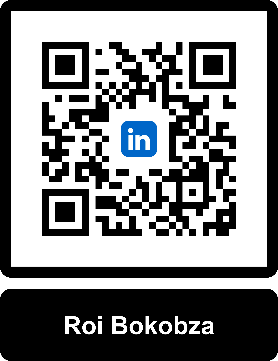
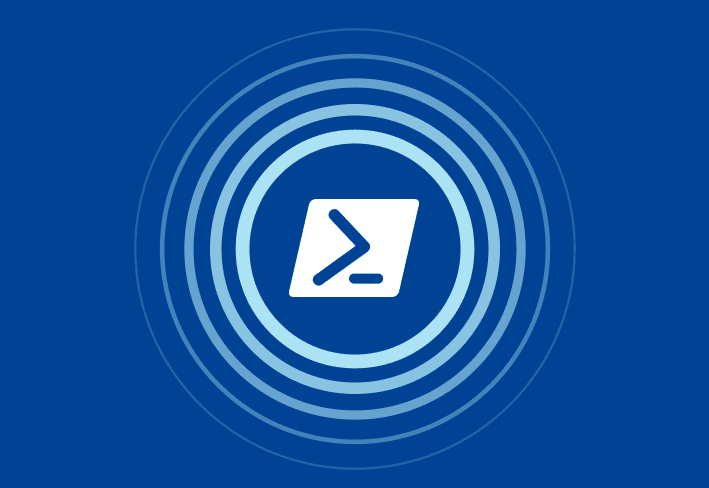


PowerShell Fundamentals

By Roi Bokobza



Roibokobza017@Gmail.com

**Table of Contents**

**Introduction**

* **What is PowerShell?.............................................................3**
* **PowerShell Features…………………………………………………….5**

**Cmdlets**

* **What are cmdlets?.................................................................7**
* **Examples of common cmdlets……………………………………….8**

**Objects and Pipeline**

* **Working with objects……………………………………………………..9**
* **Chaining cmdlets with the pipeline………………………………..10**

**Aliases**

* **What are Aliases?..................................................................11**
* **Create Aliases………………………………………………………………13**

**Files and Directory management**

* **Basic commands………………………………………………………….14**
* **Creating Files and Directories……………………………………….15**

**Variables and data types**

* **What are variables?..............................................................22**
* **Data types…………………………………………………………………..27**

**Scripting and Automation**

* **Writing scripts…………………………………………………………….32**
* **Functions and parameters………...…………………………………39**

**Advanced Topics**

* **Modules, remote management, and execution policies.…40**

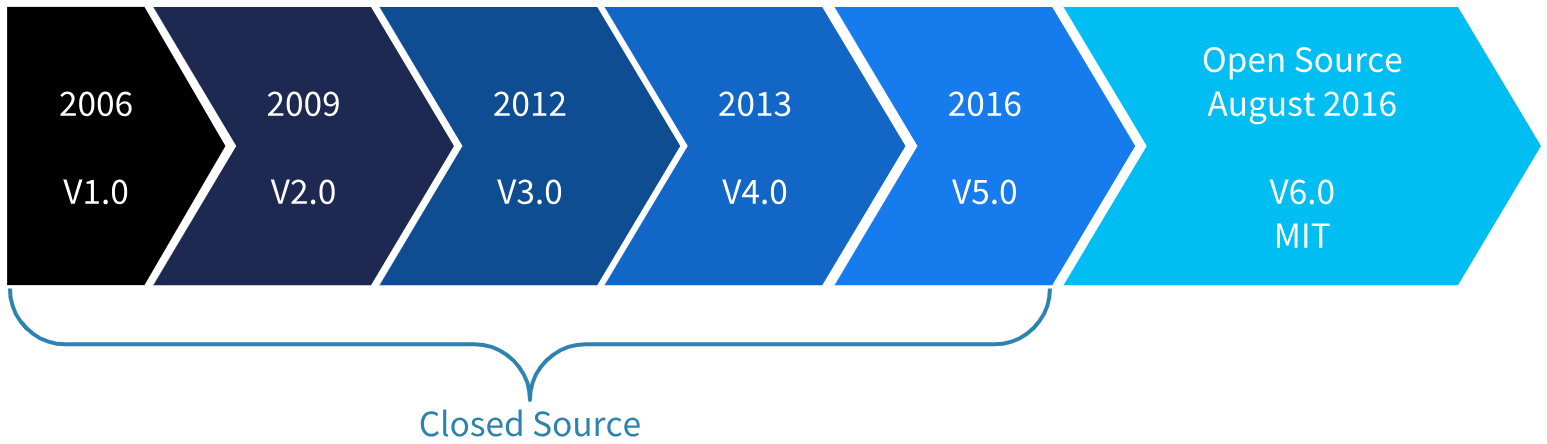
**Introduction**

**What is PowerShell?**

**PowerShell is a cross-platform scripting language and shell developed by Microsoft. It originally launched in 2006 as a command-line tool for Windows, designed to help system administrators automate tasks and manage configurations.**

**In 2016, PowerShell Core was introduced as an open-source, cross-platform version, allowing it to run on macOS and Linux as well. The transition to PowerShell 7 in 2020 unified both the original PowerShell and PowerShell Core, and it continued the cross-platform support.**

**For this Project I will be using PowerShell 7.4 on Windows 11.**



**PowerShell Attributes:**

* **PowerShell is a programming language used to automate tasks on both local and remote computers.**
* **Initially, it was developed as a framework to automate administrative tasks in Windows.**
* **Over time, PowerShell has evolved into a cross-platform tool that supports various types of tasks.**
* **PowerShell comprises two main components: a command-line shell and a scripting language.**
* **It was created as a task engine that utilizes** [**cmdlets**](#cmdlets) **to simplify task execution.**
* **With PowerShell, you can perform tasks such as managing users and automating workflows.**
* **It can be used for a variety of operations, including managing cloud resources and supporting continuous integration and delivery (CI/CD).**
* **PowerShell provides numerous built-in commands, and its functionality can be extended at any time by installing additional modules.**

**Features**

**Features Similar to Traditional Shells:**

* **Built-in Help System: PowerShell includes a help system that provides details about commands and integrates with online documentation. *Get-Help <command>***
* **Pipeline Support: Like traditional shells, PowerShell allows chaining multiple commands, where the output of one serve as the input for the next. *|***
* **Aliases: PowerShell supports alternative command names, enabling users to execute common commands using familiar shortcuts like cls (to clear the screen) and ls (to list files).**

**Differences from Traditional Command-Line Shells:**

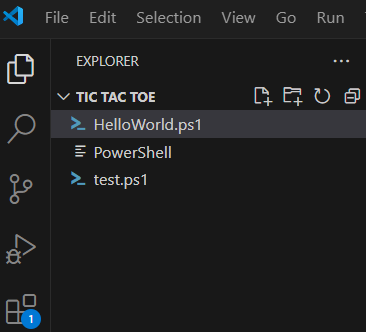
* **Object-Based Processing: PowerShell works with** [**objects**](#Objects) **instead of text, minimizing formatting and extraction efforts.**
* **Cmdlets: PowerShell commands (cmdlets) run within a shared runtime, unlike standalone executables in other shells.**
* **Multiple Command Types: PowerShell supports executables, cmdlets, functions, scripts, and** [**aliases**](#Aliases)**.**
* **PowerShell is NOT case sensitive.**

**PowerShell Types**

 PowerShell (Console): Lightweight  
 Command line interface where you execute  
 Commands one at a time.  
 Ideal for quick tasks and automation.

PowerShell ISE: A GUI (Graphical user  
 interface.)   
 Includes features like multi-line editing  
 and debugging tools, making it better for  
 Writing and testing scripts.

I will be using the PowerShell console for this Project, for later topics I will be using PowerShell on Visual Code Studio (It's recommended to use Visual Code but it's not a must.)  
Create a PowerShell File on VCS:



Click on this icon.

Name your new file with a .ps1 extension for it to be recognized as a PowerShell Script.

**Cmdlets**

**Cmdlets (pronounced "command-lets") are single-purpose and lightweight commands in PowerShell.   
They are the building blocks of PowerShell and are designed to follow a verb-noun naming convention, for example; *Get-Process, Set-Location, New-Item)*.**

**Key Characteristics:**

**Verb-Noun Naming:**

* **Verbs describe the action, like; Get, Set, Remove etc.**
* **Nouns describe the object being acted upon; Process, Location, Item.**
* **Example: Get-Process retrieves information about running processes.**

**Pipeline Support:**

* **Cmdlets can be chained together using the pipeline (|) to pass output from one cmdlet as input to another.**

**Object Oriented:**

* **Cmdlets return .NET objects, not plain text. This allows for powerful manipulation and filtering of data.**

**Cmdlets Examples:**

|  |  |
| --- | --- |
| Cmdlets | Description |
| Get-Process | **Retrieves information about running tasks.** |
| Get-Content | **Retrieves the contents of a file.** |
| Write-output | **Sends output to the console or** [**Pipeline**](#Objects)**.** |
| Remove-Item | **Deletes a File or Directory.** |
| New-Item | **Creates a File or Directory.** |
| Set-Location | **Change the current directory you are on.** |

What are Objects?

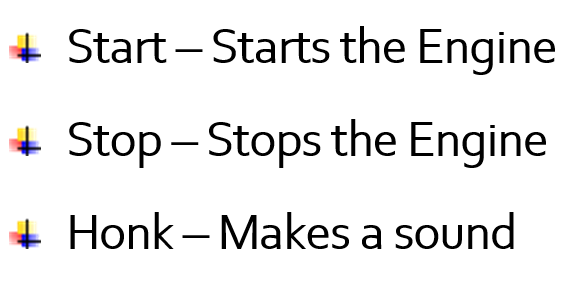
**Objects and Pipelines**

**In PowerShell, everything is an object. An object is a 'container' that holds information (properties) and actions (methods).   
For example:**

* **A file object has properties like Name, Size, and LastModified.**
* **A process object has properties like Name, Id, and CPUUsage.**

**Let's take a car for an object:**

**Properties: Methods:**

* **Color: Red**
* **Make: Toyota**
* **Model: Corolla**
* **Year: 2020**

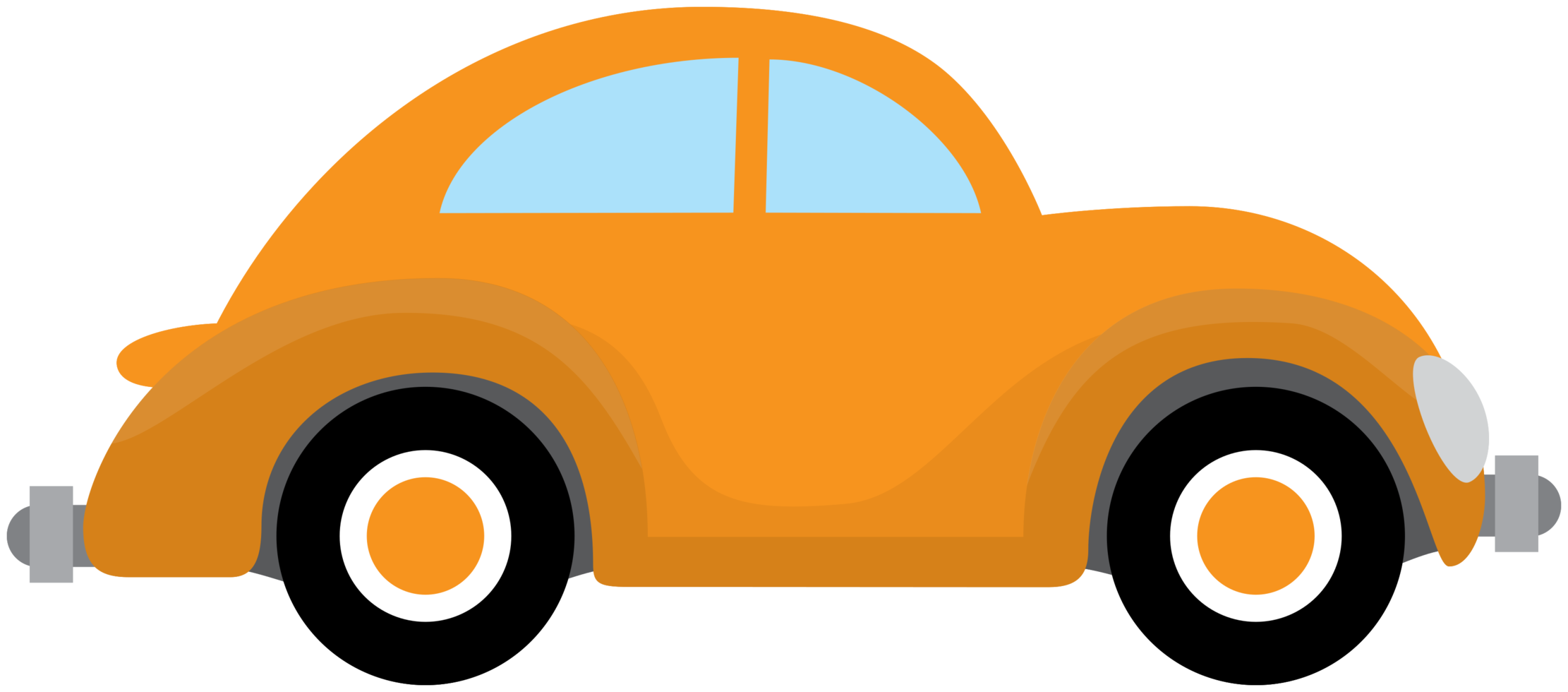
**Properties: These are the details about the car.**

Methods: These are actions performed by the car**.**

Object:

Properties

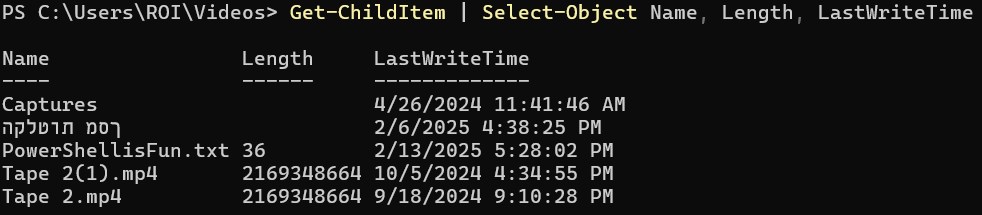
Methods



**Chaining cmdlets with Pipeline:**

**Pipes ( | ) is a way to pass the output of one cmdlet to another, using pipes help with chaining commands together that perform complex tasks in a single line.**

**Let's demonstrate how Objects and Pipeline work in PowerShell with a simple command:**

****

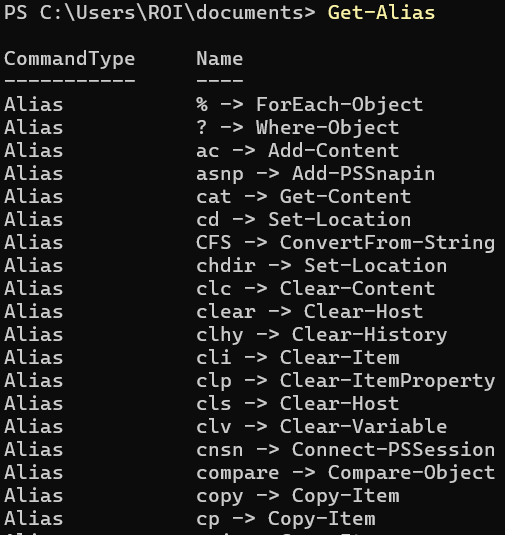
Get-ChildItem – Retrieves all the files and folders in the current directory. (ls)

| (Pipeline) – Pipes the output of the previous command (in this case, 'GetChildItem') and passes it as input to the next command.

**Select-Object Name, Length, LastWriteTime – This command takes the Object passed through the pipeline and selects specific properties to display (Name, Length, and last time it was modified).**

**Aliases are defiantly one of the coolest features on PowerShell is being able to give Aliases to commands to keep things simpler to type and remember.**

**Aliases**

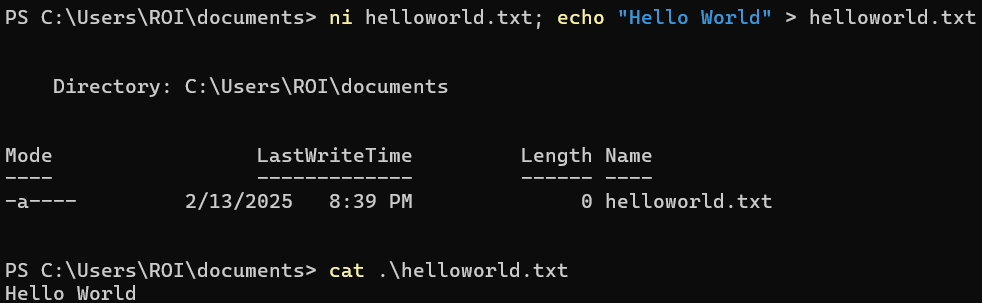
**By typing *'Get-Alias'* we are shown all Aliases that are already built into PowerShell, if you are familiar with Linux's bash or Unix you might recognize some of these Aliases that will make you feel right at home.**

**To name a few:**

|  |  |  |
| --- | --- | --- |
| Alias | Cmdlet | Description |
| ls | **Get-ChildItem** | **Lists files and directories.** |
| rm | **Remove-Item** | **Deletes files or directories.** |
| cp | **Copy-Item** | **Copies files or directories.** |
| mv | **Move-Item** | **Moves files or directories.** |
| cat | **Get-Content** | **Display the contents of a file.** |
| echo | **Write-output** | **Outputs text to console.** |
| cd | **Set-Location** | **Changes the current directory.** |

**Write a command using Aliases:**

**This simple command tells the system to make a new item, call it "helloworld.txt" and put the text "Hello World" inside it.**

**Here is a breakdown:**

* **ni – Alias for *'New-Item'.***
* **helloworld.txt – the name I chose for this file, and a txt format.**
* **; -- separates the two commands in one line**
* **echo "Hello World" – Alias for *'Write-Output'*.**
* **'>' -- the alias for *'Out-File'*.**
* **Writes "Hello World" into helloworld.txt.**

**You can make the command even simpler by typing:**

***echo "Hello World" > helloworld.txt***

**Finally use 'cat <file-name>' to verify if the file has been created with the output inside.**

**Make your own Aliases:**

**This is the cool part of aliases: if you're familiar with another programming language or shell, you can feel right at home by using PowerShell's alias tools to change commands into more relatable terms to give the shell your personal touch!**

Get-ChildItem

*****New-Alias -Name list -Value Get-ChildItem***

List

* **Use the New-Alias cmdlet to create a new alias.**
* **- Value tells PowerShell: *"When I use this alias,   
  run this command or script."***

**Aliases can also be used for scripts, for example, you  
have a script called "C:\Scripts\whoami.ps1".**

**Give it an Alias by typing:**

***New-Alias -Name backup -Value "C:\Scripts\whoami.ps1"*  
Now, you can run the script by simply typing 'backup'.**

**New-Alias are temporary and only last for the current  
PowerShell session.  
To make them permeant add them to your PowerShell profile which I will show** [**how to do that**](#Scripting) **later in the guide.**

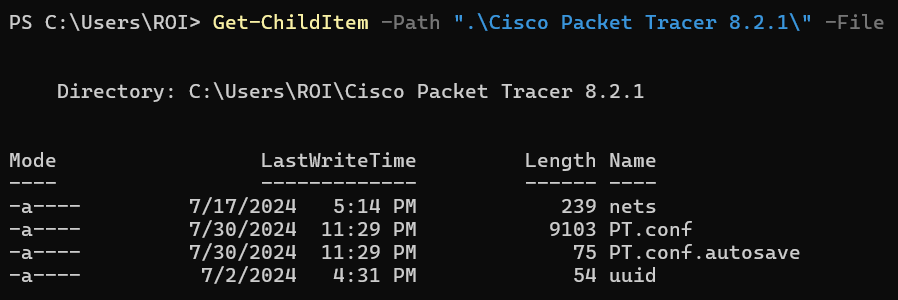
**File and Directory Management is one of the most commonly used aspects of PowerShell, as it allows you to create, read, update, delete, and organize files and folders.**

**File and Directory Management**

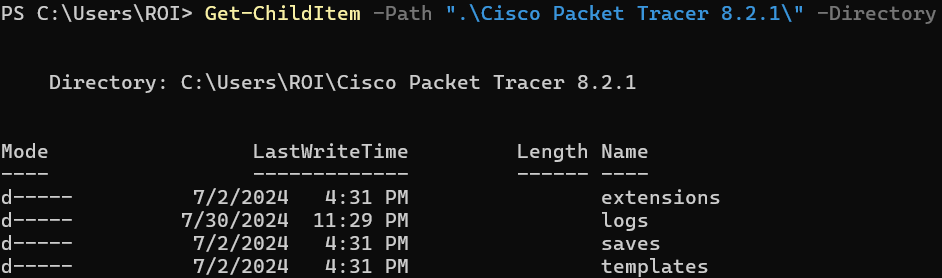
**Get-ChildItem (or its aliases: ls, dir, gci): Lists all files and folders in the current directory.**

**Retrieve content from a file in a specific path and filter by file type:**

**Filters by –File**

****

**Filters by –Directory**

****

**Creating Files and Directories:**

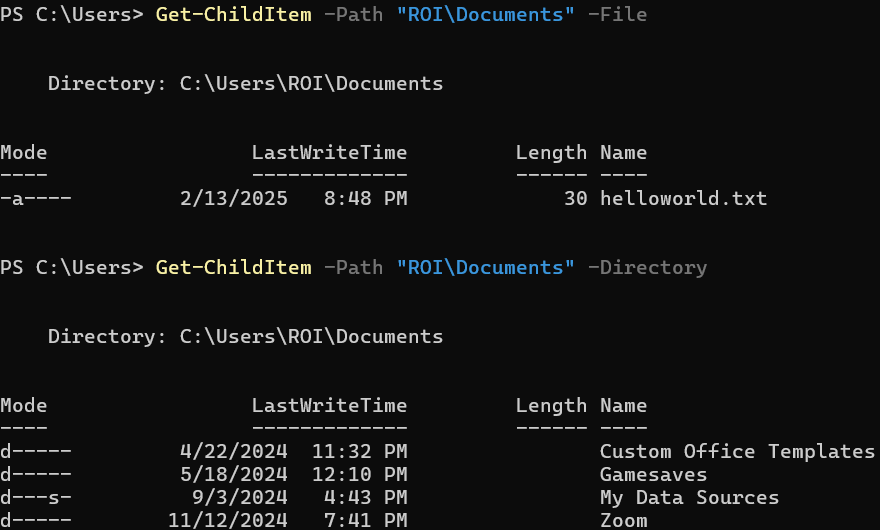
**Use Get-ChildItem (or its** [**aliases**](#Aliases)**ls, dir, gci) to list files and folders in a directory.**

**1. Listing Files and Directories:**

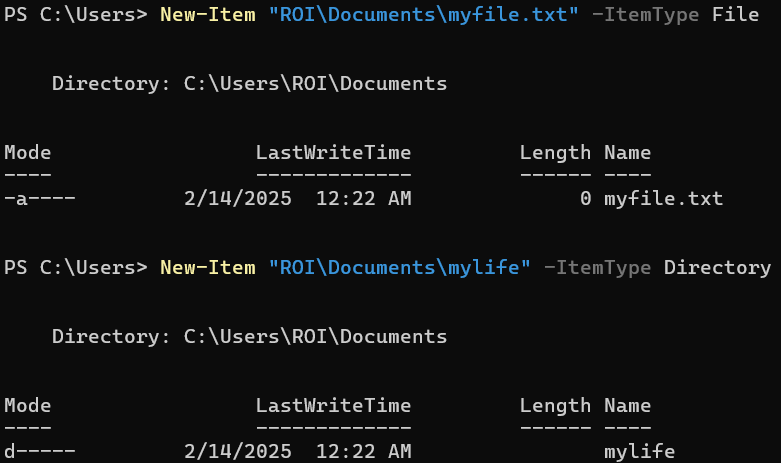
***Get-ChildItem -Path <Your-path> -type***

**Filter for files or directories simply by specifying -File or -Directory**

**\*Using '-path' is not mandatory, the first positional parameter is often -Path. This means you can just provide the path directly without typing -Path.**

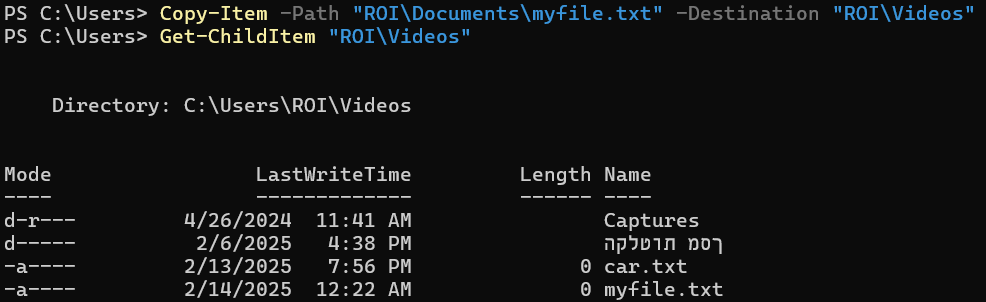
****

**2. Creating Files and Directories:**

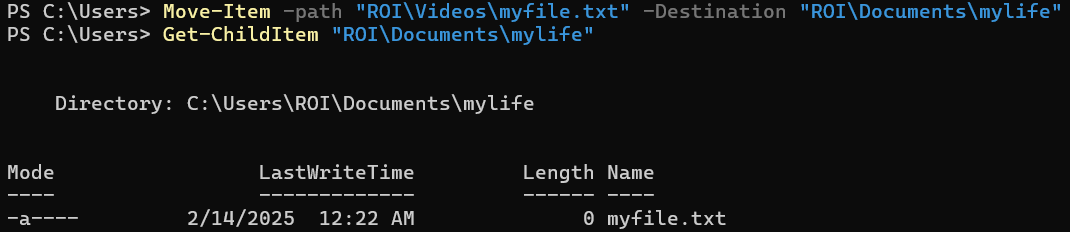
****

* **New-Item – Creates a new file or directory (ni alias)**
* **-ItemType – Specify the type of item you want to create.**
* **To create an item on the current directory simply put the name for your Item in quotations where the path is.  
  Example: *New-Item "myfile.txt" -ItemType File***

**3. Copying and moving items:**

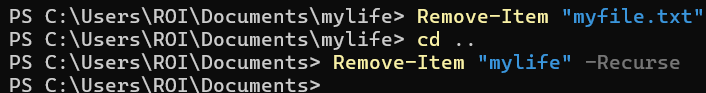
****

* **Copy-Item (or its alias cp) – Copies files and folders.**
* **- Destination – Copy destination.**

****

* **Move-item (or its alias mv) – Moves files or directories.**

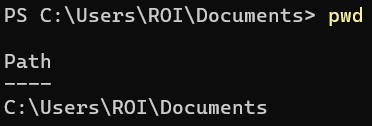
**4. Deleting files and directories:**

****

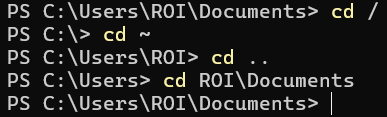
* **Remove-Item (its alias rm) – Removes files or directories**
* **-Recurse – Deletes a folder with its contents.**

**5. Change directories (cd):**

**When creating or removing files and folders we must understand how to navigate between them, sure we can write their path and make changes to them, but its as important to know how to reach them via the console.**

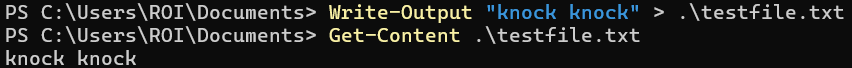
**pwd – Shows the path you are currently on.** ****

**cd – changes the current working directory.**

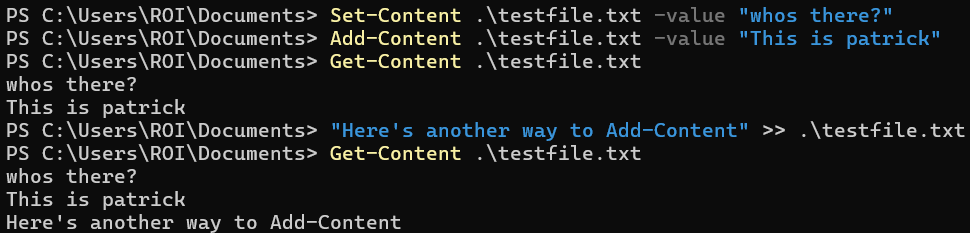
****

* **cd / -- Go to the root folder.**
* **cd ~ -- Go to your home folder.**
* **cd .. Move up one directory level.**
* **cd C:\Users\Home – Go to a specified path.**

**6. Reading and Writing File Content:**

****

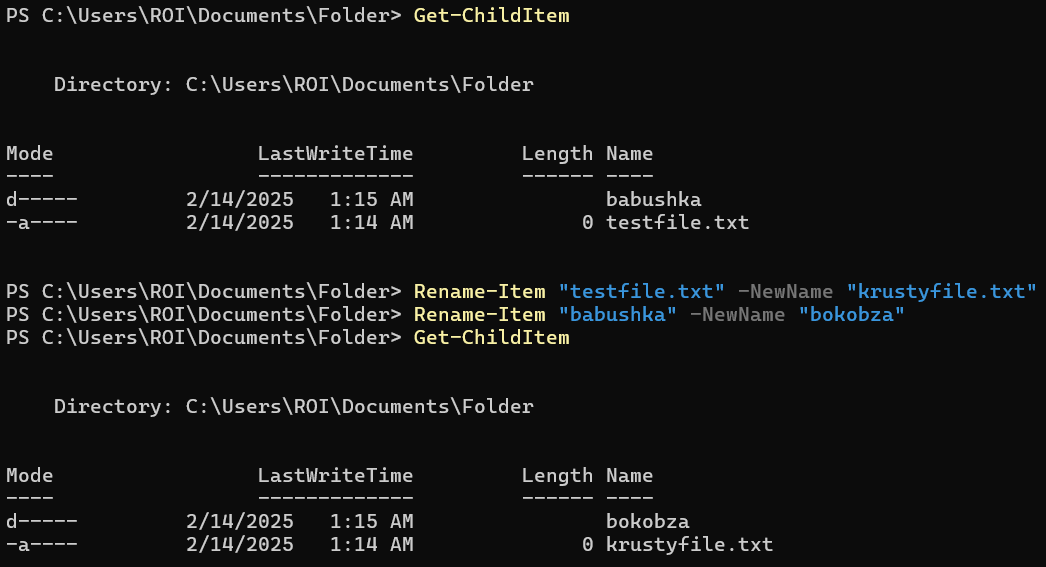
* **I used *'Write-Output'* to put the text "knock knock" into (>) testfile.txt.**
* **Then *'Get-Content'* to view the contents of the file.**

****

* **-Value: Specifies the text to write into the file.**
* **>> (Append Operator): Adds text to the file without deleting existing content.**
* **Set-Content: Overwrites the file with new text ("whos there?" replaced "knock knock").**
* **Add-Content: Adds new text ("This is patrick") below the existing content.**

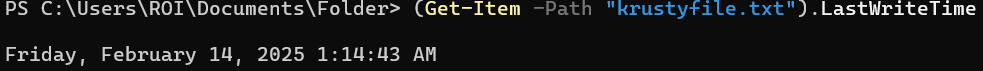
**7. Renaming Files and Directories**

**Use Rename-Item to rename files or folders.**

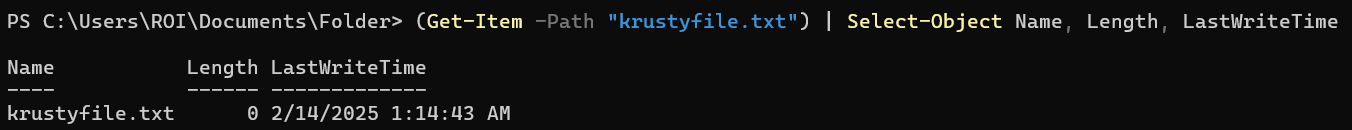
****

**8. Checking File and Directory Properties**

* **Use *'Get-Item'* to retrieve properties like Name, Length (size), and LastWriteTime.**
* **The parentheses execute *Get-Item* first, then access its properties. Without them, PowerShell wouldn’t know what object you're working with.**
* **LastWriteTime – The one property I asked 'Get-Item' to display.**

****

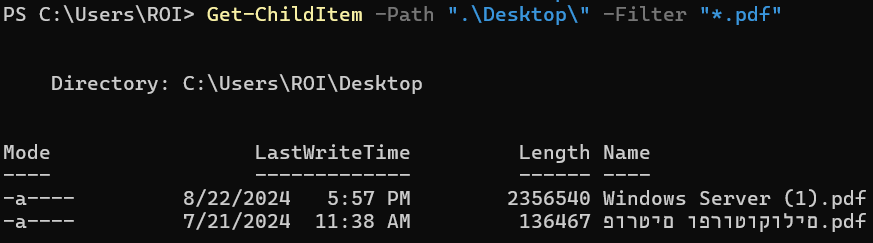
**>>>**

****

* **Display multiple items by Writing *Get-Item*, piping the output to the command *Select-Object*.**
* ***Select-Object* is a cmdlet that picks specific properties from an object Instead of showing all details, it lets you filter what you want to see.**

**9. Searching for files:**

**Find files by name:**

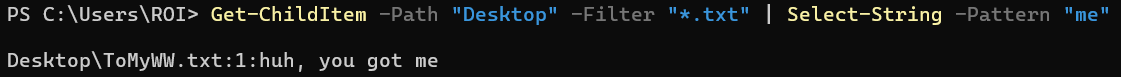
****

* ***Get-ChildItem* (gci or ls) lists files/folders.**
* **-Path specifies where to search.**
* ***-Filter "\*.pdf"* finds only .pdf files (wildcard *\** means any name).**

**You can search and filter like this with all extensions (.txt .exe etc..)**

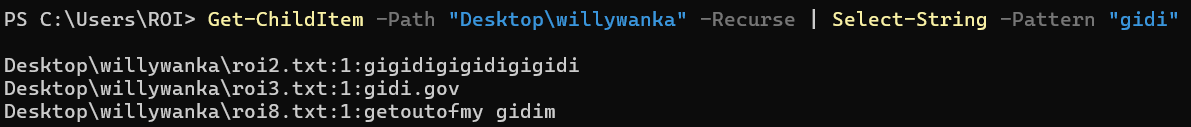
**>>>**

**Search for text in Files:**

****

***Get-ChildItem -Path "Desktop" -Filter "\*.txt"* finds all .txt files on the Desktop and pipes them to *Select-String -Pattern "me",* which scans their contents for the word "me" and returns matching lines with the file name and line number. The pipeline ensures files are first located and then searched, making it a two-step process.**

**Search for text in folders and subfolders:**

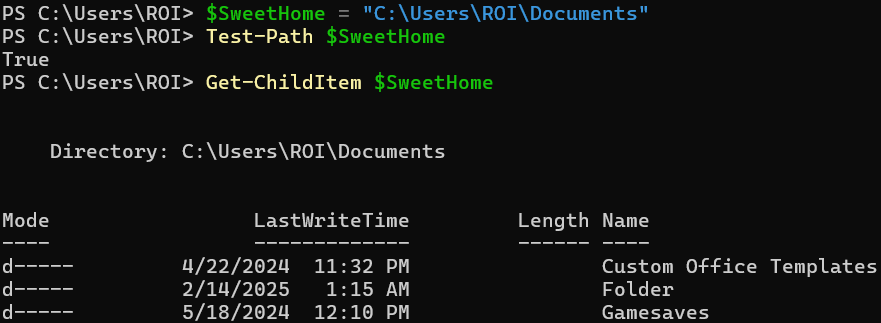
****

**This command searches for the term "gidi" in all files within the *Desktop\willywanka* folder and its subfolders *(-Recurse),* and *Select-String* filters the content for the pattern "gidi".   
The pipeline (|) connects the two, passing file data from Get-*ChildItem* to Select-String for searching.**

**10. Working with file paths:**

**What it does?  
Manages and navigates file locations efficiently.**

**Why use it?  
Ensures commands run correctly, avoids errors, and supports automation.**

****

***'$SweetHome = "C:\Users\ROI\Documents'***

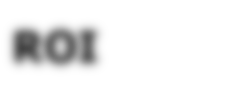
* **This creates a** [**variable**](#Variables) **($SweetHome) and assigns it the path "C:\Users\ROI\Documents".**
* **This makes it easier to reference the path later without typing it every time.**

***Get-ChildItem $SweetHome***

* **Get-ChildItem (alias: ls) lists files and folders.**
* **Since $SweetHome holds "C:\Users\ROI\Documents", it retrieves all items in that directory.**

**Variables and DATA types**

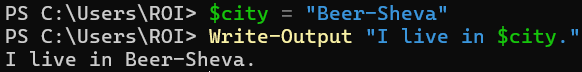
**What are variables?**

**A variable in PowerShell can be looked at as a  
small storage unit meant to store data under a  
name of your choice.**

**$name**

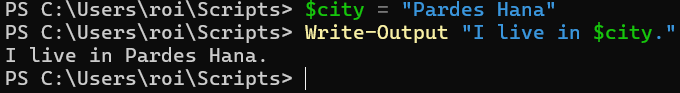
**Why use variables?**

**Variables make scripts more reusable by storing values that can change. Instead of typing the same data multiple times, you can use a variable.**

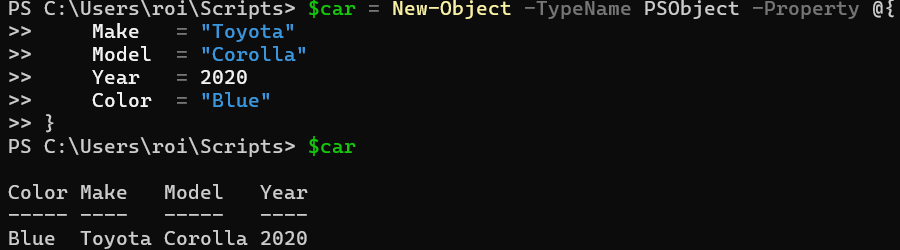
****

***$city = "Beer-Sheva"* now $city holds the value "Beer-Sheva".**

**Change the Variable simply by typing it again with a different value.**

****

**Store an Object within a Variable:  
I will now store the Object "car" I previously talked about in the variable $car;**

****

**$car – This is a variable and it's going to store the object we're creating.**

***New-Object* – This cmdlet is used to to create a custom object.**

***-TypeName PSObject* – This specifies the type of object you're creating. PSObject is a PowerShell object type that lets you add properties dynamically.**

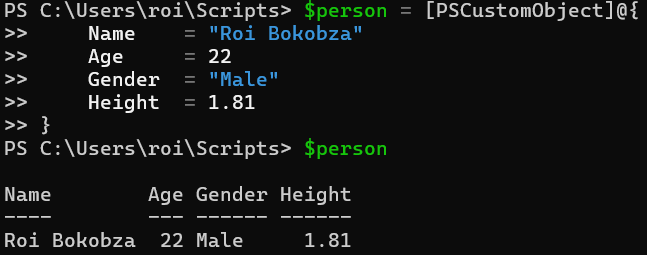
***-Property @{ ... }* – Here, you're defining the properties of the object. Inside the @{} is a** [**hashtable**](#hashtables)**, where you define the keys (properties) and values (data).**(To space-down without executing the command use Shift + Enter)

* **Make = "Toyota": The key Make gets the value "Toyota".**
* **Model = "Corolla": The key Model gets the value "Corolla".**
* **Year = 2020: The key Year gets the value 2020.**
* **Color = "Blue": The key Color gets the value "Blue".**

**Let's now use a variable to create a [PSUCustomObject]:**

***[PSCustomObject]:* This is a shorthand for creating a custom object. It's a cleaner and preferred way to create objects in PowerShell compared to New-Object.**

**I will create a Variable called $Person and store the properties of: Name, Age, and Gender and Height.**

****

***PSCustomObject* is great for quickly creating simple objects with dynamic properties when you need lightweight data.**

**On the other hand, classes (the previous method) are better when you need a structured and reusable blueprint with methods and type safety for more complex or long-term data handling.**

**The reason strings (like "John", "Male") are in quotes is because they're text data types in PowerShell, so you wrap them in quotes to signify that they're strings. Meanwhile, numbers (like 22, 1.81) don’t need quotes because they're considered numeric data types (integers or floats), and PowerShell recognizes them as numbers without needing quotes.**

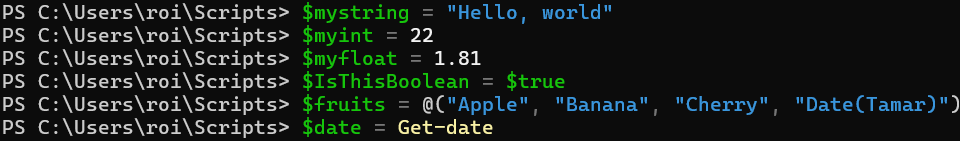
**This leads perfectly into the next topic: data types in PowerShell!**

**Data Types in PowerShell:**

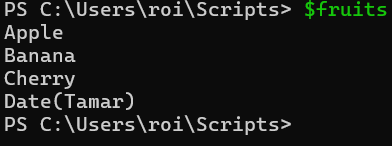
**in PowerShell, data types are like the foundation of your data. they tell PowerShell what kind of information you're working with—whether it's numbers, text, true/false values, or even more complex things.**

**You got a few key types you'll be using all the time:**

* **Strings ("Hello, world!") for text.**
* **Integers (22, 2025) for whole numbers.**
* **Floats (1.81) - Decimal numbers, for precise values.**
* **Booleans ($true, $false) for yes/no.**
* **Arrays (like lists) to hold multiple values.**
* **DateTime to work with dates and times.**

****

**I created a variable with each data type, to run them I will simply type $<my-variable-name>.**

****

**Arrays (lists):**

**arrays are collections of multiple items stored in a single variable. They let you keep a list of values—whether numbers, strings, or objects—together. You can access any item using its index (starting from 0).**

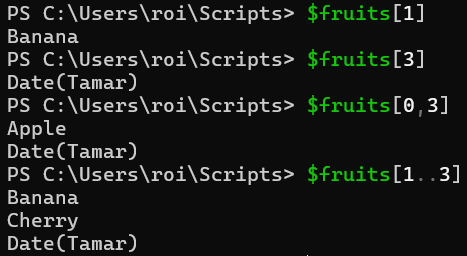
0 1 2 3

**My array (fruits): ["Apple", "Banana", "Cherry", "Date"].  
Each item in the array has an ascending index, starting from 0 (so 'Apple' would be at index 0).**

**Create an Array(@()):**

**It's a fixed-size collection, meaning once you create it, you can't change its contents.**

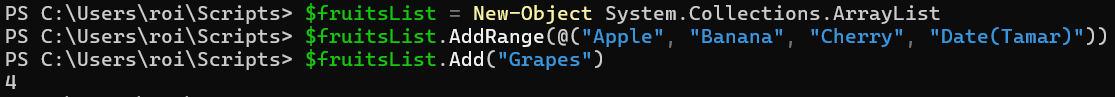
**Display specific items: inside of the square brackets you can pick an index number of an item inside of the array list to display, you can pick multiple items with [0,3] and get a range of items with [1..2]**

****

**Create an ArrayList:**

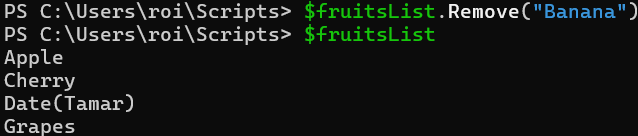
* **It's a dynamic collection, meaning you can add, remove, and modify items easily.**
* **You use *New-Object System.Collections.ArrayList* to create an ArrayList, and it allows for more flexibility in handling data.**

**Add items:**

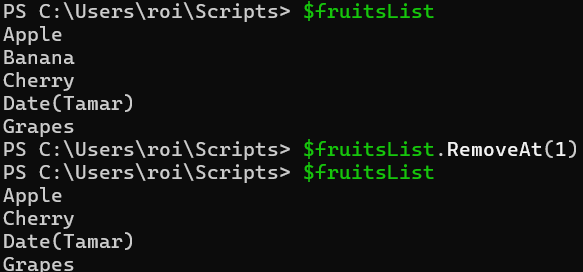
****

* **Use *New-Object System.Collections.ArrayList* to create an ArrayList (its empty as of now).**
* ***AddRange(@())* takes a collection (like an array) and adds all the items in that collection to the ArrayList.**
* **Add adds an item to the ArrayList.**

**Remove Items:**

****

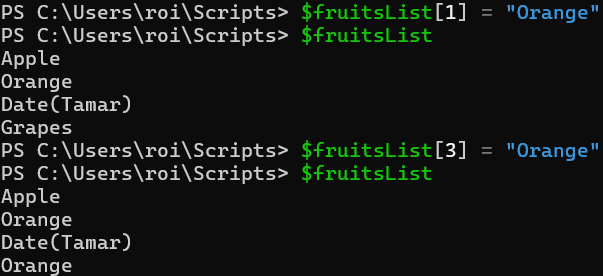
**Remove by name type; *Remove("name")***

****

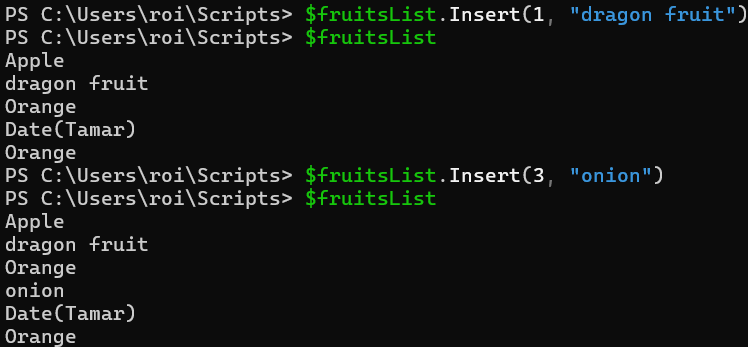
**Remove by index, type; *RemoveAt(number)***

**Replace Item:**

**Inside the square brackets put the index number of the item you want to replace.**

****

**Insert Item:**

****

**type *insert(index-number, "name")* to insert an item at the requested index number.**

**Booleans:**

**Booleans are simple true/false values, and they’re powerhouses when it comes to making decisions in your code. use them to control the flow of your program based on conditions.**

**$True – means something is correct.**

**$False – means something is incorrect.**

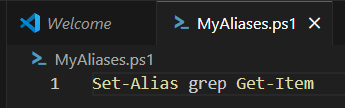
**You can use this simple terminology to apply conditions to your code, for example; if something is true execute a specific command or   
vice-versa.**

**This is another perfect segue to the next Subject!   
Scripting and automation.**

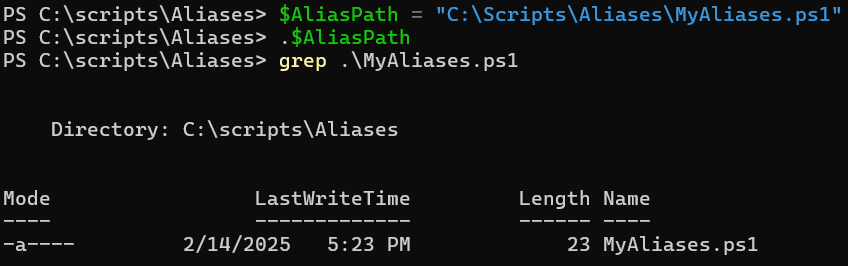
**Scripting and Automation**

**Script is a file containing a series of PowerShell commands, saved with a .ps1 extension. Instead of typing commands one by one, you can automate tasks by running the script.   
Scripts are used for things like managing files, setting up configurations, automating system tasks, and much more.**

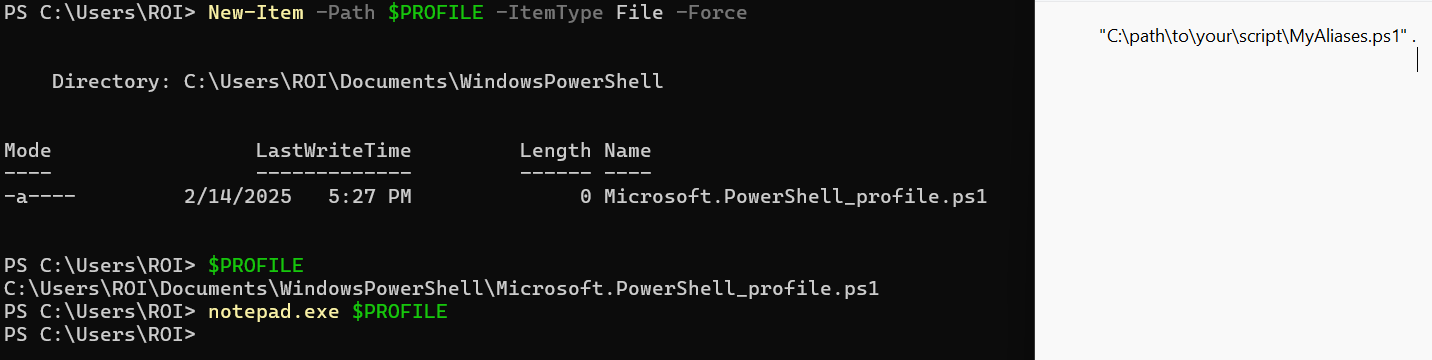
**For this topic, I will be running PowerShell on** [**Visual Code Studio**](#powershellVCS)**.**

****

**I created a script named "MyAliases.ps1" the extension .ps1 makes this a script, inside of the script I inserted the command Set-Alias and created the alias 'grep' for 'Get-Item'.**

****

**I ran the script on my PowerShell Console to see if it works.**

****

**However, PowerShell won't remember this new alias in the next session, so I needed to automate a script that will apply my aliases every time a new session starts with PowerShell, First I had to create a PowerShell profile by typing   
*New-Item -Path $PROFILE -ItemType File -Force  
. "C:\Scripts\Aliases\MyAliases.ps1"***

***Execute this command if you do not have a PowerShell Profile.***

***$PROFILE* Will direct you to the path of you PowerShell profile, there I opened the file with *notepad.exe* Make sure to use the full path to your script and wrap it in quotes.**

**After adding the line, save and close the file.**

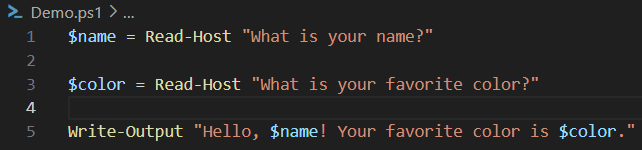
**now the script will load automatically with every PowerShell Session.**

**This was the simplest demonstration of Scripting and Automating.**

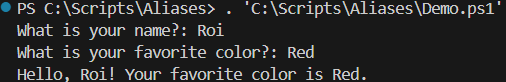
**Now I will involve Variables, Arrays, Booleans, hashtables and different data types.**

***'Read-Host'* Prompts the user for input and stores it in a variable.**

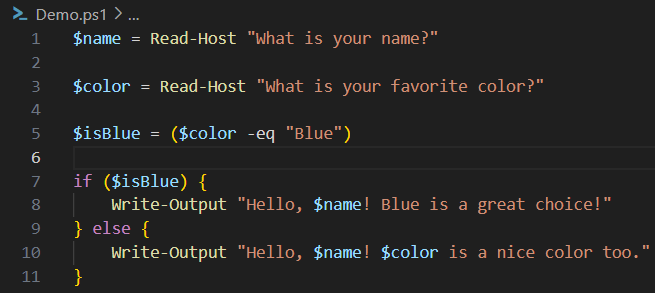
**Your input assumes the place of the variable.**

****

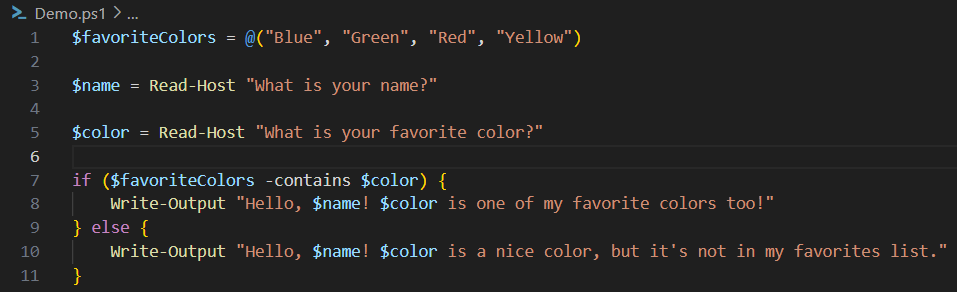
**The input provided by the user is stored in variables and displayed in the final output.**

****

**Let’s level up the script by adding conditional logic (if, else) to respond differently based on the user’s input.   
We’ll also use a boolean to check if the user’s favorite color is "Blue."**

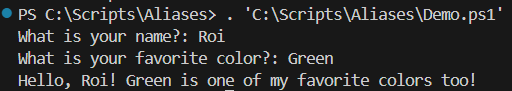
****

* **Boolean ($isBlue): Checks if the user’s favorite color is "Blue" (case-insensitive).**
* **Conditional Logic (if, else): Displays a different message based on the value of $isBlue.**

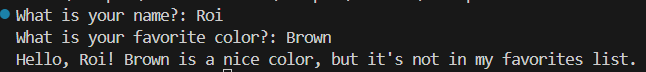


* ***Array ($favoriteColors):* Stores a list of favorite colors.**
* ***Read-Host*: Prompts the user for input and stores it in variables ($name, $color).**
* **-contains Operator: Checks if the user’s input ($color) is in the array.**
* ***Conditional Logic (if, else):* Displays a different message based on whether the color is in the array.**

**If your color is in the array**

****

**Else:**

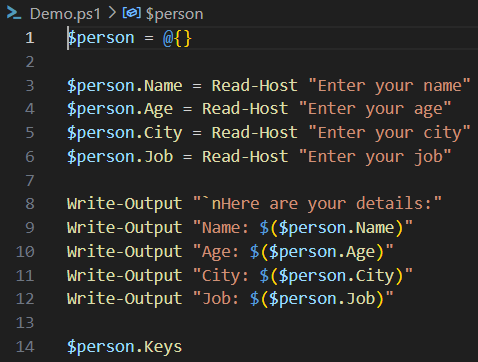
****

**Hashtables:**

**A hashtable in PowerShell is a collection of key-value pairs, similar to a dictionary in other languages.   
Each key acts as an identifier for a value, allowing for quick lookups.**

**Key value pairs are a way to store data where each "key" is unique and points to a specific "value." The key is like a label, and the value is the information linked to that label.   
For example, a key "name" might have a value "Roi".**

**Here, "Name" and "Age" are keys, while you input is their corresponding values.**

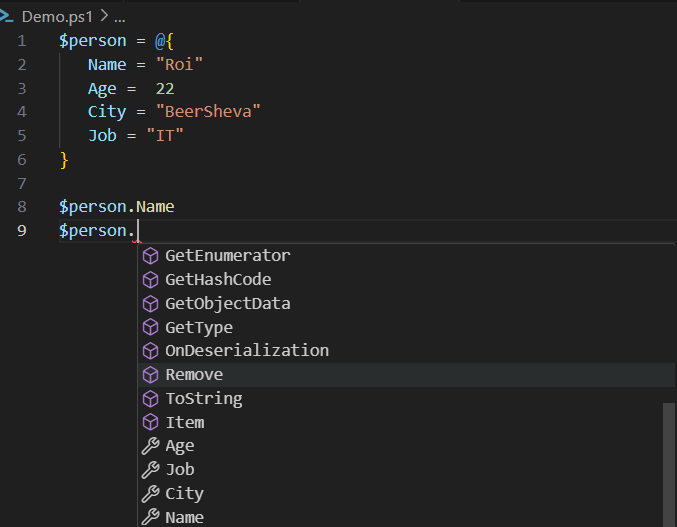
****

***@{} =* Creates an empty hashtable $person**

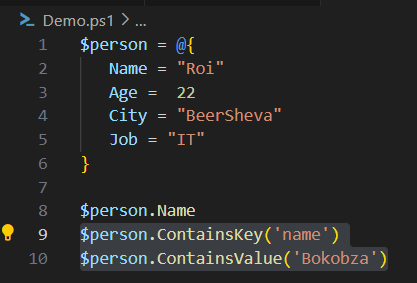
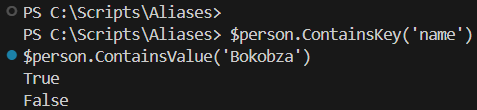
***$Person.Keys* retrieves and lists all the keys in the hashtable (i.e., Name, Age, City, Job).**

**Extract a specific key from the hashtable:**

***$myhashtable.<key-name>* as you can see I can get the value of each one of my keys.**

****

**Check If a Value or a Key exists in your hashtable:**

****

**Contains replies in a Boolean value (True or false), I checked if my hashtable contained the key "Name" and the Value "Bokobza".**

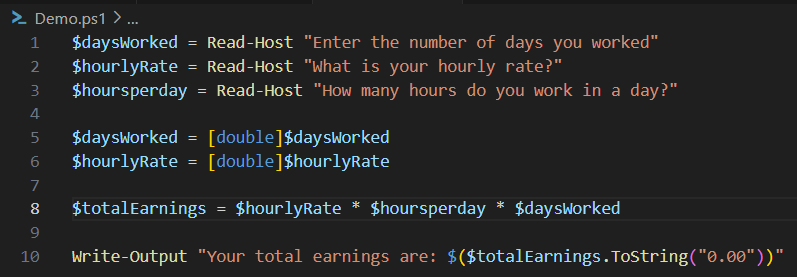
**The key exists while the value doesn't.**

**Why use Hashtables over ArrayLists?**

**Use hashtables over arrays when you need to store key-value pairs and access values based on a key.   
Hashtables offer faster lookups compared to arrays, which require searching through the entire list.**

**Scripting With string values:**

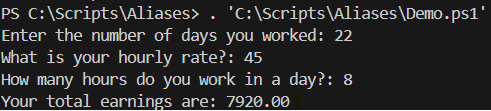
**For this next demonstration I will be making a salary calculation script.**

****

***[double]* – Converts the user input (which is initially a string) into numbers for mathematic operations to be performed.**

**Line 8 – Multiplies the hourly rate by the number of hours worked per day and the number of days worked to calculate the total earnings.**

***ToString("0.00").* – Displays the total earnings formatted to a decimal number (.00).**

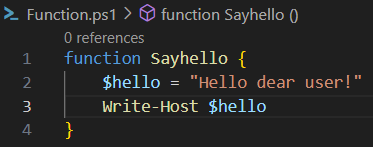
****

**What is a Function?**

**Functions and Parameters**

* **A function is a 'block of code' that that builds repeatable actions that can be reused multiple times.**
* **Functions are the bases of many Object-oriented programming languages like PowerShell.**
* **Functions are essential for us to learn before we start writing larger scripts within PowerShell.**
* **Instead of repeating actions, we can create actions inside a function and call that function throughout the code.**

**I called my function "Sayhello" and stored a variable named "$hello".  
the variable "hello" will be scoped into the function, meaning it could not be used outside of the function.**

****

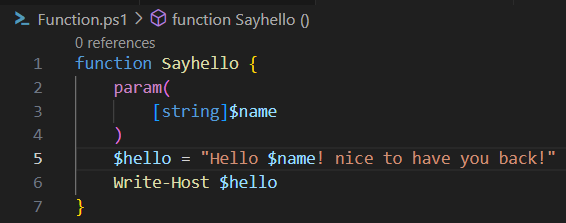
**When I run the script, nothing will show up, but when I type my function "Sayhello" the script within the curly brackets will run.**

****

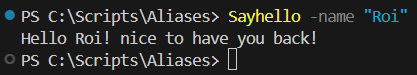
**What is a Parameter?**

**A parameter is a variable you pass to a function to change how it behaves. It makes functions more flexible and useful in different situations.**

**Almost all PowerShell cmdlets and advanced scripts use parameters, so understanding them is essential.**

****

**Under say hello I made a function called Sayhello and added a parameter *($name)* so it’s not stuck with just one name.  
Inside param(), I defined $name as a string, meaning it expects text input.**

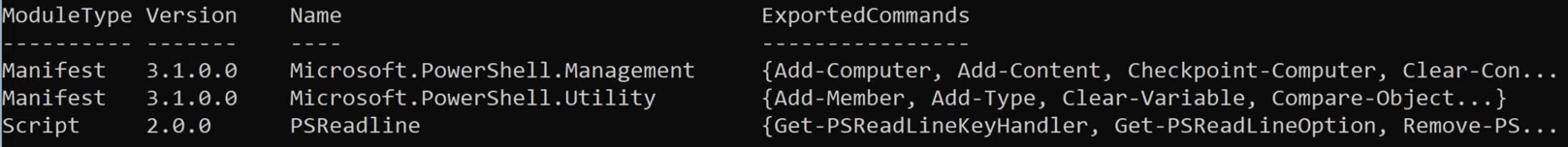
**  
Now, when I run *Sayhello -name* "Roi", it swaps in "Roi" dynamically instead of using a fixed name. This makes the function way more useful since I can use it with any name instead of rewriting code.**

**What is a Module?**

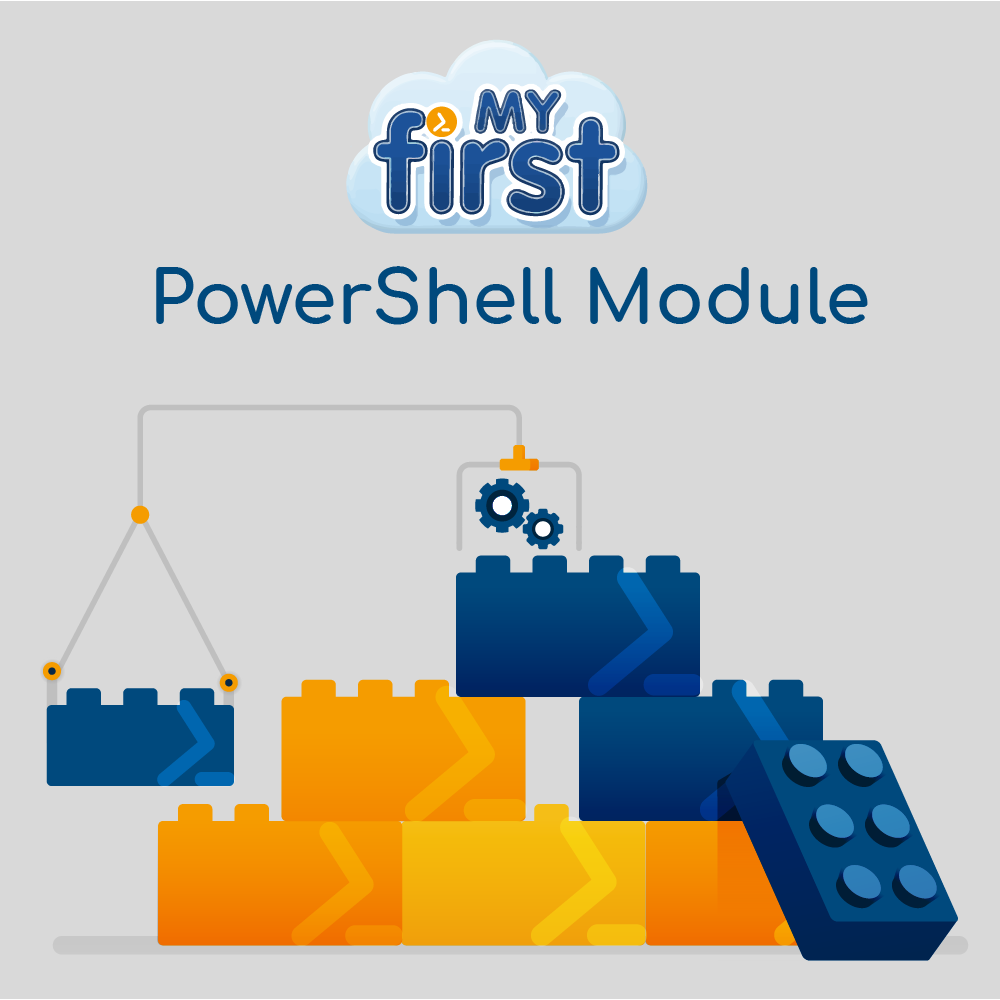
**Modules**

**In PowerShell all of the commands are stored in something called "Module".**

**Module is a collection of cmdlets and a way to organize and package related PowerShell functionality (cmdlets, functions, scripts, variables, etc.).**

****

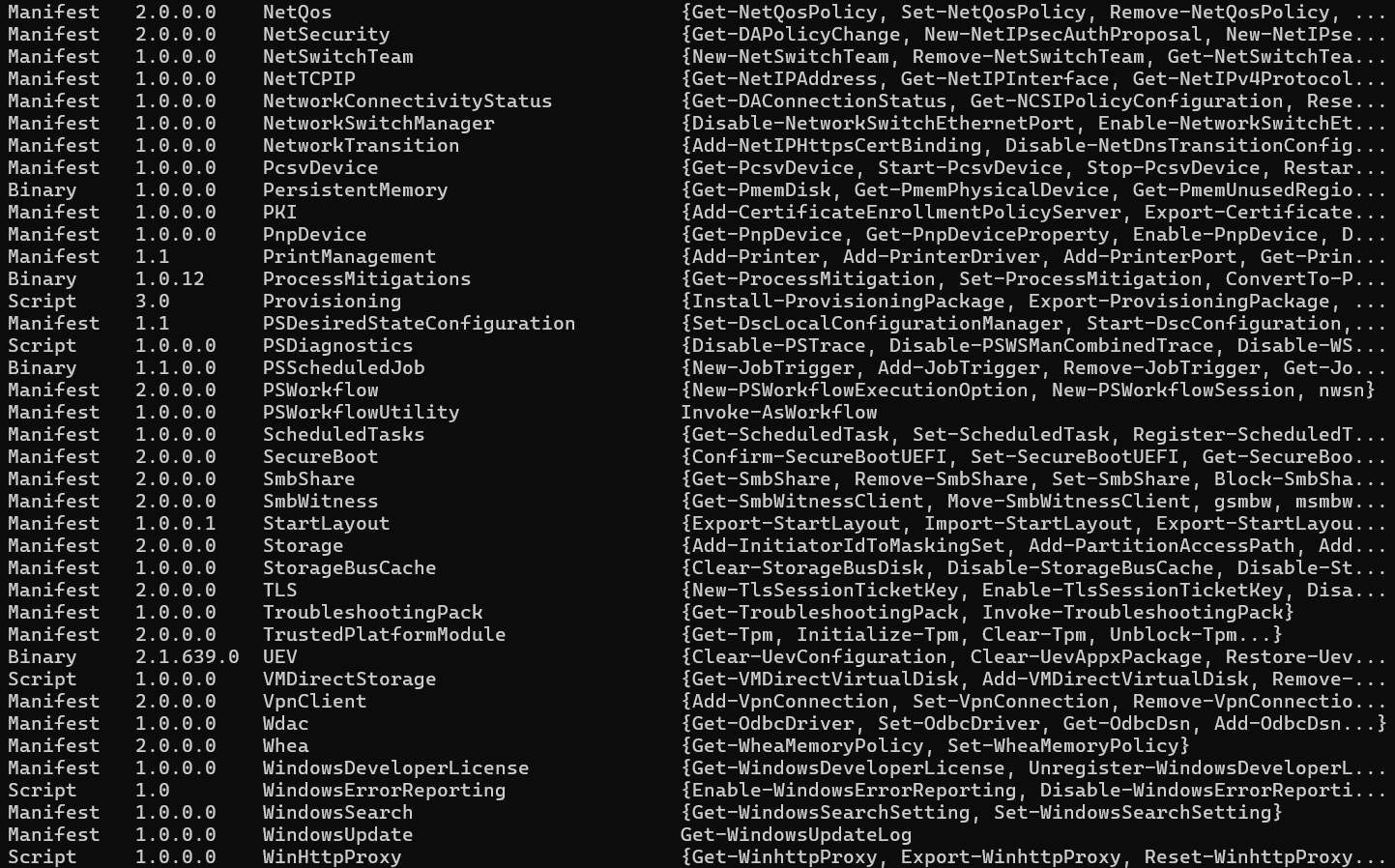
**When typing *'Get-Module'* we'll get a list of all modules that are built into PowerShell.**

**However, there are more models stored on our computer that is available to load into memory.**

41

**Load Modules:**

**Typing *Get-Modules -ListAvailable* will display all the models stored on the computer, there are quite a few here:**

****

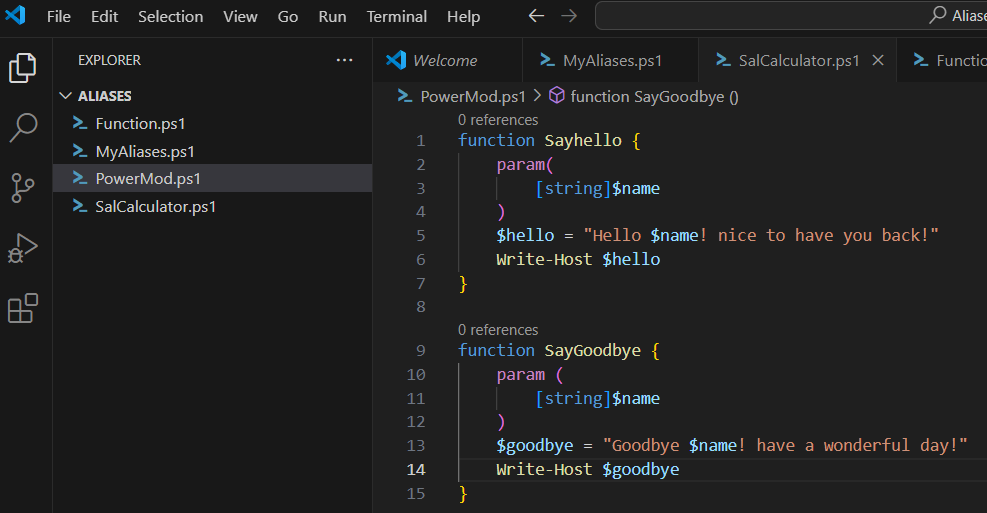
**  
To install a module type *'Install-Module -Name <Module-name>'*  
When installing PowerShell will prompt you with a message asking you if you are sure you want to install the modules from PSGallery?**

**The modules installed are linked directly to the site** [**PowerShell Gallery**](https://www.powershellgallery.com/)**, a site where people share their modules.**

**To import a module stored on the module AvailableList, type:   
*'Import-Module <Module-name>'*. and type Y to run the module.**

****

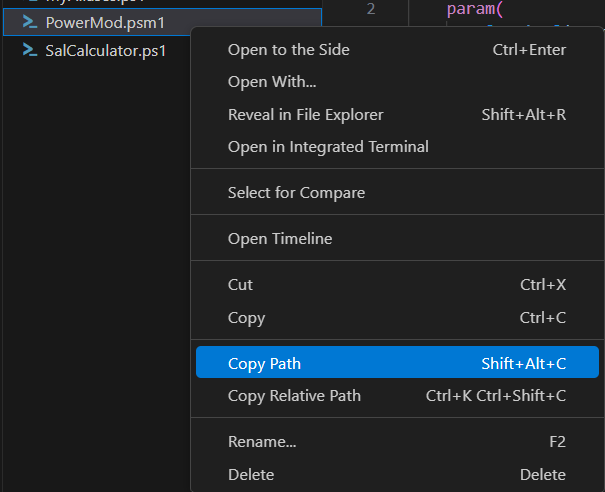
**Make your own Module:**

****

**To make a module we could take a function and do a very simple change, type pms1 instead of ps1.**

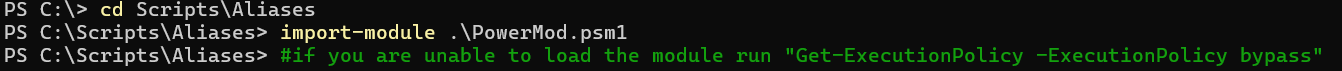
****

**This is now a module we can load and unload into PowerShell!**

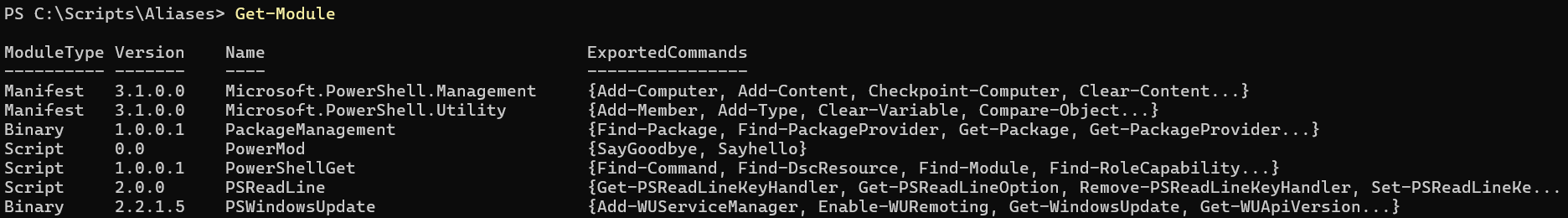
**Copy the path to your module and paste it on Shell:**

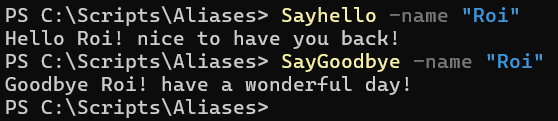
**I used cd to change the directory to the proper path where my module is at.**

***Import-Module .\PowerMod.psm* imports my module.**

****

**Type 'Get-Module' to confirm that the module has been loaded:**

****

****

**functions, parameters, and modules all work together in PowerShell to create a powerful, organized, and flexible environment for automation and management.**

**What is remoting and why use it?**

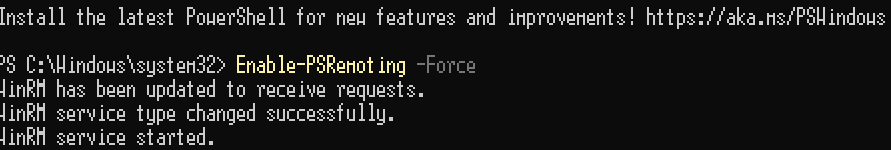
**PowerShell Remoting lets you control another computer’s PowerShell from your own, like having a remote command center.   
Instead of physically going to the machine, you can run commands, scripts, and manage it from wherever you are.   
This is a huge time saver, Whether you're automating tasks or handling issues, this feature gives you full control over remote machines with just a few commands.**

**1. Enable PowerShell remoting:**

**Remoting**

**Before you can use remoting, open PowerShell as manager and enable it on the remote computer by typing:**

***'Enable-PSRemoting -Force'***

****

**What command does:**

* **Enables the WinRM (Windows Remote Management) service.**
* **Configures the firewall to allow remote PowerShell connections.**
* **Sets up the necessary configurations for remoting.**

**2. Test the connection:**

**Before running commands, test if you can connect to the remote computer by typing**

***'Test-WSMan -ComputerName <"Your-pc-name">'***

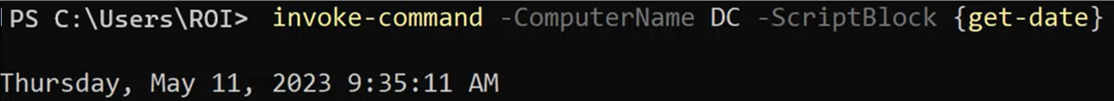
**What the command does:**

* **Checks if the remote computer is configured for PowerShell Remoting.**
* **Verifies that the WinRM service is running and accessible.**

**3. Run a command remotely:**

**Use *Invoke-Command* to run a command or script block on a remote computer.**

***Invoke-Command -ComputerName "RemotePC" -ScriptBlock { <Command> }***

******

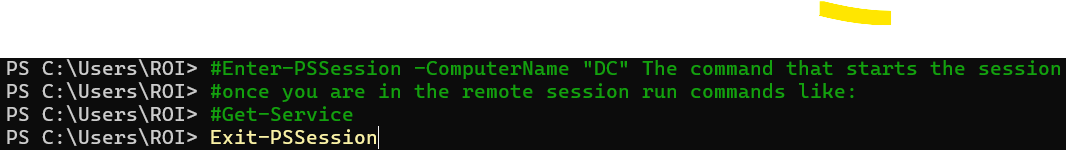
**This command executes the specified command or script block on the remote computer and returns the results to your local session.**

**4. Start an interactive session:**

**Use *Enter-PSSession* to start an interactive session with a remote computer. This allows you to run multiple commands in a remote session.**

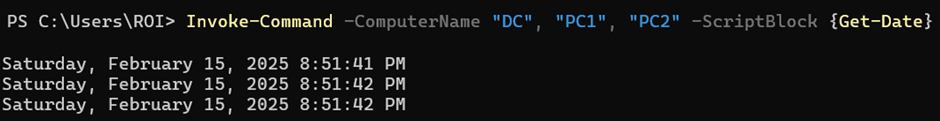
***Enter-PSSession -ComputerName "<Your-pc-name>"***

* **This command Opens an interactive PowerShell session on the remote computer.**
* **Any commands you type will run on the remote computer until you exit the session.**

****

**5. Run command on multiple computers:**

**You can run commands on multiple remote computers at once by providing a list of computer names.**

****

**Executes the specified command or script block on multiple remote computers simultaneously and returns the results from each computer.**

**6. Create a Persistent Session:**

**Use New-PSSession to create a persistent session with a remote computer. This allows you to run multiple commands in the same session without reconnecting.**

****

* **Creates a persistent connection to the remote computer.**
* **Allows you to run multiple commands in the same session.**

**7. Run scripts remotely:**

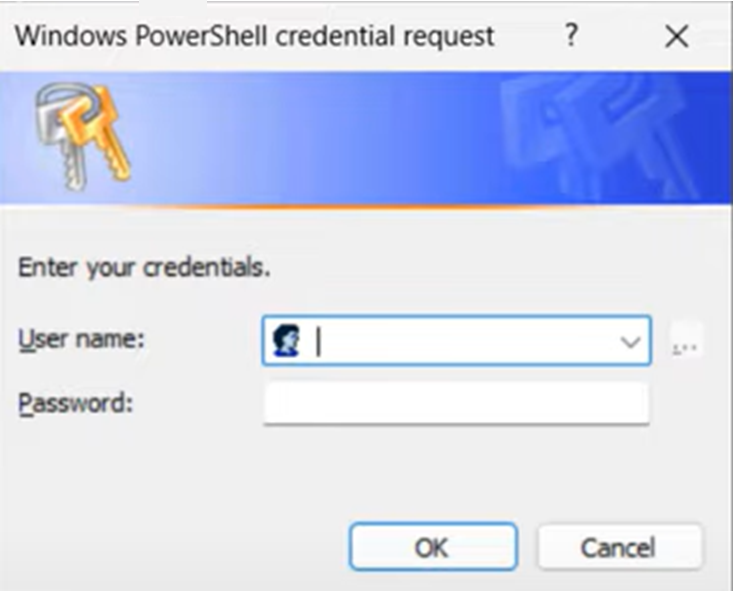
**You can also run entire scripts on a remote computer using Invoke-Command.**

**Executes the script on the given path on the remote computer.**

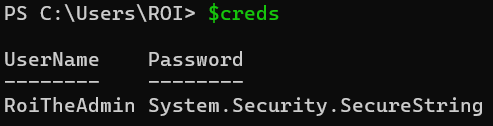
**8. Use Credentials for authentication:**

**By default, PowerShell Remoting uses your current user account, but if you need to use a different account you can specify it with the -Credential parameter.   
This is especially important when you need admin rights for some commands to work.   
If you're automating tasks, you might need to run them as a different user, like a service account. And as a general rule, don’t use your personal account for everything—only when necessary.**

****

****

**This variable ($creds) will allow us to securely store our credentials inside of it, in the prompt command box enter your credentials.**

****

**When typing in our variable $creds, my credentials show up, but the password is encrypted.**

**9. Example for real-world use!**

**Check Disk Space:**

***Invoke-Command -ComputerName "DC" -ScriptBlock {Get-PSDrive -PSProvider FileSystem}***

**Restart a Service:**

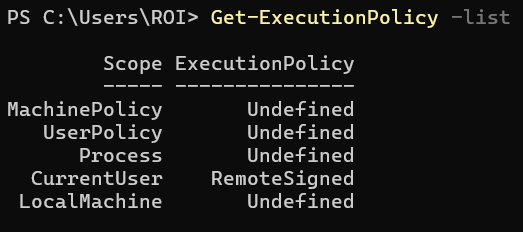
***Invoke-Command -ComputerName "DC" -ScriptBlock {Restart-Service -Name "Spooler"}***

**Collect Event Logs:**

***Invoke-Command -ComputerName "DC" -ScriptBlock {Get-WinEvent -LogName "Security"}***

**Execution Policies**

**Execution policies in PowerShell help reduce accidental risks and controls whether and how scripts can run on your system.   
They’re a security measure to prevent malicious scripts from running unchecked.   
Depending on the policy, you might be blocked from running unsigned scripts, or only allowed to run local ones.**

****

**Types of Execution Policies:**

* **undefined – no rules are enforced.**
* **RemoteSigned – any script you write will run properly, but when downloading one from the internet, it needs to be 'signed'.**
* **Restricted – This is the default policy. No scripts can run at all, not even locally written ones. You can only run individual commands.**
* **AllSigned – Scripts can only run if they are signed by a trusted publisher. That goes for both local and downloaded scripts.**

**>>>**

* **Unrestricted: Scripts can run, but there’s a warning for any downloaded script. It’s basically no restrictions, but you'll get a prompt for downloaded scripts.**
* **Bypass: There’s no restriction at all. This is basically full permission to run any script, no questions asked.**

**Scopes:**

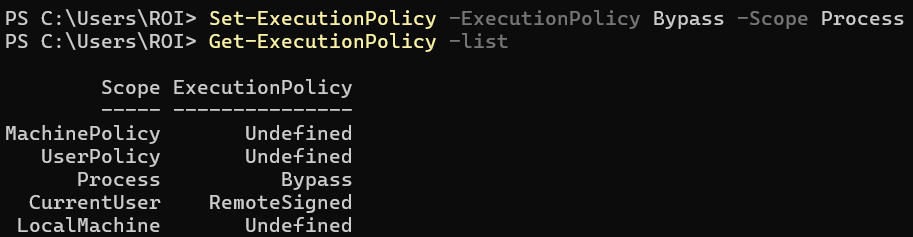
**scopes determine where the policy is enforced.**

* **MachinePolicy: Managed by group policy, affects all users.**
* **UserPolicy: Managed by group policy, affects the user.**
* **Process: Affects just the current session, goes away when you close PowerShell.**
* **CurrentUser: Affects only the current user.**
* **LocalMachine: Affects all users on the machine.**

**The Syntax to change a policy for a certain scope is:**

***Set-ExecutionPolicy -ExecutionPolicy <Policy> -Scope <Scope>***

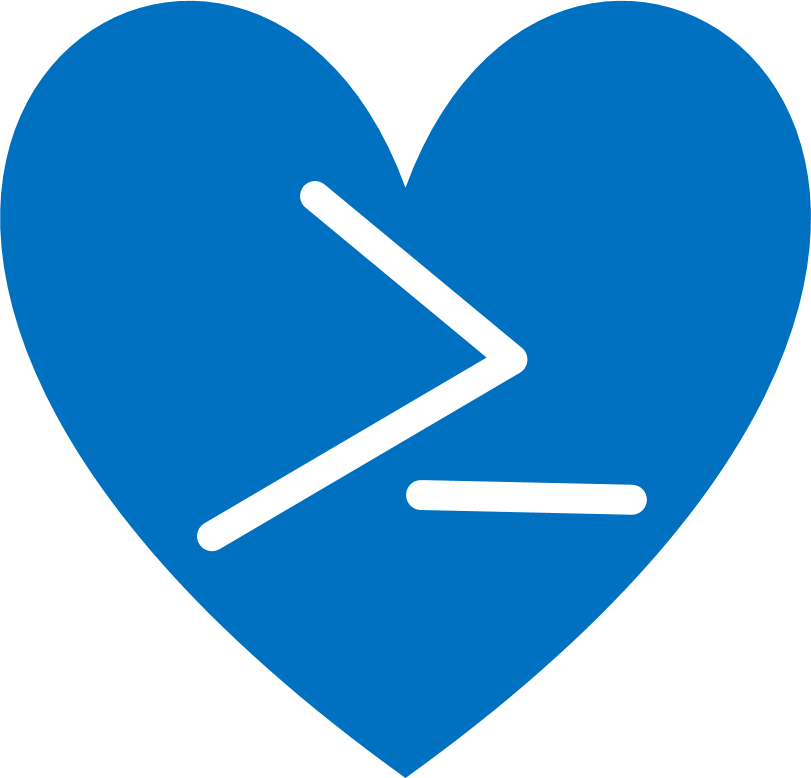
**Running the cmdlet will prompt you with a warning depending on the execution policy you have, hit Y and the policy for the scope will be updated.**

****

**Conclusion**

**Working on this project has been an incredibly exciting and rewarding experience! Diving into PowerShell has not only improved my technical skills but also opened up new ways to automate tasks, manage systems, and solve problems more efficiently. I hope this guide helps others discover the power of PowerShell and inspires them to explore its capabilities further. Whether you're a beginner or looking to sharpen your skills, PowerShell is a tool that can truly transform the way you work.**

**Happy scripting, and may your journey with PowerShell be as fulfilling as mine has been!**



**- Roi Bokobza**