PowerShell Basics

*Note: If you’d like to follow along (you should!) you’ll need access to PowerShell. If you’re on a Windows device, PowerShell is already included, simply launch it. Linux users can install PowerShell from*[*here*](https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-core-on-linux?view=powershell-6)*and MacOS users can install from*[*here*](https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-core-on-macos?view=powershell-6)*.*

**PowerShell is a command-line shell and scripting language that aims to help administrators and power-users rapidly automate tasks**.

To accomplish this, PowerShell adheres to design goals that enhance discoverability and consistency. To put it simply: PowerShell is easy to pick up and start using.

Cmdlets

PowerShell commands are referred to as [cmdlet’s](https://docs.microsoft.com/powershell/scripting/developer/cmdlet/cmdlet-overview?view=powershell-7). Cmdlets perform an action and typically return an object to the pipeline.

Try it yourself! Launch a PowerShell console window and type:

|  |  |
| --- | --- |
| Code example: | |
| 1 | Get-TimeZone |

This cmdlets returns an object containing data about your device’s currently configured time zone.

Verb-Noun

In PowerShell, cmdlets follow a verb-noun structure to help improve accessibility, and reduce command memorization.

This means that cmdlets follow a **Do**–**AThing** structure.

In the above example you were **Get**ing-the**TimeZone**.

Because we know about the Verb-Noun structure, and have previously seen the Get-TimeZone cmdlet, we can *infer* that there might also be a **Set-TimeZone** (which does indeed exist). Without any prior knowledge of these cmdlets, we can further infer that Get-TimeZone will be read-only while Set-TimeZone will take some type of action adjusting the devices time zone settings.

A list of [approved PowerShell Verbs](https://docs.microsoft.com/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands?view=powershell-7) is available, and all properly formed cmdlets adhere to this guideline.

Four commands needed to learn and use PowerShell – Get-Command, Get-Help, Get-Member, Find-Module

There are *thousands* of cmdlets. It is *impossible* for you to memorize them all. You don’t have to. You only have to memorize three. With these three cmdlets, you can find, explore, and leverage the others to perform a wide variety of tasks in your organization.

1. Get-Command

[Get-Command](https://docs.microsoft.com/en-us/powershell/scripting/learn/getting-information-about-commands?view=powershell-6) is a cmdlet that can be used to display available commands.

In fact, it can identify every cmdlet available to run on your system. Try it:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | Get-Command \*  # An asterisk (\*) in many languages acts as a wildcard. This syntax is saying:  get me ALL of the commands |

Using just a single wild card will display all available commands on the system. Often though, you’ll be after a specific objective. Lets try seeing if there is a cmdlet for dealing with system processes:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | Get-Command \*process\*  # the wild cards around process will find ANY command that contains the word  process |

This reveals all cmdlets that deal with processes. Unsurprisingly, we find the approved verb structure of:

* Get-Process
* Start-Process
* Stop-Process
* Wait-Process

1. Get-Help

Wait, what do all of those cmdlets do? Does Stop-Process do what I actually think it does?

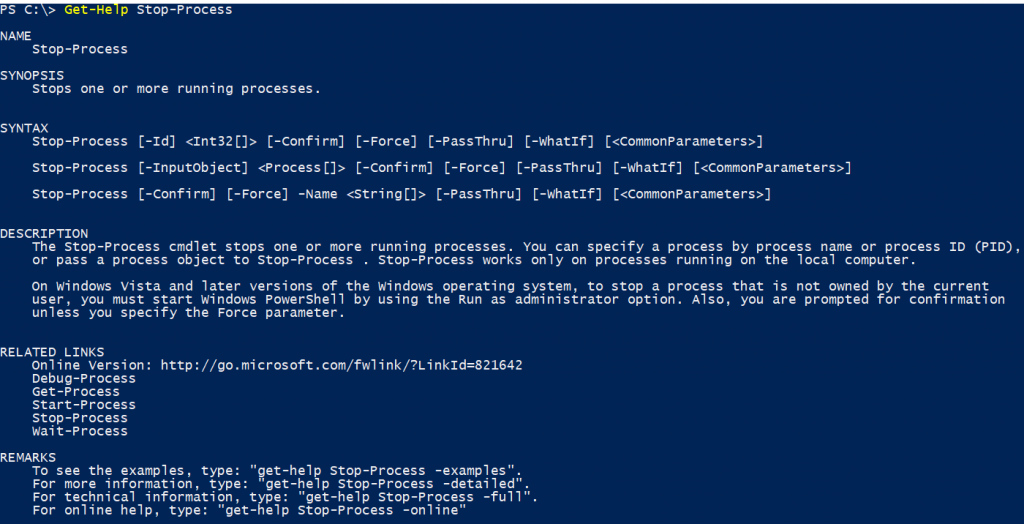
This is where [Get-Help](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/get-help?view=powershell-6) comes in. It provides information about what a cmdlet does, and how it can be used.

Before we leverage Get-Help, it’s a good idea to first run [Update-Help](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/update-help?view=powershell-6). This cmdlet will download up-to-date help information for all of your cmdlets. For Windows users, you will need to launch an [elevated PowerShell prompt](https://www.thewindowsclub.com/how-to-open-an-elevated-powershell-prompt-in-windows-10) as this command requires administrative privileges to update the help files.

*Note: Linux and MacOS users, Update-Help isn’t working as intended for you at this time so you will run slightly different commands in these short examples.*

With your help up-to-date you can now try Get-Help:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | # Windows Users:  Get-Help Stop-Process  #Linux/MacOs Users  Get-Help Stop-Process -Online |



As you can see we get a breakdown of exactly what the cmdlet does. Note that at the bottom of the return that that we can also call the same command with different parameters such as **-examples**

Lets try that now:

|  |  |
| --- | --- |
| Code example: | |
| 1 | Get-Help Stop-Process -Examples |
| Code example: | | |
| 1  2  3  4  5 | Example 1: Stop all instances of a process      PS C:\Stop-Process -Name "notepad"      This command stops all instances of the Notepad process on the computer.  Each instance of Notepad runs in its own process.      It uses the Name parameter to specify the processes, all of which have the  same name. If you were to use the Id parameter      to stop the same processes, you would have to list the process IDs of  each instance of Notepad. | |

Note that this provides detailed examples of how to run the Stop-Process cmdlet, and clearly outlines what you should expect as a result.

Armed with Get-Help you can teach yourself how to use any cmdlet on your system!

1. Get-Member

As previously mentioned, PowerShell deals with objects. [Get-Member](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-member?view=powershell-6) serves to help you identify what type of objects you are dealing with when using PowerShell. It can also help you identify what properties and methods are available to that object.

Try the following, don’t worry about the **|** for now, we’ll cover that in a later lesson:

|  |  |
| --- | --- |
| Code example: | |
| 1 | Get-Date | Get-Member |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | TypeName: System.DateTime  Date                 Property       datetime Date {get;}  Day                  Property       int Day {get;}  DayOfWeek            Property       System.DayOfWeek DayOfWeek {get;}  DayOfYear            Property       int DayOfYear {get;}  Hour                 Property       int Hour {get;}  Kind                 Property       System.DateTimeKind Kind {get;}  Millisecond          Property       int Millisecond {get;}  Minute               Property       int Minute {get;}  Month                Property       int Month {get;}  Second               Property       int Second {get;}  Ticks                Property       long Ticks {get;}  TimeOfDay            Property       timespan TimeOfDay {get;}  Year                 Property       int Year {get;} |

This indicates that Get-Date returns a **System.DateTime** object which contains many properties, such as Date, Day, Hour, Second, etc. This is a complex object return that contains a lot of data!

*Note: You may notice that if you simply run Get-Date that many of these properties are not returned to you. They are still there, but only a subset of properties is displayed to you from certain cmdlets. Try the following on your computer and see if it aligns with Get-Member:*

|  |  |
| --- | --- |
| Code example: | |
| 1 | Get-Date | Format-List |

This returns all the Get-Date Property members formatted in form of a list of key-value pairs.

|  |  |
| --- | --- |
| Code example: | |
| 1 | Get-Random | Get-Member |

This returns only a simple object of **TypeName: System.Int32**

Get-Random is just a quick cmdlet to get a random set of numbers. It has no other properties and only returns a System.Int32 number.

Understanding what type of object you are working with is very important in PowerShell. Get-Member is the cmdlet you’ll use to check if you aren’t sure!

When Get-Command doesn’t do the job

Sometimes the noun you are searching for might not be as apparent as you might like. For example, imagine you want to get some information about a file on your system. You might *infer* something like Get-File, and even try something like:

|  |  |
| --- | --- |
| Code example: | |
| 1 | Get-Command \*file\* |

What you’ll quickly find though, is that there is not a PowerShell command that gets a lot of file information that contains the word *file*. In these cases you’ll have to fall back to an internet search, and a **quick Google** reveals that Get-Item is the cmdlet that does what we need.

1. Find-Module

PowerShell can’t do *everything*. Or *can* it?

Did you know that native PowerShell can’t send Telegram messages? PowerShell supports modules though, and someone out there may have just written a module that enables PowerShell to send Telegram messages.

This is where [Find-Module](https://docs.microsoft.com/en-us/powershell/module/powershellget/find-module?view=powershell-6) comes in. It can search the PowerShell Gallery and find modules that add additional capabilities to PowerShell.

Try this out:

|  |  |
| --- | --- |
| Code example: | |
| 1 | Find-Module -Tag Telegram |

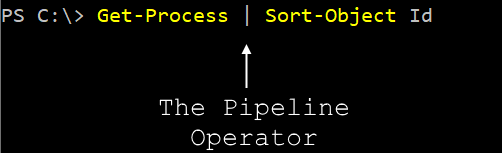
## The PowerShell Pipeline

Of all the concepts you’ll learn in this series, the PowerShell pipeline is arguably the most important.

**A**[**pipeline**](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_pipelines?view=powershell-6)**combines commands connected by pipeline operators**. Unlike many other shells, PowerShell is not limited to passing only strings along the pipeline. Complex objects can be passed from one command to the next making PowerShell very powerful!

### PowerShell Pipeline Operator

In PowerShell, the pipeline operator is:**|**



|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | # In this example, every command found will be "piped" to Get-Help  # This will run Get-Help against EVERY command found  Get-Command | Get-Help |

This operator sends the results of the preceding command to the next command. In the above example, Get-Help will return help for every command discovered by Get-Command. If 1,000 commands are found on your device, this command would find help for all 1,000 of them!

**Using the PowerShell Pipeline**

You can leverage the pipeline to perform powerful actions in your environment, as well as work with objects.

### Pipeline Passing

You can pass the results of one cmdlet to another to take various actions:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # This example would restart the BITS service!  Get-Service -Name BITS | Restart-Service |

Look at another example below. Without the use of the WhatIf parameter, what action do you think would result?

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | # try running this command on your machine to see the results.  # the WhatIf parameter will make Stop-Process take no actual action  Get-Process | Stop-Process -WhatIf |

If you were to remove the WhatIf parameter in the above example, PowerShell would immediately Get, and start stopping all the processes on your device. As you can imagine, that would lead to some pretty undesirable outcomes, likely resulting in your device crashing.

This is a good time to mention that PowerShell doesn’t contain a lot of hand holding. Its a tool that respects that you have an operational understanding of the commands that you run.

### PSItem (kinda like the “this” keyword in JS – points to object running the current function)

When working with the PowerShell pipeline there is one [automatic variable](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-6) that is of particular importance, **$PSItem**. **$PSItem** contains the current object in the pipeline.

If you are new to PowerShell, this may seem like an abstract concept, but all that PSItem is doing is “holding” the current object in the pipeline. Lets take a look at an example:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Get all processes. For each process (object) found, display that object  Get-Process | ForEach-Object {$PSItem} |

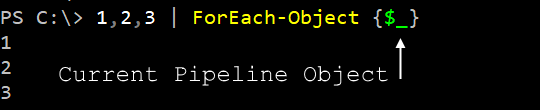
The above example gathers all processes, and then for each process discovered, outputs that object to the console window. This works because **$PSItem** contains the current object in the pipeline. Lets see this in action again:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # In this simple example, we will display the current object to the console  1,2,3 | ForEach-Object {$PSItem} |

Again, we see that $PSItem contains the current object. We loaded three simple numbers into the pipeline: 1,2,3. For each of these “objects” we returned the current object in the pipeline using $PSItem.

$PSItem is rarely fully spelled out in day-to-day PowerShell. Instead, utilize the shorthand **$\_** to denote the current pipeline object. **$\_** and **$PSItem** are one and the same. Try it yourself:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | # Spelling out $PSItem (not common)  1,2,3 | ForEach-Object {$PSItem}  # using $PSItem shorthand (common)  1,2,3 | ForEach-Object {$\_} |



### Processing data with the pipeline

As [previously mentioned](https://techthoughts.info/learn-and-use-powershell-with-just-three-commands/#Three_commands_needed_to_learn_and_use_PowerShell), many cmdlets will return a predefined subset of information that you are most likely interested in.

Many times though, you’ll be after : all OR very specific object data that the cmdlet returns. Using the pipeline you can format, sort, and select the properties you want to work with.

#### **Format-List & Format-Table**

[Format-List](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/format-list?view=powershell-6) and [Format-Table](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/format-table?view=powershell-6) alter the output view of an objects properties. Try the following example:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | # Note the way things look using Format-Table  Get-Process | Format-Table  # Now see how that differs if you pipe instead to Format-List  Get-Process | Format-List |

Format-List can be used to see more properties of an object, rather than just a predefined subset of properties.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | # Note the return from just Get-Date  Get-Date  # Now see how the whole object is returned if you pipe to Format-List  Get-Date | Format-List |

With some cmdlets, especially those with a lot of data in the object, even Format-List will only return a portion of the object. Include an asterisk to ensure all data from the object is displayed:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6 | # basic information about the notepad process  Get-Process notepad  # more information about the notepad process  Get-Process notepad | Format-List  # all available information about the notepad process  Get-Process notepad | Format-List \* |

#### **Select-Object**

On the flip-side you may find yourself wanting only a very specific set of properties from an object. [**Select-Object**](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/select-object?view=powershell-6) can help you achieve this.

For instance, **Get-Process** returns detailed information about the running processes on a device. But what if you are only interested in a few of those key properties? Try the following:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Using Select-Object you can retrieve only the properties you are after  Get-Process | Select-Object Name,Id,CPU,Responding |

#### **Sort-Object**

You can take even further control over output using [Sort-Object](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/sort-object?view=powershell-6). Lets extend the previous example by sorting the output by the busiest processes. Try the following:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6 | # This example pipes all processes to Sort-Object which sorts by CPU use  # Use to quickly identify your busiest processes. Very handy!  Get-Process | Sort-Object CPU    # You can combine multiple pipelines together as well  Get-Process | Select-Object Name,Id,CPU,Responding | Sort-Object CPU |

#### **Where-Object**

[Where-Object](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object?view=powershell-6) is one of the most powerful tools in your PowerShell toolbox. It operates a bit like Select-Object but has the added benefit of only selecting objects based on values you specify. This enables you to focus in and quickly identify specific results.

**Where-Object** will leverage the PSItem previously discussed to evaluate each object for the criteria specified. Lets start with an example where we only want to retrieve processes that are actively using the CPU:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Get Processes, return only those where current object $\_ is greater than 15  Get-Process | Where-Object {$\_.CPU -gt 15} |

Remember that **$\_** is the current object, which represents in this case, one full process object. However, we are only after objects where the CPU is greater than 15, so we are evaluating only the CPU property of the process object by specifying **$\_.CPU**

Lets take a look at another useful example of **Where-Object**:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | # Get file information for all files found in the $HOME directory  # Return information for only files that are greater than 5MB in size  Get-ChildItem $HOME -Recurse | Where-Object {$\_.Length -gt 5MB} |

There may seem like there is a lot going on here, so lets break it down:

Get-ChildItem $HOME -Recurse

This gets all files in your home directory

Where-Object {$\_.Length -gt 5MB}

Using Where-Object we can evaluate only the Length property. The Length property is the size of the file. We only return information for files greater than 5MB

Using the power of the pipeline, you can even simply count how many large files you have. Try this example:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Count the number of large file in your $HOME directory  Get-ChildItem $HOME -Recurse | Where-Object {$\_.Length -gt 50MB} | Measure-Object |

## PowerShell Comparison Operators

Note that we used a comparison operator in the above examples (greater than). Comparison operators are a common component in all programming languages. Here are several common PowerShell comparison operators:

-eq equals

-ne not equals

-gt greater than

-ge greater than or equal to

-lt less than

-le less than or equal to

-like wild card pattern match

-notlike wild card pattern match

## PowerShell Parameters

So far in the series we’ve had a few examples where we’ve used parameters. **Parameters in PowerShell are simply attributes of a cmdlet where you can specify various options or arguments.**

Each cmdlet differs in the number and type of parameters they have. Some cmdlets have no parameters, while others have mandatory parameters. You can always use Get-Help to determine what parameters are available to a cmdlet.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5 | # This standard cmdlet returns time zone information of the device  Get-TimeZone  # The ListAvailable parameter completely changes the behavior of this cmdlet  # Now, it lists the AVAILABLE timezones the device could be set to  Get-TimeZone -ListAvailable |

## Wrapping up the PowerShell Pipeline

The pipeline is what makes PowerShell so powerful. Many shells are limited to piping only strings. PowerShell is capable of passing entire objects of data between cmdlets. This ability to easily pass complex objects opens up a lot of possibilities. It enables you to perform various actions, or evaluate specific criteria in your environment quickly, easily, and accurately.

**NOTE** : You can’t just arbitrarily pipe objects around. For example **Get-Service | Stop-Process** wouldn’t work because Stop-Process is expecting a process object, not a service object.

Also keep in mind that not all cmdlets support pipeline input. You can always leverage Get-Help to determine what cmdlets support pipeline input.

The PowerShell pipeline is a critical component to master on your PowerShell journey. Feel free to comment below if you have questions on this topic!

### PowerShell ISE no more

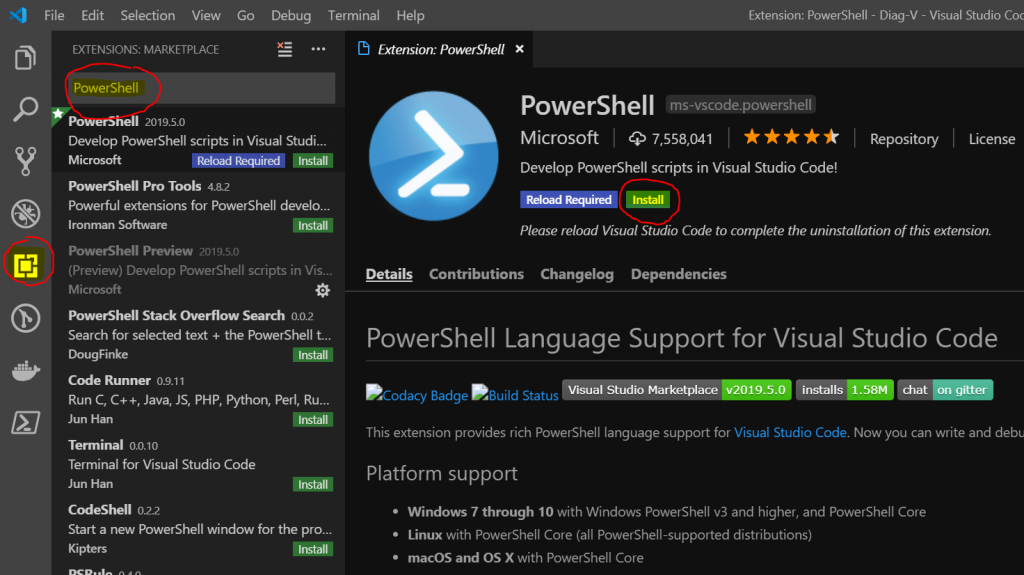
In the past the de-facto choice for writing PowerShell code was the [PowerShell Integrated Scripting Environment (ISE)](https://docs.microsoft.com/en-us/powershell/scripting/components/ise/introducing-the-windows-powershell-ise?view=powershell-6). It is a native component that shipped with Windows PowerShell. With the release of PowerShell 6 and PowerShell becoming cross-platform, the PowerShell ISE is no longer supported. You can still use it (Windows only) to write PowerShell, but the future of PowerShell development is with Visual Studio Code (VSCode).

### Components needed for PowerShell Development

#### **1 – VS Code - PowerShell Extension**

VSCode does not natively support PowerShell. To effectively write, debug, see syntax highlighting, use code snippets, and have intellisense for cmdlets, you’ll need to install the [PowerShell extension](https://marketplace.visualstudio.com/items?itemName=ms-vscode.PowerShell).

1. Select the extensions tab on the left side of VSCode
2. Search for PowerShell
3. Install the PowerShell Extension

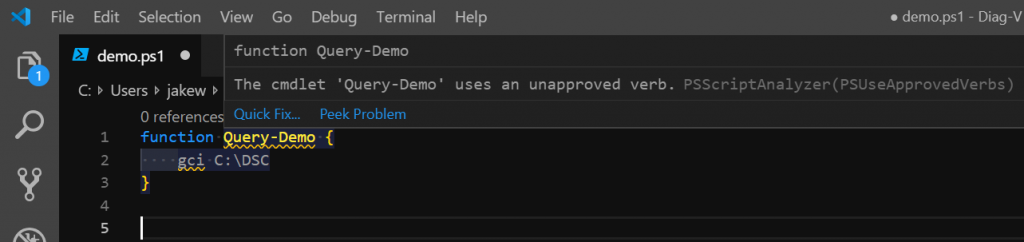


With VSCode installed you can also utilize it from the terminal by typing **code**. Here is how you can install the PowerShell extension via a terminal:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | code --list-extensions  code --install-extension ms-vscode.PowerShell |

#### **2 – PowerShell Script Analyzer**

The PowerShell Script Analyzer is a PowerShell module that acts as a static code checker. It will check your PowerShell code by running a set of rules to ensure you are adhering to best practices, and correct syntax. When you have both VSCode and the PowerShell Script Analyzer installed, VSCode will continuously check your code as you write it. Any errors will be displayed by visual queues in the editor.



In the above example the name of the function violates the PSUseApprovedVerbs rule. The example function is not using an [~~approved PowerShell verb~~](https://docs.microsoft.com/en-us/powershell/developer/cmdlet/approved-verbs-for-windows-powershell-commands).

**gci** is violating the use of aliases rule. **gci** is an alias for **Get-ChildItem**. Aliases can introduce confusion into your code for others, so they should be avoided. Note that PSScriptAnalyzer is working with VSCode to highlight these two rule violations.

#### **3 – Configure VSCode for PowerShell development**

While the PowerShell extension does add a lot of capability, there are a few additional VSCode settings you’ll want to configure to enhance your PowerShell development experience. You can access VSCode settings with the keyboard shortcut: **Ctrl +**,

VSCode has a graphical settings menu, but I’d encourage you to try using the JSON based settings file. The graphical menu is a recent addition to VSCode and the majority of online examples will include JSON examples. You can switch to the JSON version by clicking the Open Settings (JSON) button.

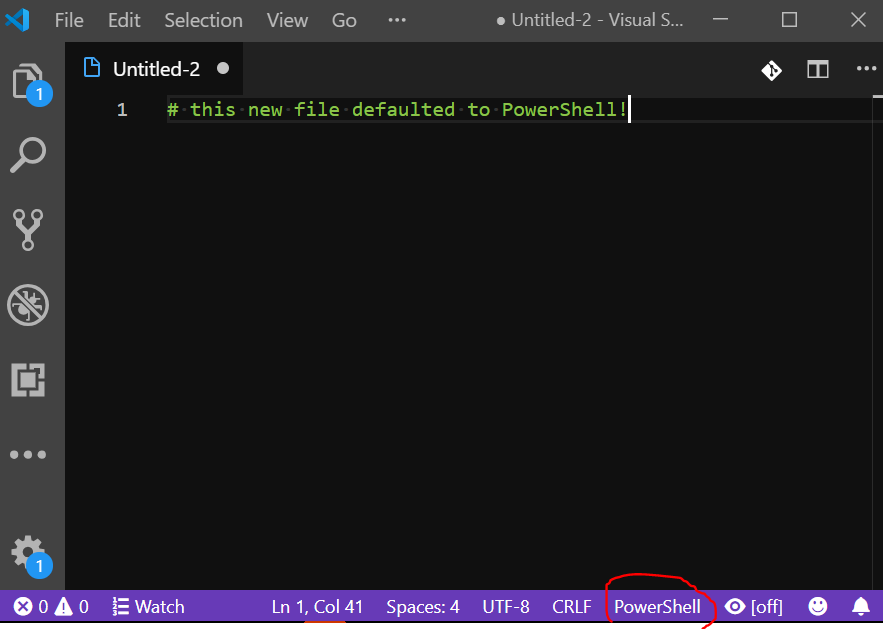


With the JSON settings file accessed, simply copy and paste the below into the file. These settings will provide a more streamlined PowerShell experience in VSCode. You’ll have tab completion, new files will open as PowerShell, and automatic script analysis will occur. Note that the location of the PowerShell path will vary based on if you are coding for PowerShell 5.1 vs 6 on Windows. It will also be different if you are developing on Linux. Review the sample below and adjust the appropriate lines to specify the version that you will be using for your terminal.

|  |  |
| --- | --- |
|  | { |
|  | // editor |
|  | "editor.quickSuggestionsDelay": 1, |
|  | "editor.tabCompletion": "on", |
|  | "files.defaultLanguage": "powershell", |
|  |  |
|  | // default shell |
|  | // Windows |
|  | // PowerShell 6 |
|  | "terminal.integrated.shell.windows": "C:\\Program Files\\PowerShell\\6\\pwsh.exe", |
|  | "powershell.powerShellExePath": "C:\\Program Files\\PowerShell\\6\\pwsh.exe", |
|  | //PowerShell 5.1 and below |
|  | // "terminal.integrated.shell.windows": "C:\\WINDOWS\\System32\\WindowsPowerShell\\v1.0\\powershell.exe", |
|  | // "powershell.powerShellExePath": "C:\\WINDOWS\\System32\\WindowsPowerShell\\v1.0\\powershell.exe", |
|  | // Linux |
|  | // Ubuntu |
|  | // "terminal.integrated.shell.linux": "/snap/powershell/36/opt/powershell/pwsh", |
|  | // "powershell.powerShellExePath": "/snap/powershell/36/opt/powershell/pwsh", |
|  |  |
|  | // powershell settings changes |
|  | "powershell.codeFormatting.preset":"Stroustrup", |
|  | "powershell.startAutomatically": true, |
|  | "powershell.scriptAnalysis.enable": true, |
|  | "powershell.integratedConsole.showOnStartup": false, |
|  | "powershell.integratedConsole.focusConsoleOnExecute": true, |

}

Save your settings.json file. You are now set up for efficient PowerShell Development using VSCode. Test things out by launching a new file either via File -> New File, or using **Ctrl + N**. With your settings changes, it should default to a PowerShell file. You can confirm this by looking in the lower right of VSCode:



## Declaring PowerShell Variables

As discussed in previous lessons PowerShell deals with objects. [PowerShell variables](https://docs.microsoft.com/en-us/powershell/scripting/learn/using-variables-to-store-objects?view=powershell-6) are just a named object that stores the object data you specify. Create named objects using the **$** character followed by the variable name. A quick example demonstrates how simple this is:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | Get-Process  $processes = Get-Process  $processes |

**Get-Process** of course gets all the processes on the system as a System.Diagnostics.Process object type. In the second line, we declare the **$processes** variable and store the object returned from **Get-Process**. We can confirm this by running the third line which returns the exact same object data.

Now that we have that information stored in **$processes** it is no longer necessary to run **Get-Process** each time we want different information. Consider the following example:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10 | #----------------------------------------  # not using variable  Get-Process | Where-Object {$\_.CPU -gt 5000} #find processes keeping the CPU busy  Get-Process | Sort-Object WorkingSet64 -Descending #sort processes by memory usage  #----------------------------------------  # using variable  $processes = Get-Process  $processes | Where-Object {$\_.CPU -gt 5000} #find processes keeping the CPU busy  $processes | Sort-Object WorkingSet64 -Descending #sort processes by memory usage  #---------------------------------------- |

In the first example notice **Get-Process** had to be run twice to get results. The second example is more efficient as it stores the first run in a variable. Afterwards, it can be manipulated endlessly to obtain additional information.

## Understanding PowerShell Variables

PowerShell in not a strong typed language. This means that PowerShell variables can be declared without specifying the data type of the variable. Have a look:

|  |  |
| --- | --- |
| Code example: | |
| 1 | $myNewVariable |

You’ve just declared a new PowerShell variable, **$myNewVariable**. What type of variable is this? Is it a string? Is it an integer? A boolean? In PowerShell, this new variable has the potential to be anything. If you were to load it with a string, it would become a string data type. Assigning it an integer would cause it to become an integer. Loading it with **Get-Process** would cause it to become a System.Diagnostics.Process object type.

### Variable Data types

Lets perform a few basic operations to highlight how variables can become different types of data.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9 | #----------------------------------------  $total = 2 + 2  $total  $total | Get-Member  #----------------------------------------  $total = '2 + 2'  $total  $total | Get-Member  #---------------------------------------- |

If you run lines two and three in this example, you’ll find that **$total** contains the result of 4. Because we provided an integer based operation in line two, **$total** becomes an integer type. You can confirm this with **Get-Member** in line four.

In line six, we wrap the operation in quotes, forming a string. This changes the data type and **$total** is now of type string. Note that **$total** now instead contains the result of 2+2. This is because **$total** is a literal string, and no math operation was performed. You can confirm this with **Get-Member** in line eight.

#### **Working with different variable types**

Because PowerShell variables can be of any data type, it is important that you understand the data type of the variable you are working with. Consider the two operations below:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11 | #----------------------------------------  $num1 = 2  $num2 = 2  $total = $num1 + $num2  $total  #----------------------------------------  $num1 = '2'  $num2 = '2'  $total = $num1 + $num2  $total  #---------------------------------------- |

Initially **$num1** and **$num2** are loaded with integer numbers. When a math operation is performed on line four, this causes **$total** to also become of type integer with a result of 4.

In the second example **$num1** and **$num2** are loaded with strings. When the same operation is performed on line 9, this causes **$total** to become a string as well, with a result of 22! This is why it is crucial to understand what variable type you are working with, as it can change the outcome of your results!

#### **Strongly typing variables**

One method of dealing with this is to strongly type your variables. Instead of letting PowerShell decide what data type your variable will be, you can specify it. Look a the following example:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | [int]$num1 = '2'  [int]$num2 = '2'  $total = $num1 + $num2 |

In our previous examples when we wrapped numbers in quotes, it loaded them into literal string variables. Note in this example that the variable is front loaded with **[int]**. This is telling PowerShell that you want this variable to be of type int. Despite that the 2’s are wrapped in quotes, PowerShell will load them as integer numbers and **$total**will be 4.

#### **Converting variable types**

You can convert variables to other data types. In the example directly above **$total** is of type integer. There may be a need to have it converted to type string though. Try the following:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | #take the $total variable from the previous example, and convert it to a string  $iAmAStringNow = $total.ToString() |

In this example we declare a new variable, **$iAmAStringNow** and load in **$total.ToString()**. **$total** of course, is an integer, but what is the **.ToString()** about? This is a method that is being used to convert **$total** to a string. How did I know that I could use the ToString method on **$total**? I used Get-Member to verify what methods were available for that variable – [covered in our first lesson in the series](https://techthoughts.info/learn-and-use-powershell-with-just-three-commands/). Look at all the methods available for **$total**:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | $total | Get-Member       TypeName: System.Int32  Name        MemberType Definition  ----        ---------- ----------  CompareTo   Method     int CompareTo(System.Object value), int CompareTo(int value), int IComparable.CompareTo(…  Equals      Method     bool Equals(System.Object obj), bool Equals(int obj), bool IEquatable[int].Equals(int ot…  GetHashCode Method     int GetHashCode()  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.IFormatProvide…  ToType      Method     System.Object IConvertible.ToType(type conversionType, System.IFormatProvider provider)  ToUInt16    Method     ushort IConvertible.ToUInt16(System.IFormatProvider provider)  ToUInt32    Method     uint IConvertible.ToUInt32(System.IFormatProvider provider)  ToUInt64    Method     ulong IConvertible.ToUInt64(System.IFormatProvider provider)  TryFormat   Method     bool TryFormat(System.Span[char] destination, [ref] int charsWritten, System.ReadOnlySpa… |

### Working with quotes and PowerShell variables

As you’ve seen previously, quotes can greatly impact the functionality of a command. An operation will perform very differently with a string vs an integer. It is possible to escape operations using different types of quotes. Lets look at an example:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | $literal = 'Two plus one equals: $(1 + 2)'  $literal  $escaped = "Two plus one equals: $(1 + 2)"  $escaped |

Single quotes will always result in a literal string. This means that **$literal** contains exactly what was loaded into it. It is a literal string. Double quotes will still create a string, but are capable of working with dynamic components within that string. Note that by escaping the operation 1 + 2 with the syntax **$(1 + 2)** that we are telling PowerShell we want the variable result of one plus two. Thus, **$escaped** results in: Two plus one equals: 3

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | Write-Host '$escaped'  Write-Host "$escaped" |

Again, we have an example with single (literal) vs doulbe (dynamic) quotes. Remember, both of these still result in a string, but note the differences by trying these out in your own console!

### Reserved (Constant) PowerShell variables

There are few PowerShell variable names that are constant and therefore reserved. This means you won’t be able to use these variable names. You can familiarize yourself with this short list by running:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Get a list of variable names in use  Get-Variable |

Attempts to load into a constant variable will result in an error similar to the one below:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7 | $HOME = 'c:\test'  Cannot overwrite variable HOME because it is read-only or constant.  At line:1 char:1  + $HOME = 'c:\test'  + ~~~~~~~~~~~~~~~~~  + CategoryInfo          : WriteError: (HOME:String) [], SessionStateUnauthorizedAccessException  + FullyQualifiedErrorId : VariableNotWritable |

#### **PowerShell Environmental variables**

In addition to a few constant variables, PowerShell has variables reserved for some environment data. You can see a list of these with the following command:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Get a list of environment variables  Get-ChildItem env: |

These contain information about the environment PowerShell is running in. Examples include computer name ( **$env:COMPUTERNAME** ), and the current user you are running as ( **$env:USERNAME** ).

## Putting it all together with an example

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5 | $path = Read-Host -Prompt 'Please enter the file path you wish to scan for large files...'  $rawFileData = Get-ChildItem -Path $path -Recurse  $largeFiles = $rawFileData | Where-Object {$\_.Length -gt 100MB}  $largeFilesCount = $largeFiles | Measure-Object | Select-Object -ExpandProperty Count  Write-Host "You have $largeFilesCount large file(s) in $path" |

Based on everything we’ve learned up to this point we can now explore this simple example. **Read-Host** prompts the user with a literal string wrapped in single quotes. The **$path** variable is loaded with the entry the user provides. **Get-ChildItem** then finds all the files in that provided path, and loads the results into **$rawFileData**. We then find the files over 100MB with **Where-Object** and load those results into **$largeFiles**. Using **Measure-Object** and **Select-Object** in the pipeline, we determine the total count of large file over 100MB. We then display back to the user using **Write-Host** the total number of large files. Note that on the last line we use double quotes because we have a dynamic variable we want to display.

## Common PowerShell Variable Types

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | [string]    String of Unicode characters    [int]       32-bit integer  [long]      64-bit integer  [decimal]   A 128-bit decimal value  [single]    32-bit floating point number  [double]    64-bit floating point number    [bool]      Boolean True/False    [DateTime]  Date and Time    [array]     An array of values  [hashtable] Hashtable object |

## Controlling PowerShell Logic Flow

PowerShell is no different than other computer languages in that more complex capabilities require the addition of logic. If you are familiar with logic controls in another programming language, you’ll find no surprises. We’ll simply cover some of the specific PowerShell syntax.

If programming logic is a relatively new concept I’d like to encourage you not to shy away from this component of PowerShell. The addition of PowerShell logic is simple, and relatively straightforward. Take a look at this quick code example to see how easy these PowerShell logic controls are to implement:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10 | # do something based on a condition  if ('a condition is met') {      # do something / take an action  }  elseif ('a different condition is met') {      # do something different  }  else {      # no conditions met, do something else  } |

### Conditional

PowerShell utilizes if, elseif, and else for conditional statements. Conditional statements perform conditional tests based on criteria you specify. The results of those tests will result in various actions that you create.

Many times in PowerShell you will need to confirm the presence of a directory or file path. Lets leverage the **Test-Path** cmdlet combined with some PowerShell logic to take various actions based on the discovered results.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9 | $path = 'C:\Windows' #windows  # $path = /home #linux  $evalPath = Test-Path $path  if ($evalPath -eq $true) {      Write-Host "$path VERIFIED"  }  elseif ($evalPath -eq $false) {      Write-Host "$path NOT VERIFIED"  } |

In this basic example we [load up the variable](https://techthoughts.info/working-with-powershell-variables/) **$path** with a directory location. We then evaluate that path with **Test-Path** and store the results in **$evalPath**. **Test-Path** returns only a boolean value (**$true**/**$false**) so we use conditional logic to determine if that variable contains a true or false value. Based on the results, we take a different action.

Because **Test-Path** only returns two possible values the above example can also be written with an else to achieve the same results:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9 | $path = 'C:\Test' #windows  # $path = /home #linux  $evalPath = Test-Path $path  if ($evalPath -eq $true) {      Write-Host "$path VERIFIED"  }  else {      Write-Host "$path NOT VERIFIED"  } |

#### **Comparison Operators**

Notice that conditional evaluations make heavy use of comparison operators. Comparison operators allow you to specify certain conditions. Here are a few common PowerShell comparison operators:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | -eq       equals  -ne       not equals  -gt       greater than  -ge       greater than or equal to  -lt       less than  -le       less than or equal to  -like     wild card pattern match  -notlike  wild card pattern match |

Visit the PowerShell documentation on [comparison operators](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators?view=powershell-6) for a full list of all available operators.

### Switch

A [switch statement](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch?view=powershell-6) is much like using an if. The difference is that it is capable of evaluating multiple conditions. You could achieve the same thing utilizing multiple if statements. Switch statements are just a bit cleaner to look at, and simpler to write. Try running the following example on your computer. Then add additional switch statements to account for numbers up to 10.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9 | [int]$aValue = Read-Host 'Enter a number'  switch ($aValue) {      1 {          Write-Host 'You entered the number ONE'    }      2 {          Write-Host 'You entered the number TWO'    }      3 {          Write-Host 'You entered the number THREE'    }      4 {          Write-Host 'You entered the number FOUR'    }      5 {          Write-Host 'You entered the number FIVE'    }      Default {        Write-Host "Sorry, I don't know what to do with $aValue"    }  } |

### Loops

Loops are one of the base logical structures of programming. They enable your code to perform repetitive tasks quickly, or evaluate a lot of data rapidly. If you have implemented loops in another language, you’ll find no surprises in PowerShell.

If you aren’t familiar with implementing loops, I provide several practical examples below. Again, don’t shy away from these logical control mechanisms. They aren’t “just for developers”. These PowerShell logic controls are easy to implement and will take your PowerShell capabilities to the next level!

#### **For Loop**

A basic [For Loop](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_for?view=powershell-6) can be used to perform an action a specific number of times. Try out the rainbow example on your computer:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | # rainbow example  for ($i = 0; $i -le 15; $i++) {      Write-Host $i -ForegroundColor $i  } |

The rainbow example started **$i** at 0. Each loop **$i** was incremented until it reached 15. You can also go the opposite direction as in this example:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6 | $aString = 'Jean-Luc Picard'  $reverseString = ''  for ($i = $aString.Length; $i -ge 0; $i--) {      $reverseString += $aString[$i]  }  $reverseString |

Lets break that example down. Variable **$aString** is created and a string loaded into it. We also declare an empty **$reverseString** which will store the reversed string. We then start looping based on the total length of **$aString** (in this example 15). In each of the loops we take the one character in that loop **$aString[$i]** and load it into **$reverseString**. Because we start at the end of **$aString**, the first letter that gets loaded into **$reverseString** is the last letter. The loop continues until each character is loaded in reverse order.

#### **Foreach Loop**

In my experience, [Foreach Loops](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_foreach?view=powershell-6) tend to be used more than For loops in PowerShell. Foreach enables you to loop through all the items in a collection. A collection could be many things. It could be a collection of strings, a collection of numbers, or even a collection of complex objects. Lets dive into an example.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7 | $path = 'C:\Test'  [int]$totalSize = 0  $fileInfo = Get-ChildItem $path -Recurse  foreach ($file in $fileInfo) {      $totalSize += $file.Length  }  Write-Host "The total size of file in $path is $($totalSize /1MB) MB." |

**Get-ChildItem** is recursively getting all files in the specified path. Using foreach we can loop through each of those objects in the **$fileInfo** collection. In each iteration of the loop **$file.Length** (size) is evaluated. **$file** represents the current item in the collection that is being handled. So, on the first loop, **$file** represents the first item in **$fileInfo**. The loop will continue until every item in the collections has gone through.

#### **While Loop**

There are two types of while loops that you will encounter, a [do while,](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_do?view=powershell-6) and a standard [while loop](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_while?view=powershell-6). The primary difference between this is that a do while will always perform the loop at least once. A while loop may or may not enter the loop depending on the criteria set. The following examples illustrate the difference:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | #-------------------------------------------------  # do while loop  $pathVerified = $false  do {      # in a do while, the user will always be prompted at least once      $path = Read-Host 'Please enter a file path to evaluate'      if (Test-Path $path) {          $pathVerified = $true      }  } while ($pathVerified -eq $false)  # the loop will continue until the path is verified  #-------------------------------------------------  # while loop  $pathVerified = $true  while ($pathVerified -eq $false) {      # in a while loop, you might never enter the loop      $path = Read-Host 'Please enter a file path to evaluate'      if (Test-Path $path) {          $pathVerified = $true      }  }  # this loop was never entered because $pathVerified is $true  #------------------------------------------------- |

Keep in mind that while loops will continue until the condition is met. Try running the do while loop example above. Provide paths that both exist and don’t exist on your compu

ter.

### Where-Object

[Where-Object](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object?view=powershell-6) is one of the most powerful cmdlets at your disposal. It behaves much like an if statement in that it uses conditional statements to evaluate criteria you specify. It also behaves like a loop in that it will evaluate every object in your pipeline. You could accomplish the same results using both if statements and loops. The benefits of Where-Object is that it allows you stay within the pipeline, and is much simpler to implement.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11 | # Get processes using more than 50MB of memory  $largeProcesses = Get-Process | Where-Object { $\_.WorkingSet64 -gt 50MB }    # do the same thing using if statements and loop  $largeProcesses = @()  $processes = Get-Process  foreach ($process in $processes) {      if ($process.WorkingSet64 -gt 50MB) {          $largeProcesses += $process      }  } |

As you can see **Where-Object** greatly simplifies evaluating each object returned by **Get-Process**. **Where-Object** will often be your primarily filtering tool for selecting specific objects returned by various cmdlets.

### ForEach-Object

[ForEach-Object](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/foreach-object?view=powershell-6) is the same as a Foreach loop except that you can use it inside the pipeline.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | $path = 'C:\Test'  $folderCount = 0  Get-ChildItem $path | ForEach-Object -Process { if ($\_.PSIsContainer) { $folderCount++ } }  $folderCount |

Each file object found by **Get-ChildItem** is being evaluated in the **ForEach-Object** to determine if it is a directory (PSIsContainer) or a file. If it is a directory, **$folderCount**is incremented.

## Putting it all together with an example

With all that covered, we can now combine various elements together to produce powerful and insightful results.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | # declare a few variables for counting  [int]$fileCount = 0  [int]$folderCount = 0  [int]$totalSize = 0    # declare our path we want to evaluate  $path = 'C:\Test'    # get the file information  $rawFileInfo = Get-ChildItem $path -Recurse    # loop through that file information  foreach ($item in $rawFileInfo) {      if ($item.PSIsContainer) {          # this is a folder/directory          $folderCount++      }      else {          # this is a file, because it is not a PSIsContainer          $fileCount++          $totalSize += $item.Length      }  }    # generate output  Write-Host "Breakdown of $path"  Write-Host "Total Directories: $folderCount"  Write-Host "Total Files: $fileCount"  Write-Host "Total Size of files: $($totalSize / 1MB) MB" |

Using basic cmdlets, a foreach loop, and a few conditional statements, we can provide rapid insight into a given file path!

Now try removing the **-Recurse** switch. How does this change the output? Why?

With everything covered so far you may have recognized other ways to code this example. A switch statement could have replaced the if/else. We could have also filtered using ForEach-Object. There are many ways to solve a problem now that you can leverage PowerShell logic!

## PowerShell Input

Often times your PowerShell code will need to evaluate some type of input. That input can be statically provided by you ahead of time. There are often use cases though where retrieving a dynamic input is desired.

### PowerShell Input From cmdlets

You can of course use the output of cmdlets for the input of your PowerShell functionality. Depending on what a cmdlet returns, you could perform a variety of different functions based on that generated input.

* run cmdlets (Get-Process)
  + cmdlet generates output
    - use output as your input
      * perform functionality based on dynamic input

#### **Dynamic PowerShell input from the web**

Lets explore an example where we use PowerShell to pull in dynamic results from a website. If you come from a more Linux focused background this is a bit like using **curl**.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6 | #retrieve dynamic content from a website  $webResults = Invoke-WebRequest -Uri '<https://reddit.com/r/powershell.json>'  $rawJSON = $webResults.Content  $objData = $rawJSON | ConvertFrom-Json  $posts = $objData.data.children.data  $posts | Select-Object Title,Score | Sort-Object Score -Descending |

Here we are using [**Invoke-WebRequest**](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest?view=powershell-6) to retrieve web content from [/r/PowerShell](https://www.reddit.com/r/PowerShell/). (A great PowerShell resource by the way!) HTML isn’t particularly useful to us as we want to work with objects in PowerShell. So note that I have visited the JSON version of this page. As a result, the output content from this cmdlet is in JSON. Because JSON is structured data, we can take control and convert it to native PowerShell objects using [**ConvertFrom-JSON**](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/convertfrom-json?view=powershell-6).

Note: There are many of these convert cmdlets. This enables you to take in data from a variety of sources and continue to work with PowerShell objects!

Once in native object format, it’s a simple matter to explore the available sub-properties. We can then drill down into them and find post information. Once we have post data, we can select and sort for the inputs we are after. In this example, post title and score (upvotes) are returned.

Run the example above on your machine to see the results. Try exploring other available sub-properties in **$posts**.

#### **Leveraging cmdlet output as input**

How would you go about using the output from that example as input? That’s entirely up to you! You could write PowerShell code to:

* Email yourself the top 3 PowerShell posts of the day
* Send yourself a text message with the top post of the day
* Display the top 10 posts of the day when you login

You’d be using PowerShell to make yourself smarter on PowerShell! We’ll be covering more how to create ps1 scripts later in the series so stay tuned!

### Read-Host Input

[**Read-Host**](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/read-host?view=powershell-6) is another way to get dynamic input from yourself or your users. Leveraging the previous example again, we could only return the desired number of posts:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7 | [int]$numPosts = Read-Host -Prompt "Enter the number of posts to read"  #retrieve dynamic content from a website  $webResults = Invoke-WebRequest -Uri '<https://reddit.com/r/powershell.json>'  $rawJSON = $webResults.Content  $objData = $rawJSON | ConvertFrom-Json  $posts = $objData.data.children.data  $posts | Select-Object Title,url | Sort-Object Score -Descending | Select-Object -First $numPosts |

The -First parameter of **Select-Object** allows us to control how many objects we return to the console. Because we are sorting by number of upvotes, we will return the number of top posts specified. Run this on your computer specifying different numbers of posts.

Here is fun example where we can retrieve web data combined with **Read-Host** to display the desired number of cat facts:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | [int]$numFacts = Read-Host -Prompt "Enter the number of cat facts you would like"  $webResults = Invoke-RestMethod -Uri "<https://catfact.ninja/facts?limit=>$numFacts&amp;max\_length=140"  $webResults.data |

Note that catfact.ninja is an ***API***and not a traditional webpage. As a result, [***Invoke-RestMethod***](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-restmethod?view=powershell-6) is the appropriate cmdlet, and not **Invoke-WebRequest**.

### Get-Content

Another source of dynamic input that you often need comes from files on a device. Operationally, you’ll often be looking at log files. Lets take a look at an example log file:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5 | [INFO][Date]Normal Operations  [INFO][Date]Normal Operations  [INFO][Date]Normal Operations  [ERROR][Date]192.168.1.5 unable to access  [INFO][Date]Normal Operations |

[**Get-Content**](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-content?view=powershell-6) can retrieve the contents of a file for you to use as input.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | $logContent = Get-Content C:\Test\SampleLog.log    # get just entries that have an IP address using regex  $regex = "\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b"  $logContent | Select-String -Pattern $regex -AllMatches    # get just entries that have an IP address using Where-Object  $logContent | Where-Object {$\_ -like "\*.\*.\*.\*"} |

The contents of the SampleLog are stored in **$logContent**. Now that you have that input data you can start doing evaluations. It may be critical that PowerShell take some form of action if an IP address is discovered in the log. You can parse the data using [regular expression](https://en.wikipedia.org/wiki/Regular_expression) (regex) or by using Where-Object. If an IP is discovered, you could ticket, pop a notification, or perform some other action.

You aren’t limited to just text or standard log files. Get-Content can pull in input data from a wide array of file types. Here is an example where you can get input from a csv file. Remember that there are many conversion cmdlets so you can get csv data into normal PowerShell object format easily!

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | $rawCSVInput = Get-Content C:\Test\SampleCSVFile.csv  $objData = $rawCSVInput | ConvertFrom-Csv |

## PowerShell Output

If you want to display results back to the console or generate a results file, you’ll need to generate some type of PowerShell output. There are a few ways to accomplish this, and one you should avoid!

### Write-Host

Write-Host can output customized strings to the console. You can customize the output by controlling the start of a new line, or -NoNewline. A foreground and background color can also be set. Try the following examples:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | # Output simple string to console  Write-Host 'Text output to console'    # Customize output to the console with colors:  Write-Host "Warning" -ForegroundColor Yellow  Write-Host "ERROR" -ForegroundColor Red  Write-Host "Works Great" -ForegroundColor Green  Write-Host "CRITICAL ERROR" -BackgroundColor Red -ForegroundColor White |

Write-Host is only capable of string output, so you need to make sure you are giving it a string. If not, Write-Host will do its best to accommodate some type of output, but the results may not be what you expect.

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | $hostInfo = Get-Host  Write-Host $hostInfo |

In the above example, you’ll get the output of: **System.Management.Automation.Internal.Host.InternalHost**

This is because **$hostInfo** is a PowerShell object, not a string. We can drill down farther into this object until we have a string that is use-able by Write-Host:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | $hostInfo = Get-Host  Write-Host $hostInfo  Write-Host $hostInfo.Version |

Write-Host is easy to implement, and it’s color capabilities make it an attractive choice for displaying results to users. However, you shouldn’t use it. It can be handy to use when writing new PowerShell to verify your code contains values you expect. Beyond that, its use should be avoided. Jeffrey Snover covers the reasons why in his post [Write-Host Considered Harmful](https://www.jsnover.com/blog/2013/12/07/write-host-considered-harmful/). In it he advocates for the use of Write-Verbose over Write-Host. We’ll be covering Write-Verbose later in the series when we begin creating PowerShell functions.

### Write-Output

If you’ve been following along in the series you’ve already used [**Write-Output**](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-output?view=powershell-6) quite a bit. This is because Write-Output is what PowerShell is using behind the scenes as the default output stream.

|  |  |
| --- | --- |
| Code example: | |
| 1 | Get-Process |

When you run **Get-Process** and see results in the PowerShell console, that was accomplished by Write-Output. All of these examples accomplish the same thing:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10 | # Example 1  Get-Process    # Example 2  $processes = Get-Process  Write-Output $processes    # Example 3  $processes = Get-Process  $processes |

As a result, you won’t often see Write-Output written in PowerShell code. It’s implied as the default output behavior.

### Out-File

The console isn’t your only avenue for PowerShell output. For times when you want to output to a file, you can leverage [**Out-File**](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/out-file?view=powershell-6). **Out-File** is capable of sending the entire output to file. This means that it’s a bit more intelligent than **Write-Host**, and can handle sending the entire object return from a cmdlet to file. See below for some examples:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6 | # Example 1  $processes = Get-Process  $processes | Out-File -FilePath C:\Test\processInfo.txt    # Example 2  Get-Process | Out-File -FilePath C:\Test\processInfo.txt |

Don’t forget about those conversion cmdlets! There are several for outputs as well! So, if your file needs to be in CSV format, you can leverage **ConvertTo-CSV**:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | $processes = Get-Process  $processes | ConvertTo-Csv -NoTypeInformation | Out-File c:\test\processes.csv |

There are a few parameters you’ll want to familiarize yourself with for **Out-File**.

* NoClobber – prevents an existing file from being overwritten
* Append – instead of overwriting an output file, add to it
* Encoding – especially useful if you are running on Linux – specify the encoding type of the file

## PowerShell Error Types

In many programming languages an error results in the program halting, or terminating. The program won’t continue until the issue is corrected. You have likely experienced this at some point as a program crashed, and wouldn’t resume. PowerShell doesn’t typically behave this way and doesn’t have many terminating errors.

Remember that PowerShell is an operationally focused language that aims to make your life easier. If you wanted to get processes information with **Get-Process**, would you expect PowerShell to crash if it couldn’t query a process? Or would you expect it to log the error, and continue to do as much operational work as possible? Because of this operational nature, PowerShell errors and exceptions are typically non-terminating.

### Non-terminating

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # non-terminating error  1/0;Write-Host 'Hello, will I run after an error?' |

The above example is a classic divide by zero error. Because dividing by zero is not possible, this should throw an exception. Note the **;** symbol. In PowerShell this allows you to chain commands together. In this example that means PowerShell will first divide by zero, and then attempt to write a string to the console with **Write-Host**.

Launch a new PowerShell console and try the example. PowerShell runs both commands. Notice that you still get a divide by zero error, but that doesn’t stop PowerShell from moving on to the next command.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9 | # non-terminating errors don't stop loops  $collection = @(      'C:\Test\newcsv.csv',      'c:\nope\nope.txt'      'C:\Test\newcsv2.csv'  )  foreach ($item in $collection) {      Get-Item $item  } |

In this example we declare an array (**$collection**) and load it with three file paths. It is very possible that one of these doesn’t actually exist, or maybe we don’t have access to it. In the foreach loop, **Get-Item** may encounter an error accessing one of these as a result. Try this example on your computer by providing files that both do and don’t exist. Notice that PowerShell will continue processing the loop, regardless of how many errors are encountered.

### Terminating

It is sometimes undesirable to have non-terminating errors. There are several use cases where you may want to force PowerShell to terminate:

* The error warrants stopping all further action
* You want to trap and evaluate the error – possibly performing various actions based on the error information
* You want to trap the error and hide it from the user – providing a more user friendly message

There are several ways for turning a non-terminating PowerShell error into a terminating PowerShell error.

#### **ErrorAction**

[ErrorAction](https://devblogs.microsoft.com/powershell/erroraction-and-errorvariable/) is a common parameter that tells PowerShell what action to take when it encounters an error. The truth is, you’re always using the ErrorAction parameter. You’re just using the default setting when you don’t specify it, which is continue. Continue will display the error to the console and keep processing.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | # Without ErrorAction  Get-Item -Path c:\nope\nope.txt;Write-Host 'Hello, will I run after an error?'  # With ErrorAction  Get-Item -Path c:\nope\nope.txt -ErrorAction Stop;Write-Host 'Hello, will I run after an error?' |

Try the two examples above on your machine. The first **Get-Item** will error because it will be unable to find the file. **Write-Host** will continue to run. This is standard non-terminating behavior.

The second example leverages ErrorAction set to Stop. This instructs PowerShell to not continue when it encounters an error. Note that **Write-Host** does not run on the example with ErrorAction. That’s because **Get-Item** terminated when it was unable to find the file.

ErrorAction supports several settings:

* **Continue** – Logs error to $Error, displays error to console, continues processing.
  + default (used even if ErrorAction isn’t specified)
* **Stop** – Logs error to $Error, displays error to console, terminates.
* **SilentlyContinue** – Logs error to $Error, does not display error, continues processing.
* **Ignore** – Does not log error to $Error, does not display error, continues processing.

#### **try/catch**

Placing your code into a [try catch](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_try_catch_finally?view=powershell-6) doesn’t necessarily make it terminating. PowerShell will simply try the code inside the try section, and if an error is encountered, it will move to the catch. What the catch does is up to you. The use of **throw** in the catch is one way for making an error terminating.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | # throw causes PowerShell to terminate  try {    1/0;Write-Host 'Hello, will I run after an error?' }  catch {    throw } |

Try will catch many standard errors in this manner like divide by zero, or incorrect syntax.

When you have cmdlets in a try/catch and you want PowerShell to go to the catch, you still need to specify ErrorAction. Just because a cmdlet is in a try/catch does not mean PowerShell will go to the catch if the cmdlet encounters an error. Remember that PowerShell defaults to **Continue**. So even if an error is encountered, it will continue to process without going to the catch.

Try the following:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6 | # this example will not go the catch and will run the Write-Host  try {    Get-Item -Path c:\nope\nope.txt;Write-Host 'Hello, will I run after an error?' }  catch {    Write-Host 'You are now in the catch' }  # this example will error and go directly to the catch  try {    Get-Item -Path c:\nope\nope.txt -ErrorAction Stop;Write-Host 'Hello, will I run after an error?' }  catch {    Write-Host 'You are now in the catch' } |

Now evaluate the code block below. If you run this on your machine it will try to get process information for both processes. What could you change to make this terminate the first time it encounters an error?

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11 | $processNames = @(      'NotAProcess',      'Notepad'  )  foreach ($item in $processNames) {      try {        Get-Process -Nam $item    }      catch {          Write-Host $item          throw      }  } |

Keep in mind that you don’t have to terminate in the catch with **throw**. You could perform a variety of actions, such as logging the error somewhere. Using [Write-Error](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-error?view=powershell-6) you can even still display the error back to the user.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | # Write-Error simply prints the error for the user  try {    1/0;Write-Host 'Hello, will I run after an error?' }  catch {    Write-Error $\_ } |

You can also leverage finally with your try/catch. Regardless if the try is performed successfully, or if the catch is run, the finally step will always be performed.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | # Finally will log results regardless of what happens  try {    Get-Content -Path c:\nope\nope.txt -ErrorAction Stop }  catch {    Write-Error $\_ }  finally {    # log results to a logging file } |

## PowerShell Errors – Exploring the error object

PowerShell deals in objects, and PowerShell errors are no exception. (A pun!) The error that is written out to the console is just a piece of the error object. When you trap the error in a try/catch you can evaluate the error’s various properties and take action accordingly.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5 | #The website exists, but the page does not  try {      $webResults = Invoke-WebRequest -Uri '<https://techthoughts.info/nope.htm>'  }  catch {  Write-Error $\_ } |

Try the example above. Because the nope.htm does not exist, **Invoke-WebRequest** will throw an exception . We will write the error to console using **Write-Error $\_**. Just like for the success pipeline **$\_** represents the current object in the error pipeline as well.

Now, visiting a webpage that doesn’t exist technically is an exception, but not an uncommon one. Many users are familiar with miss-typing an address and getting a page not found error. If you ran the code above, it is doubtful that any of that is accurately reflected in your console. The error displayed with **Write-Error** isn’t fully articulating the issue.

Depending on the nature of your code, this likely is not a good reason to terminate and stop functionality. Instead, you might advise the user to check the web address and try again. To do that, you’ll need to evaluate the error object and take the appropriate action.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #The website exists, but the page does not  try {      $webResults = Invoke-WebRequest -Uri '<https://techthoughts.info/nope.htm>'  }  catch {      $theError = $\_      if ($theError.Exception -like "\*404\*") {          Write-Warning 'Web page not found. Check the address and try again.'          #Retry code      }      else {    throw   }  } |

The error object has many different sub-properties. Here, we evaluate the Exception sub-property to determine if the exception message contains a 404. If it does, we can advise the user and retry. Otherwise, we’ve encountered a different type of error and it may be appropriate to terminate.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | #The website does not exist  try {      $webResults = Invoke-WebRequest -Uri '<https://techthoughtssssssss.info/>'  }  catch {      $theError = $\_      $theError.Exception  } |

In this example, the website doesn’t exist at all. Thus, the sub-property Exception will be different. Run this on your computer to see how it differs from the previous example.

To see all the sub-properties that are contained in the error object you will need to pipe it to **Format-List** with the force parameter.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7 | #The website exists, but the page does not  try {    $webResults = Invoke-WebRequest -Uri '<https://techthoughts.info/nope.htm>'}  catch {      $theError = $\_      # see all the sub-properties      $theError | Format-List \* -Force  } |

## $Error Variable

$Error is a reserved variable that contains a collection of all errors in the current session. If you have been trying the examples in this lesson your current PowerShell window should contain quite a few. Simply type **$Error** and click enter.

You may be wondering what the point of this variable is. Remember that errors in PowerShell are rich data objects. As you’re developing you will likely be generating some errors as you try different solutions. If you forget to capture one, you can access the error in the **$Error** history. It’s a simple array so you can access the error and all it’s sub-properties.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | $Error[5] | Format-List \* -Force    #a few other $Error commands to try  $Error  1/0;$Error  Get-Process -Name 'NotAProcess'  $Error  $Error.Clear() |

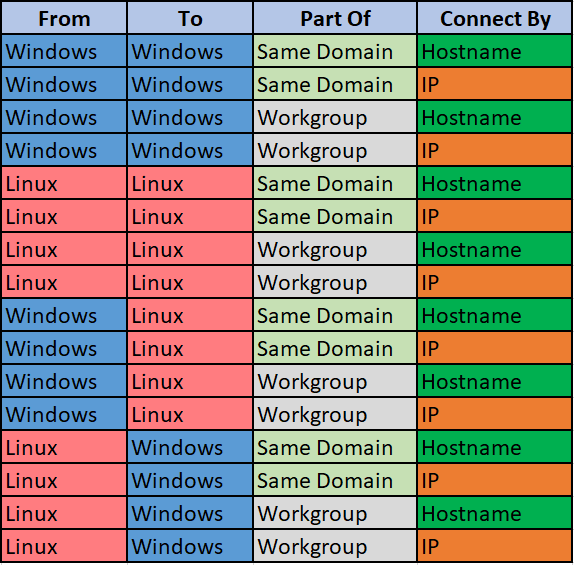
## Closing Example

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #this example will help display some helpful message to the user  #this example will only work in PowerShell 6.1+  $uri = Read-Host 'Enter the URL'  try {      $webResults = Invoke-WebRequest -Uri $uri -ErrorAction Stop  }  catch {      $statusCodeValue = $\_.Exception.Response.StatusCode.value\_\_      switch ($statusCodeValue) {          400 {              Write-Warning -Message "HTTP Status Code 400 Bad Request. Check the URL and try again."          }          401 {              Write-Warning -Message "HTTP Status Code 401 Unauthorized."          }          403 {              Write-Warning -Message "HTTP Status Code 403 Forbidden. Server may be having issues. Check the URL  and try again."          }          404 {              Write-Warning -Message "HTTP Status Code 404 Not found. Check the URL and try again."          }          500 {              Write-Warning -Message "HTTP Status Code 500 Internal Server Error."          }          Default {              throw          }      }  } |

## Understanding PowerShell Remoting Requirements

This topic is a bit different from previous episodes because it requires knowledge of several concepts beyond PowerShell. Establishing a connection from one device to another engages a lot of system technologies. An understanding of basic network protocols, ports, firewalls, listeners, and server management is assumed. Remote access using PowerShell also engages various authentication technologies: Default, Basic, Digest, NTLM, Kerberos, Negotiate, and CredSSP.

To further complicate matters there are many possible connection scenarios. Each requires a slightly different remote connection configuration!



This post can’t cover every scenario, as that would greatly increase the length of this topic. Instead, we’ll cover some of the more common configuration scenarios. Links are provided at the bottom of this article to get you started with other configuration requirements.

Don’t feel put off by the complexity of this topic. If you are wondering why you would invest a lot of effort into learning and configuring your environment for remote PowerShell connections, remember this: PowerShell remoting is one of the most powerful capabilities that PowerShell has to offer. It can be used to audit, push changes, or query thousands of devices. This is the capability that turns a 3-day task into a 15 minute task. It will take effort to learn and implement, but PowerShell remoting is well worth investing in.

## Windows PowerShell Remoting

PowerShell remote access on Windows uses the following technologies:

* [Windows Remote Procedure Call (RPC)](https://docs.microsoft.com/en-us/windows/win32/rpc/rpc-start-page)
  + Keep in mind that the RPC protocol only works in Windows PowerShell and not with PowerShell Core (6+).
* [Windows Management Instrumentation (WMI)](https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page)
* [WS-Management](https://docs.microsoft.com/en-us/windows/win32/winrm/ws-management-protocol)
  + Controlled and managed in Windows via WinRM

### ComputerName

There are many native cmdlets (even more with modules) that support the ComputerName parameter. These allow you to query and interact with remote machines using PowerShell [without the need for additional configuration](https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/running-remote-commands?view=powershell-6#windows-powershell-remoting-without-configuration). Per Microsoft documentation varying communication protocols are utilized to accomplish this which are typically enabled by default on Microsoft Operating systems.

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Per PowerShell documentation you can find a list of cmdlets that support ComputerName with the following:  Get-Command | Where-Object { $\_.parameters.keys -contains "ComputerName" -and $\_.parameters.keys -notcontains "Session"} |

This means in most cases credentials and network access to the device should be sufficient to interact with it. Lets start by first creating a credential object using [**Get-Credential**](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/get-credential?view=powershell-6).

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # this will prompt you to enter your access credentials. the creds will be securely stored in the variable  $creds = Get-Credential |

With our credentials provided, we can now use cmdlets with the ComputerName previously identified to perform remote tasks:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9 | # restart a single computer  Restart-Computer -ComputerName RemoteDevice -Credential $creds    # restart several computers  Restart-Computer -ComputerName RemoteDevice1, RemoteDevice2, RemoteDevice3 -Credential $creds    # restart an entire list of computers  $devices = Get-Content -Path C:\listOfServers.txt  Restart-Computer -ComputerName $devices -Credential $Creds -Force | |

### WinRM

Windows remote access in PowerShell is typically accomplished by establishing a session via [WinRM](https://docs.microsoft.com/en-us/windows/win32/winrm/installation-and-configuration-for-windows-remote-management). WinRM must be installed and configured on both your management device as well as all devices you will be managing remotely.

For WinRM to operate two things are required:

1. The WinRM service needs to be running
2. A WinRM listener needs to be configured

WinRM listens on a local port that you configure. By default, WinRM uses the following ports:

* Port **5985** for http connections (trusted network – typical for domain to domain)
* Port **5986** for https connections (untrusted network – also required in some non-domain scenarios)

Keep in mind that multiple listeners can be configured on a single device.

#### **Configuring WinRM**

To easily configure WinRM on a device with all defaults you can simply run the following:

|  |  |
| --- | --- |
| Code example: | |
| 1 | winrm quickconfig |

This command performs the following:

* Starts WinRM service and sets startup type to Automatic
* Configures a default listener for HTTP for any IP address
* Configures Windows Firewall to open WinRM ports

Remember, WinRM is configured as both client and a service. The client is the settings used when the device establishes an outgoing session to a remote device. The service settings configure the listeners that respond to potential clients trying to establish a remote connection.

You can see both the client and service settings of your device by running the following from an administrator PowerShell window:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | winrm get winrm/config/client  winrm get winrm/config/service |

#### **Testing WinRM**

Verify WinRM is configured and responding using [**Test-WSMan**](https://docs.microsoft.com/en-us/powershell/module/microsoft.wsman.management/test-wsman?view=powershell-6). This cmdlet is van verify WinRM on both a local and remote device.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | #verify that WinRM is setup and configured locally  Test-WSMan    #verify that WinRM is setup and responding on a remote device  #you must specify the authentication type when testing a remote device.  #if you are unsure about the authentication, set it to Negotiate  $credential = Get-Credential  Test-WSMan RemoteDeviceName -Authentication Negotiate -Credential $credential |

You can also leverage the following commands to verify basic network connectivity and open ports:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5 | #verify local device is listening on WinRM port  Get-NetTCPConnection -LocalPort 5985    #verify a remote device is listening on WinRM port  Test-NetConnection -Computername 192.168.34.13 -Port 5985 |

#### **Connecting via WinRM**

**Interactive Session**

[Enter-PSSession](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/enter-pssession?view=powershell-6) establishes an interactive remote session to the specified device. Interactive sessions are for connecting to a single remote device and running commands remotely in the console. This is a great alternative to RDP. You’ll notice that once you establish a connection that the command prompt will change.

[RemoteDeviceName]: PS C:\ >

This indicates that commands typed and run are running in the context of the remote device, and no longer you local device. You can continue to run commands in the remote context of the interactive session. When you want to drop back to your local device, simply type **Exit**.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | #establish an interactive remote session  $credential = Get-Credential  Enter-PSSession -ComputerName RemoteDeviceName -Credential $credential |

**PowerShell session (PSSession)**

Use [New-PSSession](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new-pssession?view=powershell-6) to establish remote PowerShell sessions to one, or many remote devices.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #basic session opened to remote device  $session = New-PSSession -ComputerName RemoteDeviceName -Credential domain\user    #session opened to device over SSL  $credential = Get-Credential  $sessionHTTPS = New-PSSession -ComputerName RemoteDeviceName -Credential $credential -UseSSL    #establish sessions to multiple devices  $credential = Get-Credential  $multiSession = New-PSSession -ComputerName RemoteDeviceName1,RemoteDeviceName2, RemoteDeviceName3 -Credential $credential    #establish session to an entire list of devices  $devices = Get-Content -Path C:\listOfServers.txt  $credential = Get-Credential  $multiSession = New-PSSession -ComputerName $devices -Credential $credential    #session opened with advanced session options configured  $sessionOptions = New-PSSessionOption -SkipCNCheck -SkipCACheck -SkipRevocationCheck  $advancedSession = New-PSSession -ComputerName 10.0.3.27 -Credential user -UseSSL -SessionOption $so |

In and of itself this session doesn’t accomplish anything other than opening a connection to the remote device(s). Use [Invoke-Command](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/invoke-command?view=powershell-6) to actually send commands to those established sessions.

This is hugely empowering as it enables you remotely perform tasks on devices throughout your environment. Take note of the ScriptBlock parameter. Provide code here to audit, query, or make changes to your entire fleet.

|  |  |
| --- | --- |
|  | |
|  | #get the number of CPUs for each remote device  Invoke-Command -Session $sessions -ScriptBlock {(Get-CimInstance Win32\_ComputerSystem).NumberOfLogicalProcessors}    #get the amount of RAM for each remote device  Invoke-Command -Session $sessions -ScriptBlock {Get-CimInstance Win32\_OperatingSystem | Measure-Object -Property  TotalVisibleMemorySize -Sum | ForEach-Object {[Math]::Round($\_.sum/1024/1024)}}    #get the amount of free space on the C: drive for each remote device  Invoke-Command -Session $sessions -ScriptBlock {      $driveData = Get-PSDrive C | Select-Object Used,Free      $total = $driveData.Used + $driveData.Free      $calc = [Math]::Round($driveData.Free / $total,2)      $perFree = $calc \* 100      return $perFree  }    #stop the BITS service on all remote devices  Invoke-Command -Session $sessions -ScriptBlock {Stop-Service BITS -Force} |

Invoke-Command runs commands in parallel, so keep that in mind. If you send a command to 500 servers, data that comes back will not be in the same order. For example, if server 25 in the list is the first to process the scriptblock, it will be the first to return results.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | #alternative to running Invoke-Command in parallel  #foreach forces sequential connection and return for each server in the list  #establish session to an entire list of devices  $devices = Get-Content -Path C:\listOfServers.txt  $credential = Get-Credential  foreach ($server in $devices) {      Invoke-Command -ComputerName $server -ScriptBlock {$env:COMPUTERNAME} -Credential $credential  } | |

## PowerShell Remoting on Linux

Linux and MacOS leverage SSH for PowerShell remote access. Each distro will require slightly different specific steps.

* Install OpenSSH
* Configure ssh\_config for password authentication and PowerShell subsystem
* Restart sshd service

Reference [PowerShell remoting over SSH](https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/ssh-remoting-in-powershell-core?view=powershell-6) for other step-by-step instructions for SSH configuration. I have included a basic example below from that link:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #install openssh  sudo apt install openssh-client  sudo apt install openssh-server    #Edit the sshd\_config file at location /etc/ssh  #Make sure password authentication is enabled:  PasswordAuthentication yes    #Add a PowerShell subsystem entry:  Subsystem powershell /usr/bin/pwsh -sshs -NoLogo -NoProfile    #Optionally, enable key authentication:  PubkeyAuthentication yes    #Restart the sshd service.  sudo service sshd restart |

Once configured you can establish sessions and execute remote commands. The SSHTransport parameter indicates that the remote connection will use SSH.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7 | #establish an interactive session to a remote Linux device  $session = New-PSSession -HostName RemoteDevice -UserName user -SSHTransport  Enter-PSSession $session    #execute commmands on a remote Linux device  $session = New-PSSession -HostName RemoteDevice -UserName user -SSHTransport  Invoke-Command -Session $session -ScriptBlock {Get-Process} |

## PowerShell Remoting: Advanced / Other

As shown in the graphic above there are many different connection scenarios. Most of these require additional configuration steps beyond what has been covered. Some only support SSL, which will require you to setup certificates. Others require adding remote devices to your TrustedHosts. Consult the Additional Reading section at the bottom of this post for more information.

### Linux to Windows and Windows to Linux

It is definitely possible to configure remote PowerShell access from Windows to Linux and vice-versa. Unfortunately, the complexities of that configuration are beyond the scope of this course. There are several published articles covering various methods, but I have not found one today that covers every scenario well.

## Closing Example

Use the closing example to audit remote devices for:

* ComputerName
* Number of CPUs
* Amount of Memory
* Free space on C:

The **$servers** variable could easily be replaced with a different source of devices in your environment. Note that this does not run **Invoke-Command** in a loop. This means commands will be run in parallel against all remote devices. As a result, information will come back in an unpredictable order, which is why we are also capturing the computer name.

ScriptBlock contains the logic that will be executed on the remote device. In this case we will be gathering some basic information about the device and storing it in a custom PowerShell object for return. There are a wide variety of reasons that this might encounter errors during execution. Remote devices might be:

* Down for maintenance
* Not configured for remote access
* Credentials not permitted for remote access
* You can not reach the device due to network configuration

To handle this in a large audit sweep like this closing example, we will send PSRemotingTransportException errors to a custom error variable, connectErrors. At the end of execution, you will have two final variables:

* **$remoteResults**: Will contain the results of all remote executions
* **$remoteFailures**: Will contain a list of all devices that failed to establish a remote connection to

|  |
| --- |
| #declare servers we will connect to remotely  $servers = 'Server1','Server2','Server3','Server4'  #capture credentials used for remote access  $creds = Get-Credential    #declare array to hold remote command results  $remoteResults = @()    #declare a splat for our Invoke-Command parameters  $invokeSplat = @{      ComputerName  = $servers      Credential    = $creds      ErrorVariable = 'connectErrors'      ErrorAction   = 'SilentlyContinue'  }    #execute remote command with splatted parameters.  #store results in variable  #errors will be stored in connectErrors  $remoteResults = Invoke-Command @invokeSplat -ScriptBlock {      $obj = [PSCustomObject]@{          Name      = $env:COMPUTERNAME          CPUs      = "-------"          Memory    = "-------"          FreeSpace = "-------"      }      $obj.CPUs = (Get-CimInstance Win32\_ComputerSystem).NumberOfLogicalProcessors      $obj.Memory = Get-CimInstance Win32\_OperatingSystem `          | Measure-Object -Property TotalVisibleMemorySize -Sum `          | ForEach-Object { [Math]::Round($\_.sum / 1024 / 1024) }      $driveData = Get-PSDrive C | Select-Object Used, Free      $total = $driveData.Used + $driveData.Free      $calc = [Math]::Round($driveData.Free / $total, 2)      $obj.FreeSpace = $calc \* 100      return $obj  }    #capture any connection errors  $remoteFailures = $connectErrors.CategoryInfo `      | Where-Object {$\_.Reason -eq 'PSRemotingTransportException'} `      | Select-Object TargetName,@{n = 'ErrorInfo'; E = {$\_.Reason} } |

## Creating and Saving PowerShell Scripts

Creating a PowerShell script is as simple as saving your PowerShell commands with the **.ps1** file extension. Go ahead and copy the simple code below and save it as **math.ps1**. We’ll use this simple script example moving forward.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | $total = 2 + 2  $output = "two plus two is equal to $total"  Write-Output $output |

## Running PowerShell Scripts

You can run a PowerShell script by specifying the path to the script:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | #windows  C:\scripts\math.ps1  #linux  /scripts/math.ps1 |

### PowerShell Execution Policy

Depending on the configuration of your device and environment you may not be able to run PowerShell scripts. You may need to adjust your PowerShell Execution policy.

* **Restricted** — Stops any script from running.
* **RemoteSigned** — Runs scripts created on the device. However, scripts created on another computer won’t run unless they include a signature of a trusted publisher.
* **AllSigned** — All the scripts will run as long as they’ve been signed by a trusted publisher.
* **Unrestricted** — Runs any script without any restrictions.

To determine the current execution policy run the following cmdlet:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | #get the execution policy of all scopes in the order of precedence  Get-ExecutionPolicy -List |

To change the execution policy:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | #change execution policy  Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser |

### Script Scope

When you run a script by specifying the ps1 file path, that script does not run in the scope of the current session. This means that if you open a PowerShell session, and run a script, you will not have access to the variables and information in your current session. You can see this in action for yourself. Open a PowerShell window and run the **math.ps1** script from above.

After it runs, try typing **$total** in your PowerShell session. It will not return anything. This is because **math.ps1** was run under the scope of a different PowerShell session. Once the script completed, so did that session.

If you need to access the information in the context of the current PowerShell session, you can **dot source** the PowerShell script. This is achieved by placing a dot and then a space before the script path.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | # dot source a full file path  . C:\scripts\math.ps1  # dot source a local file  . .\math.ps1 |

## PowerShell Script Example

If you’ve completed Episodes 0-9, you’re ready to start writing PowerShell scripts. You may not feel ready, but the only way to shake that feeling is to write more PowerShell. We’re going to kick things off with our first script. Before sitting down and writing code, you need to have an objective. Lets declare our script’s objective now.

We want to be notified when a drive in our home network is running low on space. The script will check the desired drive for free space, and send a [Telegram](https://telegram.org/) notification if it is below 20%. All actions should be logged. The script should support scanning multiple different drives. The script should support both Linux and Windows.

If this is your first time writing a script, you may have no idea where to start. That’s OK. Just break the objective into manageable chunks:

* Supports logging
* Checks desired drive for free space
* Sends Telegram notification
* Lets user specify drive
* Supports both windows and Linux

Try tackling these one at a time. In the video above you can watch me live-code this solution. This gives you a chance to see my thought process and workflow for tackling something like this. Alternatively, you can reference the solution below, and try scripting your own objective.

As an additional exercise try enhancing the script below to:

* Permit the user to specify the percentage threshold
  + Tip: it’s currently hard-coded to 20
* Not a Telegram user? Try adjusting the script to send an email instead.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 | param (      [Parameter(Mandatory = $true)]      [string]      $Drive  )    if ($PSVersionTable.Platform -eq 'Unix') {      $logPath = '/tmp'  }  else {      $logPath = 'C:\Logs' #log path location  }    #need linux path    $logFile = "$logPath\driveCheck.log" #log file    #verify if log directory path is present. if not, create it.  try {      if (-not (Test-Path -Path $logPath -ErrorAction Stop )) {          # Output directory not found. Creating...          New-Item -ItemType Directory -Path $logPath -ErrorAction Stop | Out-Null          New-Item -ItemType File -Path $logFile -ErrorAction Stop | Out-Null      }  }  catch {      throw  }    Add-Content -Path $logFile -Value "[INFO] Running $PSCommandPath"    #verify that the required Telegram module is installed.  if (-not (Get-Module -ListAvailable -Name PoshGram)) {      Add-Content -Path $logFile -Value '[INFO] PoshGram not installed.'      throw  }  else {      Add-Content -Path $logFile -Value '[INFO] PoshGram module verified.'  }    #get hard drive volume information and free space  try {      if ($PSVersionTable.Platform -eq 'Unix') {          $volume = Get-PSDrive -Name $Drive -ErrorAction Stop          #verify volume actually exists          if ($volume) {              $total = $volume.Free + $volume.Used              $percentFree = [int](($volume.Free / $total) \* 100)              Add-Content -Path $logFile -Value "[INFO] Percent Free: $percentFree%"          }          else {              Add-Content -Path $logFile -Value "[ERROR] $Drive was not found."              throw          }      }      else {          $volume = Get-Volume -ErrorAction Stop | Where-Object { $\_.DriveLetter -eq $Drive }          #verify volume actually exists          if ($volume) {              $total = $volume.Size              $percentFree = [int](($volume.SizeRemaining / $total) \* 100)              Add-Content -Path $logFile -Value "[INFO] Percent Free: $percentFree%"          }          else {              Add-Content -Path $logFile -Value "[ERROR] $Drive was not found."              throw          }      }  }  catch {      Add-Content -Path $logFile -Value '[ERROR] Unable to retrieve volume information:'      Add-Content -Path $logFile -Value $\_      throw  }    #evaluate if a message needs to be sent if the drive is below 20GB freespace  if ($percentFree -le 20) {        try {          Import-Module PoshGram -ErrorAction Stop          Add-Content -Path $logFile -Value '[INFO] PoshGram imported successfully.'      }      catch {          Add-Content -Path $logFile -Value '[ERROR] PoshGram could not be imported:'          Add-Content -Path $logFile -Value $\_          throw      }        Add-Content -Path $logFile -Value '[INFO] Sending Telegram notification'        $messageSplat = @{          BotToken    = "#########:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"          ChatID      = "-#########"          Message     = "[LOW SPACE] Drive at: $percentFree%"          ErrorAction = 'Stop'      }        try {          Send-TelegramTextMessage @messageSplat          Add-Content -Path $logFile -Value '[INFO] Message sent successfully'      }      catch {          Add-Content -Path $logFile -Value '[ERROR] Error encountered sending message:'          Add-Content -Path $logFile -Value $\_          throw      } } |

## When you should write a PowerShell function

At the end of the day a PowerShell script is a set of code that performs a task. That’s what a PowerShell function does too! So, why would you choose to write a function vs a script? We’ll dive into a few reasons below. Generally speaking though, a script is something for you. When you want to start sharing your code, or having your team-mates engage it, you’ll want to start engaging some of the capabilities that functions provide. There is no firm line. For a quick task that will be run on your workstation one time, a script might make sense. If you’re writing some PowerShell that others might use, or that you might use more than once, a function will likely serve better.

* **Single purposed** – a function is typically used to perform a narrow task and “do one thing, and do it great”. This makes functions highly **re-usable**. If you write a logging function for example, you can then use that PowerShell logging function with other functions or scripts.
* **Help** – functions support help based comments. This allows your users to do things like type Get-Help and figure out how to use your function.
* **Parameters** – support for simple and advanced parameters declarations and control allows your function to be dynamic and take various forms of user input.
* **Testable** – functions can be tested and mocked which greatly improves the quality of the code.

## Anatomy of a PowerShell function

Here is the basic layout and order of a PowerShell function. We’ll dive into each of these sections below and explore them in detail.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # function help - (optional but strongly encouraged)  # function name  # CmdletBinding - (optional)  # parameters - (optional)  # function logic (optional Begin / Process / End)  # return - (optional)    function Verb-Noun {      [CmdletBinding()]      param (    )       begin {   }       process {    }       end {   }  } |

### Function Help

When you include help at the top of your function it will be displayed when Get-Help is run against your function. This is a powerful capability when others are exploring your code. Taking the time to flesh out a good Synopsis and Description combined with solid examples will help others understand what your function is for, and how to use it.

All help fields are optional and you can add or remove them to your liking. I recommend you include at least the fields below at a minimum.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <#  .SYNOPSIS      Short description  .DESCRIPTION      Long description  .EXAMPLE      C:\PS>      Example of how to use this cmdlet  .EXAMPLE      C:\PS>      Another example of how to use this cmdlet  .PARAMETER InputObject      Specifies the object to be processed.  You can also pipe the objects to this command.  .OUTPUTS      Output from this cmdlet (if any)  .NOTES      General notes  .COMPONENT      The component this cmdlet belongs to  #> |

### CmdletBinding

|  |  |
| --- | --- |
| Code example: | |
| 1 | [CmdletBinding()] |

When you write a great function and give it a name like **Get-AllFileInformation**, it looks and will act like a cmdlet. But it’s an important distinction that a PowerShell function is not a cmdlet. Cmdlet’s are written and compiled in C# and your PowerShell function is written in PowerShell.

CmdletBinding is an attribute of functions that enables capabilities that makes them operate more like compiled cmdlets. Adding this to the top of your function will give your function a lot of additional capabilties such as:

* [Write-Verbose](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-verbose?view=powershell-7) – allow users to use the -Verbose switch to see what your function is doing while it’s doing it.
* [ShouldProcess](https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.cmdlet.shouldprocess?view=pscore-6.2.0) – if your function will make a change to a system that is high risk, you may want the user to confirm the action.
* [PositionalBinding](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters?view=powershell-7#position-argument) – enables your function to be run without explicitly providing each parameter name. The values can be inferred by the order they are provided to your function

### parameters

|  |  |
| --- | --- |
|  | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | param (      [Parameter(          Position = 0,          Mandatory = $false      )]      [string]      $City,        [Parameter(Position = 1)]      [ValidateSet('Metric', 'USCS')]      [string]      $Units = 'USCS',        [Parameter(Position = 2)]      [ValidateSet('ar', 'af', 'be', 'ca', 'da', 'de', 'el', 'en', 'et', 'fr', 'fa', 'hu', 'ia', 'id', 'it', 'nb', 'nl', 'pl', 'pt-br', 'ro', 'ru', 'tr', 'th', 'uk', '  vi', 'zh-cn', 'zh-tw')]      [string]      $Language = 'en',        [Parameter(Position = 3)]      [switch]      $Short  ) |

Parameters serve to provide the user a way of dynamically interacting with your function. You can take in various forms of input and perform a wide variety of checks and validations to ensure you are getting the right kind of information.

You can stay very basic and only declare the names of your parameters. Alternatively, you can get very specific on what is required for your parameters:

* Whether the parameter is mandatory
* The position of a parameter (the order)
* The parameter type (string/int/bool/etc)
* Setting default values
* Input validation
* Parameter sets
* Much more

This post can’t cover every possible parameter configuration. Even after years of writing PowerShell I find it difficult to memorize all of the parameter settings and syntax. Know that parameter declarations are your way of taking input into your function. You can provide a lot detail and controls to them, or just keep them simple depending on your needs. Bookmark the parameter links at the bottom of this post. Even today, I visit them frequently!

### Begin Process End

These three key-words are phases that you can declare within your function.

* Begin will initialize some steps at the start
* Process will process each object as received
* End can perform cleanup.

This is especially useful if you want your your function to support processing inputs from the pipeline.

This can be difficult to grasp in theory so here is an example that demonstrates it’s use.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7 | function Get-PipelineBeginEnd {      param (    [string]$SomeInput    )      begin {     "Begin: The input is $SomeInput"   }      process {     "The value is: $\_"   }      end {     "End:   The input is $SomeInput"   }  }#Get-PipelineBeginEnd  1, 2, 3 | Get-PipelineBeginEnd -SomeInput 'Test' |

When you run the code above note that Begin runs once, as does End. Process is run for each item passed into it and Process has access to the current object in the pipeline.

### Function Logic

The logic contained inside a PowerShell function is no different that that of a script. This is where you will write the core logic that performs the task that your function is intended for. This was covered in episodes 1-10 of [this course](https://techthoughts.info/learn-powershell-series/). You can also see a full live example in the [PowerShell Scripts](https://techthoughts.info/powershell-scripts/) episode.

### Function return

Not all functions return something when run. If your function will return something you can optionally use return. Keep in mind that when using return, all further actions in the function will stop and the function will end.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10 | function Get-Total {      param (          [int]$Number1,          [int]$Number2      )      $total = $Number1 + $Number2        return $total  }  Get-Total -Number1 2 -Number2 2 |

In the above example this simple function contains two parameters of type integer. The logic of the function adds the two numbers together and returns the integer to the user.

## Your first PowerShell function

Take at look at the PowerShell function below. Note the layout adheres to what we’ve previously covered. Also note that the help section provides a great amount of detail as to the purpose and use of the function.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | <#  .SYNOPSIS      Returns your public IP address.  .DESCRIPTION      Queries the ipify Public IP Address API and returns your public IP.  .EXAMPLE      Get-PublicIP        Returns the public IP.  .OUTPUTS      System.String  .NOTES  <https://github.com/rdegges/ipify-api>  #>  function Get-PublicIP {      $uri = '[https://api.ipify.org](https://api.ipify.org/)'      try {          $invokeRestMethodSplat = @{              Uri         = $uri              ErrorAction = 'Stop'          }          $publicIP = Invoke-RestMethod @invokeRestMethodSplat      }      catch {    Write-Error $\_   }       return $publicIP  }#Get-PublicIP |

As the actual logic of this function is fairly simple, it might be tempting to leave it as a script. Note though that this does one specific thing, and does it well. You could now incorporate this function into other advanced logic, instead of having to reference or import a script. Other users in your environment might also start engaging this function once they become familiar with it.

## Function Scope

Like most languages PowerShell has scoping rules for variables. When a variable is within a function you will no longer be able to “see” it’s value. Here is an example of this behavior:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6 | function Get-NumberTimesTwo {      param (     [int]$Number    )      $total = $Number \* 2      return $total  }#Get-NumberTimesTwo  Get-NumberTimesTwo -Number 2 |

The function behaves as expected and will correctly multiply a number by two. In your console though, try seeing what the value of **$total** is. You will not be able to. This is because **$total** is scoped to the function, and not your active console session.

This can be confusing for people new to PowerShell and can make writing functions more difficult. There are a couple of strategies for dealing with this.

You could write the majority of your logic in a script type format, testing variables as you go. Once you are confident the logic is working as intended, you could then wrap the logic inside a function.

Alternatively, you can use **Write-Host**, **Write-Output**, or better yet **Write-Debug** to see what is going on with your variables inside your function!

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | function Get-NumberTimesTwo {      [CmdletBinding()]      param (     [int]$Number   )      $total = $Number \* 2      Write-Debug $total      return $total  }#Get-NumberTimesTwo  Get-NumberTimesTwo -Number 2 -Debug |

Note that with debug switch we get access to the value inside $total to ensure our logic is doing what we intend.

## Closing Example

Here is a larger closing example of two PowerShell Functions that will enable you to surf the popular website ***reddit*** using PowerShell!

Visit the GitHub gist link below to access the code and load it into your console.

<https://gist.github.com/techthoughts2/cd2b720c9b291510cbd643e6ca73e05f>

As you explore these functions try to answer the following questions:

* Of the two functions, which one is the primary function?
* Are these standard or advanced functions?
* Which function is going to support pipeline input?

## PowerShell In The Cloud

Eventually you will be called upon to help manage and solve problems with various cloud-based solutions. Some of those resources will be of a more traditional nature, such as Infrastructure as a Service (IaaS). Others may be more modern solutions such as Software as a Service (SaaS), or Platform as a Service (Paas).

* SaaS examples:
  + Microsoft Office 365
  + Google Apps (Gmail, Google Docs)
* PaaS examples:
  + AWS Elastic Beanstalk
  + Azure App Service
  + Google App Engine
* IaaS examples:
  + AWS EC2
  + Azure Virtual Machine
  + Google Compute Engine (GCE)

Regardless of type, many of these still have traditional authentication, accessibility, inventory, monitoring, and cost challenges to solve. Your investment in learning PowerShell has paid off in a big way, as PowerShell is available to help you manage cloud solutions.

### Getting Practice with PowerShell and Cloud

Like anything technology based – it will help to practice some of these tasks in the real world. To do that, you’ll need a cloud account. (Ideally more than one – don’t just go “all in” on one cloud!). It does take a credit card and a valid email address to get an account setup. Even though you’ve provided your credit card each major cloud provider has a free tier that will enable to practice for no cost, or very low cost.

Cloud providers free tier information:

* [Azure Free Account Information](https://azure.microsoft.com/en-us/free/free-account-faq/)
* [AWS Free Tier Information](https://aws.amazon.com/free/)
* [Google Cloud Free Tier Information](https://cloud.google.com/free)

Don’t let cost be your excuse for not diving into cloud learning. Part of your job as a technologist in the modern landscape is to have a basic understanding of cloud costs and billing. You should be comfortable navigating the free tier to learn cloud basics without racking up any type of bill. If you do make a mistake and rack up a $15 bill, that’s money you spent investing in yourself!

## AWS Tools for PowerShell

Amazon Web Services (AWS) is the largest cloud provider and offers hundreds of cloud based services. Your ability to manage AWS resources using PowerShell will be a great addition to your resume.

What you’ll need to get started:

* [AWS Account](https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/)
* [Create a user in your AWS account](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html)
* [AWS Tools for PowerShell](https://aws.amazon.com/powershell/)

The AWS Tools for PowerShell let developers and administrators manage their AWS services and resources using PowerShell! AWS Tools is essentially just a module, or collection of modules that enables your existing PowerShell install to mange AWS Cloud! You can find and install the needed module(s) from the [PowerShell Gallery](https://www.powershellgallery.com/) using basic commands like **Find-Module** and **Install-Module**.

The AWS Tools for PowerShell has evolved over time so lets dive quickly into the differences.

* AWS Tools for PowerShell – Modular (**recommended**)
  + **AWS.Tools.\*** – this is the primary modular module, and the one that will receive support moving forward. There is only one “core” small module (**AWS.Tools.Common**) which contains a few basic AWS commands and allows you to authenticate to your AWS account. Every service offered by AWS is then broken up into a separate module. For example EC2 (Virtual Machine/Compute) cmdlets are found in the **AWS.Tools.EC2** module. This version of the module supports Windows PowerShell 5.1+ and PowerShell Core 6+ on Windows, Linux and macOS.
* AWS Tools for PowerShell
  + **AWSPowerShell.NetCore** – This is a large monolithic module that contains all the PowerShell cmdlets for AWS in one module. It is quite large and will eventually be deprecated as AWS continues to grow. It supports PowerShell 6+.
* AWS Tools For PowerShell (Legacy)
  + **AWSPowerShell** – This is a large monolithic module that contains all the PowerShell cmdlets for AWS in one module. It is quite large and is in the process of being deprecated. It was created to support PowerShell 5.1.

### Installing AWS Tools for PowerShell

Because the AWS Tools for PowerShell (modular) is the future, this blog post will focus on this version of the module.

Get started by running the following install command:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Install AWS.Tools - A Modularized Version of AWS Tools for PowerShell  Install-Module -Name AWS.Tools.Installer |

The **AWS.Tools.Installer** module provides cmdlets that enable you to install, update, and remove the modules for each of the AWS services. It also automatically install the **AWS.Tools.Common** module which provides cmdlets for configuration and authentication that are not service specific.

Use the **AWS.Tools.Common** module not only to authenticate to your AWS account, but also discover what modules are required for each AWS Service. Try running the following:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3 | # Get a list of AWS services supported by the Tools for PowerShell  Import-Module AWS.Tools.Common  Get-AWSService | Select-Object Service,ServiceName,ModuleName | Format-Table -AutoSize |

There are a lot of AWS Services, and there is a module to support each one!

AWS operates with the concept of [regions](https://aws.amazon.com/about-aws/global-infrastructure/regions_az/). You will likely want to deploy resources to the AWS region that is closest to you. Get a list of available AWS regions by running the following:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Discover available AWS regions  Get-AWSRegion |

#### **Keeping AWS.Tools in Sync**

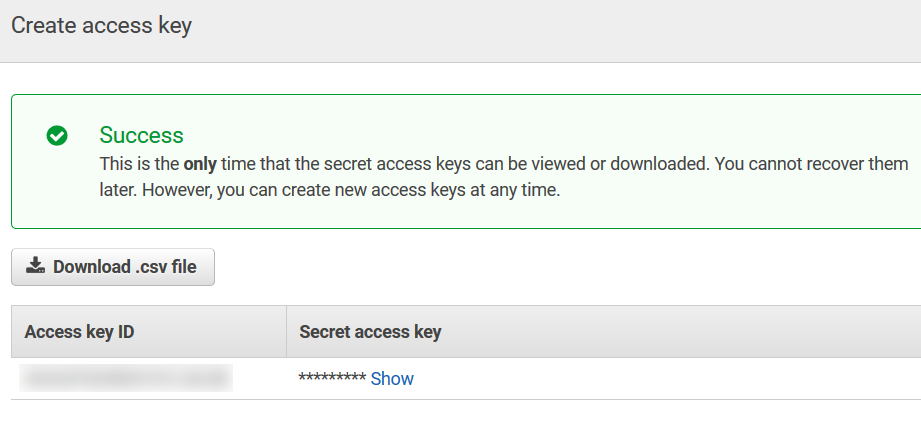
Quick note regarding the modular version of AWS.Tools. As you continue to engage various AWS services using PowerShell you will likely install additional service modules from time to time. As a result, you may find yourself in a state where your module versions are out of sync. For instance, your **AWS.Tools.S3** module might be on version 4.0.4.0 and your **AWS.Tool.EC2** might be on version 4.0.6.0. This can be easily fixed by simply running **Update-AWSToolsModule**.

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Keep AWS.Tools versions in sync  Update-AWSToolsModule |

### Authenticating to AWS using PowerShell

There are [several methods for specifying your AWS credentials in PowerShell](https://docs.aws.amazon.com/powershell/latest/userguide/specifying-your-aws-credentials.html). In this post I’ll demo one way, which is providing the Access Key and Secret key of a user in your AWS account. If you haven’t done so, go ahead and [create a user in your AWS account](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html).

Once you have a user created, navigate to Services ➝ IAM ➝ Users ➝ (Your User) ➝ Security Credentials Tab. From there, you can click the create access key button. Don’t forget to copy your credentials!



Armed with your user’s access key and secret key, you can now establish a secure connection to your AWS account in your PowerShell session. We will also set the default region that we will be creating and managing resources in.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | #\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  # Import the AWS Tools Common module  Import-Module AWS.Tools.Common  # set the credential for the session  Set-AWSCredential -AccessKey $aKey -SecretKey $sKey  # set the default region for the session  Set-DefaultAWSRegion -Region us-west-2  #\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

### Managing AWS Cloud with PowerShell

With your PowerShell session authenticated to your AWS account you might be thinking, now what? The answer is the possibilities are nearly limitless. Using PowerShell the entire power of the AWS cloud is at your fingertips.

* Create a virtual server and establish access
* Create a fleet of 500 servers pre-configured to your needed specification
* Retrieve billing information
* Create monitors and alarms
* Create serverless solutions

#### **AWS PowerShell Final Example**

In this final example we’ll use PowerShell to create a highly secure storage bucket in AWS (S3). Then we’ll upload a file to the bucket, and create a secure pre-signed URL to the file. The URL will be good for 24 hours (configurable). This is a great way to share a file securely worldwide leveraging the power of the cloud.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | # AWS FINAL EXAMPLE  # 1 - Create an S3 bucket  # 2 - Make the S3 bucket secure - NO PUBLIC ACCESS  # 3 - Upload a file to the new S3 bucket  # 4 - Create a pre-signed URL to enable people to securely download the file worldwide    # Install the AWS.Tools S3 module to work with Amazon Simple Storage Service (S3)  Install-Module AWS.Tools.S3  Import-Module AWS.Tools.S3    # 1 - This command creates a new private bucket named "techthoughts".  # <https://docs.aws.amazon.com/powershell/latest/reference/items/New-S3Bucket.html>  New-S3Bucket -BucketName 'techthoughts'  Get-S3PublicAccessBlock -BucketName 'techthoughts' # by default the bucket will not have public access blocked    # 2 - Adjust public access to the bucket to BLOCKED  # <https://docs.aws.amazon.com/powershell/latest/reference/items/Add-S3PublicAccessBlock.html>  $publicAccessBlockSplat = @{      PublicAccessBlockConfiguration\_BlockPublicAcl       = $true      PublicAccessBlockConfiguration\_IgnorePublicAcl      = $true      PublicAccessBlockConfiguration\_BlockPublicPolicy    = $true      PublicAccessBlockConfiguration\_RestrictPublicBucket = $true      BucketName                                          = 'techthoughts'  }  Add-S3PublicAccessBlock @publicAccessBlockSplat  Get-S3PublicAccessBlock -BucketName 'techthoughts' # verify the new BLOCKED policy is in place    # 3 - Upload a local file on your drive to the new S3 bucket  $filePath = 'C:\rs-pkgs\techthoughts\_text\_file.txt'  $key = 'techthoughts\_text\_file.txt'  $bucketName = 'techthoughts'  #  [https://docs.aws.amazon.com/powershell/latest/reference/index.html?page=Write-S3Object.html&amp;tocid=Wri](https://docs.aws.amazon.com/powershell/latest/reference/index.html?page=Write-S3Object.html&amp;tocid=Wri       te-S3Object)  [te-S3Object](https://docs.aws.amazon.com/powershell/latest/reference/index.html?page=Write-S3Object.html&amp;tocid=Wri       te-S3Object)  # <http://iwantmyreal.name/s3-download-only-presigned-upload>  $writeS3Splat = @{      BucketName       = $bucketName      File             = $filePath      Key              = $key      HeaderCollection = @{          'Content-Disposition' = "attachment; filename=""$key"""      }  }  Write-S3Object @writeS3Splat    # 4 - Create a pre-signed URL to securely allow others to download the file for a set period of time  # <https://docs.aws.amazon.com/powershell/latest/reference/items/Get-S3PreSignedURL.html>  # This URL will be good for 1 day - 24 hours!  $url = Get-S3PreSignedURL -BucketName $bucketName -Key $key -Expire (Get-Date).AddDays(1) |

### Additional AWS PowerShell Resources

* [AWS Tools for PowerShell Documentation](https://docs.aws.amazon.com/powershell/index.html)
* [AWS Tools for PowerShell Cmdlet Reference](https://docs.aws.amazon.com/powershell/latest/reference/Index.html)
* [GitHub aws-tools-for-powershell](https://github.com/aws/aws-tools-for-powershell)

## Azure PowerShell

What you’ll need to get started:

* [Azure Account](https://docs.microsoft.com/learn/modules/create-an-azure-account/)
* [Azure PowerShell](https://github.com/Azure/azure-powershell)

PowerShell for Azure has evolved over time so lets dive quickly into the difference:

* Az module (**recommended**) – this is a replacement for AzureRM and AzureRM.Netcore and is now the primary PowerShell module for Azure. It supports PowerShell 5.1 and 6.1+.
* AzureRM module – officially maintained through December 2020 but will no longer receive new cmdlets or features.

### Installing Azure Az PowerShell

Because the Az module is the future, this blog post will focus on this version of the module.

Get started by running the following install command:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Install the Az module  Install-Module -Name Az |

[Installing the Az module](https://docs.microsoft.com/en-us/powershell/azure/install-az-ps) will install a [full list of modules](https://github.com/Azure/azure-powershell/blob/master/documentation/azure-powershell-modules.md) for all services within Azure.

There are many Azure Services, and the Az modules are split up to support each one! You can import all of the Azure cmdlets at once, or you can import just the cmdlets you need. As there are quite a few cmdlets you can save a lot of time by only engaging the Az modules you actually need.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6 | # Import all of the Az cmdlets - time in seconds  (Measure-Command {Import-Module Az}).Seconds  11  # Import just the Sql Az cmdlets - time in seconds  (Measure-Command {Import-Module Az.Sql}).Seconds  1 |

You can explore which Azure cmdlets are available by running the following:

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4 | # Get a list of Azure services supported by the Az modules  Get-Command -Noun Az\* | Sort-Object Source  # List all (read only) Get cmdlets that contain VM in the Az.Compute module  Get-Command -Verb Get -Noun AzVM\* -Module Az.Compute |

Azure operates with the concept of [Azure geographies](https://azure.microsoft.com/global-infrastructure/geographies/). Each geography contains one or more regions. You will likely want to deploy resources to the geography that is closest to you. Discover the available locations to deploy in Azure by running the following:

|  |  |
| --- | --- |
| Code example: | |
| 1  2 | # Get a list of available Azure locations to deploy resources to  Get-AzLocation | Select-Object DisplayName, Location |

With Azure you will not set a default region. Instead, you first create a container called a resource group in a region of your choosing. Then, all resources created in the resource group will also be located in that region.

### Authenticating to Azure using PowerShell

[Azure supports several PowerShell authentication methods](https://docs.microsoft.com/en-us/powershell/azure/authenticate-azureps). This post will cover one way, the interactive sign in. Use the [**Connect-AzAccount**](https://docs.microsoft.com/powershell/module/az.accounts/connect-azaccount?view=azps-4.5.0) cmdlet which will prompt you to launch a browser window and authenticate using your Azure account login.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8 | Import-Module Az.Accounts  # Connect to Azure with a browser sign in token  Connect-AzAccount  # If you have more than one subscription associated with your mail account, you can choose the default subscription.  Get-AzSubscription  Select-AzSubscription -Subscription "My Subscription"  # Verify your current and active subscription  Get-AzContext |

If your credentials have access to multiple Azure subscriptions you can retrieve a list of them using [**Get-AzSubscription**](https://docs.microsoft.com/powershell/module/az.accounts/get-azsubscription?view=azps-4.5.0). If you do have more than one, you can choose the one you want to work with using [**Select-AzSubscription**](https://docs.microsoft.com/powershell/module/servicemanagement/azure.service/select-azuresubscription?view=azuresmps-4.0.0).

Once you have authenticated via your browser window, you will have now have the ability to run commands against the active Azure subscription. If you are not sure which subscription you are currently under, you can always verify by running [**Get-AzContext**](https://docs.microsoft.com/powershell/module/az.accounts/get-azcontext?view=azps-4.5.0).

### Managing Azure Cloud with PowerShell

With your PowerShell session authenticated to your Azure account you can now engage the power of the Azure cloud. Again, the possibilities are nearly limitless. Using PowerShell you can create, manage, delete, monitor, and maintain Azure cloud resources around the world.

#### **Azure PowerShell Final Example**

In this final example we’ll use PowerShell to create a highly secure blob storage in Azure. Then we’ll upload a file to the blob, and create a pre-signed URL to the file. The URL will be good for 24 hours (configurable). This is a great way to share a file securely worldwide leveraging the power of the cloud.

|  |  |
| --- | --- |
| Code example: | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | # AZURE FINAL EXAMPLE  # 1 - Create a resource group to hold storage account  # 2 - Create a storage account  # 3 - Create a storage container  # 4 - Upload a file to the new storage container  # 5 - Create a pre-signed URL to enable people to securely download the file worldwide    #Resource Group  $resourceGroupName = "techthoughts"  $location = "westus"    # 1 - Create a resource group to hold storage account  # <https://docs.microsoft.com/powershell/module/az.resources/new-azresourcegroup?view=azps-4.4.0>  $newResourceGroupSplat = @{      Name     = $resourceGroupName      Location = $location  }  New-AzResourceGroup @newResourceGroupSplat    # 2 - Create a storage account  # <https://docs.microsoft.com/powershell/module/az.storage/new-azstorageaccount?view=azps-4.4.0>  $storageAccountName = 'techthoughtsstorage'  $newStorageAccountSplat = @{      ResourceGroupName = $resourceGroupName      AccountName       = $storageAccountName      Location          = $location      SkuName           = 'Standard\_LRS'      Kind              = 'StorageV2'  }  New-AzStorageAccount @newStorageAccountSplat    # 3 - Create a storage container  # <https://docs.microsoft.com/powershell/module/az.storage/get-azstorageaccount?view=azps-4.4.0>  $getStorageAccountSplat = @{      ResourceGroupName = $resourceGroupName      Name              = $storageAccountName  }  $storageContext = Get-AzStorageAccount @getStorageAccountSplat  # <https://docs.microsoft.com/powershell/module/az.storage/new-azstoragecontainer?view=azps-4.4.0>  $containerName = 'techthoughtscontainer'  $newStorageContainerSplat = @{      Context    = $storageContext.Context      Name       = $containerName      Permission = 'Off'  }  New-AzStorageContainer @newStorageContainerSplat    # 4 - Upload a file to the new storage container  # <https://docs.microsoft.com/powershell/module/az.storage/set-azstorageblobcontent?view=azps-4.4.0>  $setAzStorageBlobSplat = @{      Context   = $storageContext.Context      Container = $containerName      File      = 'C:\rs-pkgs\techthoughts\_text\_file.txt'      Blob      = 'techthoughts\_text\_file.txt'  }  Set-AzStorageBlobContent @setAzStorageBlobSplat    # 5 - Create a pre-signed URL to enable people to securely download the file worldwide  # <https://docs.microsoft.com/powershell/module/az.storage/new-azstorageblobsastoken?view=azps-4.4.0>  $newStorageSASSplat = @{      Context    = $storageContext.Context      Container  = $containerName      Blob       = 'techthoughts\_text\_file.txt'      ExpiryTime = (Get-Date).AddDays(1)      Permission = 'r'      FullUri    = $true  }  $url = New-AzStorageBlobSASToken @newStorageSASSplat |

### Additional Azure PowerShell Resources

* [Azure PowerShell documentation](https://docs.microsoft.com/powershell/azure/?view=azps-4.5.0)
* [Introducing the new Az module](https://docs.microsoft.com/powershell/azure/new-azureps-module-az?view=azps-4.5.0)
* [azure-powershell GitHub](https://github.com/Azure/azure-powershell)

## Google Cloud Tools for PowerShell

*Currently the project is in maintenance mode and we do not plan to support adding any new cmdlets. Pull requests are welcomed though if you want to add new cmdlets for this.*

As you can see on [***this issue***](https://github.com/GoogleCloudPlatform/google-cloud-powershell/issues/638#issuecomment-562287952) from the google-cloud-powershell GitHub repo, Google has elected to abandon PowerShell support moving forward.

Google Cloud also has the [lowest market share and lowest growth rate](https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/) of the three major cloud providers. Coincidence? You decide.

The Google Cloud Tools for PowerShell, such as they are, do function and you can leverage them to manage some resources in Google Cloud. As they are end of life though, this post will not dive into details of their use.