PowerShell 7 Guide & Documentation

Useful Resources for Learning PowerShell 7

- PowerShell Command Index: A-Z Index of PowerShell Commands
- This PowerShell 7 Guide & Documentation was created with heavy inspiration from "PowerShell 7 Essential Training" on LinkedIn Learning: PowerShell 7 Essential Training - LinkedIn Learning
    - The "PowerShell 7 Essential Training" can also be accessed on Workday, for Workday Training Credit: **[Link REDACTED For Security Purposes]**
- PowerShell 7 Master Guide: Learn / PowerShell / PowerShell 7.3 / Scripting and development / PowerShell Language Specification 3.0 / 3. Basic concepts

Requirements (How to Obtain PowerShell 7)

1. **The Latest Version of PowerShell 7** can be downloaded here: PowerShell 7 Releases - PowerShell/PowerShell - GitHub
    a. Click on the "Latest" release of PowerShell (Example: "v7.3.9 Release of PowerShell").
    b. Once you are on the GitHub Repository for the "Latest" PowerShell 7 version, scroll-down to the "Assets" section, and click the PowerShell version that matches your operating system, to download **PowerShell 7**.
        i. Example: For a 64-Bit Windows 10 OS, you would click-on, and download, "PowerShell-7.3.9-win-x64.msi".
2. Download the "Ex_Files_PowerShell_7_EssT.zip" and "Additional Example Scripts.zip" files attached to this documentation.
    a. The "Ex_Files_PowerShell_7_EssT.zip" file contains examples of PowerShell 7 Scripts from the PowerShell 7 Essential Training notes.
    b. The "Additional Example Scripts.zip" file contains PowerShell 7 Script examples made for this documentation.
3. Download, and install the "Stable" version of **Visual Studio Code:** Visual Studio Code
    a. Example: For a Windows 10 Operating System, the installer would be, "VSCodeUserSetup-x64-1.82.3.exe".
    b. **Note:** If your Operating System, or Operating System version is not listed, you can also click on "Other downloads", below the "Linux x64 " text, to view, and install other supported versions of Visual Studio Code.
4. In order to access, modify, and obtain **ActiveDirectory** objects, both with and without PowerShell, the **RSAT for Windows 10** file must be installed - RSAT for Windows 10

How to Install PowerShell 7 & Additional Applications

If you would like to install PowerShell 7 on a **Red Hat Enterprise Linux (RHEL)** device, follow this guide: How to Install PowerShell 7 on Red Hat Enterprise Linux (RHEL)

1. **Installing PowerShell 7 (.zip):** Once you have downloaded all of the required files for PowerShell 7, create a new folder called "PowerShell 7" where you want to store the PowerShell 7 files.
    a. Once you have created the folder, parse to where you downloaded **PowerShell 7** (PowerShell-7.3.9-win-x64.zip), and extract the files to the newly-created "PowerShell 7" folder.
    b. **To Launch PowerShell 7:** Double-click the **pwsh.exe** file in the newly-created "PowerShell 7" folder.
    c. Note: The "Other Settings & Useful Applications For PowerShell 7" page contains instructions on how to install **PowerShell Core Policy Definitions** and **PowerShell Remoting** functionality.

2. **Installing PowerShell 7 (.msi):** Browse to where you downloaded **PowerShell 7** (PowerShell-7.3.9-win-x64.msi), right-click the **.msi** file, and run the installer as Administrator.
    a. **Welcome to the PowerShell 7-x64 Setup Wizard:** Click "Next"
    b. **Destination Folder:** Use the **default option**, and click "Next".
    c. On the **Optional Actions** screen**:** Make sure only the following options are selected, and click "Next".
    [x] Add PowerShell to Path Environment Variable
    [x] Register Windows Event Logging Manifest
    [x] Add 'Open here' context menus to Explorer
    [x] Add 'Run with PowerShell 7' context menus for PowerShell files
        i. Do not check the "Enable PowerShell remoting" box, *unless you would like to use PowerShell Remoting on this device.*
            1. **DO NOT ENABLE POWERSHELL REMOTING, UNLESS THE DEVICE IS ON AN ISOLATED NETWORK, OR YOU KNOW WHAT YOU'RE DOING**
            2. **On Open/Unprotected Networks:** Enabling PowerShell Remoting can **open your device to vulnerabilities**, and can even lead to your **device being compromised by an attacker**.
    d. **Use Microsoft Update to help keep your computer secure and up to date:** Use the **default options**, and click "Next".
    e. **Ready to install PowerShell 7-x64:** Click the "Install" button, and PowerShell 7 will begin to install on your device, and when it says "Installation completed successfully." click "Finish".
    f. Open/Click on the Windows Search Bar, type "PowerShell", and click on **PowerShell 7 (x64)**.
        i. **Note:** Depending on what commands, or scripts, you will be executing, you may need to run PowerShell 7 as Administrator.

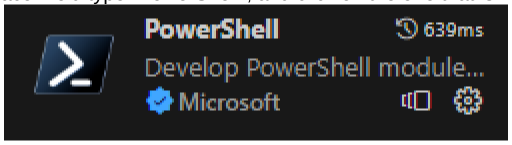
3. **Installing Visual Studio Code:** Parse to where you downloaded the **Visual Studio Code Installer** (VSCodeUserSetup-x64-1.82.2.exe), right-click the **.exe** file, and run the installer as Administrator.
    a. **License Agreement:** Click "I accept the agreement", and click "Next >".
    b. **Select Destination Location:** Use the **default option**, and click "Next >".
    c. **Select Start Menu Folder:** Use the **default option**, and click "Next >".
    d. **Select Additional Tasks:** Make sure **all of the options are selected**, and click "Next >".
    e. **Ready to Install:** Click the "Install" button and Visual Studio Code will begin to install on your device.

            i. When it says "Completing the Visual Studio Code Setup Wizard", un-check the box that says "Launch Visual Studio Code", and click "Finish".
        f. **To Open Visual Studio Code:** Open the Windows Search Bar, type "Visual Studio Code", and click on **Visual Studio Code**.
            i. **Note:** Setting-up Visual Studio Code for PowerShell 7 will be covered later in the guide.

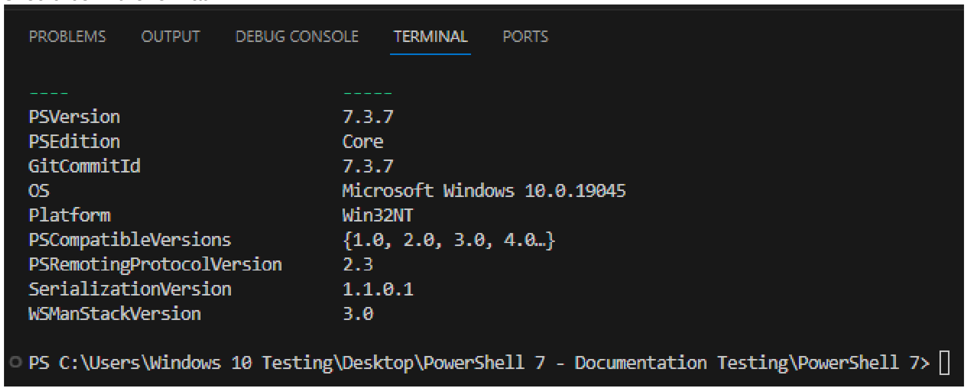
4. After you have **Finished Installing All of the Applications and Tools, Restart Your Device**.

5. **Setting-Up Visual Studio Code for PowerShell 7:** Once you have installed PowerShell 7 and Visual Studio Code, open the Windows Search Bar, type "Visual Studio Code", and click on **Visual Studio Code**.
    a. In Visual Studio Code, click "File" in the top-left corner, and click "Open Folder...".
    b. In the "Open Folder..." window, parse to the directory of your choice (where you want to Store PowerShell Code/Scripts), create a new folder called "PowerShell 7", click on the newly-created **PowerShell 7** folder, and click "Select Folder".
        i. If you get a prompt, "Do you trust the authors of the files in this folder?", click the button that says, "Yes, I trust the authors".

    c. In the top-left corner of Visual Studio, under "Explorer", to the right of the "PowerShell 7" folder name, click the **New File... Icon**, type "test.ps1" in the prompt that appears, and press the "Enter" key.
        i. If done correctly Visual Studio will open a blank PowerShell file, called **test.ps1**.
    d. After you create this file, there should be a prompt in the bottom-right corner that says, "Do you want to install the recommended 'PowerShell' extension from Microsoft for the PowerShell language?", click the blue "Install" button, and the **PowerShell Extension for Visual Studio Code** will begin to install.

        i. If this prompt **did not appear**, click on the **Extensions Icon** on the left-hand side-bar, in the "Search Extensions in Marketplace" field type "PowerShell", and click on the one that is **Verified by Microsoft**, and click the blue "Install" button.
            1. 

    e. Now that the "PowerShell" extension has been installed, click on the "test.ps1" tab at the top, and type **$PSVersionTable** on Line 1.

    f. Click "File" in the top-left corner, click "Save", and click the **Run button** on the top-right.
        i. If done correctly, the "Terminal" should have output a table with all the information for your PowerShell Version. Your **PSVersion** should be in the format: **7.#.#.**
        ii. 

```
----                           -----
PSVersion                      7.3.7
PSEdition                      Core
GitCommitId                    7.3.7
OS                             Microsoft Windows 10.0.19045
Platform                       Win32NT
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0…}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0

PS C:\Users\Windows 10 Testing\Desktop\PowerShell 7 - Documentation Testing\PowerShell 7>
```

        iii. If the "PSVersion**" was not 7.#.#**, make sure your "test.ps1" file has **.ps1** at the end of it, and that the "PowerShell" Visual Studio Code Extension has been installed.
    g. Visual Studio Code is now all setup for PowerShell 7!

6. All necessary applications and tools are installed and setup, now you can start experimenting with, and learning about PowerShell 7! 🙂

PowerShell 7 Use-Cases

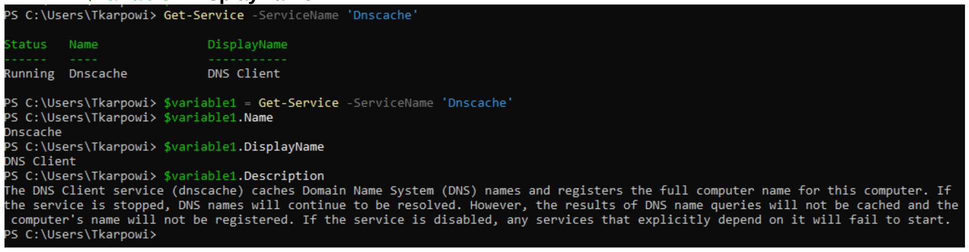
- PowerShell 7 has a variety of use cases:
    - Manage ActiveDirectory users and groups
    - Save command outputs to files
    - Search for, and select, a variety of different objects and contents from files, such as log files.
    - Save a great deal of clicking various different applications, and folders, when it comes to searching for, moving, and copying files.
- Scripting in PowerShell 7 can also be used to automate a variety of tedious tasks, and drastically increase the efficiency of various tasks that you are working on.
- Make sure you have completed the "PowerShell 7 - Information & Installation/Setup Guide" page, so that you have all the necessary tools and applications for utilizing PowerShell 7.
- "PowerShell 7 Essential Training" example files: Ex_Files_PowerShell_7_EssT.zip

Utilizing PowerShell 7 - Table of Contents

PowerShell 7 Syntax

1. PowerShell commands use a **Verb-Noun** structure.
    a. PowerShell commands will always start with **Verb**, and end with a **Noun**.
    b. Examples
        i. **Get-Module**
        ii. **Save-Help**
        iii. **Out-Host**

2. Command Options: **<Command> -Option**
    a. Command options target, and execute, a specific function that is part of a command, rather than executing all functions in the command.
    b. If you type "**-**", you can press the "Tab" key to cycle through the available **options** for a given command.
        i. **Get-Command -Verb** "Add"
        ii. **Get-Command -Noun** "Computer"
    c. **Note:** Anything that is in single **''**, or double quotes **""** tells PowerShell, whatever is in **""**, spaces included, is the phrase you would like to match to a cmdlet.

3. Variables: **$variable1**
    a. PowerShell variables are objects which store data, such as:
        i. Commands
        ii. Integers
        iii. Strings, and more.
    b. Variables in PowerShell usually follow the naming convention, lower-case first letter of the first word, and then Upper-Case First Letter of the following words.
        i. **$varTest**
        ii. **$varGetService**
    c. PowerShell variables can even be used like commands.
        i. **$variable1** = **Get-Service -ServiceName** 'Dnscache'
        ii. **$variable1.Name**
        iii. **$variable1.DisplayName**
    d. 

```
PS C:\Users\Tkarpowi> Get-Service -ServiceName 'Dnscache'

Status   Name               DisplayName
------   ----               -----------
Running  Dnscache           DNS Client

PS C:\Users\Tkarpowi> $variable1 = Get-Service -ServiceName 'Dnscache'
PS C:\Users\Tkarpowi> $variable1.Name
Dnscache
PS C:\Users\Tkarpowi> $variable1.DisplayName
DNS Client
PS C:\Users\Tkarpowi> $variable1.Description
The DNS Client service (dnscache) caches Domain Name System (DNS) names and registers the full computer name for this computer. If
the service is stopped, DNS names will continue to be resolved. However, the results of DNS name queries will not be cached and the
 computer's name will not be registered. If the service is disabled, any services that explicitly depend on it will fail to start.
PS C:\Users\Tkarpowi>
```

    e. For variables in PowerShell, if you type a period "**.**", you can press the "Tab" key to cycle through the available options for a given variable, just like using the "-" for commands.

Operators In PowerShell 7

1. **Pipe Operator: |**
    a. This will take the output from one command (on the left-side of the **|**), and send it to the next command (on the right side of the **|**) as an i nput.
    b. **Example:** Get-Command **|** Where-Object { **$_.parametersets.count** -gt **2** } **|** Format-List **Name**
        i. First the "Get-Command" command is executed, and that output is sent to the "Where-Object" command.
        ii. Next, the "Where-Object" command takes that input, and simplifies it to where the objects meet the defined parameters "{}".
        iii. Then, the output of the "Where-Object" command is sent to the "Format-List" command, where the output is further-simplified to only show the **Name** of the command objects.
        iv. This last "Format-List" command output is then output to the terminal.

2. **Asterisk: ***
    a. This operator accepts zero, or more, values before <"*****.txt">, or after <"text*****">, in a specified phrase.
    b. **Example:** Get-ChildItem **.\Downloads** -Filter **"*.zip"**
        i. First the "Get-ChildItem" command is executed on the ".\Downloads\" directory.
        ii. Next, the "-Filter" option takes that input, and simplifies it to where the objects contain a **".zip"** at the end of the file's name.
    c. **Example:** Get-ChildItem **.\Downloads** -Filter **"U_*"**
        i. First the Get-ChildItem command is executed on the ".\Downloads\" directory.
        ii. Next, the "-Filter" option takes that input, and simplifies it to where the object has a **"U_"** at the beginning of the file's name.
        iii. **Tip:** This can be used for obtaining a sorted list of all DISA STIG files.

3. **Double Ampersand: &&**
    a. This operator executes the command to the right of it, if the first command was successful.
    b. **Example:** Write-Host "Primary Message" **&&** Write-Host "Secondary Message"
        i. First the "Write-Host" command is executed, and then "Primary Message" is output to the terminal.
        ii. Next, since the **first command worked successfully**, the second "Write-Host" command is executed, and outputs "Secondary Message" to the terminal.
    c. **Example:** Primary Message **&&** Write-Host "Secondary Message"
        i. Since "Primary Message" is not an actual cmdlet, the terminal will output an error to the console.

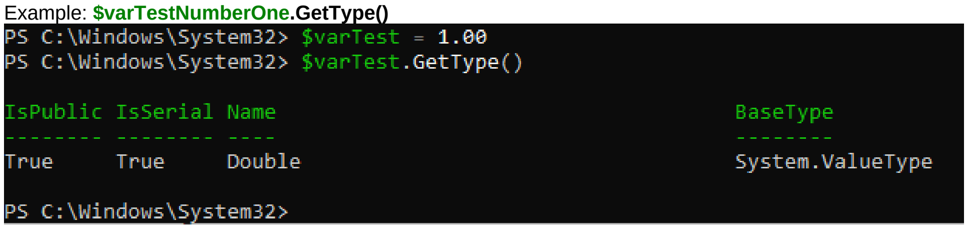
        ii. Next, since the **first command failed**, the second Write-Host command **will not be executed**.

4. **Double Pipe Operator: ||**
    a. The **Double Pipe Operator** will execute the command to the right-side of it, only if the first command was **unsuccessful**.
    b. **Example:** Write-Error "Primary Error" **||** Write-Host "Secondary Message"
        i. First the "Write-Error" command is executed, and outputs an error message "Write-Error: Primary Error" to the terminal.
        ii. Next, since the **first command failed**, the "Write-Host" command is executed, and outputs "Secondary Message", after the error message, to the terminal.
    c. **Example:** Write-Host "Primary Error" **||** Write-Host "Secondary Message"
        i. First the "Write-Error" command is executed, and outputs "Primary Error" to the terminal.
        ii. Next, since the **first command worked successfully**, the second "Write-Host" command is **not executed**, and the cmdlet has completed.

5. **Coalescing Operators: ??, ??=, ?., ?[]**
    a. These operators are **NULL Conditional Access Operators**.
        i. This means, if a variable on the left-hand side of the operator doesn't have anything (the variable is NULL), execute the action, or command, on the right side of the operator.
    b. Example 1 (Part 1): **$variable1** = "test"
    c. Example 1 (Part 2): **$variable1** **??** "No Value is Found"
        i. First **$variable1** is initialized to "test", a non-NULL value.
        ii. Next, since the variable is a String value, **not-NULL**, the terminal will output "test".
    d. Example 2 (Part 1): **$variable2**
    e. Example 2 (Part 2): **$variable2** **??** "No Value is Found"
        i. First **$variable2** is initialized, but not given a value, which makes it a NULL value.
        ii. Next, since the variable **is a NULL** value, the terminal will output "No Value is Found".

Variable Types & Casting Variables In PowerShell 7

Useful Page on PowerShell 7 Variables: Learn / PowerShell / PowerShell 7.3 / Scripting and development / PowerShell Language Specification 3.0 / 4. Types

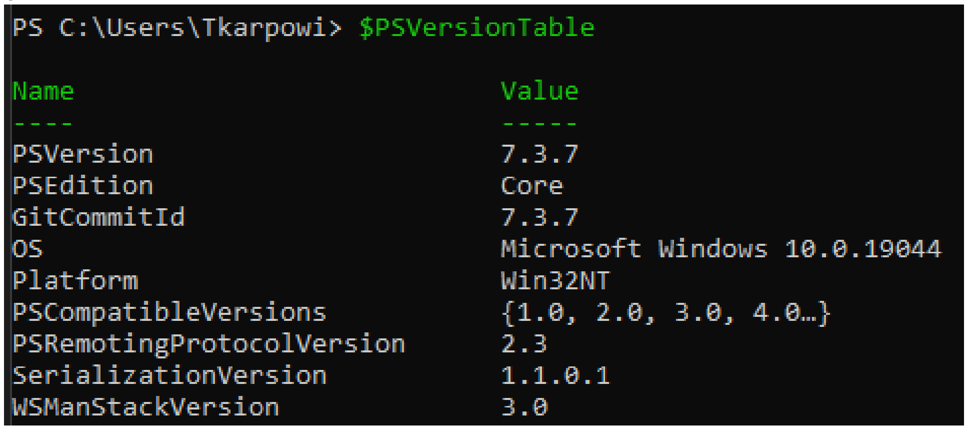
1. When declaring PowerShell variables, the formatting is usually lower-case first letter of the first word, and then Upper-Case First Letter of the following words.
    a. Example: **$varTest**
    b. Example: **$varTestNumberOne**
2. Variables can be created in PowerShell to hold multiple different types of values.
    a. Likewise, variables can also be casted to other types of values, to fit a specific use-case.
3. **GetType()**
    a. To check a variable's type, you can use the **GetType()** method.
    b. Example: **$varTest.GetType()**
    c. Example: **$varTestNumberOne.GetType()**
    d. 

```
PS C:\Windows\System32> $varTest = 1.00
PS C:\Windows\System32> $varTest.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Double                                   System.ValueType

PS C:\Windows\System32>
```

Variable Type	Cast Type (Notation)	Details	Example 1	Example 2
32-Bit Integer	**[int32]**	This is an **Integer Cast Type**, and it will temporarily cast the variable to a **32-Bit Integer** value.	PS> **$varTest** = **"1.00"** PS> **[int32]** **$varTest** **Output:** **1** The above example takes a variable, with the type [string], and casts it into a **32-Bit Integer** value.	PS> **$varTest** = **1.00** PS> **[int32]** **$varTest** **Output:** **1** The above example takes a variable, with the type [double], and casts it into a **32-Bit Integer** value.
64-Bit Integer (Long)	**[int64]** and **[long]**	These are both **Long Integer Cast Types**, and it will temporarily cast the variable to a **64-Bit Integer** value.	PS> **$varTest** = **"1.00"** PS> **[int64]** **$varTest** **Output:** **1** The above example takes a variable, with the type [string], and casts it into a **64-Bit Integer** value.	PS> **$varTest** = **1.00** PS> **[long]** **$varTest** **Output:** **1** The above example takes a variable, with the type [double], and casts it into a **64-Bit Integer** value.

Single (Float)	**[single]** and **[float]**	This is a **Single (Float) Cast Type**, and it will temporarily cast the variable to a **32-Bit Single-Precision** value.	PS> $varTest = **"1.00"** PS> **[float]** $varTest **Output:** **1** This example takes a variable, with the type [string], and casts it into a **Single (Float)** value.	
Double	**[double]**	This is a **Double Cast Type**, and it will temporarily cast the variable to a **64-Bit Double-Precision** value.	PS> $varTest = **"1.00"** PS> **[double]** $varTest **Output:** **1** This example takes a variable, with the type [string], and casts it into a **Double** value.	
String	**[string]**	This is a **String Cast Type**, and it will temporarily cast the variable to a **String** value.	PS> $varTest = **1** PS> **[string]** $varTest **Output:** **1** This example takes a variable, with the type [int32], and casts it into a **String** value.	
Boolean	**[boolean]**	This is a **Boolean Cast Type**, and it will temporarily cast the variable to a **Boolean (True or False)** value.	PS> $varTest = **1** PS> **[boolean]** $varTest **Output:** **True** This example takes a variable, with the type [int32], and casts it into a **Boolean** value, in this case **True**.	PS> $varTest = **0** PS>**[boolean]** $varTest **Output:** **False** This example takes a variable, with the type [int32], and casts it into a **Boolean** value, in this case "**False**".
Date-Time	**[datetime]**	This is a **Date-Time Cast Type**, and it will temporarily cast the variable to a variable that has both a **Date and Time** associated with it. • This can be used for creating logs with time-stamps on them, • This can be used for sending calculated time to various programs.	PS> $varTest = **"Monday, October 9, 2023, 7:44:00pm"** PS> **[datetime]** $varTest **Output:** **Monday, October 9, 2023 7:44:00 PM** This example takes a variable, with the type [string], and casts it into a variable **Formatted for Date and Time.**	

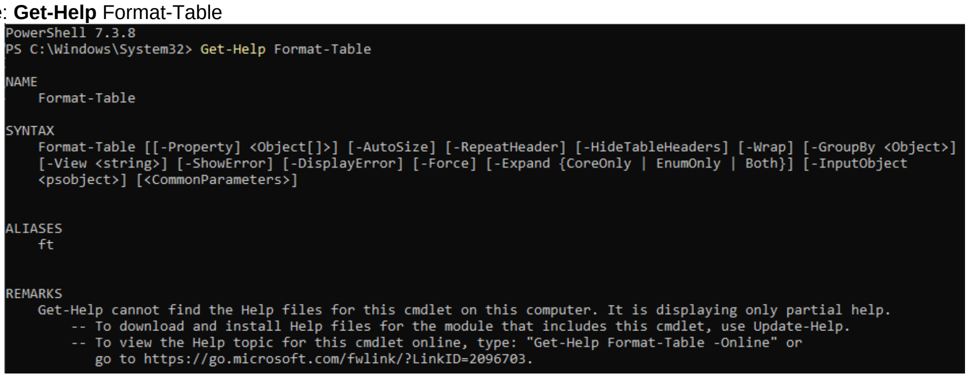
Useful Commands In PowerShell 7

1. **$PSVersionTable.<Value>**
    a. **$PSVersionTable** - This built-in PowerShell 7 variable outputs a table of the PowerShell version, and other details for your instance of PowerShell.
        i. 

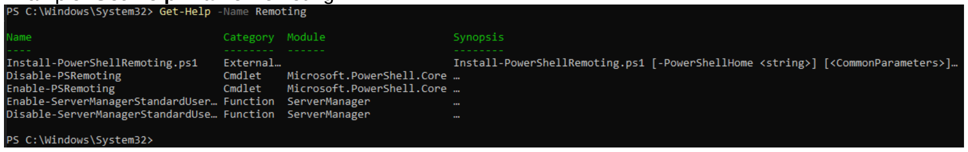
```
PS C:\Users\Tkarpowi> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      7.3.7
PSEdition                      Core
GitCommitId                    7.3.7
OS                             Microsoft Windows 10.0.19044
Platform                       Win32NT
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0…}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

    b. **$PSVersionTable.PSVersion** - Outputs only the PowerShell version number.

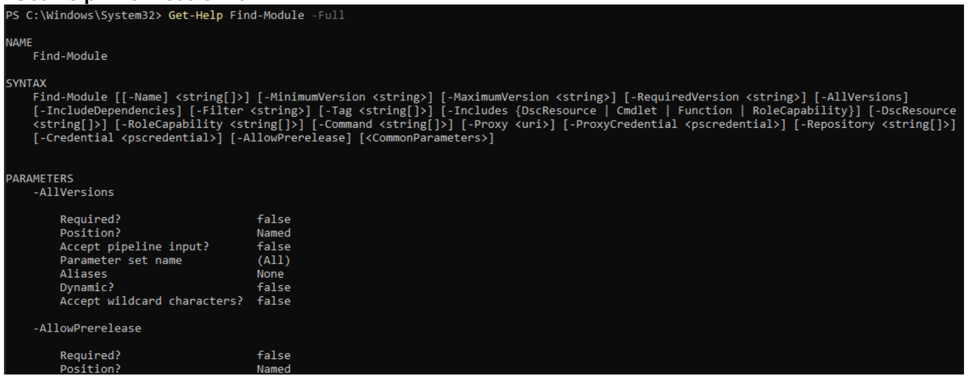
   c. **$PSVersionTable.OS** - Outputs only the name and version of the operation system.

2. **Get-Help** <Command>
   a. This command will display information about PowerShell commands and concepts to the terminal.
   b. **Information Displayed**
      i. **Name:** The name of the command.
      ii. **Syntax:** The syntax for the command, including available options and the command's structure.
      iii. **Aliases:** The short-hand aliases for the command.
      iv. **Remarks:** Displays any "Help files" for the cmdlet, and module, as well as a link for the "Help topic" for the given cmdlet online, on Microsoft's website (if applicable).
   c. Example: **Get-Help** Format-Table
      i. 

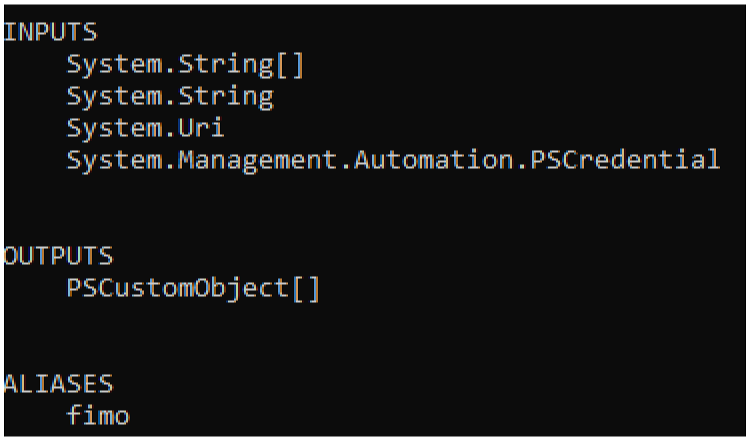
```
PowerShell 7.3.8
PS C:\Windows\System32> Get-Help Format-Table

NAME
    Format-Table

SYNTAX
    Format-Table [[-Property] <Object[]>] [-AutoSize] [-RepeatHeader] [-HideTableHeaders] [-Wrap] [-GroupBy <Object>]
    [-View <string>] [-ShowError] [-DisplayError] [-Force] [-Expand {CoreOnly | EnumOnly | Both}] [-InputObject
    <psobject>] [<CommonParameters>]


ALIASES
    ft


REMARKS
    Get-Help cannot find the Help files for this cmdlet on this computer. It is displaying only partial help.
        -- To download and install Help files for the module that includes this cmdlet, use Update-Help.
        -- To view the Help topic for this cmdlet online, type: "Get-Help Format-Table -Online" or
           go to https://go.microsoft.com/fwlink/?LinkID=2096703.
```

   d. **<**Command**> -?**
      i. You could also put **-?** at the end of a command, or cmdlet, to get help with the command itself.
      ii. This short-hand expression works essentially the same way as the **Get-Help** command.
      iii. **Example:** Format-Table **-?**
   e. Example: **Get-Help -Name** <String>
      i. The **-Name** option allows you to specify the name of a cmdlet, function, provider, script, workflow, conceptual article name, and an alias, that you need help with.
      ii. The **-Name** option will also allow you to specify the name of the command that you are looking to get help with, like the normal **Get-Help** <Command> cmdlet.
         1. This option is only necessary for the name of a command, if you have multiple options in the **Get-Help** command.
      iii. Example: **Get-Help -Name** Remoting
      iv. 

```
PS C:\Windows\System32> Get-Help -Name Remoting

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
Install-PowerShellRemoting.ps1    External…                           Install-PowerShellRemoting.ps1 [-PowerShellHome <string>] [<CommonParameters>]…
Disable-PSRemoting                Cmdlet    Microsoft.PowerShell.Core …
Enable-PSRemoting                 Cmdlet    Microsoft.PowerShell.Core …
Enable-ServerManagerStandardUser… Function  ServerManager             …
Disable-ServerManagerStandardUse… Function  ServerManager             …

PS C:\Windows\System32>
```

3. **Get-Help** <Command> **-Full**
   a. The **-Full** option will display the entire help article for a cmdlet.
      i. This option will display everything that the **Get-Help** command displays to the terminal.
      ii. This option will also display the "Parameters", "Inputs", "Outputs", and command examples, for a given command.
   b. **Information Displayed**
      i. **Name:** The name of the command.
      ii. **Syntax:** The syntax for the command, including available options and the command's structure.
      iii. **Parameters:** All of the available command options, and some details about these options.
      iv. **Inputs:** The acceptable input object types for the command in question.
      v. **Outputs:** The resulting output object types for the command in question.
      vi. **Aliases:** The short-hand aliases for the command.
      vii. **Remarks:** Displays any "Help files" for the cmdlet, and modules (if applicable), as well as a link for the "Help topic" for the given cmdlet online, on Microsoft's website.
   c. Example: **Get-Help** Find-Module **-Full**
      i. 

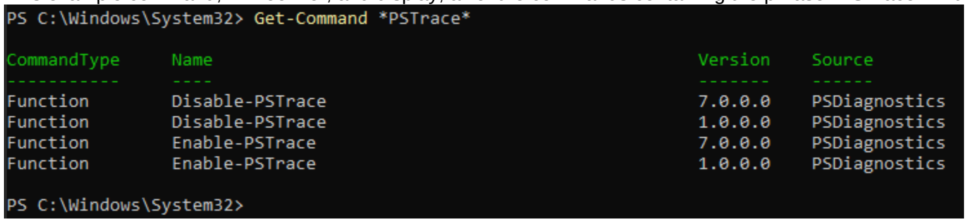
```
PS C:\Windows\System32> Get-Help Find-Module -Full

NAME
    Find-Module

SYNTAX
    Find-Module [[-Name] <string[]>] [-MinimumVersion <string>] [-MaximumVersion <string>] [-RequiredVersion <string>] [-AllVersions]
    [-IncludeDependencies] [-Filter <string>] [-Tag <string[]>] [-Includes {DscResource | Cmdlet | Function | RoleCapability}] [-DscResource
    <string[]>] [-RoleCapability <string[]>] [-Command <string[]>] [-Proxy <uri>] [-ProxyCredential <pscredential>] [-Repository <string[]>]
    [-Credential <pscredential>] [-AllowPrerelease] [<CommonParameters>]


PARAMETERS
    -AllVersions

        Required?                    false
        Position?                    Named
        Accept pipeline input?       false
        Parameter set name           (All)
        Aliases                      None
        Dynamic?                     false
        Accept wildcard characters?  false

    -AllowPrerelease

        Required?                    false
        Position?                    Named
```

ii. 

```
INPUTS
    System.String[]
    System.String
    System.Uri
    System.Management.Automation.PSCredential


OUTPUTS
    PSCustomObject[]


ALIASES
    fimo
```

d. Example: **Get-Help** Add-Member **-Full** | Out-String -Stream | Select-String -Pattern "MemberType"
   i. This cmdlet is only useful if you have help files, for given a cmdlet, installed your computer.
   ii. This cmdlet will search for a word in a locally available cmdlet help file, to find help information for the command that you are looking for.
   iii. This example will also show you the cmdlet position of the "MemberType" parameter, in the Add-Member cmdlet.

e. **Help [Command]**
   i. The **Help** command will display everything that the **Get-Help** command, with the **-Full** option, displays to the terminal.
   ii. The **Help** command will output part of the **-Full** output, and you will have to press the "Enter" key to manually load the next line of help information.
   iii. This command is essentially the short-hand version of the **Get-Help** <Command> **-Full** cmdlet.
   iv. Example: **Help** Find-Module

4. **Get-Command [-Verb | -Noun]** <"Phrase">
   a. This command will get all of the commands related to the <"Phrase"> that is specified, and output them to the terminal.
   b. **Note:** PowerShell is built on a Verb-Noun structure, when it comes to commands (cmdlets), the **first word** will be a **verb**, and the **secon d word** will be a **noun**.
   c. Example: **Get-Command** *PSTrace*
      i. This example command, will look for, and display, all of the commands containing the phrase "PSTrace" in it.
      ii. 

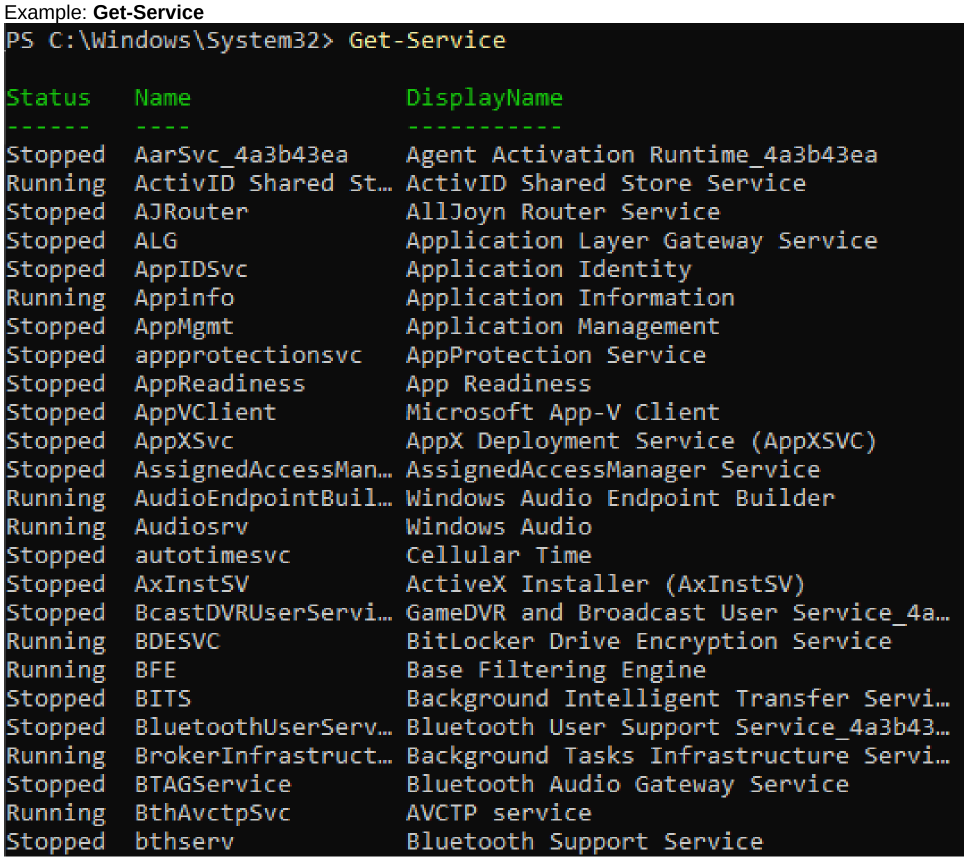
```
PS C:\Windows\System32> Get-Command *PSTrace*

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Disable-PSTrace                                    7.0.0.0    PSDiagnostics
Function        Disable-PSTrace                                    1.0.0.0    PSDiagnostics
Function        Enable-PSTrace                                     7.0.0.0    PSDiagnostics
Function        Enable-PSTrace                                     1.0.0.0    PSDiagnostics

PS C:\Windows\System32>
```

   d. **Get-Command -Verb** <"Phrase">
      i. The **-Verb** option specifies that the phrase in question belongs at the beginning of the command.
      ii. Example: **Get-Command -Verb** "Add"
      iii. This example command, will look for, and display, all of the commands containing the phrase "Add", at the start of the cmdlet.
   e. **Get-Command -Noun** <"Phrase">
      i. The **-Noun** option specifies that the phrase in question belongs at the end of the command.
      ii. Example: **Get-Command -Noun** "Computer"
      iii. This example command, will look for, and display, all of the commands containing the phrase "Computer", at the end of the cmdlet.
   f. **Find-Command** **[**-Verb | -Noun **]** <"Phrase">
      i. The **Find-Command** command was made for PowerShellGet 2.x.
      ii. **Find-Command** works essentially the same way as "Get-Command", but with less options, and more limited functionality.
      iii. "Get-Command" is used in, and built for PowerShell 7. Get-Command also has way more available options than the **Find-Command** command.
      iv. Example: **Find-Command** -Verb "Add"

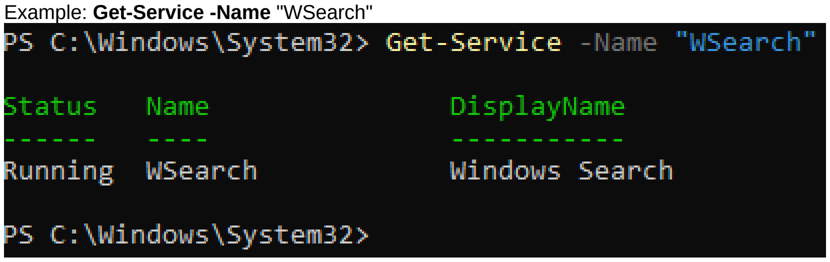
5. **Get-ComputerInfo**
   a. This command outputs a variety of different details, and information, about the computer hosting the current PowerShell 7 terminal session.
   b. **Key Information Displayed**
      i. **WindowsProductId:** The Windows License Key for the PC.
      ii. **BiosBIOSVersion:** Information about the BIOSDescription, BIOSName, and BIOSManufacturer, all in one section.
      iii. **CsName:** The name of the computer, as it appears to other devices on the network.
      iv. **OSName:** The type of Windows Operating System that is installed on the PC (Example: Microsoft Windows 10 Pro).
      v. **OSVersion:** The Windows Operating System version number for the PC.

6. **Get-Service [** -Name **]** <"Search String">
   a. This command will obtain all of the services on a computer, matching the <"Search String">, and output them as object values.
      i. This command will retrieve both running and stopped services.
      ii. The <"Search String"> is where you put the name for the service that you want to obtain information on.
   b. **Information Displayed**
      i. **Status:** The state of the service ("Running" or "Stopped").

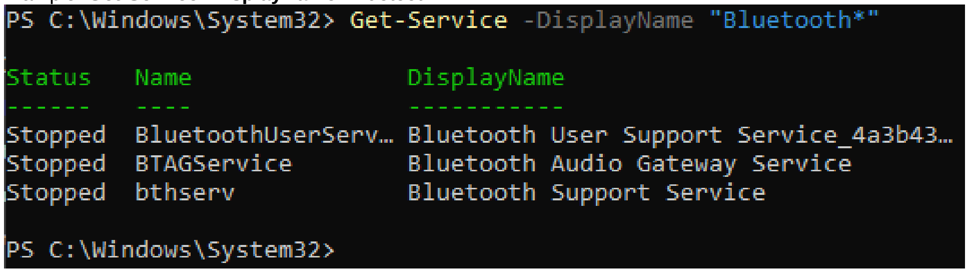
ii. **Name:** The system's name for the service.
iii. **DisplayName:** The user-recognized display name for the system service.

c. If the **Get-Service** command is run without parameters, without a <"Search String">, all of the local computer's running and stopped services will be output to the terminal.
   i. Example: **Get-Service**
   ii. 

```
PS C:\Windows\System32> Get-Service

Status   Name               DisplayName
------   ----               -----------
Stopped  AarSvc_4a3b43ea    Agent Activation Runtime_4a3b43ea
Running  ActivID Shared St… ActivID Shared Store Service
Stopped  AJRouter           AllJoyn Router Service
Stopped  ALG                Application Layer Gateway Service
Stopped  AppIDSvc           Application Identity
Running  Appinfo            Application Information
Stopped  AppMgmt            Application Management
Stopped  appprotectionsvc   AppProtection Service
Stopped  AppReadiness       App Readiness
Stopped  AppVClient         Microsoft App-V Client
Stopped  AppXSvc            AppX Deployment Service (AppXSVC)
Stopped  AssignedAccessMan… AssignedAccessManager Service
Running  AudioEndpointBuil… Windows Audio Endpoint Builder
Running  Audiosrv           Windows Audio
Stopped  autotimesvc        Cellular Time
Stopped  AxInstSV           ActiveX Installer (AxInstSV)
Stopped  BcastDVRUserServi… GameDVR and Broadcast User Service_4a…
Running  BDESVC             BitLocker Drive Encryption Service
Running  BFE                Base Filtering Engine
Stopped  BITS               Background Intelligent Transfer Servi…
Stopped  BluetoothUserServ… Bluetooth User Support Service_4a3b43…
Running  BrokerInfrastruct… Background Tasks Infrastructure Servi…
Stopped  BTAGService        Bluetooth Audio Gateway Service
Running  BthAvctpSvc        AVCTP service
Stopped  bthserv            Bluetooth Support Service
```

d. **Get-Service -Name** <"Search String">
   i. With the **-Name** option, the <"Search String"> is used to specify the system's name for the service(s) you want to be displayed.
   ii. Example: **Get-Service -Name** "WSearch"
   iii. 

```
PS C:\Windows\System32> Get-Service -Name "WSearch"

Status   Name               DisplayName
------   ----               -----------
Running  WSearch            Windows Search

PS C:\Windows\System32>
```

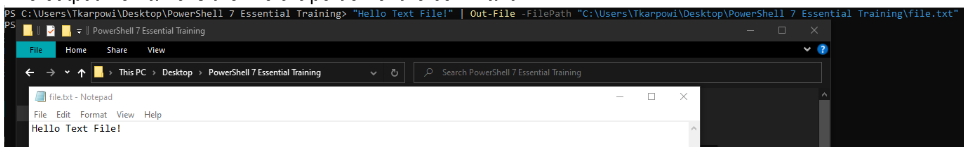
e. **Get-Service -DisplayName** <"Search String">
   i. With the **-DisplayName** option, the <"Search String"> is used to specify the user-recognized display name for the system service(s) you want to be displayed.
   ii. Example: **Get-Service -DisplayName** "Bluetooth*"
   iii. 

```
PS C:\Windows\System32> Get-Service -DisplayName "Bluetooth*"

Status   Name               DisplayName
------   ----               -----------
Stopped  BluetoothUserServ… Bluetooth User Support Service_4a3b43…
Stopped  BTAGService        Bluetooth Audio Gateway Service
Stopped  bthserv            Bluetooth Support Service

PS C:\Windows\System32>
```

f. **Showing Only Running Services**
   i. Example: **Get-Service |** Where-Object **{** $_.Status -eq "Running" **} |** Select-Object **DisplayName**
   ii. The output of the **Get-Service** command is piped to the "Where-Object" command.
   iii. The "Where-Object" command will make it so that only services with "Running" as their "Status" will be displayed to the terminal.
   iv. Then, the "Select-Object" command will filter the terminal output, to only the user-recognized display name for the system services currently running.
   v. To view the "Status", "Name", and "DisplayName" of only the "Running" services, you would enter this command: **Get-Service |** Where-Object **{** $_.Status -eq "Running" **}**

g. **Organizing Stopped & Running Services**

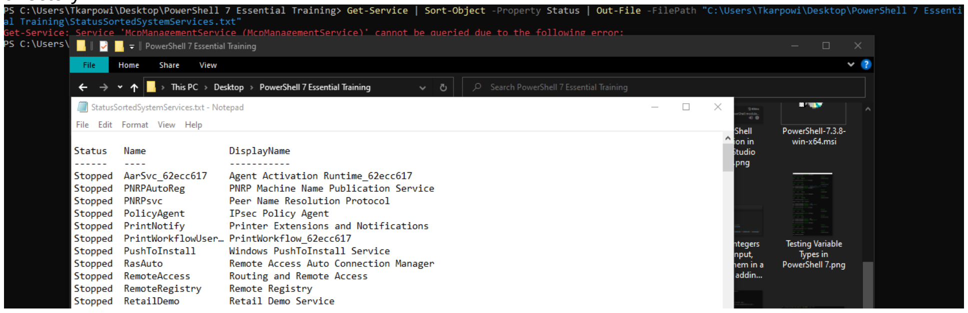
        i. Example: **Get-Service |** Sort-Object -Property **Status**
        ii. The output of the **Get-Service** command is piped to the Sort-Object command with the -Property option.
        iii. The -Property option tells the Sort-Object command to sort the output of the **Get-Service** command, based on the processes' status ("Running" or "Stopped").

7. **Out-File** -FilePath <"C:\File_Path\File_Name">
    a. This command outputs a command, or function result, into a file (usually a **.txt** file).
        i. When you use the **Out-File** command, whatever is, or would be, in the terminal output, is output to the resulting file.
    b. If you output a file to a folder, the folder has to be created before-hand.
        i. You can make the folder manually, or you could use the command: **New-Item** -Path <File Path> -ItemType **Directory**
        ii. Example: **New-Item** -Path "C:\Users\Tkarpowi\Desktop\New Folder" -ItemType **Directory**
        iii. The above example creates a new folder, with the folder name "New Folder".
    c. <Cmdlet or "Text"> **| Out-File -FilePath** <"C:\File_Path\File_Name">
        i. The **-FilePath** option allows you to send the terminal output to a file, with a specific file name and folder location.
        ii. **Note:** If you don't use the **-FilePath** option and don't specify the directory, the resulting output file will output to the current working directory in the PowerShell Terminal.
    d. **Outputting Text to a File**
        i. <"Text"> **| Out-File -FilePath** <"C:\File_Path\File_Name">
        ii. Example: "Hello Text File!" **| Out-File -FilePath** "C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training**file.txt**"
        iii. The "Hello Text File!" portion of the command, is the contents that will be **output to the specified directory**, "C:\Users\".
        iv. The output file name is the **file.txt** portion of the command.
        v. 

```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training> "Hello Text File!" | Out-File -FilePath "C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\file.txt"
PS
```

PowerShell 7 Essential Training
File
Home
Share
View
This PC > Desktop > PowerShell 7 Essential Training
Search PowerShell 7 Essential Training
file.txt - Notepad
File Edit Format View Help

```
Hello Text File!
```

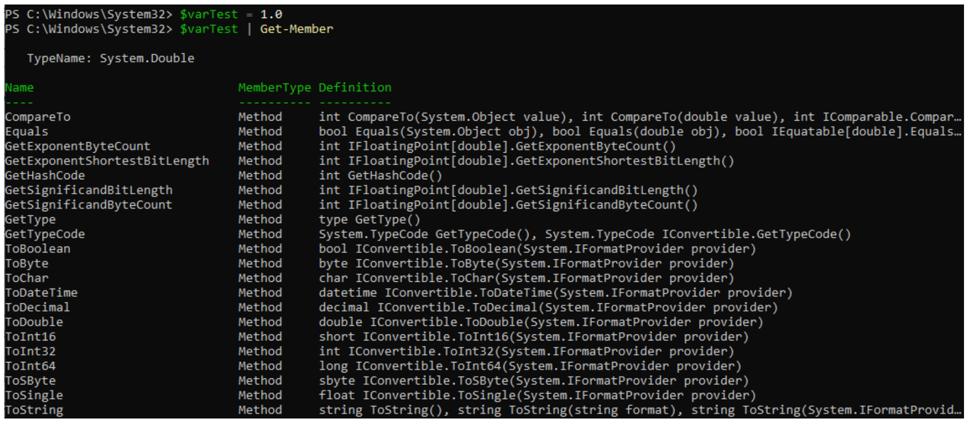
    e. **Sending a Cmdlet Output to a File**
        i. **Format:** <Cmdlet> **| Out-File -FilePath** <"C:\File_Path\File_Name">
        ii. Example: **Get-Service** | Sort-Object -Property Status **| Out-File -FilePath** "C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training**StatusSortedSystemServices.txt**"
        iii. The above cmdlet will obtain all of the system services on the local computer, and sort them by their "Status", "Stopped" or "Running".
        iv. Then, the resulting output of that cmdlet will be sent to the **StatusSortedSystemServices.txt** file, in the specified "C:\Users\" directory.
        v. 

```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training> Get-Service | Sort-Object -Property Status | Out-File -FilePath "C:\Users\Tkarpowi\Desktop\PowerShell 7 Essenti
al Training\StatusSortedSystemServices.txt"
Get-Service: Service 'McpManagementService (McpManagementService)' cannot be queried due to the following error:
PS C:\Users\
```

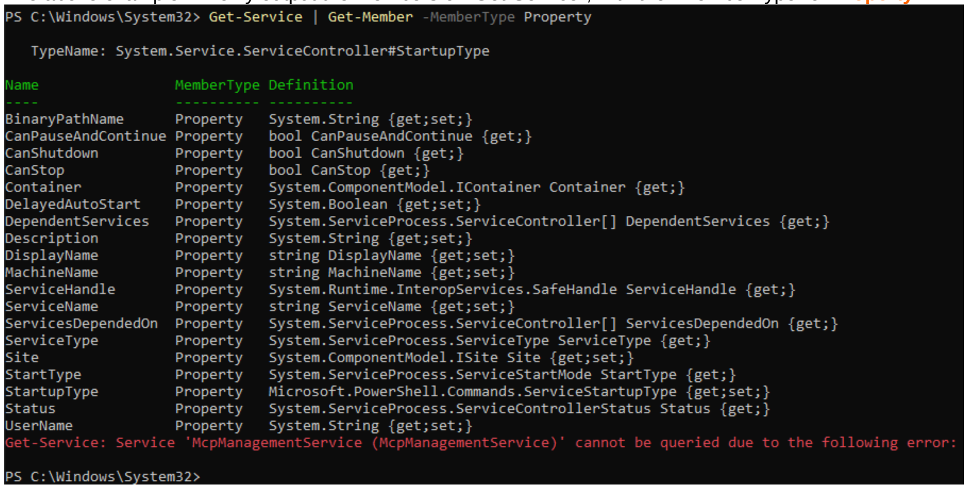
PowerShell 7 Essential Training
File
Home
Share
View
This PC > Desktop > PowerShell 7 Essential Training
Search PowerShell 7 Essential Training
StatusSortedSystemServices.txt - Notepad
File Edit Format View Help
PowerShell-7.3.8-win-x64.msi
Testing Variable Types in PowerShell 7.png

```
Status   Name                DisplayName
------   ----                -----------
Stopped  AarSvc_62ecc617     Agent Activation Runtime_62ecc617
Stopped  PNRPAutoReg         PNRP Machine Name Publication Service
Stopped  PNRPsvc             Peer Name Resolution Protocol
Stopped  PolicyAgent         IPsec Policy Agent
Stopped  PrintNotify         Printer Extensions and Notifications
Stopped  PrintWorkflowUser… PrintWorkflow_62ecc617
Stopped  PushToInstall       Windows PushToInstall Service
Stopped  RasAuto             Remote Access Auto Connection Manager
Stopped  RemoteAccess        Routing and Remote Access
Stopped  RemoteRegistry      Remote Registry
Stopped  RetailDemo          Retail Demo Service
```

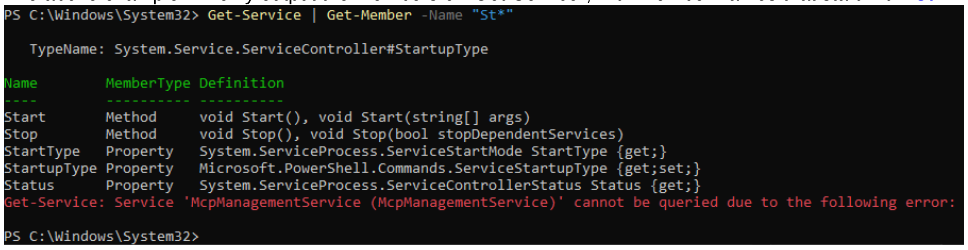
8. **Get-Member**
    a. The **Get-Member** cmdlet gets all of the members, the properties, and methods from the <Cmdlet or Object> piped input.
    b. This cmdlet returns a list of members that's sorted alphabetically.
        i. Methods are listed first, followed by the properties.
    c. **Information Displayed**
        i. **TypeName:** The PowerShell, *system*, recognized object type, for the given parent object.
        ii. **Name:** The child objects, *members*, of the given parent object.
        iii. **MemberType:** The type value for the given child object (Example: Method, Property).
        iv. **Definition:** A brief system description for a given child object.
    d. The **Get-Member** cmdlet can be used to save specific object values, along with their children object values, to a variable.
        i. For instance, an ActiveDirectory account can be saved, along with the components of that user's ActiveDirectory profile, such as the individual's username, and groups that the *member* is a part of.
        ii. This cmdlet can also be used to output only certain object types, as well as certain properties belonging to objects, and save these contents to a variable.
    e. The **Get-Member** cmdlet can be useful for finding input object types, which may be unknown, to allow a script, or command, to work with a given object, or variable, type.
        i. This cmdlet can also be useful for determining what object type a variable needs to be cast to, when it comes to entering input values of that specific object type, in a script or command.
    f. <Cmdlet or Object> **| Get-Member**
        i. Example (pt1): **$variable1** = **1.0**
        ii. Example (pt2): **$variable1 | Get-Member**
        iii. The above example takes a variable object, with the type "Double", and outputs all of the members belonging to the **TypeName:** "System.Double".



```
PS C:\Windows\System32> $varTest = 1.0
PS C:\Windows\System32> $varTest | Get-Member

   TypeName: System.Double

Name                            MemberType Definition
----                            ---------- ----------
CompareTo                       Method     int CompareTo(System.Object value), int CompareTo(double value), int IComparable.Compar…
Equals                          Method     bool Equals(System.Object obj), bool Equals(double obj), bool IEquatable[double].Equals…
GetExponentByteCount            Method     int IFloatingPoint[double].GetExponentByteCount()
GetExponentShortestBitLength    Method     int IFloatingPoint[double].GetExponentShortestBitLength()
GetHashCode                     Method     int GetHashCode()
GetSignificandBitLength         Method     int IFloatingPoint[double].GetSignificandBitLength()
GetSignificandByteCount         Method     int IFloatingPoint[double].GetSignificandByteCount()
GetType                         Method     type GetType()
GetTypeCode                     Method     System.TypeCode GetTypeCode(), System.TypeCode IConvertible.GetTypeCode()
ToBoolean                       Method     bool IConvertible.ToBoolean(System.IFormatProvider provider)
ToByte                          Method     byte IConvertible.ToByte(System.IFormatProvider provider)
ToChar                          Method     char IConvertible.ToChar(System.IFormatProvider provider)
ToDateTime                      Method     datetime IConvertible.ToDateTime(System.IFormatProvider provider)
ToDecimal                       Method     decimal IConvertible.ToDecimal(System.IFormatProvider provider)
ToDouble                        Method     double IConvertible.ToDouble(System.IFormatProvider provider)
ToInt16                         Method     short IConvertible.ToInt16(System.IFormatProvider provider)
ToInt32                         Method     int IConvertible.ToInt32(System.IFormatProvider provider)
ToInt64                         Method     long IConvertible.ToInt64(System.IFormatProvider provider)
ToSByte                         Method     sbyte IConvertible.ToSByte(System.IFormatProvider provider)
ToSingle                        Method     float IConvertible.ToSingle(System.IFormatProvider provider)
ToString                        Method     string ToString(), string ToString(string format), string ToString(System.IFormatProvid…
```

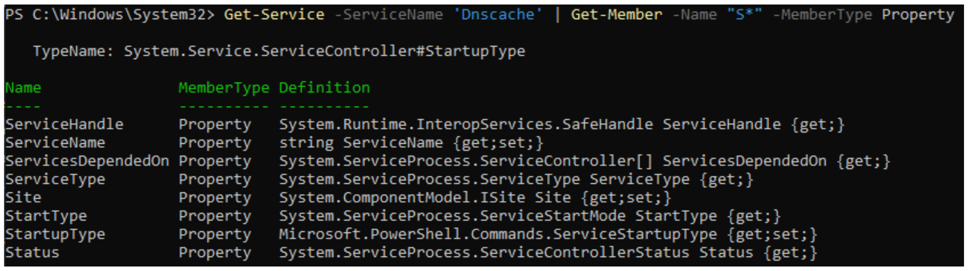
g. <Cmdlet or Object> **| Get-Member -MemberType [Property | Properties | AliasProperty | Methods | Method | All | ScriptMethod]**
   i. The **-MemberType** option will only output members with the "MemberType" value specified after the **-MemberType** option, belonging to the input value(s).
   ii. The options in orange are all **-MemberType** properties that can be selected.
   iii. **Example:** Get-Service | **Get-Member -MemberType Property**
   iv. The above example will only output the members of "Get-Service", with the "MemberType" of **Property**.
   v. 

```
PS C:\Windows\System32> Get-Service | Get-Member -MemberType Property

   TypeName: System.Service.ServiceController#StartupType

Name                MemberType Definition
----                ---------- ----------
BinaryPathName      Property   System.String {get;set;}
CanPauseAndContinue Property   bool CanPauseAndContinue {get;}
CanShutdown         Property   bool CanShutdown {get;}
CanStop             Property   bool CanStop {get;}
Container           Property   System.ComponentModel.IContainer Container {get;}
DelayedAutoStart    Property   System.Boolean {get;set;}
DependentServices   Property   System.ServiceProcess.ServiceController[] DependentServices {get;}
Description         Property   System.String {get;set;}
DisplayName         Property   string DisplayName {get;set;}
MachineName         Property   string MachineName {get;set;}
ServiceHandle       Property   System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName         Property   string ServiceName {get;set;}
ServicesDependedOn  Property   System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType         Property   System.ServiceProcess.ServiceType ServiceType {get;}
Site                Property   System.ComponentModel.ISite Site {get;set;}
StartType           Property   System.ServiceProcess.ServiceStartMode StartType {get;}
StartupType         Property   Microsoft.PowerShell.Commands.ServiceStartupType {get;set;}
Status              Property   System.ServiceProcess.ServiceControllerStatus Status {get;}
UserName            Property   System.String {get;set;}
Get-Service: Service 'McpManagementService (McpManagementService)' cannot be queried due to the following error:

PS C:\Windows\System32>
```

h. <Cmdlet or Object> **| Get-Member -Name <**"The Name of the Object(s)" **|** '\Directory_Location\'**>**
   i. The **-Name** option will only output members, pertaining to the input value(s), containing a "Name" value that matches the specified string value, after the **-Name** option.
   ii. The options in blue only apply to the **-Name** option.
   iii. **Example:** Get-Service | **Get-Member -Name** "St*"
   iv. The above example will only output the members of "Get-Service", with member names that start with "St".
   v. 

```
PS C:\Windows\System32> Get-Service | Get-Member -Name "St*"

   TypeName: System.Service.ServiceController#StartupType

Name        MemberType Definition
----        ---------- ----------
Start       Method     void Start(), void Start(string[] args)
Stop        Method     void Stop(), void Stop(bool stopDependentServices)
StartType   Property   System.ServiceProcess.ServiceStartMode StartType {get;}
StartupType Property   Microsoft.PowerShell.Commands.ServiceStartupType {get;set;}
Status      Property   System.ServiceProcess.ServiceControllerStatus Status {get;}
Get-Service: Service 'McpManagementService (McpManagementService)' cannot be queried due to the following error:

PS C:\Windows\System32>
```

i. <Cmdlet or Object> **| Get-Member [** -Name **(<**"The Name of the Object(s)" **|** '\Directory_Location\'**>) |** -MemberType **[Property | Properties | AliasProperty | Methods | Method | All | ScriptMethod]]**
   i. The full **Get-Member** Command Structure.
   ii. The **-Name** and **-MemberType** parameters can be combined together to filter for even more specific MemberTypes.
   iii. **Example:** Get-Service -ServiceName 'Dnscache' **| Get-Member -Name** "S*" **-MemberType Property**
   iv. The above example takes the output of the **Get-Service** command, with the service name of "Dnscache", which is then piped to the **Get-Member** command, to only output names that start with "S" and have the **Property** MemberType.

v. 

```
PS C:\Windows\System32> Get-Service -ServiceName 'Dnscache' | Get-Member -Name "S*" -MemberType Property

   TypeName: System.Service.ServiceController#StartupType

Name               MemberType Definition
----               ---------- ----------
ServiceHandle      Property   System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName        Property   string ServiceName {get;set;}
ServicesDependedOn Property   System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType        Property   System.ServiceProcess.ServiceType ServiceType {get;}
Site               Property   System.ComponentModel.ISite Site {get;set;}
StartType          Property   System.ServiceProcess.ServiceStartMode StartType {get;}
StartupType        Property   Microsoft.PowerShell.Commands.ServiceStartupType {get;set;}
Status             Property   System.ServiceProcess.ServiceControllerStatus Status {get;}
```

Practicing Commonly Used Commands In PowerShell 7

1. Open the PowerShell 7, or Terminal, application as Administrator.
2. **View details about your PowerShell 7 version**

```
$PSVersionTable
```

3. **Obtain the name of your computer, as it appears to other devices on the network**

```
Get-ComputerInfo -Property "CsName"
```

4. **Obtain details about your computer, using ActiveDirectory**

```
Get-ADGroupMember -Identity "<CsName from previous command output>"

# Example
Get-ADGroupMember -Identity "MA14203TKARPOWI"
```

5. **Obtain details about your ActiveDirectory user account**

```
Get-ADUser -Identity "<Your AD Username>"

# Example
Get-ADUser -Identity "TKARPOWI"
```

6. **Change the current working directory for PowerShell 7 to your user account**

```
cd "C:\Users\<Your AD Username>"

# Example
cd "C:\Users\Tkarpowi"
```

7. View the folders and files in your user account folder.
   a. **ls**
8. **View the hidden folders and files in your user account folder**

```
ls -Hidden
```

9. **Obtain a list of services running on the system, and output it to a text file in your "~\Desktop" folder**

```
Get-Service | Out-File -Path "C:\Users\<Your AD Username>\Desktop\Services.txt"
```

10. View the details of the newly created text file in your "~\Desktop" folder.

a. **Using the Get-Item Command**

```
Get-Item -Path "C:\Users\<Your AD Username>\Desktop\Services.txt"
```

b. **Using the Get-Content Command**

```
Get-Content -Path "C:\Users\<Your AD Username>\Desktop\Services.txt"
```

11. Create a log of services running on the system, with a timestamp, and output it to a text file in your "~\Desktop" folder.

a. **Create the new text file**

```
New-Item -Path "C:\Users\<Your AD Username>\Desktop\Services_With_Timestamp.txt" -ItemType File
```

b. **Output the current date and time to the newly created text file**

```
Get-Date | Out-File -Path "C:\Users\<Your AD Username>\Desktop\Services_With_Timestamp.txt"
```

c. **View the contents of the newly updated text file**

```
Get-Content -Path "C:\Users\<Your AD Username>\Desktop\Services_With_Timestamp.txt"
```

d. **Obtain a list of all the "Stopped" and "Running" Services on the machine, and append this list to the end of the text file**

```
Get-Service | Out-File -Path "C:\Users\<Your AD Username>\Desktop\Services_With_Timestamp.txt" -
Append
```

e. **View your newly created system services log file!**

```
Get-Content -Path "C:\Users\<Your AD Username>\Desktop\Services_With_Timestamp.txt"
```

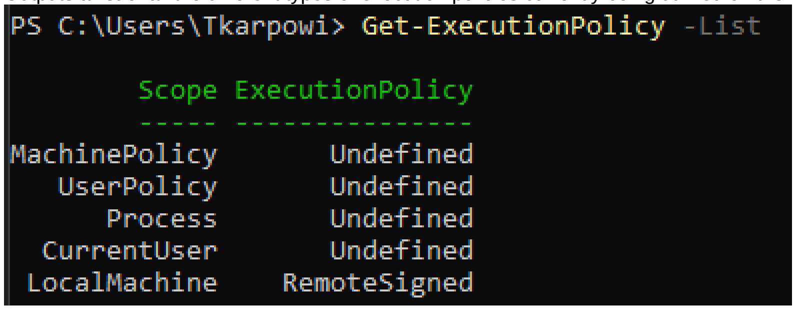
What is PowerShell Scripting?

- PowerShell Scripting can be used to automate a variety of different tedious tasks, and drastically increase the efficiency of various of tasks that you are working on.
- There are a vast amount of concepts and commands that go along-side PowerShell 7 Scripting.
  - This guide aims to streamline this learning process, and provide a solid foundation for making your own PowerShell 7 Scripts.
- If you want to learn about certain parts of PowerShell Scripting, click the links in the **PowerShell 7 Scripting - Table of Contents**, to jump to those sections.
- Make sure you have completed, **6. Setting-Up Visual Studio Code for PowerShell 7**, in the "How to Install PowerShell 7 & Additional Applications" section, on the "PowerShell 7 - Information & Installation/Setup Guide" page.
  - This is how you create new PowerShell Script files (.ps1).
- Example PowerShell Script files made for concepts in this documentation: Additional_Example_Scripts.zip
  - The scripts in this **.zip** file have plenty of comments, to illustrate what each part of the script actually does.
- "PowerShell 7 Essential Training" example files: Ex_Files_PowerShell_7_EssT.zip

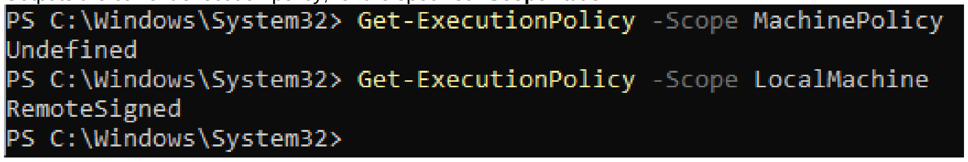
PowerShell 7 Scripting - Table of Contents

PowerShell 7 Execution Policies

PowerShell Execution Policies define the parameters for which PowerShell Scripts should be executable, depending on what types of scripts they are, as well as the origin of where the PowerShell Script came from.

1. **Get-ExecutionPolicy**
   a. This command will output the current configuration for the PowerShell Script Execution Policy to the terminal.
   b. **-List**
      i. Outputs a list of all the different types of execution policies currently being utilized on the machine.
      ii. 

```
PS C:\Users\Tkarpowi> Get-ExecutionPolicy -List

        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser       Undefined
 LocalMachine    RemoteSigned
```

   c. **-Scope [LocalMachine | MachinePolicy | Process | UserPolicy | CurrentUser]**
      i. Outputs the current execution policy, for the specified **-Scope** value.
      ii. 

```
PS C:\Windows\System32> Get-ExecutionPolicy -Scope MachinePolicy
Undefined
PS C:\Windows\System32> Get-ExecutionPolicy -Scope LocalMachine
RemoteSigned
PS C:\Windows\System32>
```

2. **Set-ExecutionPolicy**
   a. This command will set the current PowerShell 7 execution policy.
   b. By default, without the "-Scope" option, this command will only set the execution policy for the "LocalMachine".
   c. **-ExecutionPolicy [Execution Policy Type]**
      i. Format: **Set-ExecutionPolicy -ExecutionPolicy [AllSigned | Bypass | Default | RemoteSigned | Restricted | Undefined | Unrestricted]**
      ii. This option is used to define which PowerShell Script execution policy you would like to set.
      iii. Example: **Set-ExecutionPolicy -ExecutionPolicy RemoteSigned**
      iv. The above example will set the PowerShell 7 execution policy for the "LocalMachine" to **RemoteSigned**.
   d. **-Scope [Execution Policy Scope]**
      i. Format: **Set-ExecutionPolicy** -ExecutionPolicy **<**Execution Policy Type**> -Scope [LocalMachine | MachinePolicy | Process | UserPolicy | CurrentUser]**
      ii. This option defines the PowerShell execution policy for a specific "Scope" value (user, group, machine).
      iii. This option changes the current execution policy for only the specified value after the **-Scope** parameter.
      iv. Example: **Set-ExecutionPolicy** -ExecutionPolicy Restricted **-Scope LocalMachine**
      v. The above example will set the PowerShell 7 execution policy for only the "LocalMachine" to Restricted.
   e. **-Force**
      i. Format: **Set-ExecutionPolicy** -ExecutionPolicy **<**Execution Policy Type**>** -Scope **<Execution Policy Scope>** -Force
      ii. This will force the change in the current execution policy, and suppress all warning and confirmation prompts, when changing the current execution policy.
      iii. Use caution with this parameter. Only use this **-Force** option, if you know what you are doing.
      iv. Example: **Set-ExecutionPolicy** -ExecutionPolicy Unrestricted -Scope **CurrentUser -ForceSet-ExecutionPolicy**
   f. Combined Format: **Set-ExecutionPolicy** -ExecutionPolicy **<AllSigned | Default | RemoteSigned | Restricted>** -Scope **<LocalMachine | MachinePolicy | Process | UserPolicy | CurrentUser>** -Force

3. **-ExecutionPolicy** Fields
   a. **-ExecutionPolicy AllSigned**
      i. Scripts can run, but it requires the oldest script's configuration.
      ii. Files must be signed by a trusted publisher, including the scripts that you write on your own local computer.
      iii. You will get prompts before running scripts, if you have not classified a script as "Trusted" or "Untrusted" yet.
      iv. Example: **Set-ExecutionPolicy** -ExecutionPolicy **AllSigned**
   b. **-ExecutionPolicy Bypass**
      i. Nothing is blocked, any scripts can run, and there are no warning messages or prompts.
      ii. **---Extremely Dangerous and Un-Secure---**
      iii. Example: **Set-ExecutionPolicy** -ExecutionPolicy **Bypass**
   c. **-ExecutionPolicy Default**
      i. This option applies the default execution policy, based on the system's default configuration (i.e. the system's configuration file).
      ii. If the execution policy in all scopes is default, ***usually***, the effective execution policy is ***Restricted*** for **Windows Clients** (Windows 10, Windows 11), and ***RemoteSigned*** for **Windows Server**.
      iii. Example: **Set-ExecutionPolicy** -ExecutionPolicy **Default**
   d. **-ExecutionPolicy RemoteSigned**
      i. This is the default **-ExecutionPolicy** configuration for Windows Server Computers.
      ii. Scripts can run, but it requires a digital signature from a trusted publisher on scripts, and configuration files, that are downloaded from the Internet.
      iii. Digital signatures are not required on scripts that are written on the local computer, as well as files not downloaded from the Internet (PowerShell Script files obtained from a file transfer, or USB).
      iv. Example: **Set-ExecutionPolicy** -ExecutionPolicy **RemoteSigned**
   e. **-ExecutionPolicy Restricted**

            i. This is the default **-ExecutionPolicy** configuration for Windows Client Devices (**Windows 10, Windows 11**).
            ii. This allows for individual commands to be run through PowerShell, but not through PowerShell Scripts themselves.
            iii. This policy prevents running all PowerShell Script files, including:
                1. Formatting and Configuration Files (**.ps1 .xml**)
                2. Module Script Files (**.psm1**)
                3. PowerShell Profile Files (**.ps1**)
            iv. Example: **Set-ExecutionPolicy** -ExecutionPolicy **Restricted**
    f. **-ExecutionPolicy Undefined**
            i. There is no execution policy set in the current scope.
            ii. If the execution policy in all scopes is undefined, the effective execution policy is ***Restricted*** for **Windows Clients** (Windows 10, Windows 11), and ***RemoteSigned*** for **Windows Server**.
            iii. Example: **Set-ExecutionPolicy** -ExecutionPolicy **Undefined**
    g. **-ExecutionPolicy Unrestricted**
            i. This is the default execution policy for Non-Windows Computers (Example: **Linux**) and ***Cannot be Changed***.
            ii. Unsigned PowerShell Scripts cannot run.
            iii. There is a risk of running Malicious Scripts.
            iv. It warns the user before running scripts and configuration files that are not from the Local Internet Zone.
            v. Example: **Set-ExecutionPolicy** -ExecutionPolicy **Unrestricted**

4. **-Scope** Fields
    a. **-Scope LocalMachine**
            i. This is the default scope that affects, and applies to, all users of the computer.
            ii. Example: **Set-ExecutionPolicy** -ExecutionPolicy AllSigned -Scope **LocalMachine**
    b. **-Scope MachinePolicy**
            i. This scope is set by a Group Policy, for all users of the computer.
            ii. Example: **Set-ExecutionPolicy** -ExecutionPolicy Default -Scope **MachinePolicy**
    c. **-Scope Process**
            i. This scope focuses only on the current PowerShell Session.
            ii. Example: **Set-ExecutionPolicy** -ExecutionPolicy Unrestricted -Scope **Process**
    d. **-Scope UserPolicy**
            i. This scope is set by a Group Policy for the current user of the computer.
            ii. Example: **Set-ExecutionPolicy** -ExecutionPolicy RemoteSigned -Scope **UserPolicy**
    e. **-Scope CurrentUser**
            i. This scope affects only the current user of the computer.
            ii. Example: **Set-ExecutionPolicy** -ExecutionPolicy Unrestricted -Scope **CurrentUser**

Conditional Operations

Example code for this section can be found in the attached "Additional Example Scripts.zip" file, with the file name, **if_statements.ps1**.

1. **Conditional Operators: if ()**, **elseif ()**, **else**
    a. The code inside the conditional operators will only execute, if one of the **if ()** or **elseif ()** parameters' **(**conditions**)** are met.
            i. Order of checking conditions: **if () elseif () else**
            ii. Once an **if ()** or **elseif ()** statement's parameter's conditions are met, PowerShell 7 will not check the other conditional operations after it, unless there is another **if** statement.
    b. **Conditional Options:**
            i. Greater than: **-gt**
            ii. Greater than or Equal to: **-ge**
            iii. Less than: **-lt**
            iv. Less than or Equal to: **-le**
            v. Equal to: **-eq**
    c. An example PowerShell Script, **if_statements.ps1**, can be seen below, containing examples for the majority of the conditional operators, and conditional options.
    d. **if_statements.ps1**

```
# Example PowerShell 7 Script for if statements

# Get input from the user, in the PowerShell Terminal
[Int32] $numberInput = Read-Host "Please enter a number 1-10"

# Checks if the number is less than 1
if ($numberInput -lt 1) {

    Write-Output "$numberInput is less than 1. The number must be between 1 and 10."

}

# Checks if the number is greater than 10
elseif ($numberInput -gt 10) {

    Write-Output "$numberInput is greater than 10. The number must be between 1 and 10."

}
```

```
# Checks if the number is between 1 and 10
# Checks if the number is Greater than or equal to 1, and Less than or equal to 10
elseif ($numberInput -ge 1 && $numberInput -le 10) {

    Write-Output "$numberInput is a number between 1 and 10!"

}

else {

    Write-Output "$numberInput is not a number. Make sure you enter a number between 1 and 10."

}
```

PowerShell 7 Loops

Example code for this section can be found in the attached "Additional Example Scripts.zip" file, with the file name, **loops.ps1**.

1. **Types of Loops**
    a. **while ()**
        i. **while (<Conditional Statement>) { }**
        ii. Executes the code inside the loop, while the **<Conditional Statement>** inside the parameters is true.
        iii. Once the statement inside the parameters is false, it will jump to the next line of code outside the **while ()** loop.
        iv. **Best Use-Case:** Executing a certain action, until a change is met. Such as removing every file in a directory, until the directory is empty.

    b. **for ()**
        i. **for ($i = 0; <Conditional Statement>; $i++) { }**
        ii. **$i = 0;** Creates a temporary variable to count with, starting from a specific number, usually zero.
        iii. Executes the code inside the loop, from a given instance variable, until the **<Conditional Statement>** is false.
        iv. Once all of the code inside the loop has been executed, the temporary counter variable **$i++** will be increased (or decreased **$i--**) by one.
        v. Once the "<Conditional Statement>" is false, it will jump to the next line of code outside the **for ()** loop.
        vi. **Best Use-Case:** Listing off objects in an array, starting from a specific index.
        vii. **Best Use-Case:** Executing commands a specific number of times. It can be used to execute commands a certain number of times based on a user's input as well.

    c. **ForEach-Object { }**
        i. <Variable/ObjectArray> **| ForEach-Object { }**
        ii. Executes the code inside the loop, for every index in the specified array of objects.
        iii. **Example:** If there are 15 items in the specified variable array, $varArray15, or array of objects, the code inside of the **ForEach-Object { }** will be executed 15 times.
        iv. **Best Use-Case:** Going through every instance in an array of objects.
        v. <Variable/Object> **| ForEach-Object { } -Parallel**
        vi. **-Parallel** Executes the code inside the loop, in a parallel (multi-threaded) format, for every instance in the specified array of objects.
        vii. **Example:** If there are 11 items in the specified variable array, $varArray11, or Array of Objects, the code inside of the **ForEach-Object { }** will be executed 11 times, without emphasis on order of execution.
        viii. **Best Use-Case:** Going through every instance in an array of objects, when the output order doesn't matter, and you want to increase the efficiency of the script.

2. **Conditional Loop Options**
    a. Greater than: **-gt**
    b. Greater than or Equal to: **-ge**
    c. Less than: **-lt**
    d. Less than or Equal to: **-le**
    e. Equal to: **-eq**

3. **Example PowerShell Script for Loops**
    a. An example PowerShell Script **loops.ps1** can be seen below, containing examples for the majority of commonly used loops, when it comes to scripting in PowerShell 7.
    b. **loops.ps1**

```
# Example PowerShell 7 Script for PowerShell 7 Scripting loops

# Get input from the user, in the PowerShell Terminal
[Int32] $numberInput = Read-Host "Please enter a number between 5 and 15"

# Checks if the number is in-between 5 and 15.
if ($numberInput -ge 5 && $numberInput -le 15) {
    # Used to store temporary number values, counting up and down.
    [Int32] $tempCounter = $numberInput - 1

    Write-Host "The following numbers are less than $numberInput - " -NoNewline -ForegroundColor
DarkGreen
    # Counts down from the provided number, until it gets to the limit (5).
    # Example: 9 -> Ouput: "8 7 6 5"
    while ($tempCounter -ge 5) {
        Write-Host "$tempCounter " -NoNewline
        $tempCounter = $tempCounter - 1
    }

    # Resets the temporary number counting variable
    $tempCounter = $numberInput + 1
    Write-Host ""

    Write-Host "The following numbers are greater than $numberInput - " -NoNewline -
ForegroundColor DarkYellow
    # Counts up from the provided number, until it gets to the limit.
    # Example: 9 -> Ouput: "10 11 12 13 14 15"
    for ([Int32] $i = $tempCounter; $i -le 15; $i++) {
        Write-Host "$i " -NoNewline
    }

    # Resets the temporary number counting variable
    $tempCounter = 5
    Write-Host ""

    # Creates an Int32 Array with all numbers 5 through 15.
    [Array] $allPossibleNumbers = 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

    Write-Host "The following numbers are all the possible numbers you could've input - " -
NoNewline -ForegroundColor Magenta
    # Writes every number, individually, 5 through 15.
    # The -Parallel option allows the loop to run in a parallelized form (multithreaded).
    $allPossibleNumbers | ForEach-Object -Parallel {
        Write-Host "$_ " -NoNewline
    }

}
# The user-inputted number value was not a number betweeen 5 and 15.
else {
    Write-Output "$numberInput is not a number between 5 and 15. Make sure you enter a number
between 5 and 15."
}
```

Functions & Function Parameters

The following example code can be found in the attached "Ex_Files_PowerShell_7_EssT.zip" file. One example script has the file name, **Define-Custom-Help.ps1**, with the file path, ".\Ex_Files_PowerShell_7_EssT\Exercise Files\CH04\04_09\Define-Custom-Help.ps1". Another example code has the file name, **Parameters-Attributes-for-Scripts-and-Functions.ps1**, with the file path, ".\Ex_Files_PowerShell_7_EssT\Exercise Files\CH04\04_03\Parameters-Attributes-for-Scripts-and-Functions.ps1".

1. **Function** Function-Name**(**$inputParameter**) { }**
    a. Functions will only be executed if, and when, they are called by the script themselves, or the user of the script.
        i. The user can only call the function if the specific function's section of code is run in Visual Studio Code, and they input the function name into the terminal.
        ii. To do this, select all of the code, or all of the code for the function that you want to run, and press "F8" on you keyboard.
        iii. This will cache the function(s) and let you execute them, by simply typing in the function's name and any applicable parameters /arguments.

2. **Calling Functions in a Script**
    a. Format: **FunctionName** -VariableName <**argument**> -VariableName <**argument**> -VariableName <**argument**> -VariableName <**argument**>
    b. Example: **Add-FourNumbers** -first **1** -second **2** -third **3** -fourth **4**
        i. **Add-FourNumbers** is the target function's name
        ii. **Variable Names:** "-first", "-second", "-third", and "-fourth", target the variable names "$first", "$second", "$third", and "$fourth", accordingly.
        iii. **Variable Input Values:** "1", "2", "3", and "4" are the input values for "$first", "$second", "$third", and "$fourth", accordingly.
    c. **Define-Custom-Help.ps1**

```
# Basic Function
function Add-FourNumbers()
{
    param(
        [Int32]$first,
        [Int32]$second,
        [Int32]$third,
        [Int32]$fourth
    )

    $result = $first + $second + $third + $fourth

    Write-Host "$($first) + $($second) + $($third) + $($fourth) = $($result)"
}

Add-FourNumbers -first 1 -second 1 -third 1 -fourth 1
```

3. **Passing Multiple Arguments to a Function**
    a. **Function** Function-Name**(**$inputParameter1**,** $inputParameter2**) { }**
        i. This function would take 2 input parameters (arguments).
    b. If you are passing multiple arguments to a function, there are two ways to do it with the exact same result.
        i. One way to do this is by using **commas** "**,**", to separate the different parameters
        ii. The other is to just **space the inputs out** " ".
    c. **Using Commas:** **Display-Message** "Value 1"**,** "Value 2"
    d. **Using Spaces:** **Display-Message** "Value 1" "Value 2"
    e. In both of the above examples, the function name is **Display-Message**, and $inputParameter1 = "Value 1" and $inputParameter2 = "Value 2".

4. **Using a Function with Multiple Parameters**
    a. If a function supports more than one parameter, and you only supply one, or more, parameters, the function will only utilize the number of parameters you have provided.
    b. **Parameters-Attributes-for-Scripts-and-Functions.ps1**

```
# Change the function to use arguments
Function Display-Message()
{
        [String]$Value1 = $args[0]
        [String]$Value2 = $args[1]

        Write-Host $Value1 $Value2
}
```

        i. **Input:** **Display-Message** "Value 1" "Value 2"
            1. **Output: Value 1**
        ii. **Input:** **Display-Message** "Value 1" "Value 2"
            1. **Output: Value 1 Value 2**
        iii. **Input:** **Display-Message** "Value 1"**,** "Value 2"
            1. **Output: Value 1 Value 2**

5. **Mandatory Parameter Inputs & Mandatory Input Types**
    a. To make a parameter input mandatory, the code below must be included in the function.
        i. **Param ([parameter(Mandatory=$true)])**
    b. If you want a function to only accept a specific parameter input type, the code below must be included in the function.
        i. **Param ([String]$Text)**
            1. This example would make a **[string]** input type the only acceptable input type for the function.
        ii. **Param ([Int32]$Text)**
            1. This example would make a **[int32]** input type the only acceptable input type for the function.
    c. **Parameters-Attributes-for-Scripts-and-Functions.ps1**

```
# Change the function to use parameter
Function Display-Message()
{
        Param(
                    [parameter(Mandatory=$true)]
                    [String]$Text
        )
        Write-Host $Text
}
```

6. **Taking Parameter Inputs From a Specific Set of Options**
   a. To make a function only take parameter inputs from a specific set of options, **[ValidateSet()]** must be included in the **Param()** section of the function.
      i. **Param ([ValidateSet("Lexus","Porsche","Toyota","Mercedes-Benz","BMW","Honda","Ford","Chevrolet")])**
   b. When a function has the **[ValidateSet()]** option specified, you can press the "**TAB**" key to cycle through all of the function's available options.
      i. **Parameters-Attributes-for-Scripts-and-Functions.ps1**

```
Function Display-Message()
{
        Param(
                    [parameter(Mandatory=$true)]
            [ValidateSet("Lexus","Porsche","Toyota","Mercedes-Benz","BMW","Honda","Ford","
Chevrolet")]
                    [String]$Text
        )
        Write-Host "I like to drive a "$Text
}
```

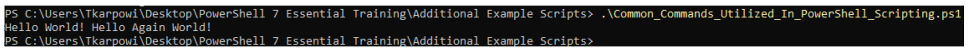
      ii. Example: **Display-Message** -Text **<"TAB" Key>**
         1. **Display-Message** -Text **Toyota**
         2. **Display-Message** -Text **Ford**

Writing Terminal Outputs & Taking Terminal Inputs

Example code for this section can be found in the attached "Additional Example Scripts.zip" file, with the file name, **Common_Commands_Utilized_In_PowerShell_Scripting.ps1**.

1. **Write-Host**
   a. This command outputs a text message to the terminal.
      i. Write-Host should only be used when there are no variables being output, along side a message, in the terminal.
      ii. Example: **Write-Host** "Hello World!"
   b. The **-ForegroundColor** option can be used to change the color of the text being output to the terminal.
      i. Example: **Write-Host** "Hello World, with color!" **-ForegroundColor** Blue
   c. The **-BackgroundColor** option can be used to change the color of the background of the text being output to the terminal.
      i. Example: **Write-Host** "Hello World, with two colors!" **-ForegroundColor** Magenta **-BackgroundColor** Green
   d. The **-NoNewline** option will make sure the next message output to the terminal starts on the same line as the message with the **-NoNewline** option.
      i. **Common_Commands_Utilized_In_PowerShell_Scripting.ps1**

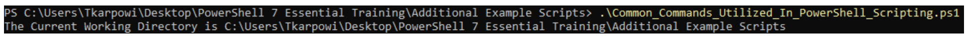
```
Write-Host "Hello World! " -NoNewline
Write-Host "Hello Again World!"
```

      ii. 

```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example Scripts> .\Common_Commands_Utilized_In_PowerShell_Scripting.ps1
Hello World! Hello Again World!
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example Scripts>
```

2. **Write-Output**
   a. This command outputs a text message to the terminal.
   b. Write-Output **cannot utilize** the **-NoNewline** option, there will always be a new line entered after the command has completed.
   c. Write-Output should be used instead of "Write-Host", when objects, such as ActiveDirectory usernames (SamAccountNames), are involved.
      i. Write-Output should really only be used, when there are variables in the output.
      ii. **Common_Commands_Utilized_In_PowerShell_Scripting.ps1**

```
$currentDirectory = Get-Location
Write-Output "The Current Working Directory is $($currentDirectory)"
```

      iii. 

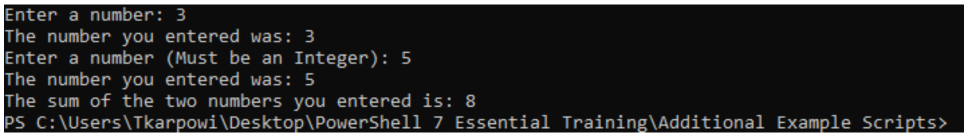
```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example Scripts> .\Common_Commands_Utilized_In_PowerShell_Scripting.ps1
The Current Working Directory is C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example Scripts
```

3. **Read-Host**
   a. This command takes a user's input from the terminal.
      i. The user's input can be used for choosing between options, or even something to make calculations off of.
   b. The user's input **can be stored** by setting the Read-Host command to a variable.
      i. **Common_Commands_Utilized_In_PowerShell_Scripting.ps1**

```
$number = Read-Host "Enter a number"
Write-Output "The number you entered was: $($number)"
```

   ii. The first part of the command "$number" stores the users input to the **$number** variable.
   iii. The second part of the command "Enter a number" is the **prompt message** that will be output to the terminal.
c. **Note:** You can ensure that the input value is a certain type value, by casting the variable before-hand.
   i. This is important for when you need to make calculations based on a user's input.
   ii. **Common_Commands_Utilized_In_PowerShell_Scripting.ps1**

```
$number = Read-Host "Enter a number"
Write-Output "The number you entered was: $($number)"
[Int32] $castedNumber = Read-Host "Enter a number (Must be an Integer)"
Write-Output "The number you entered was: $castedNumber"
$combinedNumber = [Int32] $number + $castedNumber
Write-Output "The sum of the two numbers you entered is: $combinedNumber"
```

   iii. 

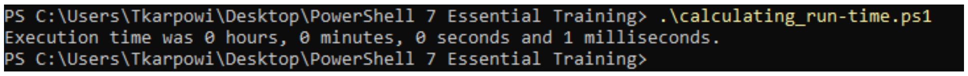
```
Enter a number: 3
The number you entered was: 3
Enter a number (Must be an Integer): 5
The number you entered was: 5
The sum of the two numbers you entered is: 8
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example Scripts>
```

Calculating Script Run-Time

Example code for this section can be found in the attached "Additional Example Scripts.zip" file, with the file name, **calculating_run-time.ps1**.

1. To calculate the run-time of PowerShell 7 Scripts, you need 2 variables.
   a. **$StartTime** - This variable will be used for starting the timer, for the run-time of the script.
   b. **$RunTime** - This variable will be used for ending the timer, for the run-time of the script, after the script has completed.
2. The code block below is a template for calculating the run-time for a PowerShell Script.
   a. **Calculating/Outputting Run-Time for PowerShell Scripts**

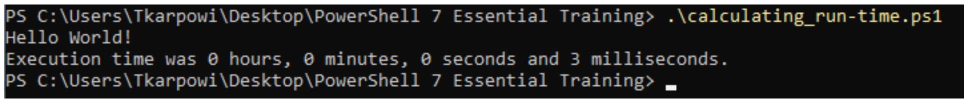
```
#Calculate the run-time of the script
$StartTime = get-date

<#
##-----Script Code-----
#>

#Calculate the run-time of the script
$RunTime = New-TimeSpan -Start $StartTime -End (get-date)
"Execution time was {0} hours, {1} minutes, {2} seconds and {3} milliseconds." -f $RunTime.Hours,
$RunTime.Minutes,  $RunTime.Seconds,  $RunTime.Milliseconds
```

   b. 

```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training> .\calculating_run-time.ps1
Execution time was 0 hours, 0 minutes, 0 seconds and 1 milliseconds.
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training>
```

3. The code block below shows the run-time being calculated for a basic PowerShell Script.
   a. **calculating_run-time.ps1**

```
#Calculate the run-time of the script
$StartTime = get-date

Write-Host "Hello World!"

#Calculate the run-time of the script
$RunTime = New-TimeSpan -Start $StartTime -End (get-date)
"Execution time was {0} hours, {1} minutes, {2} seconds and {3} milliseconds." -f $RunTime.Hours,
$RunTime.Minutes,  $RunTime.Seconds,  $RunTime.Milliseconds
```

   b. 

```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training> .\calculating_run-time.ps1
Hello World!
Execution time was 0 hours, 0 minutes, 0 seconds and 3 milliseconds.
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training>
```

Running Variable Processes In Parallel (Multi-Threaded Variable Operations)

1. When it comes to any type of coding, including scripting, efficiency is extremely important, and should be made a top priority.
2. PowerShell 7 supports running variables, and functions, in **-Parallel** mode.
    a. This allows multiple variables, or functions, to be executed simultaneously, drastically increasing efficiency in PowerShell Scripts.
    b. The ForEach-Object **-Parallel** mode is essentially the same concept as using multiple CPU Cores to run, and execute operations, in an application or program.
3. The **-Parallel** mode only works with a **ForEach-Object** loop.
    a. The **-Parallel** mode will not work with "for ()", "while ()", and "foreach ()" loops.
    b. The input parameter must be an array of objects.
        i. $amiMembers | **ForEach-Object** -Parallel { #function-code }
4. The code block below is a template for running PowerShell variable operations in **-Parallel** mode.
    a. **Running PowerShell Scripts in Parallel**

```
# Creates variables for the AD groups
$g1Group = "g1-group"
$g2Group = "g2-group"

# Creates a variable that holds the "username" of each member in the g#-group groups
$g1Members = (Get-ADGroupMember -Identity $g1Group | Select-Object -ExpandProperty SamAccountName)
$g2Members = (Get-ADGroupMember -Identity $g2Group | Select-Object -ExpandProperty SamAccountName)

# -----G1 Section-----
$g1Members | ForEach-Object -Parallel { # This line looks for each "item" in g1Members
  # Get the groups that the member belongs to
  $g1GroupUS = Get-ADPrincipalGroupMembership -Identity $_ | Where-Object {$_.SamAccountName -eq
'G1_Group_US'} # This allows us to check membership of G1_Group_US
  $g1GroupNonUS = Get-ADPrincipalGroupMembership -Identity $_ | Where-Object {$_.SamAccountName -
eq 'G1_Group_NON-US'} # This allows us to check membership of G1_Group_NON-US

   if ($g1GroupUS) {
    Write-Output "$($_) is a member of G1_Group_US"
   }
   elseif ($g1GroupNonUS) {
     Write-Output "$($_) is a member of G1_Group_NON-US"
   }
   else {
    Write-Output "$($_) is not a member of either G1_Group_US or G1_Group_NON-US"
    $g1NonMember += "$_" + " "
   }
}

# -----G2 Section------
$g2Members | ForEach-Object -Parallel { # This line looks for each "item" in g2Members
  # Get the groups that the member belongs to
  $g2GroupUS = Get-ADPrincipalGroupMembership -Identity $_ | Where-Object {$_.SamAccountName -eq
'G2_Group_US'} # This allows us to check membership of G2_Group_US.
  $g2GroupNonUS = Get-ADPrincipalGroupMembership -Identity $_ | Where-Object {$_.SamAccountName -
eq 'G2_Group_NON-US'} # This allows us to check membership of G2_Group_NON-US.

   if ($g2GroupUS) {
    Write-Output "$($_) is a member of G2_Group_US"
   }
   elseif ($amiGroupNonUS) {
     Write-Output "$($_) is a member of G2_Group_NON-US"
   }
   else {
    Write-Output "$($_) is not a member of either G2_Group_US or G2_Group_NON-US"
    $g2NonMember += "$_" + " "
   }
}

Write-Host "The following users are not members of G1_Group_US or G1_Group_NON-US:" -
ForegroundColor Blue -NoNewline
Write-Output " $($g1NonMember)" | Format-Table -AutoSize $g1NonMember

Write-Host "The following users are not members of G2_Group_US or G2_Group_NON-US:" -
ForegroundColor Magenta -NoNewline
Write-Output " $($g2NonMember)" | Format-Table -AutoSize $g2NonMember
```

5. The code block below is a template for running PowerShell variable operations in **-Parallel** mode, through the use of a **Script Block**.

   a. **Running PowerShell Scripts in Parallel with Script Block**

```
# Creates a variable that holds the "username" of each member in the g#-group groups
$g1Members = (Get-ADGroupMember -Identity $g1Group | Select-Object -ExpandProperty SamAccountName)
$g2Members = (Get-ADGroupMember -Identity $g2Group | Select-Object -ExpandProperty SamAccountName)
$g3Members = (Get-ADGroupMember -Identity $g3Group | Select-Object -ExpandProperty SamAccountName)
$g4Members = (Get-ADGroupMember -Identity $g4Group | Select-Object -ExpandProperty SamAccountName)


# -----G1 Section-----
$g1MemberScriptBlock = {
  # Assigns the current index of the G1 member that is being looked at, to a variable
  $newG1Member = $_

  # Gets the groups that the member belongs to
  $g1GroupUS = Get-ADPrincipalGroupMembership -Identity $newG1Member | Where-Object {$_.
SamAccountName -eq 'G1_Group_US'} # This allows us to check membership of G1_Group_US
  $g1GroupNonUS = Get-ADPrincipalGroupMembership -Identity $newG1Member | Where-Object {$_.
SamAccountName -eq 'G1_Group_NON-US'} # This allows us to check membership of G1_Group_NON-US

  if ($g1GroupUS) {
    # Null these out because it'll go into the array otherwise
    Write-Output "$($_) is a member of G1_Group_US" | Out-Null
  }
  elseif ($g1GroupNonUS) {
    # Null these out because it'll go into the array otherwise
    Write-Output "$($_) is a member of G1_Group_NON-US" | Out-Null
  }
  # This G1 member does not belong to the "G1_Group_US" or "G1_Group_NON-US" groups
  else {
    # Outputs/Returns the SamAccountName for the G1 member, when it is called via the parallel
script block
    $newG1Member
  }

}

# Creates an Array to hold all of the G1 members that do not belong to the 'G1_Group_US' and
'G1_Group_NON-US' groups.
# This Array creation will also run the $g1MemberScriptBlock in parallel mode.
[array] $g1NonMember = $g1Members | ForEach-Object -Parallel $g1MemberScriptBlock


# -----G2 Section-----
$g2MemberScriptBlock = {
  # Assigns the current index of the G2 member that is being looked at, to a variable
  $newG2Member = $_

  # Gets the groups that the member belongs to
  $g2GroupUS = Get-ADPrincipalGroupMembership -Identity $_ | Where-Object {$_.SamAccountName -eq
'G2_Group_US'} # This allows us to check membership of G2_Group_US
  $g2GroupNonUS = Get-ADPrincipalGroupMembership -Identity $_ | Where-Object {$_.SamAccountName -
eq 'G2_Group_NON-US'} # This allows us to check membership of G2_Group_NON-US

  if ($g2GroupUS) {
    # Null these out because it'll go into the array otherwise
    Write-Output "$($_) is a member of G2_Group_US" | Out-Null
  }
  elseif ($g2GroupNonUS) {
    # Null these out because it'll go into the array otherwise
    Write-Output "$($_) is a member of G2_Group_NON-US" | Out-Null
  }
  # This G2 member does not belong to the "G2_Group_US" or "G2_Group_NON-US" groups
  else {
    # Outputs/Returns the SamAccountName for the G2 member, when it is called via the parallel
script block
    $newG2Member
  }
}

# Creates an Array to hold all of the G2 members that do not belong to the 'G2_Group_US' and
'G2_Group_NON-US' groups.
# This Array creation will also run the $g2MemberScriptBlock in parallel mode.
[array] $g2NonMember = $g2Members | ForEach-Object -Parallel $g2MemberScriptBlock


Write-Host "The following users are not members of G2_Group_US or G2_Group_NON-US:" -
ForegroundColor Blue -NoNewline
Write-Output " $($g2NonMember)" | Format-Table -AutoSize $g2NonMember

Write-Host "The following users are not members of G2_Group_US or G2_Group_NON-US:" -
ForegroundColor Magenta -NoNewline
Write-Output " $($g2NonMember)" | Format-Table -AutoSize $g2NonMember
```

Even More Useful PowerShell 7 Commands

- This page outlines a variety of additional useful PowerShell 7 commands, to be used in a specific type of script or command, along with the category of use-case pertaining to them.
- This page of the guide aims to make it easier to find these useful commands, by pointing the reader towards a variety of different commands that may be useful, or even essential, for particular use-cases.
- All of these commands can be found on Microsoft's Website: Learn / PowerShell / PowerShell 7.3 / Reference / Microsoft.PowerShell.Utility
  - Additional commands that are not in the "~/PowerShell 7.3" website section will have their respective parent guides linked, in their corresponding section.

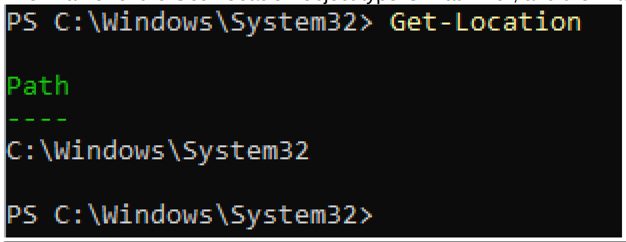
Even More Useful PowerShell 7 Commands - Table of Contents

Navigating the File System

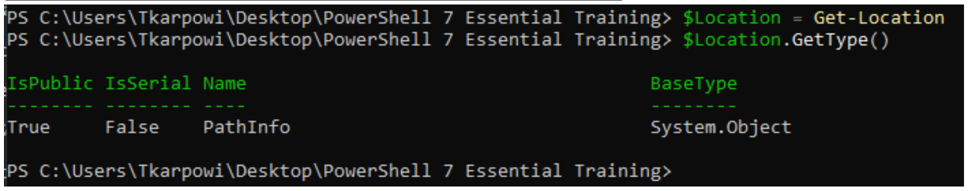
These commands can be useful for accessing source files for data, as well as saving data, a command, or even a script's output to a file, in a given directory. Example code for these concepts can be found in the attached "Ex_Files_PowerShell_7_EssT.zip" file. The example code has the file name, **Managing-Files-and-Folders.ps1**, with the File Path, "**.\Ex_Files_PowerShell_7_EssT**\Exercise Files\CH05\05_02**\Managing-Files-and-Folders.ps1**". Useful page on PowerShell 7 file system commands: Learn / PowerShell / PowerShell 7.3 / Learning PowerShell / Sample Scripts / Manage drives & files / Working with files and folders

1. **Get-ChildItem**
   a. This command will list all folders and files within a given directory.
      i. Directories will be highlighted in Blue.
      ii. .ZIP files will be displayed in Red text.
      iii. All other file types will be displayed in Gray text.
   b. **Get-ChildItem** -Path <File Path>
      i. Example: **Get-ChildItem** -Path "C:\Users\Tkarpowi\Desktop\"
      ii. The above example will output details for the contents of the "~\Desktop" directory to the terminal.
   c. **Get-ChildItem** -Path <File Path> **-Recurse**
      i. Example: **Get-ChildItem** -Path "C:\Users\Tkarpowi\Documents\" **-Recurse**
      ii. The above example will output details for the contents of the "~\Documents" directory, to the terminal.
      iii. Due to the -**Recurse** option, this cmdlet will also output the contents of the sub-folders in the "~\Documents" directory, as well as their sub-folders and files to the terminal as well.
   d. **Get-ChildItem** -Path <File Path> **-Force**
      i. Example: **Get-ChildItem** -Path "C:\" -Force
      ii. The **-Force** option will display both hidden, and non-hidden files to the terminal.
2. **New-Item**
   a. This command will create a new folder or file within a given directory.
   b. **New-Item** -Path <File Path> -ItemType **Directory**
      i. Example: **New-Item** -Path "C:\Users\Tkarpowi\Desktop\New Folder" -ItemType **Directory**
      ii. The **Directory** mode for the command specifies that a new folder will be created.
      iii. The above example creates a new folder in the "~\Desktop" directory, called **New Folder**.
   c. **New-Item** -Path <File Path> -ItemType **File**
      i. Example: **New-Item** -Path "C:\Users\Tkarpowi\Desktop\New Folder**file.txt**" -ItemType **File**
      ii. The **File** item type will create a new file.
      iii. The above example creates a file in the "~\New Folder" directory, called **file.txt**.
   d. **New-Item** -Path <File Path> -ItemType Directory **-Force**
      i. **Ex (Pt 1): $Location** = "C:\Users\Tkarpowi\Desktop\New Folder"
      ii. **Ex (Pt 2): New-Item** -Path "$(**$Location**)\PSFolderNew)" -ItemType Directory **-Force**

        iii. The **-Force** option overwrites an existing folder, if one exists with the same name.
        iv. The above example creates a new folder called "\PSFolderNew", in the "~\New Folder" directory.

3. **Out-File**
    a. This command can redirect a PowerShell command, function result, or even a script output to a file.
        i. If you redirect output to a file in a folder, that folder must be created before-hand.
    b. <Cmdlet | Text> **| Out-File** <File Path>
        i. Example: "I can now use PowerShell Pipe Commands!!" **| Out-File** "C:\Users\Tkarpowi\Desktop\New Folder**out-file.txt**"
        ii. The "I can now use PowerShell Pipe Commands!!" portion of the command is the contents that will be output to the target file, **out-file.txt**, in the "~\New Folder" directory.
    c. **Note:** With the **Out-File** command, you don't have to specify the **-Path** option, but it is good practice to do so.
        i. Example: "I can now use PowerShell Pipe Commands!!" **| Out-File -Path** "C:\Users\Tkarpowi\Desktop\New Folder**out-file.txt**"
        ii. The above example will function exactly the same as the previous cmdlet example.
    d. <Cmdlet | Text> **| Out-File** <File Path> **-Append**
        i. Example: "I can now append files in PowerShell 7!" **| Out-File** "C:\Users\Tkarpowi\Desktop\New Folder**out-file.txt**" **-Append**
        ii. The **-Append** option will add the target contents to the end of the specified file.
        iii. The "I can now append files in PowerShell 7!" portion of the command is the contents that will be appended to the target file, **out-file.txt**, in the "~\New Folder" directory.
4. **Copy-Item**
    a. This command can be used to copy files and folders, and various other items within the same namespace.
        i. For instance, a file can be copied to a folder, but a file cannot be copied to a certificate drive.
    b. **Copy-Item** -Path <File Path> **-Destination** <File Path>
        i. **Copy-Item** -Path "C:\Users\Tkarpowi\Desktop\New Folder**out-file.txt**" **-Destination** "C:\Users\Tkarpowi\Desktop\New Folder**out-file-copy.txt**"
        ii. A copy of the previously created text file, **out-file.txt**, is created as a new file, **out-file-copy.txt**, in the "~\New Folder" directory.
    c. **Copy-Item** -Path <File Path> -Destination <File Path> **-Recurse**
        i. **Copy-Item** -Path "C:\Users\Tkarpowi\Desktop\New Folder" -Destination "C:\Users\Tkarpowi\Desktop\New_Folder-Copy" **-Recurse**
        ii. The **-Recurse** option copies all of the child files and folders, along with the folder itself, to the specified "-Destination" location.
        iii. A copy of the folder **~\New Folder**, and the folder's contents, is created as a new folder, ~**\New_Folder-Copy**, in the "~\Desktop" directory.
5. **Remove-Item**
    a. This command can be used to delete files and folders, and various other items.
    b. **Items that can be Deleted:**
        i. Files and Folders
        ii. Registry Keys
        iii. Variables
        iv. Aliases
        v. Functions
    c. If there are sub-folders/files in a folder you are attempting to delete, you will get a prompt to confirm if you want to delete the folder.
    d. **Remove-Item** -Path <File Path>
        i. **Remove-Item** -Path "C:\Users\Tkarpowi\Desktop\New Folder**out-file-copy.txt**"
        ii. This command will delete the **out-file-copy.txt** file, in the "~\New Folder" directory.
    e. **Remove-Item** -Path "C:\Users\Tkarpowi\Desktop**\New Folder**"
        i. This command will give you a confirmation prompt to delete the ~**\New Folder** folder, since the folder in question still has the "file.txt" and "out-file.txt" files inside of it.
    f. **Remove-Item** -Path <File Path> **-Recurse**
        i. **Remove-Item** -Path "C:\Users\Tkarpowi\Desktop**\New_Folder-Copy**" **-Recurse**
        ii. The **-Recurse** option removes all of the child files and folders, along with the folder itself, in the specified "-Path" location.
        iii. This command will **not** give you a confirmation prompt to delete the ~**\New_Folder-Copy** folder, since the **-Recurse** option has been provided, even though there are still files in the folder.
6. **Get-Location**
    a. This command outputs/returns an object that represents the current directory, much like the "pwd" command.
    b. The "Name" of the **Get-Location** object type is "PathInfo", and the "BaseType" is "System.Object".
    c. 

```
PS C:\Windows\System32> Get-Location

Path
----
C:\Windows\System32

PS C:\Windows\System32>
```

    d. 

```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training> $Location = Get-Location
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training> $Location.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    PathInfo                                 System.Object

PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training>
```

7. **Set-Location**
    a. This command can be used to set the current working directory for PowerShell, much like the "cd" command.
    b. **Set-Location** <Directory Path>
        i. **Set-Location** "C:\Users\Tkarpowi\Desktop"
        ii. The current working directory for PowerShell is set to the "C:\Users\Tkarpowi\Desktop" directory.
    c. **Note:** With the **Set-Location** command, you don't have to specify the **-Path** option, but it is good practice to do so.
        i. Example: **Set-Location -Path** "C:\Users\Tkarpowi\Desktop"

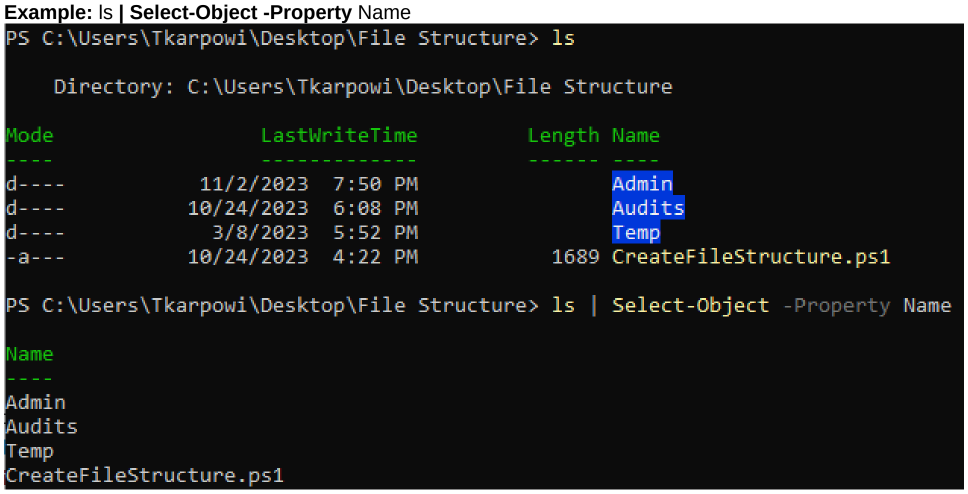
    ii. The above example will function exactly the same as the previous cmdlet example.

d. **Set-Location** <Directory Path> **-PassThru**
    i. **Set-Location** "C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training" **-PassThru**
    ii. The **-PassThru** option will set the current working directory for PowerShell, and output the absolute path for that directory to the console as well.
    iii. The new current working directory for PowerShell is set to the "C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training" fold er, and since the **-PassThru** option is there, that directory's file path is output to the console as well.

Selecting, Filtering, Sorting, and Obtaining Data In PowerShell 7

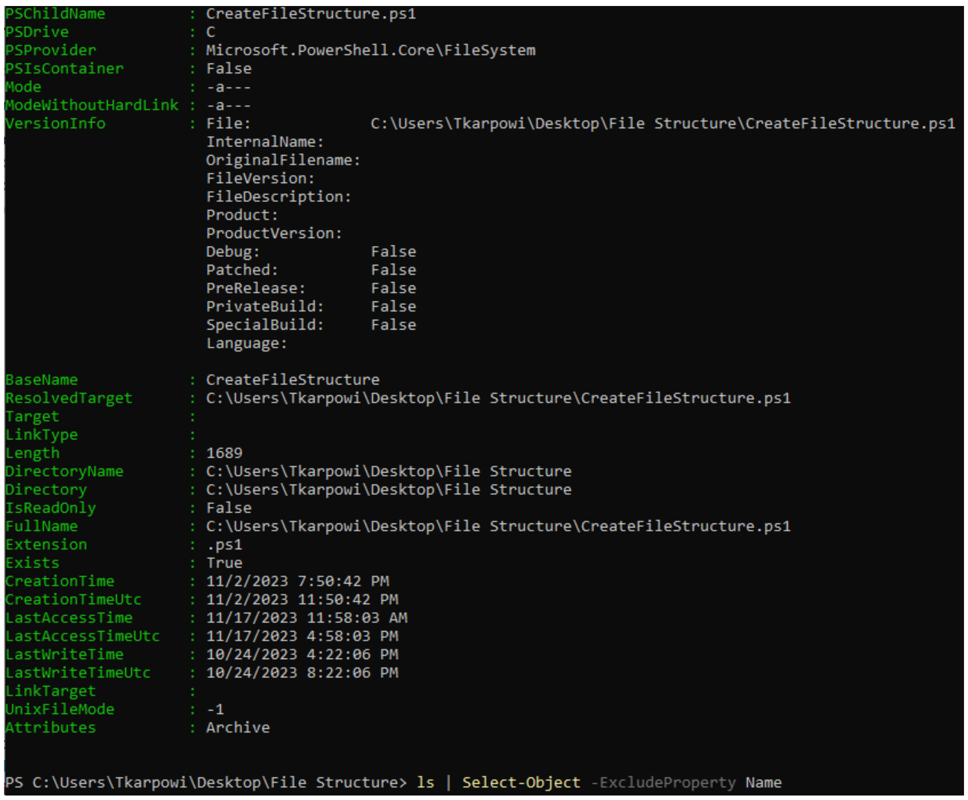
These commands can be useful for selecting, filtering, sorting, and obtaining data in PowerShell 7. These commands can also be used for parsing through and filtering out large amounts of data, as well as narrowing-down searches in PowerShell.

Useful page on working with PowerShell 7 objects: Learn / PowerShell / PowerShell 7.3 / Learning PowerShell / Sample Scripts / Working with objects / Viewing object structure

1. **Select-Object**
    a. This command selects PowerShell objects, based off a given object's properties, or a set of object's properties.
    b. This command can be used for both selecting data in PowerShell, as well as filtering out additional data.
    c. <Command or Object> **| Select-Object -Property** <Property of Command or Object>
        i. **Example:** ls **| Select-Object -Property** Name
        ii. 

```
PS C:\Users\Tkarpowi\Desktop\File Structure> ls

    Directory: C:\Users\Tkarpowi\Desktop\File Structure

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----           11/2/2023  7:50 PM                Admin
d----          10/24/2023  6:08 PM                Audits
d----            3/8/2023  5:52 PM                Temp
-a---          10/24/2023  4:22 PM           1689 CreateFileStructure.ps1

PS C:\Users\Tkarpowi\Desktop\File Structure> ls | Select-Object -Property Name

Name
----
Admin
Audits
Temp
CreateFileStructure.ps1
```

        iii. The above example takes the output from the ls command, and filters it to only output the **Name** property of the command output.
    d. <Command or Object> **| Select-Object -ExcludeProperty** <Property of Command or Object>
        i. **Example:** ls **| Select-Object -ExcludeProperty** Name

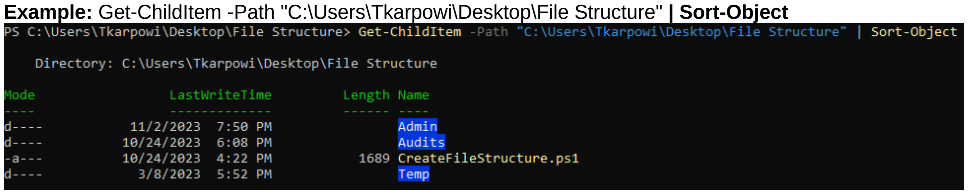


```
PSChildName          : CreateFileStructure.ps1
PSDrive              : C
PSProvider           : Microsoft.PowerShell.Core\FileSystem
PSIsContainer        : False
Mode                 : -a---
ModeWithoutHardLink  : -a---
VersionInfo          : File:             C:\Users\Tkarpowi\Desktop\File Structure\CreateFileStructure.ps1
                       InternalName:
                       OriginalFilename:
                       FileVersion:
                       FileDescription:
                       Product:
                       ProductVersion:
                       Debug:            False
                       Patched:          False
                       PreRelease:       False
                       PrivateBuild:     False
                       SpecialBuild:     False
                       Language:

BaseName             : CreateFileStructure
ResolvedTarget       : C:\Users\Tkarpowi\Desktop\File Structure\CreateFileStructure.ps1
Target               :
LinkType             :
Length               : 1689
DirectoryName        : C:\Users\Tkarpowi\Desktop\File Structure
Directory            : C:\Users\Tkarpowi\Desktop\File Structure
IsReadOnly           : False
FullName             : C:\Users\Tkarpowi\Desktop\File Structure\CreateFileStructure.ps1
Extension            : .ps1
Exists               : True
CreationTime         : 11/2/2023 7:50:42 PM
CreationTimeUtc      : 11/2/2023 11:50:42 PM
LastAccessTime       : 11/17/2023 11:58:03 AM
LastAccessTimeUtc    : 11/17/2023 4:58:03 PM
LastWriteTime        : 10/24/2023 4:22:06 PM
LastWriteTimeUtc     : 10/24/2023 8:22:06 PM
LinkTarget           :
UnixFileMode         : -1
Attributes           : Archive


PS C:\Users\Tkarpowi\Desktop\File Structure> ls | Select-Object -ExcludeProperty Name
```

ii.

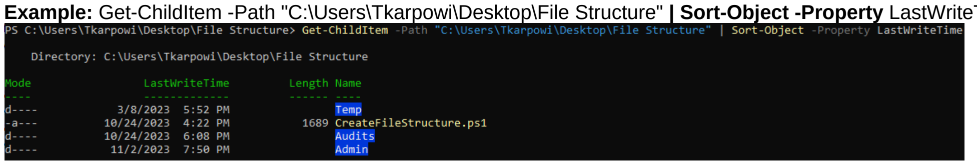
iii. The above example takes the output from the ls command, and filters it to output all of the properties, for all of the objects, output from the "ls" command, except for the **Name** property.

2. **Sort-Object**
   a. This command will sort object outputs, based on their property values.
   b. By default, this command will sort object output, based on the objects' "Name"s, from a-z.
   c. <Command or Object> **| Sort-Object**
      i. **Example:** Get-ChildItem -Path "C:\Users\Tkarpowi\Desktop\File Structure" **| Sort-Object**



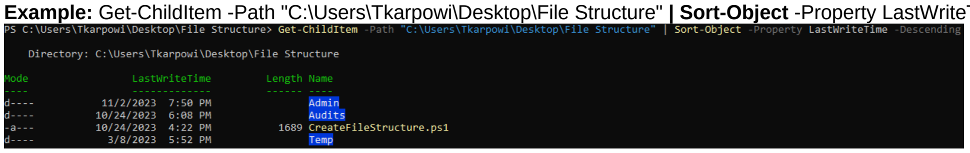
```
PS C:\Users\Tkarpowi\Desktop\File Structure> Get-ChildItem -Path "C:\Users\Tkarpowi\Desktop\File Structure" | Sort-Object

    Directory: C:\Users\Tkarpowi\Desktop\File Structure

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----           11/2/2023  7:50 PM                Admin
d----          10/24/2023  6:08 PM                Audits
-a---          10/24/2023  4:22 PM           1689 CreateFileStructure.ps1
d----            3/8/2023  5:52 PM                Temp
```

      ii.
      iii. The above example sorts the output of the "Get-ChildItem" cmdlet, by the objects' **Name** property value, in alphabetical order.
   d. <Command or Object> **| Sort-Object -Property** <Property of Command or Object>
      i. The **-Property** option will sort the output objects by the specified property.
      ii. **Example:** Get-ChildItem -Path "C:\Users\Tkarpowi\Desktop\File Structure" **| Sort-Object -Property** LastWriteTime



```
PS C:\Users\Tkarpowi\Desktop\File Structure> Get-ChildItem -Path "C:\Users\Tkarpowi\Desktop\File Structure" | Sort-Object -Property LastWriteTime

    Directory: C:\Users\Tkarpowi\Desktop\File Structure

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----            3/8/2023  5:52 PM                Temp
-a---          10/24/2023  4:22 PM           1689 CreateFileStructure.ps1
d----          10/24/2023  6:08 PM                Audits
d----           11/2/2023  7:50 PM                Admin
```

      iii.
      iv. The above example sorts the output of the "Get-ChildItem" cmdlet, by the objects' **LastWriteTime** property value, from the oldest "LastWriteTime", to the most recent "LastWriteTime".
   e. <Command or Object> **| Sort-Object** -Property <Property of Command or Object> **-Descending**
      i. The **-Descending** option will sort the output objects by the specified property in descending order.
      ii. By default, the "Sort-Object" command sorts output in ascending order.
      iii. **Example:** Get-ChildItem -Path "C:\Users\Tkarpowi\Desktop\File Structure" **| Sort-Object** -Property LastWriteTime **-Descending**

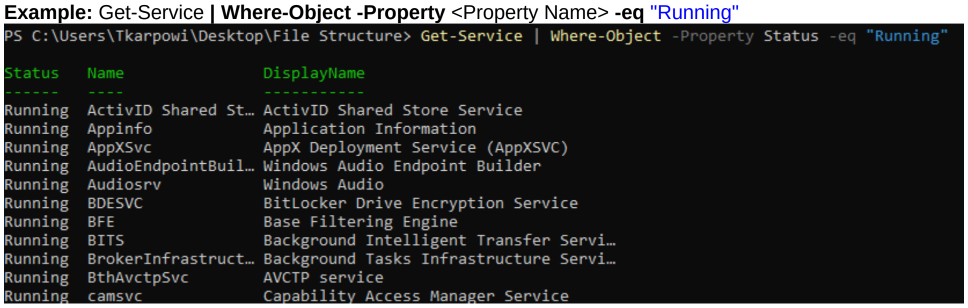


```
PS C:\Users\Tkarpowi\Desktop\File Structure> Get-ChildItem -Path "C:\Users\Tkarpowi\Desktop\File Structure" | Sort-Object -Property LastWriteTime -Descending

    Directory: C:\Users\Tkarpowi\Desktop\File Structure

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----           11/2/2023  7:50 PM                Admin
d----          10/24/2023  6:08 PM                Audits
-a---          10/24/2023  4:22 PM           1689 CreateFileStructure.ps1
d----            3/8/2023  5:52 PM                Temp
```

      iv.
      v. The above example sorts the output of the "Get-ChildItem" cmdlet, by the objects' **LastWriteTime** property value, from the most recent "LastWriteTime", to the oldest "LastWriteTime".

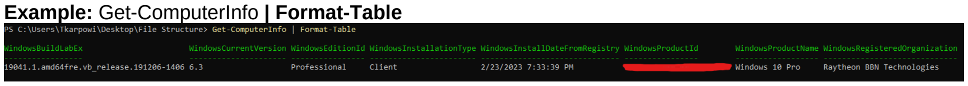
3. **Where-Object**
   a. This command will select objects, from a cmdlet or object output, based the specified child objects' property value.
      i. Learn / PowerShell / Reference / Microsoft.PowerShell.Core / Where-Object
   b. Get-Service **| Where-Object -Property** <Property Name> **-eq** "<Property Value>"
      i. The **-Property** option is used to specify the property of the object(s) that you would like to target.
      ii. The **-eq** option is used to specify that you want the object(s) to be output, only if they have the property value equal to the specified "<Property Value>".

iii. **Example:** Get-Service **| Where-Object -Property** <Property Name> **-eq** "Running"

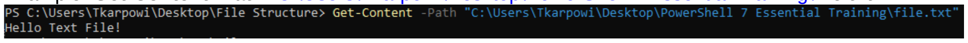
iv. 

```
PS C:\Users\Tkarpowi\Desktop\File Structure> Get-Service | Where-Object -Property Status -eq "Running"

Status   Name               DisplayName
------   ----               -----------
Running  ActivID Shared St… ActivID Shared Store Service
Running  Appinfo            Application Information
Running  AppXSvc            AppX Deployment Service (AppXSVC)
Running  AudioEndpointBuil… Windows Audio Endpoint Builder
Running  Audiosrv           Windows Audio
Running  BDESVC             BitLocker Drive Encryption Service
Running  BFE                Base Filtering Engine
Running  BITS               Background Intelligent Transfer Servi…
Running  BrokerInfrastruct… Background Tasks Infrastructure Servi…
Running  BthAvctpSvc        AVCTP service
Running  camsvc             Capability Access Manager Service
```

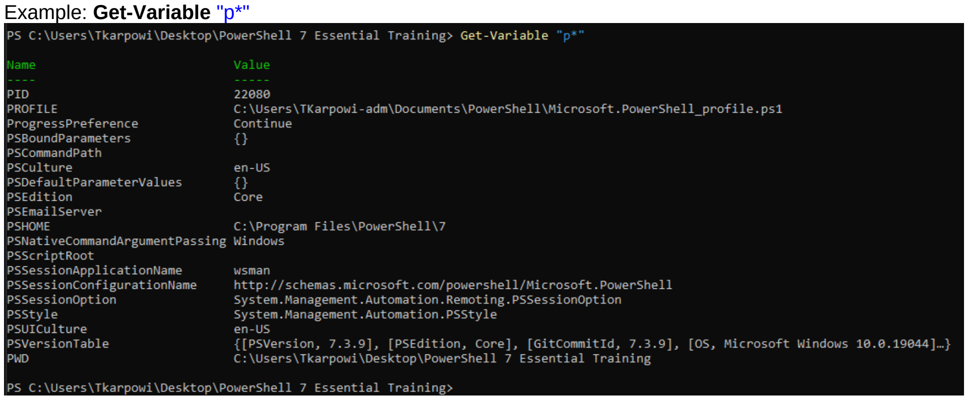
v. The above example outputs all of the system services that have the "Status" property value of "Running", to the terminal.

4. **Format-Table**
   a. This command will take command, or cmdlet input, and output these contents in an organized table format.
      i. Learn / PowerShell / Reference / Microsoft.PowerShell.Management / Get-Content
   b. <Cmdlet | Object> **| Format-Table**
      i. **Example:** Get-ADGroupMember -Identity "g1-group" **| Format-Table**
      ii. **[Image REDACTED For Security Purposes]**
      iii. The above example executes the "Get-ADGroupMember" cmdlet, and because of the **Format-Table** command, organizes the cmdlet's output contents to a table, which is then output to the terminal.
   c. <Cmdlet | Object> **| Format-Table**
      i. **Example:** Get-ComputerInfo **| Format-Table**
      ii. 

```
PS C:\Users\Tkarpowi\Desktop\File Structure> Get-ComputerInfo | Format-Table

WindowsBuildLabEx                      WindowsCurrentVersion WindowsEditionId WindowsInstallationType WindowsInstallDateFromRegistry WindowsProductId        WindowsProductName WindowsRegisteredOrganization
-----------------                      --------------------- ---------------- ----------------------- ------------------------------ ----------------        ------------------ -----------------------------
19041.1.amd64fre.vb_release.191206-1406 6.3                  Professional     Client                  2/23/2023 7:33:39 PM           [redacted]              Windows 10 Pro     Raytheon BBN Technologies
```

      iii. The above example executes the "Get-ComputerInfo" cmdlet, and because of the **Format-Table** command, organizes the cmdlet's output contents to a table, which is then output to the terminal.
5. **Get-Content**
   a. This command will obtain the contents of an item, such as a text file, and output those contents to the terminal.
      i. Learn / PowerShell / Reference / Microsoft.PowerShell.Management / Get-Content
   b. If you set a variable equal to the "Get-Content" command, you can store the output contents in a variable.
      i. This command can be used to read a text file, and store each line in a variable array.
   c. **Get-Content -Path** <File Path>
      i. The **-Path** option can be used to specify the item's absolute file path, if it is a file.
      ii. Example: **Get-Content -Path** "C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training**file.txt**"
      iii. 

```
PS C:\Users\Tkarpowi\Desktop\File Structure> Get-Content -Path "C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\file.txt"
Hello Text File!
```

      iv. The above example outputs the contents of the **file.txt** file to the terminal.
6. **Get-Variable**
   a. This command will search throughout the current PowerShell session, and find any variables, both system and user-made, that match the provided search string.
   b. This command can be extremely useful for creating a PowerShell 7 script, where some system, or file, information that you need is already stored in a pre-defined system variable.
      i. Since this command will allow you to find out the system name for that variable.
   c. **Get-Variable** <Search String>
      i. Example: **Get-Variable** "p*"
      ii. 

```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training> Get-Variable "p*"

Name                           Value
----                           -----
PID                            22080
PROFILE                        C:\Users\TKarpowi-adm\Documents\PowerShell\Microsoft.PowerShell_profile.ps1
ProgressPreference             Continue
PSBoundParameters              {}
PSCommandPath
PSCulture                      en-US
PSDefaultParameterValues       {}
PSEdition                      Core
PSEmailServer
PSHOME                         C:\Program Files\PowerShell\7
PSNativeCommandArgumentPassing Windows
PSScriptRoot
PSSessionApplicationName       wsman
PSSessionConfigurationName     http://schemas.microsoft.com/powershell/Microsoft.PowerShell
PSSessionOption                System.Management.Automation.Remoting.PSSessionOption
PSStyle                        System.Management.Automation.PSStyle
PSUICulture                    en-US
PSVersionTable                 {[PSVersion, 7.3.9], [PSEdition, Core], [GitCommitId, 7.3.9], [OS, Microsoft Windows 10.0.19044]…}
PWD                            C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training

PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training>
```

      iii. The above example will obtain, and output, the **Name** of all variables starting with the letter "p", alongside their **Value** details.

ActiveDirectory PowerShell Commands

Useful page on ActiveDirectory PowerShell commands: Learn / Windows / PowerShell / ActiveDirectory

Make sure you have already installed **RSAT for Windows 10**, to ensure the functionality of your ActiveDirectory PowerShell commands RSAT for Windows 10

1. **Get-ADUser**
   a. This command will obtain details about an ActiveDirectory user account.
      i. Learn / Windows / PowerShell / ActiveDirectory / Get-ADUser
   b. **Get-ADUser** -Identity <ADUser>
      i. Example: **Get-ADUser** -Identity "TKARPOWI"

        ii. The -Identity option is used to specify the username of the ActiveDirectory user that is being looked at.

2. **Get-ADGroup**
    a. This command will obtain details about ActiveDirectory groups.
        i. Learn / Windows / PowerShell / ActiveDirectory / Get-ADGroup
    b. **Get-ADGroup** -Identity <ADGroup>
        i. This cmdlet will only obtain details about a specific ActiveDirectory group.
        ii. The **-Identity** option is used to specify the ActiveDirectory group that is being looked at.
        iii. Example: **Get-ADGroup** -Identity "Administrators"
        iv. The above example will obtain details about the "Administrators" ActiveDirectory group.
    c. **Get-ADGroup** -Filter **'<Property>** -eq "<Property Value>" -and **<Property>** -ne "<Property Value>"**'**
        i. This cmdlet will obtain all of the groups matching the search criteria provided after the "-Filter" option.
        ii. The **-Filter** option is used to specify a specific search criteria to find any matching ActiveDirectory groups, as well as their associated group details.
        iii. Example: **Get-ADGroup** -Filter **'GroupCategory** -eq "Security" -and **GroupScope** -ne "DomainLocal"**'**
        iv. The above example will search through the ActiveDirectory group database, to find any groups that have "Security" as their group category, while also not being a part of the Local Domain, "DomainLocal".

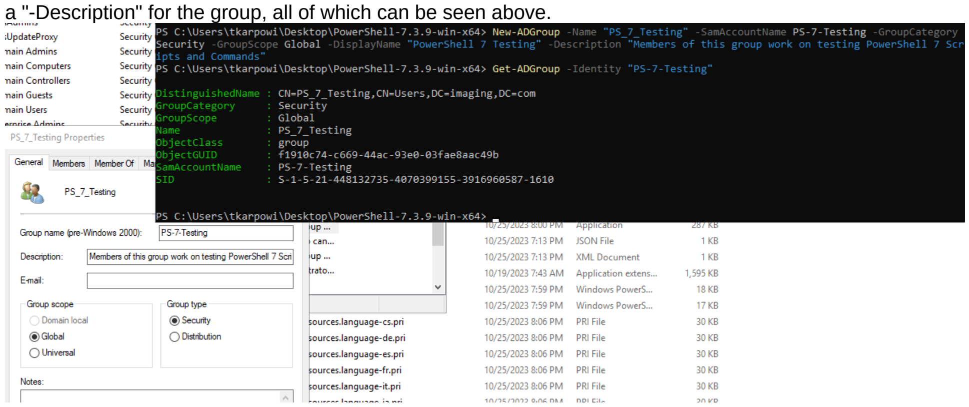
3. **Get-ADComputer**
    a. This command will obtain details about an ActiveDirectory computer.
        i. Learn / Windows / PowerShell / ActiveDirectory / Get-ADComputer
    b. **Get-ADComputer** -Filter { **<Property>** -Like <Property Value> }
        i. The **-Filter** option is used to search for, and specify, the search string property value for a given, or multiple, ActiveDirectory computer(s).
        ii. Example: **Get-ADComputer** -Filter { **OperatingSystem** -Like "Windows 10*" }
        iii. In the above example, ActiveDirectory details for every computer with a "Windows 10" operating system is output to the terminal.

4. **Get-ADGroupMember**
    a. This command will obtain, and list, all of the members of a given ActiveDirectory group.
        i. Learn / Windows / PowerShell / ActiveDirectory / Get-ADGroupMember
    b. **Get-ADGroupMember** -Identity <ADGroup>
        i. The **-Identity** option is used to specify the ActiveDirectory group members to search for.
        ii. Example: **Get-ADGroupMember** -Identity "g2-group"
        iii. The above example looks at all of the users in the "g2-group" group, and outputs ActiveDirectory details for all of the individual accounts, belonging to that group, to the terminal.
    c. **Get-ADGroupMember** -Identity <ADGroup> | **Select-Object** -ExpandProperty <ADGroupMember Property>
        i. The "Get-ADGroupMember" command can be utilized alongside the **Select-Object** command, to obtain only specific details about ActiveDirectory group members, for a particular group.
        ii. Example: **Get-ADGroupMember** -Identity "g1-group" | **Select-Object** -ExpandProperty SamAccountName
            1. The above example looks at all of the users in the "g1-group" group, and because of the **Select-Object** command, only their ActiveDirectory username is output to the terminal.
            2. The SamAccountName attribute is for selecting only the ActiveDirectory user's logon name (username).
        iii. Example: **$g3Members** = (**Get-ADGroupMember** -Identity "g3-group" | **Select-Object** -ExpandProperty SamAccountName)
        iv. The above example looks at all of the users in the "g3-group" group, obtains only their ActiveDirectory usernames, and stores the resulting output to the **$g3Members** variable.

5. **Get-ADPrincipalGroupMembership**
    a. This command will obtain all of the groups, and their associated details, that the specified ActiveDirectory user, computer, or group is a part of.
        i. Learn / Windows / PowerShell / ActiveDirectory / Get-ADPrincipalGroupMembership
    b. Example: **Get-ADPrincipalGroupMembership** -Identity <ADUsername | ADComputer | ADGroup>
        i. Example: **Get-ADPrincipalGroupMembership** -Identity "TKARPOWI"
        ii. The **-Identity** option is used to specify the ActiveDirectory user that you wish to obtain the associated ActiveDirectory groups from.
        iii. The above example will output details about all of the groups that "TKARPOWI" belongs to.

6. **New-ADUser**
    a. This command will create a new ActiveDirectory user account, with any specified settings and options that you provide.
        i. Learn / Windows / PowerShell / ActiveDirectory / New-ADUser
    b. **New-ADUser** -Name <ADUsername>
        i. This cmdlet will create a new ActiveDirectory user account, with no password, and no details.
        ii. The **-Name** option is used to specify the ActiveDirectory username for that account.
        iii. Example: **New-ADUser** -Name "TKARPOWI_2"
        iv. The above example will create a new ActiveDirectory user account, with the username "TKARPOWI_2", with no password, and no details.
    c. **New-ADUser** -Name <ADUsername> -AccountPassword <Password for ADUser>
        i. This cmdlet will create a new ActiveDirectory user account, with a specified password.
        ii. The **-AccountPassword** option is used to specify the password for the new ActiveDirectory user account being created.
        iii. Example: **New-ADUser** -Name "TKARPOWI_2-ADM" -AccountPassword (**Read-Host** -AsSecureString "AccountPassword") -Enabled **$true**
            1. The above example will create a new ActiveDirectory user account, with the username "TKARPOWI_2-ADM", and a password that is input to the terminal.
            2. The PowerShell terminal will take an input for the given ActiveDirectory user account's password, and read it through a "SecureString", which is basically an encrypted channel.
            3. The "-Enabled **$true**" part of the cmdlet, specifies that the ActiveDirectory account should be active, and able to be used/logged-in to.

7. **New-ADGroup**

- a. This command will create a new ActiveDirectory group.
    - i. Learn / Windows / PowerShell / ActiveDirectory / New-ADGroup
- b. **New-ADGroup** -Name <AD Group DisplayName>
    - i. The **-Name** option is used to specify the ActiveDirectory group name, for the new ActiveDirectory group.
    - ii. If you do not provide a "-SamAccountName", it will use the value you put after the "-Name" parameter, for the group's "-SamAccountName".
    - iii. If you do not supply the "-GroupScope" option, the command-line will prompt you to input the "GroupScope" for the specified group.
    - iv. **-GroupScope Values**
        1. DomainLocal or 0
        2. Global or 1
        3. Universal or 2
    - v. Example: **New-ADGroup** -Name "SysAdminClub"
        1. The above example will create a new ActiveDirectory group, with the ActiveDirectory group name of "SysAdminClub".
        2. For this example, the "GroupScope" that was entered into the terminal was "Global".
- c. **New-ADGroup** -Name <AD Group Name> -DisplayName <AD Group DisplayName>
    - i. The **-DisplayName** option is used to specify the ActiveDirectory group's object name, which can be utilized in PowerShell Scripting.
    - ii. Example: **New-ADGroup** -Name "PS_7_Testing" -SamAccountName **PS-7-Testing** -GroupCategory **Security** -GroupScope **Global** -DisplayName "PowerShell 7 Testing" -Description "Members of this group work on testing PowerShell 7 Scripts and Commands"
        1. The above example will create a new ActiveDirectory group, with the group name "PS 7 Testing", the "SamAccountName" **PS-7-Testing**, and the display name of "PowerShell 7 Testing".
        2. The above example also gives this new group properties for the group's "-GroupCategory", "-GroupScope", as well as a "-Description" for the group, all of which can be seen above.
        3. 

```
PS C:\Users\tkarpowi\Desktop\PowerShell-7.3.9-win-x64> New-ADGroup -Name "PS_7_Testing" -SamAccountName PS-7-Testing -GroupCategory
Security -GroupScope Global -DisplayName "PowerShell 7 Testing" -Description "Members of this group work on testing PowerShell 7 Scr
ipts and Commands"
PS C:\Users\tkarpowi\Desktop\PowerShell-7.3.9-win-x64> Get-ADGroup -Identity "PS-7-Testing"

DistinguishedName : CN=PS_7_Testing,CN=Users,DC=imaging,DC=com
GroupCategory     : Security
GroupScope        : Global
Name              : PS_7_Testing
ObjectClass       : group
ObjectGUID        : f1910c74-c669-44ac-93e0-03fae8aac49b
SamAccountName    : PS-7-Testing
SID               : S-1-5-21-448132735-4070399155-3916960587-1610


PS C:\Users\tkarpowi\Desktop\PowerShell-7.3.9-win-x64>
```

PS_7_Testing Properties

Group name (pre-Windows 2000): PS-7-Testing

Description: Members of this group work on testing PowerShell 7 Scri

E-mail:

Group scope: ○ Domain local [✓ Global] ○ Universal

Group type: [✓ Security] ○ Distribution

PowerShell 7 Remoting Commands

DO NOT INSTALL/ENABLE POWERSHELL REMOTING, UNLESS THE DEVICE IS ON AN ISOLATED NETWORK, OR YOU KNOW WHAT YOU'RE DOING

PowerShell 7 Remoting allows an individual to run PowerShell 7 commands from one device that is remotely connected to another device. These commands are sent from the remote device, and executed on the target device, as if the user was using the target device's PowerShell 7 terminal directly. To utilize PowerShell 7 Remoting, and PowerShell 7 Remoting commands, make sure you have installed the functionality for PowerShell Remoting, or executed the **.\Install-PowerShellRemoting.ps1** PowerShell script. There is a guide on how to do this in the **Other Settings & Useful Applications For PowerShell 7** page, with the section title, **Installing PowerShell Remoting**.

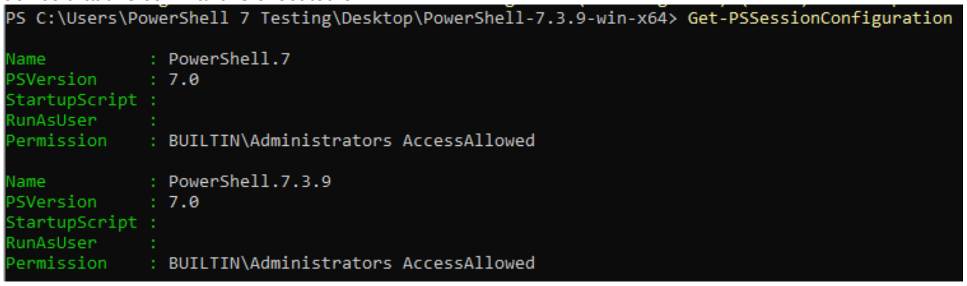
Useful page on PowerShell 7 Remoting: Learn / PowerShell / Learning PowerShell / PowerShell 101 / PowerShell remoting

1. **Enable-PSRemoting**
    - a. **THIS CAN OPEN YOUR DEVICE UP TO VULNERABILITIES AND POTENTIAL ATTACKS, MAKE SURE YOU ONLY UTILIZE THIS IN AN ISOLATED/CLOSED ENVIRONEMENT/NETWORK**
    - b. This command will open-up, and allow, other devices to remotely execute PowerShell 7 commands on the current device you are utilizing.
        - i. Learn / PowerShell / Reference / Microsoft.PowerShell.Core / Enable-PSRemoting
    - c. PowerShell commands cannot be remotely executed on a given device, unless this command is enabled, and the firewall rules in-place allow for remote PowerShell 7 connections to be established.
        - i. PowerShell 7 Remoting can be enabled if you are on a windows-recognized "Private Network".
        - ii. To set the status of your current network connection to a "Private Network" in PowerShell, utilize the command below

```
1. Set-NetConnectionProfile -Name "Network_Name" -NetworkCategory Private
```

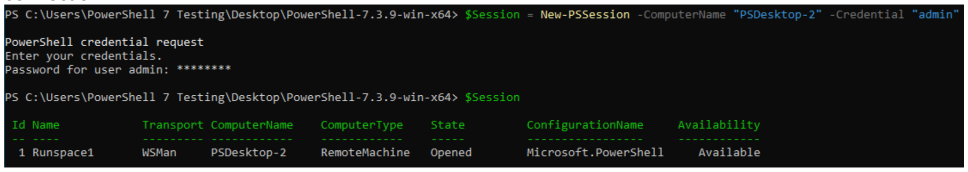
    - d. **Enable-PSRemoting** -Force
        - i. The **-Force** option will forcefully enable PowerShell Remoting on the current machine that you are utilizing.
        - ii. This means that you can forcefully open your machine to other users wishing to run PowerShell 7 commands on your device, remotely.
    - e. **Enable-PSRemoting -SkipNetworkProfileCheck** -Force
        - i. The **-SkipNetworkProfileCheck** option will allow you to enable PowerShell Remoting on the current device, even if the network that you are on is classified in Windows as a "Public Network".
        - ii. Make sure that you are on a secure, or trustworthy, network if you do execute this cmdlet.

2. **Hostname**
    a. This command will tell you the computer name of the system that you are working on.

3. **Get-PSSessionConfiguration**
    a. This command will show details about the registered Remote PowerShell session configurations, for the computer that this command is run on.
        i. Learn / PowerShell / Reference / Microsoft.PowerShell.Core / Get-PSSessionConfiguration
    b. This command will show the **Name**s and versions of PowerShell, **PSVersion**, as well as their associated **Permission**s, for the given device that this command is executed on.
    c. 

```
PS C:\Users\PowerShell 7 Testing\Desktop\PowerShell-7.3.9-win-x64> Get-PSSessionConfiguration

Name          : PowerShell.7
PSVersion     : 7.0
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed

Name          : PowerShell.7.3.9
PSVersion     : 7.0
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed
```

        i. In the above example output, the versions of PowerShell that can be utilized are PowerShell 7.0 and PowerShell 7.3.9.
        ii. Additionally, only local accounts that are part of the "Administrators" group, can execute PowerShell commands remotely on this device.

4. **New-PSSession**
    a. This command will establish a new Remote PowerShell 7 connection.
        i. Learn / PowerShell / Reference / Microsoft.PowerShell.Core / New-PSSession
    b. **New-PSSession** -ComputerName <Target Computer's Machine Name> -Credential <Target Computer's Local Account>
        i. The **-ComputerName** option is used to specify the target machine that you wish to remotely execute PowerShell commands on.
        ii. The **-Credential** option is used to specify the local account, that you wish to log-in as, on the target machine that you wish to remotely execute PowerShell commands on.
        iii. When you are starting a New-PSSession, you cannot use the Primary Account for the device, or an account that's currently logged in, for the **-Credential** <Target Computer's Local Account> section.
            1. If you try to do this, it will error-out, and say "Access denied".
        iv. Example: **New-PSSession** -ComputerName "localhost" -Credential "admin"
        v. The above example will establish a new Remote PowerShell 7 session, on the specified device, in this case the host machine, "l ocalhost", with the same permissions as the account that they provided, in this case "admin".
    c. **$<Session Variable Name> = New-PSSession** -ComputerName <Target Computer's Machine Name> -Credential <Target Computer's Local Account>
        i. **The best way to establish a new Remote PowerShell 7 connection**, is to create a variable that stores the given connection inside of it.
        ii. This will allow you to obtain details about a given Remote PowerShell session, by simply typing the variable name into the terminal.
        iii. Example: **$Session = New-PSSession** -ComputerName "PSDesktop-2" -Credential "admin"
        iv. This command will establish a new Remote PowerShell 7 connection with itself, and store details about this Remote PowerShell session in the variable **$Session**.
        v. If you type **$Session** in the terminal, after the example command, it will outputs details about the current Remote PowerShell connection.
        vi. 

```
PS C:\Users\PowerShell 7 Testing\Desktop\PowerShell-7.3.9-win-x64> $Session = New-PSSession -ComputerName "PSDesktop-2" -Credential "admin"

PowerShell credential request
Enter your credentials.
Password for user admin: ********

PS C:\Users\PowerShell 7 Testing\Desktop\PowerShell-7.3.9-win-x64> $Session

 Id Name            Transport ComputerName    ComputerType    State         ConfigurationName     Availability
 -- ----            --------- ------------    ------------    -----         -----------------     ------------
  1 Runspace1       WSMan     PSDesktop-2     RemoteMachine   Opened        Microsoft.PowerShell     Available
```

    d. If you are having issues with this **New-PSSession** command, inputting some of these commands, **on the machine that you want to remote into**, may help, but could open your device up to more vulnerabilities.
        i. **Both Machines: Enable PSRemoting, Regardless of Connected-Network Type**

```
Enable-PSRemoting -SkipNetworkProfileCheck -Force
```

        ii. **Both Machines: Change Connected-Network Type to Private**

```
Set-NetConnectionProfile -Name "<Connected Network's Name>" -NetworkCategory Private

# Example
Set-NetConnectionProfile -Name "Unidentified network" -NetworkCategory Private
```

        iii. **Both Machines: PowerShell - Change Network to Private - For Remoting**

```
Set-Item WSMan:\localhost\Client\TrustedHosts -Value "<Target PC-Name>,<Target PC's IP
Address>"

# Example
Set-Item WSMan:\localhost\Client\TrustedHosts -Value "PSDesktop-2,192.168.197.130"
```

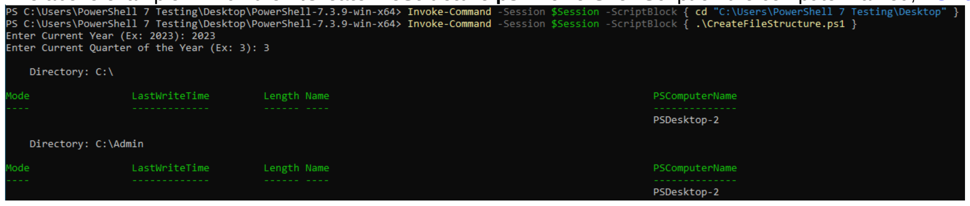
iv. **THE TWO CMDLETS BELOW CAN OPEN YOUR DEVICE UP TO VULNERABILITIES AND POTENTIAL ATTACKS, MAKE SURE YOU ONLY UTILIZE THIS IN AN ISOLATED/CLOSED ENVIRONEMENT/NETWORK**

v. **Target Machine: Allow Device to Send Remote Requests to the Host Machine**

```
Set-NetFirewallRule -Name "WINRM-HTTP-In-TCP" -RemoteAddress Any
```

vi. **Target Machine: Inserting Registry Key to Enable Account Credentials to be Remotely Entered**

```
reg add HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System /v
LocalAccountTokenFilterPolicy /t REG_DWORD /d 1 /f
```

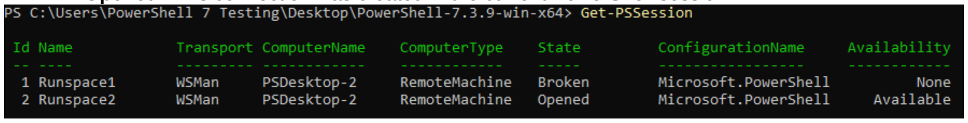
5. **Invoke-Command**
   a. This command will invoke, run, a given command, or cmdlet, on a Remote PowerShell connection.
      i. Learn / PowerShell / Reference / Microsoft.PowerShell.Core / Invoke-Command
   b. **Invoke-Command** -Session **$<Session Variable Name>** -ScriptBlock **{** <Command to be Executed on Target Machine> **}**
      i. This cmdlet will allow you to run a given command, or cmdlet on a specific Remote PowerShell connection.
      ii. The **-Session** option is used to specify the Remote PowerShell session that you want to run the command on, by variable, or session name.
      iii. The **-ScriptBlock** option is used to specify the actual command, or cmdlet, that you wish to run on the target device.
      iv. Whatever is in the curly braces **{ }**, will be the command, or cmdlet that you wish to run on the target device.
      v. Example: **Invoke-Command** -Session **$Session** -ScriptBlock **{** Get-ComputerInfo **}**
      vi. The above example executes the "Get-ComputerInfo" command, on the Remote PowerShell session stored in the **$Session** var iable.
   c. **Executing PowerShell Scripts Remotely**
      i. **Invoke-Command** -Session **$<Session Variable Name>** -ScriptBlock **{ .\<PowerShell Script Name> }**
      ii. This command will allow you to run a PowerShell Script on a remote computer.
      iii. A Remote PowerShell session must be established, and active, before running this cmdlet.
      iv. Example (Part 1): **Invoke-Command** -Session **$Session** -ScriptBlock **{ cd** "C:\Users\PowerShell 7 Testing\Desktop" **}**
         1. The above example will set the current working directory, for the remotely connected device's PowerShell terminal, to the location of the PowerShell 7 Script.
         2. Example (Part 2): **Invoke-Command** -Session **$Session** -ScriptBlock **{ .\CreateFileStructure.ps1 }**
         3. The above example will run the **~\CreateFileStructure.ps1** PowerShell Script on the computer named, PSDesktop-2.
         4. 

```
PS C:\Users\PowerShell 7 Testing\Desktop\PowerShell-7.3.9-win-x64> Invoke-Command -Session $Session -ScriptBlock { cd "C:\Users\PowerShell 7 Testing\Desktop" }
PS C:\Users\PowerShell 7 Testing\Desktop\PowerShell-7.3.9-win-x64> Invoke-Command -Session $Session -ScriptBlock { .\CreateFileStructure.ps1 }
Enter Current Year (Ex: 2023): 2023
Enter Current Quarter of the Year (Ex: 3): 3

    Directory: C:\

Mode                 LastWriteTime         Length Name                                                            PSComputerName
----                 -------------         ------ ----                                                            --------------
                                                                                                                  PSDesktop-2

    Directory: C:\Admin

Mode                 LastWriteTime         Length Name                                                            PSComputerName
----                 -------------         ------ ----                                                            --------------
                                                                                                                  PSDesktop-2
```

6. **Enter-PSSession**
   a. Format: **Enter-PSSession [$<Session Variable Name> | <Target Computer's Machine Name>]**
   b. This command will put you into the remote device's terminal instance.
      i. Learn / PowerShell / Reference / Microsoft.PowerShell.Core / Enter-PSSession
   c. This allows you to run commands directly on the remotely accessed device's PowerShell terminal.
   d. This removes the need for utilizing the **Invoke-Command** cmdlet, every time you want to execute a command on the remote device.
   e. Example: **Enter-PSSession $Session**
      i. Since all of the Remote PowerShell session data is stored in the variable, this won't create a new Remote PowerShell session, it will simply utilize the existing session.
      ii. This means that you do not need to authenticate again, before entering the remote device's PowerShell terminal instance.
   f. Example: **Enter-PSSession** -ComputerName "Server01" -Credential "admin"
      i. This will initialize a new Remote PowerShell session.
      ii. If you need to authenticate remotely, you will have to supply the "-Credential" option, even if you established a Remote PowerShell session recently.

7. **Exit-PSSession**
   a. This command will exit the remote device's terminal instance, and put you back in your own local PowerShell terminal instance.
      i. Learn / PowerShell / Reference / Microsoft.PowerShell.Core / Exit-PSSession
   b. You can also use the **exit** command to switch back to your host device's PowerShell terminal instance.

8. **Get-PSSession**
   a. This command display all of the Remote PowerShell sessions, and connections, that are currently active on the local machine, or the target machine if you are accessing the device's terminal instance remotely.

            i. Learn / PowerShell / Reference / Microsoft.PowerShell.Core / Get-PSSession
   b. This command will display "Disconnected", "Broken", and "Opened" Remote PowerShell sessions, under in the **State** column.
      i. **Disconnected:** This connection was created in a different PowerShell session.
      ii. **Opened:** This connection was created in the current PowerShell session.
   c. 

```
PS C:\Users\PowerShell 7 Testing\Desktop\PowerShell-7.3.9-win-x64> Get-PSSession

 Id Name            Transport ComputerName    ComputerType    State        ConfigurationName     Availability
 -- ----            --------- ------------    ------------    -----        -----------------     ------------
  1 Runspace1       WSMan     PSDesktop-2     RemoteMachine   Broken       Microsoft.PowerShell          None
  2 Runspace2       WSMan     PSDesktop-2     RemoteMachine   Opened       Microsoft.PowerShell     Available
```

   d. **Get-PSSession** -ComputerName <Target Computer's Machine Name> -Credential <Target Computer's Local Account>
      i. This cmdlet will obtain the Remote PowerShell sessions established on the specified <Target Computer's Machine Name>.
      ii. Example: **Get-PSSession** -ComputerName "PSDesktop-2" -Credential "admin"
      iii. The above example will obtain all of the Remote PowerShell sessions established on the PSDesktop-2 device.
      iv. If you need to authenticate remotely, you will have to supply the "-Credential" option, even if you established a Remote PowerShell session recently.
   e. **Get-PSSession** -Id <#>
      i. This cmdlet will obtain a specific Remote PowerShell session, for the machine belonging to the PowerShell terminal instance, based on a given PowerShell session ID #.
      ii. Example: **Get-PSSession** -Id 2
      iii. The above example will obtain details about the Remote PowerShell session, for the PowerShell session with the ID "2".

9. **Remove-PSSession**
   a. This command will close all of the Remote PowerShell sessions, on the current machine hosting the PowerShell terminal.
   b. If this command is executed on a remotely-accessed device, it will close all of the Remote PowerShell sessions on that given machine, including the one between the given machine and your local machine.
   c. **Remove-PSSession** -Session **$<Session Variable Name>**
      i. This cmdlet will close the Remote PowerShell session stored in the given **$<Session Variable Name>**.
      ii. Example: **Remove-PSSession** -Session **$Session**
      iii. This cmdlet will close the Remote PowerShell session stored in the **$Session** variable.
   d. **Remove-PSSession** -ComputerName <Target Computer's Machine Name>
      i. This cmdlet will close all of the Remote PowerShell sessions on the specified <Target Computer's Machine Name>.
      ii. Example: **Remove-PSSession** -ComputerName localhost
      iii. The above example will close all of the Remote PowerShell sessions on the localhost device (in this case the current device).
   e. **Remove-PSSession** -Id <#>
      i. This cmdlet will close the Remote PowerShell session, based on a given PowerShell session ID #.
      ii. Example: **Remove-PSSession** -Id 1
      iii. The above example will close the Remote PowerShell session, for the PowerShell session ID "1".

10. **Get-UICulture**
   a. This command will obtain various information about the current user interface, for the system that you are working on.
   b. If you are **not remotely connected** to a device, this command will display information about your **host device**.
   c. If you use this command on a remotely-connected device, using "Invoke-Command", it will also display a "PSComputerName" field.
   d. If you run this command on a **remotely-connected** device, this command will display information about the **target remotely-connected device**,

Defining Custom "Get-Help" Outputs

Custom "Get-Help" outputs can be extremely useful when it comes to documenting, and explaining certain parts of a custom PowerShell Script, or even a function with custom options inside of a PowerShell Script.

Example code for this section can be found in the attached "Additional Example Scripts.zip" file, with the file name, **if_statements_with_Get-Help_functionality.ps1**.

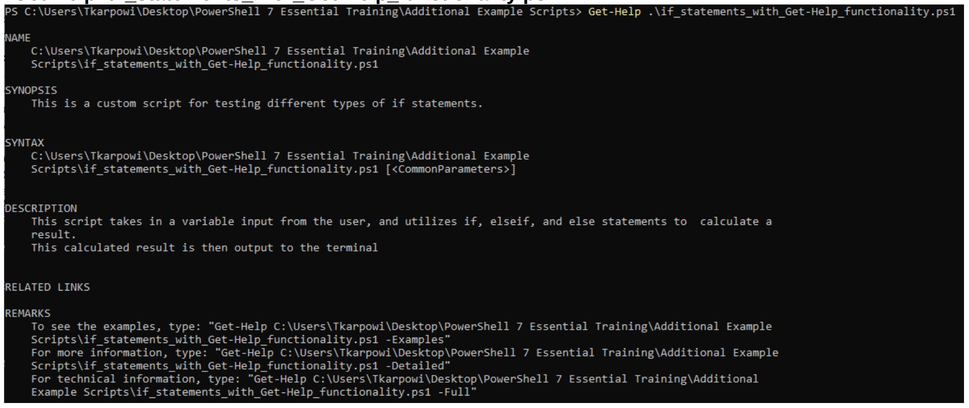
Even more example code for this section can be found in the attached "Ex_Files_PowerShell_7_EssT.zip" file. One example script has the file name, **Define-Custom-Help.ps1**, with the file path, ".\Ex_Files_PowerShell_7_EssT\Exercise Files\CH04\04_09\Define-Custom-Help.ps1". Another example code has the file name, **Parameters-Attributes-for-Scripts-and-Functions.ps1**, with the file path, ".\Ex_Files_PowerShell_7_EssT\Exercise Files\CH04\04_03\Parameters-Attributes-for-Scripts-and-Functions.ps1".

1. The following Code Block below shows the structure for adding custom "Get-Help" command functionality into a PowerShell Script.
   a. The code below can be utilized as a template for adding "Get-Help" functionality into a PowerShell Script.
   b. **Template_for_Adding_Get-Help_Functionality_in_PSScript.ps1**

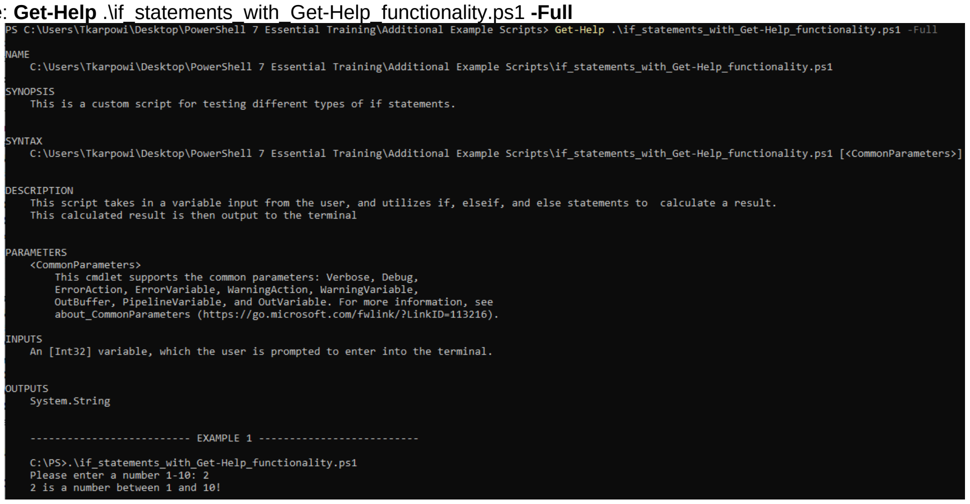
```
<#
    .SYNOPSIS
    <A brief description about the functionality of the PowerShell Script.>

    .DESCRIPTION
    <A detailed description about the functionality of the PowerShell Script.>

    .INPUTS
    [<The acceptable input object types for the PowerShell Script.>]

    .OUTPUTS
    System.<The resulting output object types for the PowerShell Script.>
    .EXAMPLE
    C:\PS> .\<Name of PowerShell Script>.ps1
    <Example Prompt (if applicable)>
    <Example Output (if applicable)>
#>

<#
##-----Script Code-----
#>
```

2. The following Code Block below shows how adding custom "Get-Help" command functionality can be incorporated into a PowerShell Script.
   a. **if_statements_with_Get-Help_functionality.ps1**

```
<#
    .SYNOPSIS
    This is a custom script for testing different types of if statements.

    .DESCRIPTION
    This script takes in a variable input from the user, and utilizes if, elseif, and else
statements to  calculate a result.
    This calculated result is then output to the terminal

    .INPUTS
    An [Int32] variable, which the user is prompted to enter into the terminal.

    .OUTPUTS
    System.String

    .EXAMPLE
    C:\PS> .\if_statements_with_Get-Help_functionality.ps1
    Please enter a number 1-10: 2
    2 is a number between 1 and 10!
#>


# Example PowerShell 7 Script for if statements

# Get input from the user, in the PowerShell Terminal
[Int32] $numberInput = Read-Host "Please enter a number 1-10"

# Checks if the number is less than 1
if ($numberInput -lt 1) {

    Write-Output "$numberInput is less than 1. The number must be between 1 and 10."

}

# Checks if the number is greater than 10
elseif ($numberInput -gt 10) {

    Write-Output "$numberInput is greater than 10. The number must be between 1 and 10."

}

# Checks if the number is between 1 and 10
# Checks if the number is Greater than or equal to 1, and Less than or equal to 10
elseif ($numberInput -ge 1 && $numberInput -le 10) {

    Write-Output "$numberInput is a number between 1 and 10!"

}

else {

    Write-Output "$numberInput is not a number. Make sure you enter a number between 1 and 10."

}
```

3. **Get-Help .\<PowerShell Script>**
    a. This cmdlet will display a short summary of the "Get-Help" details supplied in the **<# #>** section, starting with ".SYNOPSIS" at the top of the PowerShell Script, for a given PowerShell Script.
    b. If certain sections are not provided in the script, then it will either try to auto-fill them, or just not display them at all.
    c. **Information Displayed**
        i. **Name:** The name of the PowerShell Script (usually with the script's absolute file path).
        ii. **Synopsis:** A brief description about the functionality of the PowerShell Script.
        iii. **Syntax:** The syntax for how to utilize the PowerShell Script, including any available options and cmdlet structure.
        iv. **Description:** A detailed description about the functionality of the PowerShell Script.
        v. **Remarks:** Displays the syntax for how to obtain even more help contents, pertaining to the PowerShell Script.
    d. Example: **Get-Help .\if_statements_with_Get-Help_functionality.ps1**
        i. 

```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example Scripts> Get-Help .\if_statements_with_Get-Help_functionality.ps1

NAME
    C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example
    Scripts\if_statements_with_Get-Help_functionality.ps1

SYNOPSIS
    This is a custom script for testing different types of if statements.


SYNTAX
    C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example
    Scripts\if_statements_with_Get-Help_functionality.ps1 [<CommonParameters>]


DESCRIPTION
    This script takes in a variable input from the user, and utilizes if, elseif, and else statements to  calculate a
    result.
    This calculated result is then output to the terminal


RELATED LINKS

REMARKS
    To see the examples, type: "Get-Help C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example
    Scripts\if_statements_with_Get-Help_functionality.ps1 -Examples"
    For more information, type: "Get-Help C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example
    Scripts\if_statements_with_Get-Help_functionality.ps1 -Detailed"
    For technical information, type: "Get-Help C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional
    Example Scripts\if_statements_with_Get-Help_functionality.ps1 -Full"
```

        ii. Notice how the **NAME** section was auto-filled with the PowerShell Script's absolute path, and it's name.
            1. Since the **NAME** section was not provided in the PowerShell Script's "Get-Help" documentation, the "Get-Help" command auto-filled it into the terminal.
4. **Get-Help** <Command> **-Full**
    a. The **-Full** option will display all of the "Get-Help" details for the PowerShell Script.
        i. This option will display everything that the **Get-Help** command displays to the terminal, except for "Remarks".
        ii. This option will also display the "Parameters", "Inputs", "Outputs", and "Example"s, for the PowerShell Script.
    b. **Information Displayed**
        i. **Name:** The name of the PowerShell Script (usually with the script's absolute file path).
        ii. **Synopsis:** A brief description about the functionality of the PowerShell Script.
        iii. **Syntax:** The syntax for how to utilize the PowerShell Script, including any available options and cmdlet structure.
        iv. **Description:** A detailed description about the functionality of the PowerShell Script.
        v. **Parameters:** All of the available command options, and some details about these options, for the PowerShell Script.
        vi. **Inputs:** The acceptable input object types for the PowerShell Script.
        vii. **Outputs:** The resulting output object types for the PowerShell Script.
        viii. **Example:** An example of the syntax for utilizing the PowerShell Script, as well as an example output.
    c. Example: **Get-Help** .\if_statements_with_Get-Help_functionality.ps1 **-Full**
        i. 

```
PS C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example Scripts> Get-Help .\if_statements_with_Get-Help_functionality.ps1 -Full

NAME
    C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example Scripts\if_statements_with_Get-Help_functionality.ps1

SYNOPSIS
    This is a custom script for testing different types of if statements.


SYNTAX
    C:\Users\Tkarpowi\Desktop\PowerShell 7 Essential Training\Additional Example Scripts\if_statements_with_Get-Help_functionality.ps1 [<CommonParameters>]


DESCRIPTION
    This script takes in a variable input from the user, and utilizes if, elseif, and else statements to  calculate a result.
    This calculated result is then output to the terminal


PARAMETERS
    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (https://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    An [Int32] variable, which the user is prompted to enter into the terminal.


OUTPUTS
    System.String


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>.\if_statements_with_Get-Help_functionality.ps1
    Please enter a number 1-10: 2
    2 is a number between 1 and 10!
```

        ii. Notice how the **NAME** section was auto-filled with the PowerShell Script's absolute path, and it's name.
            1. Since the **NAME** section was not provided in the PowerShell Script's "Get-Help" documentation, the "Get-Help" command auto-filled it into the terminal.
    d. **Help** <Command>
        i. The **Help** command will display everything that the **Get-Help** command, with the **-Full** option, displays to the terminal.
        ii. The **Help** command will output part of the **-Full** output, and you will have to press the "Enter" key to manually load the next line of help information.
        iii. This command is essentially the short-hand version of the **Get-Help** <Command> **-Full** cmdlet.
        iv. Example: **Help** .\if_statements_with_Get-Help_functionality.ps1

Other Settings & Useful Applications For PowerShell 7

- This page contains guides on how to install a variety of different applications that may be helpful, or even essential, for particular PowerShell 7 use-cases.
- This page also contains a guide on how to configure personalization settings, to change the look of your terminal, in PowerShell 7.

Other Settings & Useful Applications For PowerShell 7 - Table of Contents

Installing PowerShell Core Policy Definitions

This section utilizes files from the "PowerShell-7.3.9-win-x64.zip" file.

The Latest Version of the PowerShell 7 **.zip** file can be downloaded here: PowerShell 7 Releases - PowerShell/PowerShell - GitHub

1. Once you have the .zip files extracted, right-click the "pwsh.exe" file, and click "Run as administrator" to launch PowerShell 7 in Administrator Mode.
2. In the terminal, type **.\InstallPSCorePolicyDefinitions.ps1**, and press the "Enter" key.
   a. If the above command **does not work**, type "Get-ExecutionPolicy", press the "Enter" key, and make note of the command output.
   b. Next, type "Set-ExecutionPolicy RemoteSigned", press the "Enter" key, and then input the above command again.
   c. Once you have entered the ".\InstallPSCorePolicyDefinitions.ps1" command, type "Set-ExecutionPolicy <Previous Execution Policy>", and press the "Enter" key.

Installing PowerShell Remoting

DO NOT INSTALL/ENABLE POWERSHELL REMOTING, UNLESS THE DEVICE IS ON AN ISOLATED NETWORK, OR YOU KNOW WHAT YOU'RE DOING

This section utilizes files from the "PowerShell-7.3.9-win-x64.zip" file.

The Latest Version of the PowerShell 7 **.zip** file can be downloaded here: PowerShell 7 Releases - PowerShell/PowerShell - GitHub

1. Right-click the "pwsh.exe" file, and click "Run as administrator" to launch PowerShell 7 in Administrator Mode.
2. In the terminal, type **.\Install-PowerShellRemoting.ps1**, and press the "Enter" key.
   a. If the above command **does not work**, type "Get-ExecutionPolicy", press the "Enter" key, and make note of the command output.
   b. Next, type "Set-ExecutionPolicy RemoteSigned", press the "Enter" key, and then input the above command again.
   c. Once you have entered the ".\Install-PowerShellRemoting.ps1" command, type "Set-ExecutionPolicy <Previous Execution Policy>", and press the "Enter" key.

Personalize PowerShell (Non-Windows Terminal)

Personalizing PowerShell, using the **PowerShell 7 (x64)** Application.

1. Press the **[Windows Key]+[Q]** keys at the same time (to open the Windows Search Bar), type "PowerShell 7 (x64)", right-click on **PowerShell 7 (x64)**, and click on "Run as administrator".
2. Right-click the top-left corner of the application bar, click on **Properties**, and this will let you customize how PowerShell looks, as well as the **Buffe r Size:** of the PowerShell Terminal itself.
   a. The **Buffer Size:** is essentially the PowerShell command execution history. It indicates how many commands the terminal (window) instance will remember.
3. Note: When you change any of the PowerShell terminal properties, such as the "Colors" or "Buffer Size:", you need to **open a new PowerShell 7 (x64) instance** for these changes to take effect.

Downloading & Installing Windows Terminal (Microsoft Store)

Download, and install the **Windows Terminal** application from the **Microsoft Store**: Windows Terminal (Microsoft Store)

1. Click on the Windows Terminal (Microsoft Store) link above, and click on the blue box on the top-right of the page that says "Get in Store app".
2. You will get a prompt that says, "Open Microsoft Store?", click the box that says "**Open Microsoft Store**".
3. When the Microsoft Store application opens-up, you should be presented with a page that says "Windows Terminal".
   a. Click the blue box that says "Get" to install the Windows Terminal application.

Downloading & Installing Windows Terminal (Non-Microsoft Store)

Download, and install the **Windows Terminal** application from **GitHub**: Windows Terminal - GitHub

1. Click on the Windows Terminal - GitHub link above, and scroll down to the "**Latest**" Release of Windows Terminal, and click on the heading title.

    a. Example: The latest version of Windows Terminal at the time of creating this guide is "Windows Terminal v1.18.2681.0", so I would click on the heading title "Windows Terminal v1.18.2681.0".
2. Scroll-down to the "**Assets**" section, and click the "**Microsoft.WindowsTerminal**" version that matches your Operating System, to download **Windows Terminal**.
    a. Example: I'm using a 64-Bit Windows 10 Operating System, so I would click on "Microsoft.WindowsTerminal_1.18.2681.0_x64.zip".
3. The application is called "WindowsTerminal.exe", and it is in the "~\terminal-1.18.2681.0" folder.