Int32

#Cast a Unicode hex string into a character
[char]0x263a
```

Automatic Variables

Built-in variables that are created and maintained by PowerShell.

- **$?:** Execution status of the last command. `True` if command succeeded, otherwise `False`.
- **$_:** Called ***pipeline variable,*** represents the current item being processed in a pipeline or loop.

```
Get-ChildItem -Path C:\ -Directory -Force -ErrorAction
SilentlyContinue | ForEach-Object {
    Write-Host $_.FullName
}
```

- **$Error:** Contains the most recent errors, collected in an array. Indexed using `$Error[0]`
- **$false:** Represents traditional Boolean value of *False*.
- **$LastExitCode:** Contains the last exit code of the program that was run.
- **$null:** Contains *null* or an empty value, used to check whether a variable contains a value or set to undefined.
- **$PSScriptRoot:** Location of directory from which the script is being run.
- **$true:** Contains True. use `$true` to represent True in commands and scripts.

Environment Variables

Store information about the operating system and paths that are frequently used by the system.

To show all the environment variables within a session, using `dir env:`

> 💡 Accessing and reusing variables by prefixing `$env`:

```
# Accessing PSModulePath value.
$env:PSModulePath
```

Reserved words and language Keywords

Words are reserved by the system and should not be used as variables or function names.

```
# More information on reserved words
Get-Help about_reserved_words

# Get detailed overview and explanation of all language keywords:
Get-Help about_Language_Keywords
```

```
Begin           Enum            Param
Break           Exit            Process
Catch           Filter          Return
Class           Finally         Static
Continue        For             Switch
Data            ForEach         Throw
Define          From            Trap
Do              Function        Try
DynamicParam    Hidden          Until
Else            If              Using
Elseif          In              Var
End             InlineScript    While
```

Overview of all the language keywords

```
#Learn more about a certain language keyword
Get-Help break

#Find help pages that write about the word we're looking for
Get-Help filter -Category:HelpFile
```

Variable Scope

Variable scope determines where a variable is accessible within a script, function, or session.

> 💡 In general, variables are only available in the context in which they are set.

```
#Set the scope of the variable $ModuleRoot to script
$script:ModuleRoot = $PSScriptRoot
```

Scope Modifier

- **global:** Sets the scope to *global*, effective when PowerShell starts or a new session is created.

> 💡 Variables with *global* scope defined in a module are available in the session once the module is loaded.

- **local:** The current scope. The local scope can be the global scope, the script scope, or any other scope.
- **script:** Scope is only effective within the script that sets this scope.

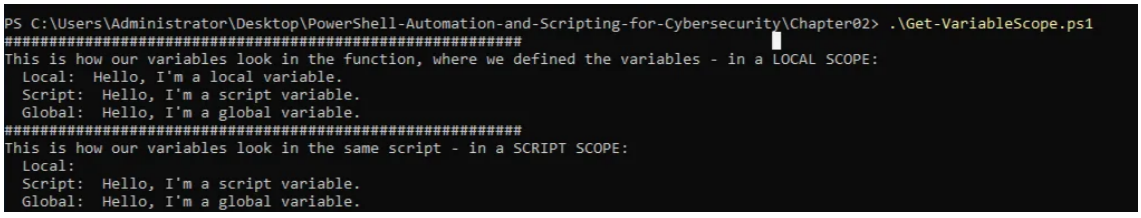
```
function Set-Variables {
    $local_variable = "Hello, I'm a local variable."
    $script:script_variable = "Hello, I'm a script variable."
    $global:global_variable = "Hello, I'm a global variable."

    Write-Host "############################################################"
    Write-Host "This is how our variables look in the function,
        where we defined the variables - in a LOCAL SCOPE:"
    Write-Host "  Local: " $local_variable
    Write-Host "  Script: " $script_variable
    Write-Host "  Global: " $global_variable
}

Set-Variables

Write-Host "############################################################"
Write-Host "This is how our variables look in the same script - in a
SCRIPT SCOPE:"
Write-Host "  Local: " $local_variable
Write-Host "  Script: " $script_variable
Write-Host "  Global: " $global_variable
```

- *Set-Variable* function is declared first, once function is called, it sets variables of three scopes.



Calling variables with a local, script, and global scope

Working with *script* and *global* scope variables, it is a good practice to always use the variable with the modifier: `$script:script_variable/$global:global_variable`

▼ Operators

Arithmetic Operators

```
# Addition
$a = 3; $b = 5; $result = $a + $b

#Substraction
$a = 3; $b = 5; $result = $b - $a

#Multiplication
$a = 3; $b = 5; $result = $a * $b

#Division
$a = 3; $b = 5; $result = $a / $b

#Modulus
$a = 12; $b = 4; $result = $a % $b
```

Comparison Operators

- **Equal (-eq):** Returns True if both values are equal.

  ```
  $a = 1; $b = 1; $a -eq $b
  ```

- **Not equal (-ne):** Returns True if both values are not equal.

  ```
  $a = 1; $b = 2; $a -ne $b
  ```

- **Less equal(-le):** Return True if the first value is less than or equal to the second value.

  ```
  $a = 1; $b = 2; $a -le $b
  ```

- **Greater equal(-ge):** Return True if the first value is greater than or equal to the second value.

  ```
  $a = 1; $b = 2; $a -ge $b
  ```

- **Less than(-lt):** Returns True if the first value is less than the second value.

  ```
  $a = 1; $b = 2; $a -lt $b
  ```

- **Greater than(-gt):** Returns True if the first value is greater than the second value.

```
$a = 1; $b = 2; $b -gt $a
```

- `-like`: Check whether a value matches a wildcard expression when used with a scalar.

```
>"PowerShell" -like "*owers*"
True
```

  - Used in an *array context*, the `-like` operator returns only the elements that match the specified wildcard expression.

```
>"PowerShell", "Dog", "Cat", "Guinea Pig" -like "*owers*"
PowerShell
```

- `-notlike`:Check whether a value does not match a wildcard expression when used with a scalar.

```
>"PowerShell" -notlike "*owers*"
    False
```

  - Used in an *array context,* the `-notlike` operator returns only te elements that do not match the specified wildcard expression.

```
>"PowerShell", "Dog", "Cat", "Guinea Pig" -notlike "*owers*"
Dog
Cat
Guinea Pig
```

- `-match`: Check whether a value matches a regular expression.

```
"Cybersecurity scripting in PowerShell 7.3" -match "shell\s*(\d)"
True

# shell: Matches the literal word "shell."
# \s*: Matches zero or more whitespace characters.
# (\d): Captures a single digit (0-9) in a capture group.
```

- `-notmatch`: Check whether a value does not match a regular expression.

```
"PowerShell Scripting and Automation" -notmatch "^Cyb"
```

Assignment Operators

- **=** : Assigns a value

```
$a = 1; $a
```

- **+= :** Increases value by the amount defined after the operator and stores the result in initial variable.

```
$a += 1; $a += 2; $a
```

- **-= :** Decreases value by the amount defined after the operator and stores the result in initial variable.

```
$a -= 1; $a
```

- ***= :** Multiplies value by the amount defined after the operator and stores the result in initial variable.

```
$a *= 5; $a
```

- **/= :** Divides the value by the amount defined after the operator and stores the result in the initial variable.

```
$a /= 2; $a
```

- **%= :** Performs a modulo operation on the variable using the amount after the operator and stores the result in the initial variable.

```
$a %= 2; $a
```

- **++ :** Increases the variable by 1

```
$a = 1; $a++; $a
```

- **- - :** Decreases the variable by 1

```
$a = 10; $a--; $a
```

Logical Operators

- `-and` **:** Combine conditions, action is triggered only if both conditions are met.

```
$a = 1; $b = 2
if (($a -eq 1) -and ($b -eq 2)) {Write-Host "Conditon is true!"}
```

- `-or` **:** One of the defined conditions is met, the action is triggered.

```
$a = 2; $b =2
if (( $a -eq 1) -or ($b -eq 2)) {Write-Host "Condition is true!"}
```

- `-not` or `!` : Used to negate a condition.

```
$path = $env:TEMP + "\TestDirectory"
if ( -not (Test-Path -Path $path)) {
    New-Item -ItemType directory -Path $path
}

if (!(Test-Path -Path $path)){
    New-Item -ItemType directory -Path $path
}
```

- `-xor` : Logical exclusive `-or`. Is *True* if *only one* statement is *True* (but returns *False* if both are *True*)

```
$a = 1; $b = 2; ($a -eq 1) -xor ($b -eq 1)
```

▼ Control Structures

A control structure is a logic that assesses conditions and variables and decides which defined action to take if certain condition is met.

Conditions

if / elseif / else :

Syntax:

```
if (<condition>)
{
    <action>
}
elseif (<condition>)
{
    <action 2>
}
...
else
{
    <action 3>
}
```

Code:

```
$color = "green"
if ($color -eq "blue") {
        Write-Host "The color is blue!"
}
elseif ($color -eq "green") {
        Write-Host "The color is green!"
}
else {
        Write-Host "Tahat is also a very beautiful color!"
}
#returns: The color is green!
```

Switch :

Check a variable against a long list of values.

Syntax:

```
switch (<value to test>) {
        <condition 1> {<action 1>}
        <condition 2> {<action 2>}
        <condition 3> {<action 3>}
        ...
        default {}
}
```

Code:

```
$color = Read-Host "What is your favourite color?"
switch ($color){
        "blue" { Write-Host "I'm BLUE, Da ba dee da ba di..." }
        "yellow" { Write-Host "YELLOW is color of my favourite IPL team." }
        "red" { Write-Host "Be alert!" }
        "purple" { Write-Host "PURPLE rain, purple rain!"}
        "black" { Write-Host "Men in Black..." }
        default { Write-Host "The color is not in this list" }
}
```

- **Using regular expression:**

  `-Regex` parameter allow the use of regex expression to match against the input.

  ```
  switch -Regex ($userInput){
          "^[A-Z]" { "User input starts with a letter." }
          "^[0-9]" { "User input starts with a number." }
  ```

```
        default { "User input dosen't start with a letter or number" }
}
```

- **Processing the content of a file:**

  The `-Wildcard` parameter enables the use of wildcard logic in code.

```
$path = $env:TEMP + "\example.txt"
switch -Wildcard -File $path {
        "*Error*" { Write-Host "Error was found!: $_"}
}
```

Loops and iterations:

Run an action over and over again until a certain condition is met.

ForEach-Object :

Accepts a list or an array of items and allows to perform an action against each of them.

> 💡 Best use case when pipeline the pipe objects to ForEach-object.

- **Processing all files that are in a folder:**

```
$path = $env:TEMP + "\baselines"
Get-ChidItem -Path $path | ForEach-Object {Write-Host $_}
```

  - To perform specific actions before processing each item use the `-Begin` and `-End` parameters.
  - Use `-Process` parameter to specify the script block that is run for each item in the pipeline.

Foreach :

Works similar to `Foreach-Object`, but it doesn't accept pipeline objects.

Foreach statement loads all items into a collection before they are processed, making it quicker but consuming more memory than `ForEach-Object` .

Code:

- *Foreach statement:*

```
$path = $env:TEMP + "\baselines"
$items = Get-ChildItem -Path $path

foreach ($file in $items){
```

```
        Write-Host $file
}
```

- *Foreach method:*

```
$path = $env:TEMP + "\baselines"
$items  = Get-ChildItem -Path $path

$items.foreach ({
        Write-Host "Current item: $_"
})
```

💡 The `$_` variable is used to reference the current item being iterated over.

while :

Does something (*<actions>*) as long as defined *condition* is *True*.

Syntax:

```
while ( <condition> ) { <action> }
```

Code:

```
while (($input = Read-Host -Prompt "Choose a command (type in 'help'
for an overview)") -ne "quit") {
        switch($input) {
                "hello" {Write-Host "Hello World!"}
                "color" {Write-Host "What's your favourtie sport?")
                "help" {Write-Host "Options: 'Hello', 'color', 'help' 'quit'"
        }
}
```

for:

Defines the initializing statement, a condition, and loops through until the defined condition is not fulfilled.

Syntax:

```
for (<initializing statement>; <condition>; <repeat>)
{
        <actions>
}
```

Code:

```
for ($i=1; $i -le 5; $i++) {Write-Host "i: $i"}
```

do-until / do-while:

Starts running the defined commands, and then checks whether the condition is still met or not.

Syntax:

```
do{
        <action>
}
<while/until> <condition>
```

> 💡 *do-while* runs as long as the condition is *True*, *do-until* runs as long as the *condition* is not met.

break:

Used to *exit* the loop.

Code:

```
for ($i=1; $i -le 10; $i++) {
        Write-Host "i: $i"
        if ($i -eq 3) {break}
}
```

continue:

Used to *skip* the current iteration of a loop and move to the next one.

Code:

```
for ($i=1; $i -le 10; $i++) {
        if (($i % 2) -ne 0) {continue}
        Write-Host "i: $i"
}
```

▼ Naming Conventions

Cmdlets and functions both follow the schema *verb-noun,* such as *Get-Help* or *Stop-Process.*

> 💡 Microsoft has released a list of approved verbs.

Finding the approved verbs

`Get-Verb` command to get the list of approved *verbs*.

- Sort the output of `Get-Verb`:

```
# By the name "Verb"
Set-Verb | Sort-Object Verb

# Using wildcards to prefilter the list:
Get-Verb re*
```

- List *verbs* from certain group(in this case Security group):

```
Get-Verb | Where-Object Group -eq Security
```

▼ PowerShell profiles

PowerShell profiles are configuration files that allow to personalize the PowerShell environment.

> 💡 Profiles are *scripts* that are executed when a PowerShell session is started, allow setting variables, define functions, create aliases, and more.

Types of PowerShell profiles

- **All Users, All Hosts** (`$profile.AllUsersAllHosts`): This profile applies to all users for all PowerShell hosts.
- **All Users, Current Host** (`$profile.AllUsersCurrentHost`): This profile applies to all users for the current PowerShell host.
- **Current User, All Hosts** (`$profile.CurrentUserAllHosts`): This profile applies to the current user for all PowerShell hosts.
- **Current User, Current Host** (`$profile.CurrentUserCurrentHost`): This profile applies only to the current user and the current PowerShell host.

📣 A ***PowerShell host*** is an application that hosts the PowerShell engine.

> **PowerShell hosts** include the Windows PowerShell console, the PowerShell **Integrated Scripting Environment(ISE)**, and the PowerShell terminal in Visual Studio Code.

Finding location of local PowerShell profiles:

```
$PROFILE | Format-List * -force
```

- Applies to local shells and all users: `%windir%\system32\WindowsPowerShell\v1.0\profile.ps1`
- Applies to all shells and all users:
  `%windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1`
- Applies to all local ISE shells and all users:
  `%windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1`

  💡 This profile is loaded when using the PowerShell ISE and can be viewed by running the `$profile | f1 * -force` command within ISE.

- Applies to current user ISE shells on the local host:
  `%UserProfile%\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1`

Accessing file path of one particular profile:

```
# Syntax:
# $profile.<profile name>

#Access CurrentUserCurrentHost profile path
$profile.CurrentUserCurrentHost
```

Creating a User profile:

- Check if profile file already exists, if not create one.

```
if ( !(Test-Path $profile.CurrentUserCurrentHost)) {
        New-Item -ItemType File -path $profile.CurrentUserCurrentHost
}
```

- Add the commands, functions, or aliases to the user profile:

```
Add-Content -Path $profile -Value "New-Alias -Name Get-Ip `
-Value ipconfig.exe"
```

▼ Understanding PSDrives

PowerShell drives in PowerShell are similar to filesystem drives in Windows, but instead of accessing files or folders, use **PSDrives** to access a variety of data stores.

Data stores include directories, registry keys, and other data sources.

> 💡 **PSDrives** are powered by ***PSProviders***, which are underlying components that provide access to data stores.

> `Env:\` is a built-in PowerShell drive that provides access to environment variables.

> 💡 To access a **PSDrive**, use a special *prefix* in the path, like we do C: to access filesystem drive.

- **Retrieving all environment variables with `path` string in their name:**

```
Get-ChildItem Env:\*path*
```

Built-in PSDrives in PowerShell includes:

- `Alias`: Provides access to PowerShell aliases
- `Environment`: Provides access to environment variables
- `Function`: Provides access to PowerShell functions
- `Variable`: Provides access to PowerShell variables
- `Cert`: Provides access to certificates in the Windows certificate store
- `Cert:\CurrentUser`: Provides access to certificates in the current user's certificate store
- `Cert:\LocalMachine`: Provides access to certificates in the local machine's certificate store
- `WSMan`: Provides access to **Windows Remote Management (WinRM)** configuration data
- `C: and D`: (*and other drive letters*): Used to access the filesystem, just like in Windows Explorer
- `HKCU`: Provides access to the `HKEY_CURRENT_USER` registry hive
- `HKLM`: Provides access to the `HKEY_LOCAL_MACHINE` registry hive

▼ Making your code reusable

Reusability is an important aspect of coding that allows to create function, cmdlet, or module once and use it multiple times without having to rewrite the same code again and again.

Cmdlets

PowerShell command that performs a specific task and can be written in *C#* or in another *.NET* language.

- To find all ***cmdlets*** that are currently installed on machine:

```
Get-Command -CommandType Cmdlet
```

Functions

Functions are a collection of PowerShell commands that should be run following a certain logic.

Basic Structure of a Function:

```
function Verb-Noun {
<#
            <Optional help text>
#>
param(
        [data type] $Parameter
)
<...Code: Function Logic...>
}
```

Calling a Function:

```
Verb-Noun -Parameter "test"
```

Parameters:

Allow to pass values to functions, enhancing their flexibility and reusability.

Defining parameters:

```
function Invoke-Greeting {
        param (
                [string] $Name
        )
        Write-Output "Hello $Name!"
}
```

cmdletbinding:

cmdletbinding is a feature in PowerShell that allows to add common parameters like (`-Verbose`, `-Debug`, `-ErrorAction`) to functions and cmdlets without defining.

- `[CmdletBinding()]` makes PowerShell functions behave like advanced cmdlets.

Make a parameter mandatory in a function:

```
function Invoke-Greeting {
        [cmdletbinding()]
        param (
                [Parameter(Mandatory)]
                [string] $Name
        )
        Write-Output "Hello $Name!"
}
```

SupportsShouldProcess:

Adding `[CmdletBinding(SupportsShouldProcess)]`, enables the `-WhatIf` and `-Confirm` parameters in function.

To use `SupportsShouldProcess` effectively, will need to call `ShouldProcess()` for each item being processed.

Example Code:

```
function Invoke-Greeting {
        [CmdletBinding (SupportsShouldProcess) ]
        param (
            $Name
        )

        foreach ($item in $Name) {
            if ($PSCmdlet.ShouldProcess($item)) {
                    Write-Output "Hello $item!"
            }
    }
}
```

Accepting input via the pipeline:

Accepting input through pipeline can be done in two ways, ***by value*** or ***by property name***.

1. By Value (Positional Parameters):

Accept the input by passing values directly to the parameters of the function when calling it, and PowerShell automatically matches the positional parameters.

2. By Property Name (Named Parameters):

Specify the parameter names when calling the function, This allows to pass arguments in any order.

Example Code :

```
function Invoke-Greeting {
        [CmdletBinding()]
    param (
            [Parameter(Mandatory, ValueFromPipeline,
ValueFromPipelineByPropertyName)]
            [string] $Name
        )

        process {
                Write-Output "Hello $Name!"
        }
}
```

Calling the Function:

1. **By Value:**

   ```
   "Alice", "Bob" | Invoke-Greeting
   ```

2. **By property name:**

   ```
   [pscustomobject]@{Name = "Prash"} | Invoke-Greeting
   ```

Comment-based help:

Comments simplify the adjustment or reuse of function code.

```
<#
    .SYNOPSIS
    <Describe the function shortly.>

    .DESCRIPTION
    <More detailed description of the function.>

    .PARAMETER Name
    <Add a section to describe each parameter, if function has one or
    more parameters.>

    .EXAMPLE
    <Example how to call the function>
```

```
    <Describes what happens if the example call is run.>
    #>
```

Error Handling:

```
try{
        New-PSSession -ComputerName $Computer -ErrorAction Stop
}
catch {
        Write-Warning -Message "Couldn't connect to Computer: $Computer"
}
```

- Setting `ErrorAction` to `Stop` will treat the error as a terminating error.
- As terminating errors are caught, the action defined in the `catch` block is triggered.

Demo: Comprehensive PowerShell Script

```
function Write-HelloWord(){
        <#
            .SYNOPSIS
            This function writes "Hello World!" to the commanline.

            .DESCRIPTION
            This is just for learning purpose

            .PARAMETER Identity
            If the parameter is specified, an individual greetin is added.

            .EXAMPLE
            Write-HelloWord -Ideentity "prash"

            Writes the output "Hello Word! Hello prash!"
        #>

        [cmdletbinding()]
        param(
                [string]$Identity
        )

        if (![string]::IsNullOrEmpty($Identity)) {
                $appendStr = " Hello $Identity!"
        }
```

```
        else{
                $appendstr = ""
        }

        Write-Host "Hello World!$appendStr"
}

# Calling function without parameter
Write-HelloWord

# Calling function with added parameter identity
Write-HelloWord -Identity "prash"
```

Difference b/w *`cmdlets`* and `script cmdlets` (advanced functions):

Aspect	Cmdlets	Script Cmdlets (Advanced Functions)
Implementation	Written in a compiled language like C#	Written in PowerShell scripting language
Performance	Faster and more efficient due to compilation	Slightly slower due to interpretation
Complexity	More complex to develop and require compilation	Easier to write, modify, and debug
Deployment	Packaged as a DLL and loaded via modules	Included in scripts or modules as `.ps1` files
Extensibility	Can access the full power of the .NET framework	Limited to what PowerShell scripting can achieve
Example	`Get-Process` (built-in cmdlet)	`function Get-HelloWorld { [CmdletBinding()] ... }`
Use Case	When performance and low-level operations matter	For simpler automation tasks and rapid development

▼ Aliases

Aliases are shorthand or alternate names for cmdlets, functions, scripts, or commands.

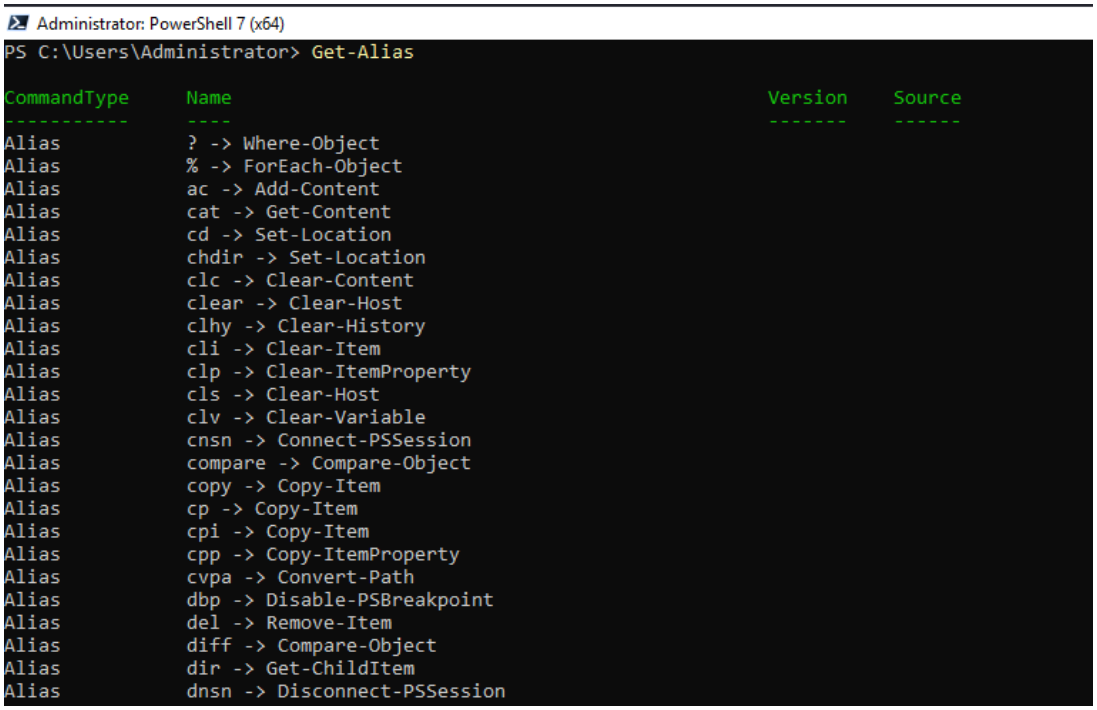
- To see all available cmdlets that have word `Alias` in name:

```
Get-Command -Name "*Alias*"
```

Working with Aliases:

Get-Alias:

To see all aliases that are currently configured on the computer.



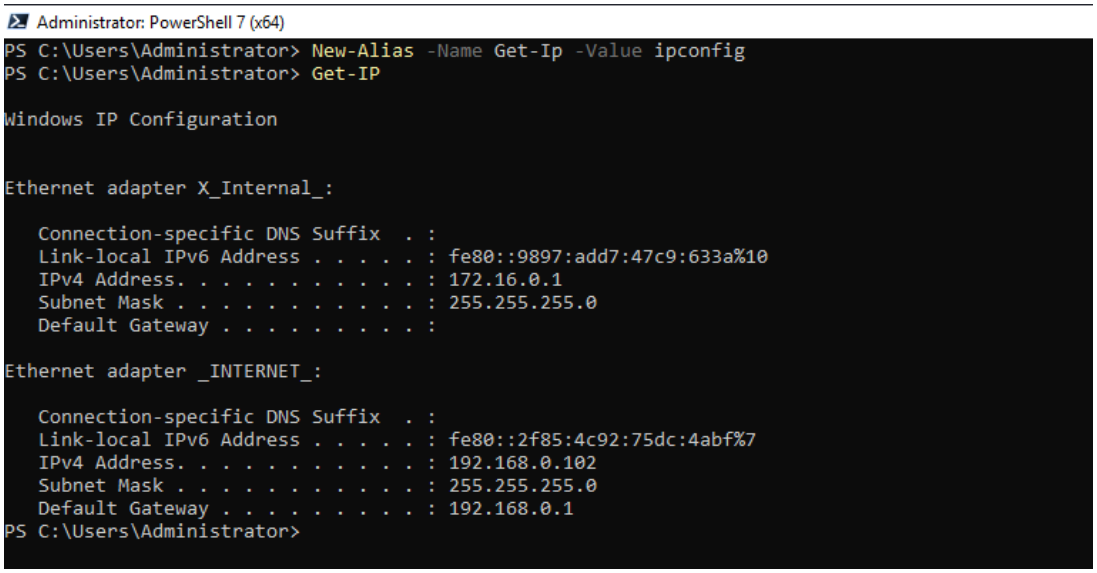
Output of the Get-Alias command

💡 `Get-Alias` can also be used to check if specific *alias* exists using the `-Name` parameter.

New-Alias:

Use `New-Alias` to create a new alias within the current PowerShell session.

```
New-Alias -Name -Get-Ip -Value ipconfig
```



Output of the GetIp alias

These *alias*es are not set *permanently*, so once session exit, alias will not work anymore.

> To make these aliases permanent configure them in PowerShell profile.

Export-Alias:

Export one or more aliases with `Export-Alias`.

- Export all aliases to a *.csv* file:

```
Export-Alias -Path "alias.csv"
```

- Export all aliases as script that can be executed :

```
Export-Alias -Path "alias.ps1" -As Script
```

- Export single alias using `-Name` parameter:

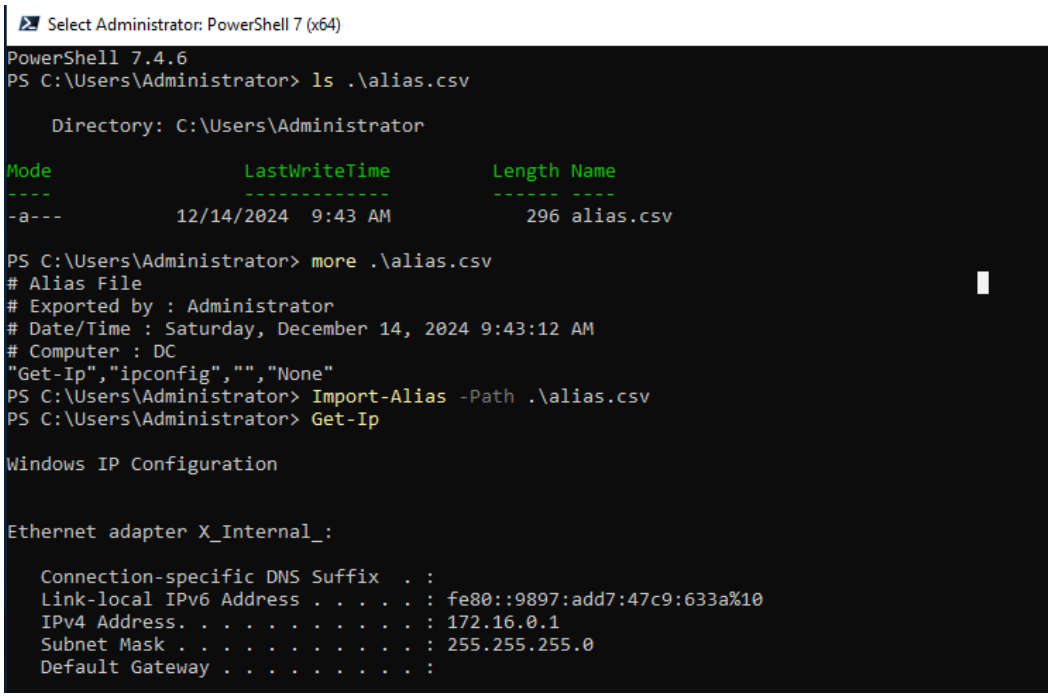
```
Export-Alias -Path "alias.ps1" -Name Get-Ip -As Script
```

Import-Alias:

Used to **import aliases from a file** into current PowerShell session.

- Importing above exported aliases through file:

```
Import-Alias -Path .\alias.csv
```



Importing aliases through file

▼ Modules

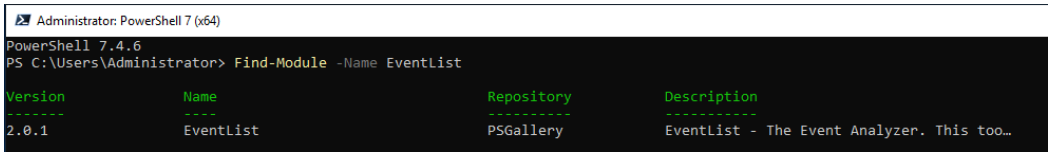
Modules are a collection of PowerShell commands and functions that can be easily shipped and installed on other systems.

> 💡 All installed modules on the system can be found in `PSModulePath` folders, part of the `Env:\` PSDrive:
>
> `Get-Item -Path Env:\PSModulePath`

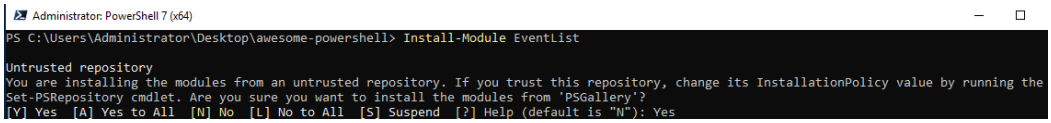
Working with modules:

1. Finding and installing modules

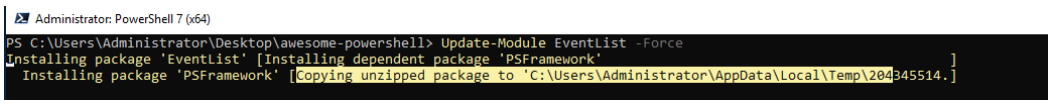
- Search for modules using `Find-Module -Name <modulename>`, which queries the repositories that are configured on operating system.



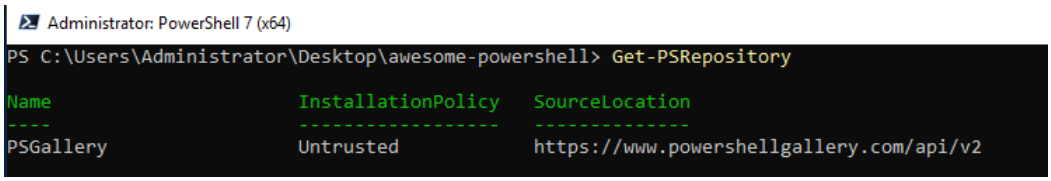
- Install module in to local system using `Install-Module <modulename>` :



- Updating module with `Update-Module <modulename> -Force` :



- To see which repositories available on system using `Get-PSRepository` :

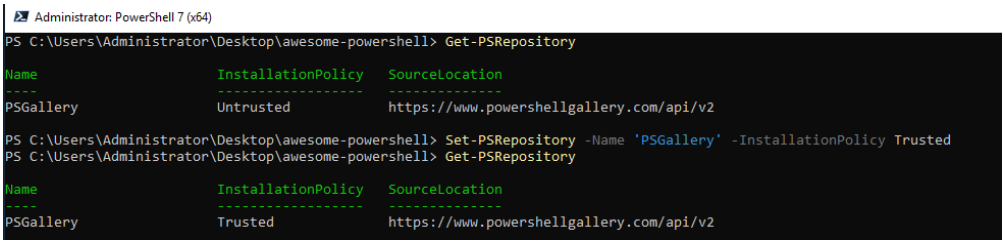


The ***PowerShell Gallery***, is the central repository for PowerShell content, which contains thousands of helpful modules, scripts and ***Desired State Configuration(DSC)*** resources.

> Using PowerShell Gallery to install modules directly, requires the `NuGet` and `PowerShellGet` to be installed.

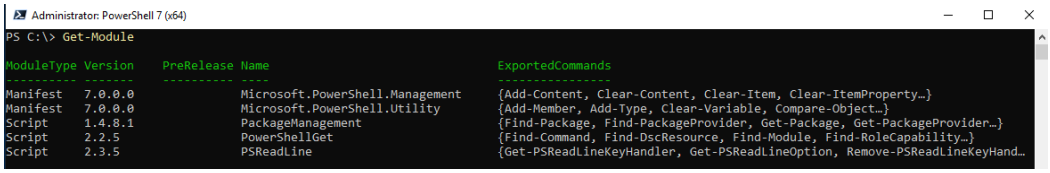
Configure PSGallery as trusted repository:

```
Set-PSRepository -Name 'PSGallery' -InstallationPolicy Trusted
```



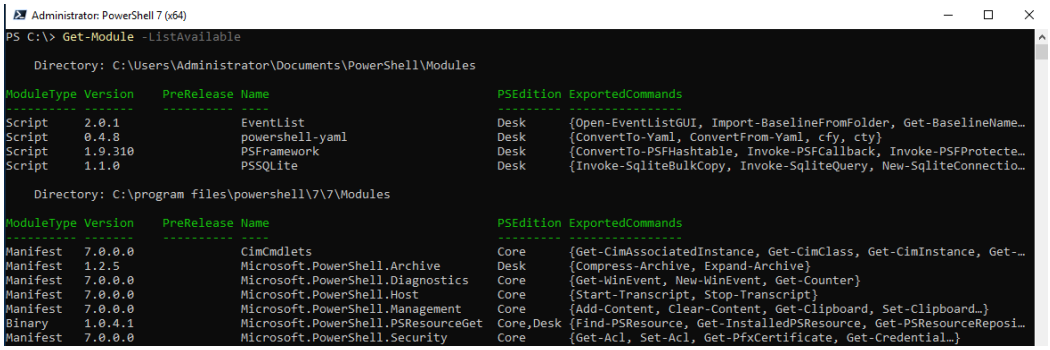
Configuring PSGallery as trusted repository

- To find already available module in the current session using `Get-Module` :



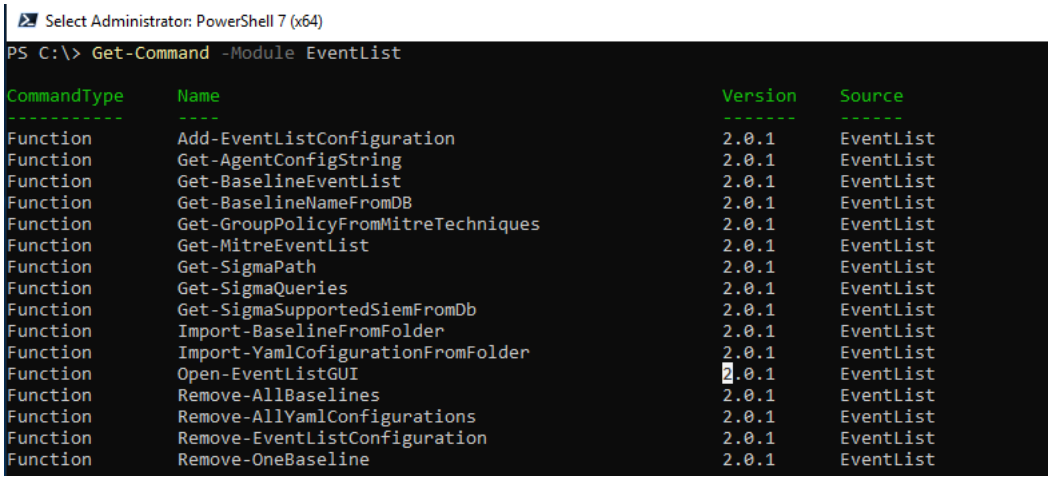
List of all available module in current session

- To see which modules are available to import, including that come pre-installed with Windows, using the `Get-Module -ListAvailable` :



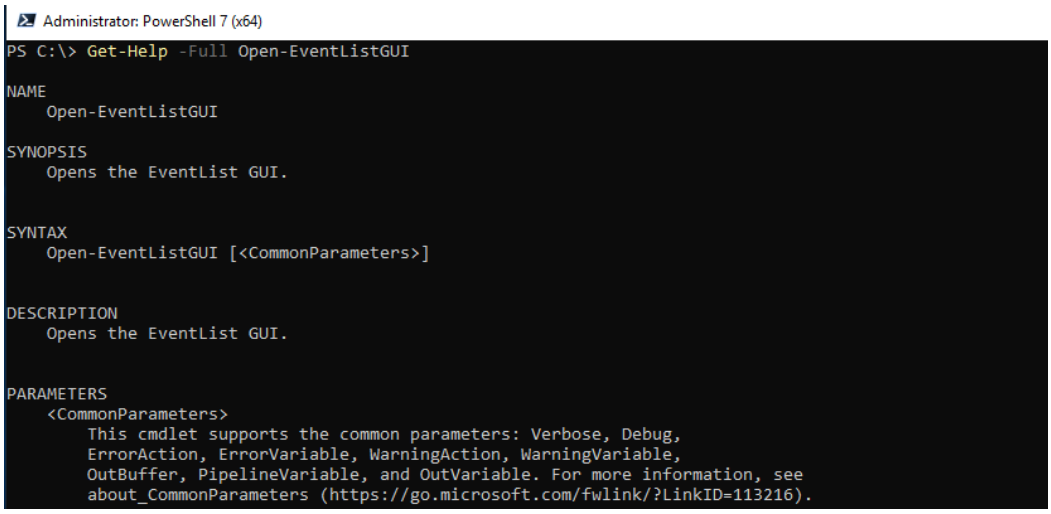
List of all available modules

- To find which commands are available in a module using `Get-Command -Module <modulename>`:



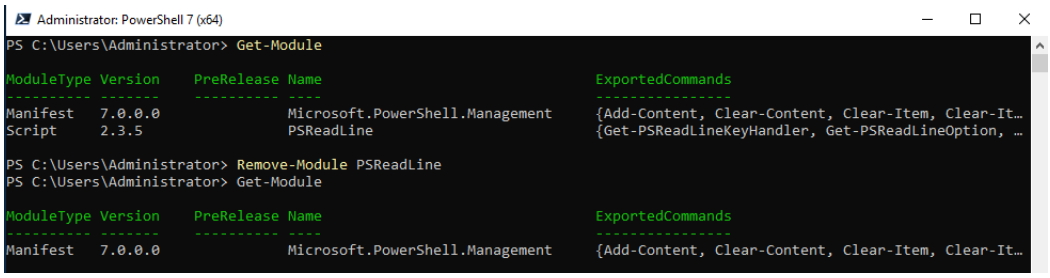
Listing all available commands of a module

- To know about usage of specific command use , `Get-Help -Full <command>` :



Getting help page of a command

- Unloads the module from current session using `Remove-Module <modulename>` :



Removing PSReadLine module from current session

2.Creating Your own Modules:

To make functions easier to ship to other systems, creating a module is a great way.

> Most necessary files commonly seen in modules — `.psm1` file and `.psd1` file.

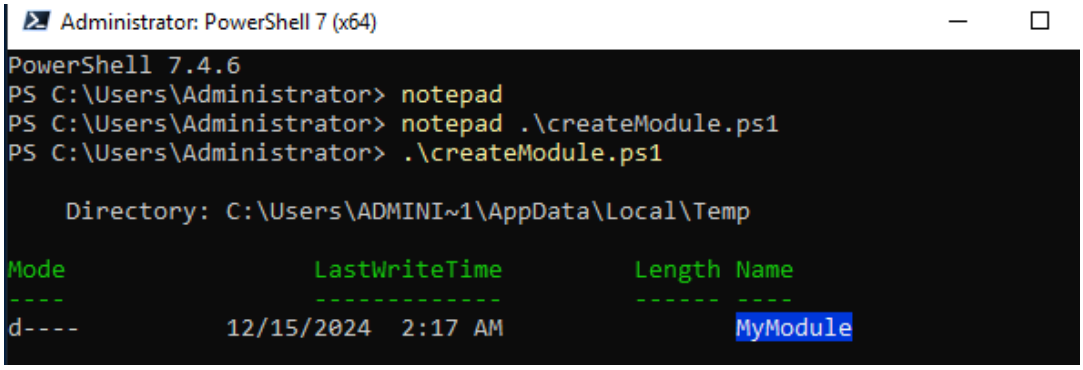
- **.psm1** file contains the scripting logic that module should provide, and can also use it to import other functions within a module.
- **.psd1** file is the manifest of module, which include information about the module.

Developing a basic module:

Module file ends with file extension of `.psm1`

1. Define the path where module should be save in the `$path` variable.
2. Use `New-ModuleManifest` cmdlet to create a new module manifest file.
3. The `-RootModule` parameter specifies the name of the PowerShell module file.
4. Using Set-Content cmdlet, create the `Module.psm1` file which contains the code logic.

```
$path = $env:TEMP + "\MyModule\"
if (!(Test-Path -Path $path)) {
        New-Item -ItemType directory -Path $path
}
New-ModuleManifest -Path $path\MyModule.psd1 -RootModule MyModule.psm1
Set-Content -Path $path/MyModule.psm1 -Value {
        function Invoke-Greeting{
            [Cmdletbinding()]
            param(
                [Parameter(Mandatory=$true)]
                [string] $Name
            )
            "Hello $Name!"
        }
}
```

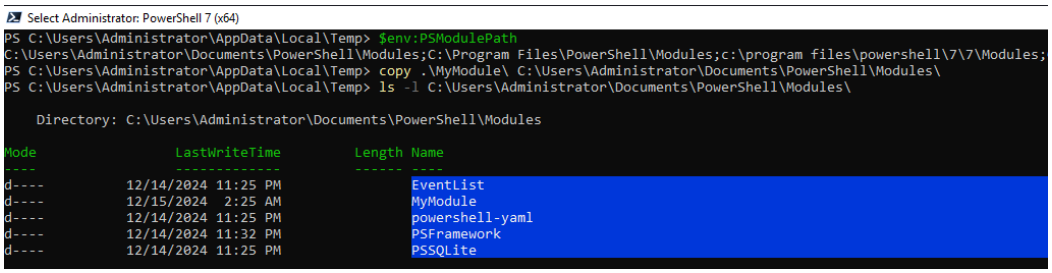


Executing the above code

> 💡 To use the module in PowerShell session, either import it directly into session or copy it into one of the `PSModule` paths.
>
> `PSModule` path are directories that are searched for modules when using `Import-Module` cmdlet.

- To see `PSModule` path using `$env:PSModulePath`:

5. Copy the module directory to location of `PSModule` path



Copying MyModule to one of the PSModulePath

6. Import the module into current session using `Import-Module MyModule` :

```
Import-Module MyModule
```

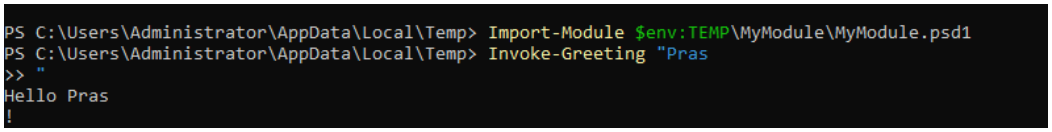
OR

- Importing the module directly from the `PSModule` path:

```
Import-Module $env:TEMP\MyModule\MyModule.psd1
```

6. Calling the function that defined in the MyModule module:

```
Invoke-Greeting -Name "Prashant"
```



Importing Module and calling defined function

Module Manifest Options allow to specify the author,the description, or modules that are required to install the module, using the `RequiredModules` hashtable.

Tools such as `PSModuleDevelopment` , are worth exploring.

▼ Further Readings

1. **System Namespace**: https://learn.microsoft.com/en-us/dotnet/api/system
2. **System.Management.Automation Namespace**: https://learn.microsoft.com/en-us/dotnet/api/system.management.automation
3. **about_Automatic_Variables**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables

4. **about_Environment_Variables**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_environment_variables
5. **about_Scopes**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scopes
6. **about_Comparison_Operators**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators
7. **about_Assignment_Operators**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_assignment_operators
8. **about_Operators**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operators
9. **ForEach-Object**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/foreach-object
10. **about_Break**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_break
11. **Approved Verbs for PowerShell Commands**: https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands
12. **Strongly Encouraged Development Guidelines**: https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/strongly-encouraged-development-guidelines
13. **Cmdlet Overview**: https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/cmdlet-overview
14. **Windows PowerShell Cmdlet Concepts**: https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/windows-powershell-cmdlet-concepts
15. **about_Functions_CmdletBindingAttribute**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_cmdletbindingattribute
16. **PowerShell Basics for Security Professionals Part 6 – Pipeline by Carlos Perez**: https://youtube.com/watch?v=P3ST3lat9bs
17. **About Pipelines**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_pipelines
18. **about_Aliases**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_aliases
19. **PowerShell Gallery**: https://www.powershellgallery.com/
20. **Writing a Windows PowerShell Module**: https://docs.microsoft.com/en-us/powershell/scripting/developer/module/writing-a-windows-powershell-module
21. **PowerShell Framework**: https://psframework.org/documentation/documents/psmoduledevelopment.html
22. **Everything you want to know about arrays**: https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-arrays

23. **Everything you want to know about hashtables**: https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-hashtable
24. **Everything you want to know about $null**: https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-null
25. **Everything you want to know about PSCustomObject**: https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-pscustomobject
26. **About functions**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions
27. **Functions 101**: https://docs.microsoft.com/en-us/powershell/scripting/learn/ps101/09-functions
28. **About functions' advanced parameters**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters
29. **Cmdlets versus functions**: https://www.leeholmes.com/blog/2007/07/24/cmdlets-vs-functions/
30. **Modules help pages**: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_modules