@PoSh

Created	@January 14, 2025 8:57 AM
Class	Scripting
Date	@January 14, 2025

13. RegEx for an IP Address

RegEx Cheat Sheet:

Show Data-type

Powershell Kata

`get-<something> | where-object | where | ? {<property> -eq | -like <something>} | select-object -property | -expandproperty <property(ties)>`

PowerShell Fundamentals

What is PowerShell?

PowerShell is a **cross-platform automation solution** made up of:

- A **command-line shell**.
- A **scripting language**.
- A **configuration management framework**.

PowerShell runs on Windows, Linux, and macOS.

As a scripting language:

- PowerShell is commonly used for automating the management of systems.
- It is also used to build, test, and deploy solutions, often in CI/CD environments.

CI/CD:

- Stands for Continuous Integration/Continuous Delivery or Deployment.
- Bridges development and operations by automating building, testing, and deployment.

Technical foundation:

- PowerShell is built on the `.NET Common Language Runtime (CLR)`.
- All inputs and outputs in PowerShell are `.NET objects`.

Objects in PowerShell:

- An object is an **enriched data container** representing data in an organized structure.
- **Methods**: Actions that can be taken by an object.

> 💡 `$_` is short for `PSItem`. It is a placeholder for items in the pipeline. It references the output of the previous command and is used after a pipe.

PowerShell Features

PowerShell shares some features with traditional shells:

- **Built-in help system**: Provides information about commands and integrates with online help articles. `Get-Help`
  - use `Get-Help` followed by the command you would like help with to see a help page similar to Linux man pages.
  - You can search the help page using `-ShowWindow` followed by the search term.
  - At the bottom of the Help page you can see examples of command usage under `REMARKS`
    - you can also use `-examples` following `Get-Help` and the command
- **Pipeline**: Allows running many commands sequentially.
- **Aliases**: Alternate names used to run commands.
  - `Get-Alias` will show you aliases for common Linux commands.

> 💡 **Important RegEx:**
>
> `[ ]` Range
>
> `{ }` Quantifier
>
> `( )` Group

How is PowerShell Different?

- **Object-oriented**: Operates on objects rather than text (bash outputs text).
  - A PowerShell object is made up of three types of data:
    - the object **type (ObjectType)**,
    - its **properties**,
    - and its **methods**.
  - No need to spend time formatting output and extracting data as objects retain their properties.
- **Cmdlets**: Commands in PowerShell are called `cmdlets` (pronounced "commandlets").
  - Cmdlets are built on a common runtime rather than being separate executables.

Cmdlets in Detail

- Cmdlets typically **take object input** and **return objects**.

> 💡 All inputs and outputs are **.NET objects**.

- Core cmdlets in PowerShell are built in `.NET Core` and are open source.
- You can build custom cmdlets in `.NET Core` or PowerShell.

Types of commands in PowerShell:

- Native executables.
- Cmdlets.
- Functions.
- Scripts.
- Aliases.

Check on Learning

- What are some features PowerShell shares with traditional shells?
  - Pipelines
  - Help-Pages
  - Aliases
- What are commands in PowerShell called?
  - cmdlets

Commandlets

A `cmdlet` is a lightweight command used in the PowerShell environment.

Key differences from traditional commands:

- Cmdlets are instances of `.NET classes`; they are not stand-alone executables.
- Cmdlets process input objects from the pipeline rather than streams of text.
- Cmdlets deliver objects as output to the pipeline.

Commandlets: Verb-Noun Naming Convention

- Cmdlets follow a **verb/noun naming convention**.
- Example: `Get-ChildItem`.
  - References the directory that you are already in and delivers all items within it. (similar to `ls` in Linux)

Functionality:

- Lists or returns items in one or more specified locations.
- If items are in a container, the command retrieves the items inside the container — child items.

Common Verbs include:

- Get-*

- Start-*
- Stop-*
- Read-*
- Write-*
- New-*
- Out-*

PowerShell Pipeline

- A pipe (`|`) takes the output of the first part of the pipeline and uses it as the input for the next part.
- Example: Using the `Select-Object` cmdlet to display only the `DisplayName` property.

```
Steve: C:\Users\student >>>get-service | Select-Object Name

Name
----
AarSvc_4639e
AJRouter
ALG
```

Parameters and Arguments

- **Cmdlets**: Most cmdlets support parameters as part of their input mechanism. These are similar to command *options* in Linux.
- Parameters can be added:
  - At the command line.
  - Passed through the pipeline as output from a previous cmdlet.

Arguments:

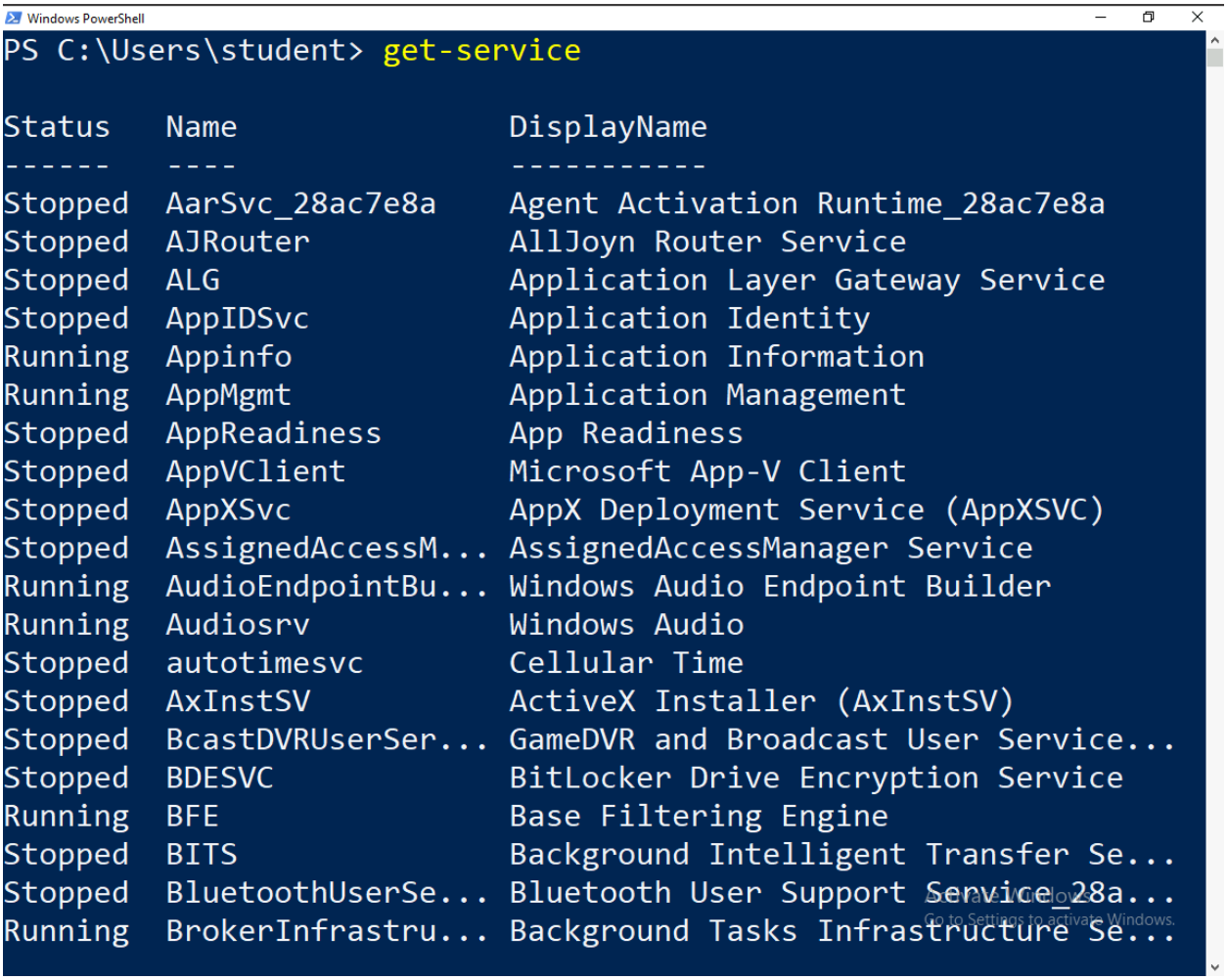
- Specify the input a cmdlet accepts.

- Control how the cmdlet operates.
- Define what data, if any, the cmdlet outputs.

Example:

To filter services that are running:

```
Get-Service | Where-Object {$_.Status –eq “Running”}
```



Check on Learning

1. What type of naming convention do cmdlets use?
    - Verb-Noun

2. Which statement is true about how cmdlets differ from commands?
    - a. Cmdlets are stand-alone executables.
    - b. Cmdlets are instances of .NET classes.
    - c. Both a & b.

PowerShell Help System

- Most shells include a help system for learning about commands, their functions, and supported parameters.

PowerShell Help Cmdlets:

- `Get-Help`
- `Get-Command`
- `Get-Member`

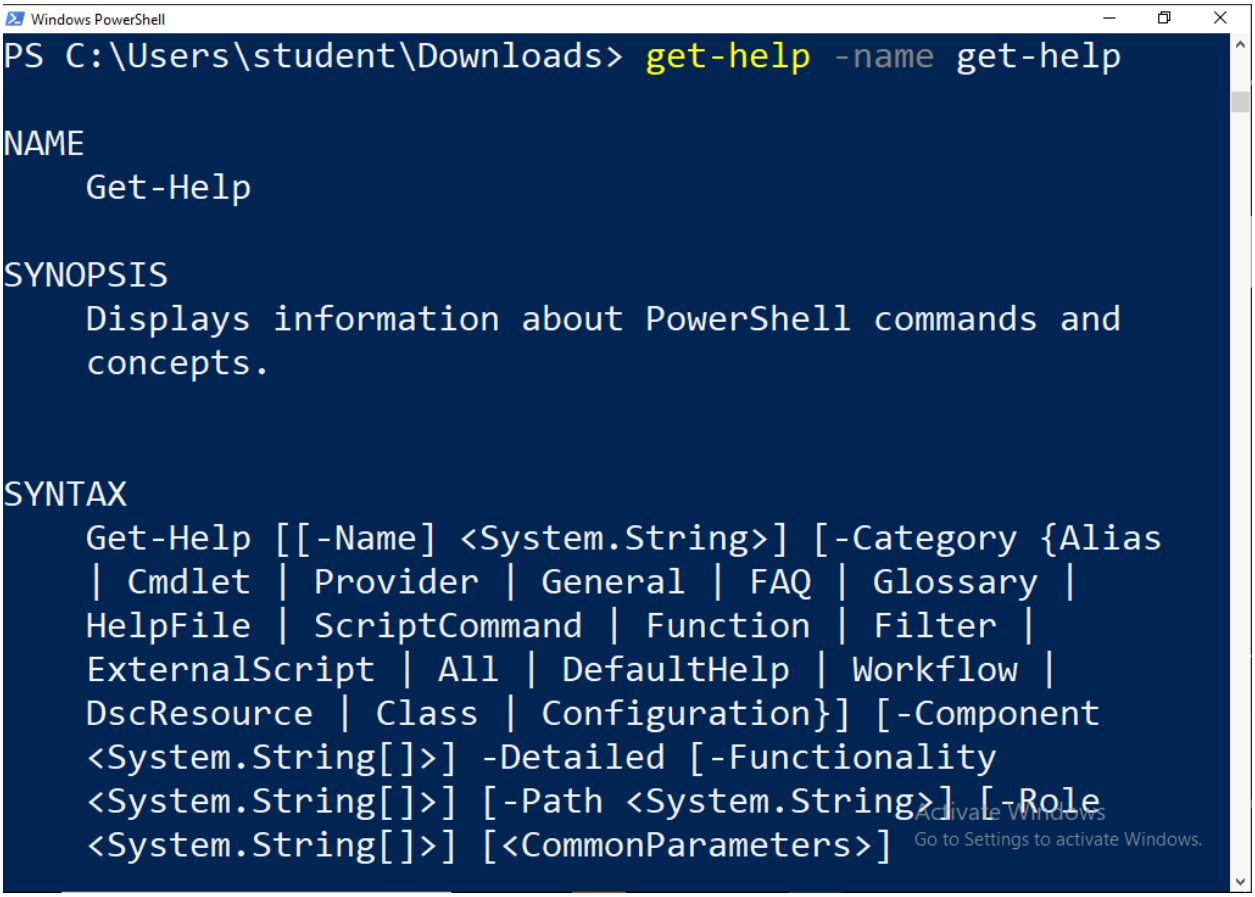
Get-Help

- `Get-Help` is a multipurpose cmdlet that helps you learn how to use commands.
- It searches:
    1. For wildcard matches of command names based on provided input.
    2. Through help topics if no direct match is found.

Example:

```
Get-Help -Name Get-Help
```

We are calling `Get-Help` on the cmdlet named `Get-Help`. Similar to `man man` in Linux.



Get-Help (Microsoft.PowerShell.Core) - PowerShell

The Get-Help cmdlet displays information about PowerShell concepts and commands, including cmdlets, functions, Common Information Model (CIM) commands, workflows,

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/get-help?view=powershell-7.4&viewFallbackFrom=powershell-7.3&WT.mc_id=ps-gethelp

Parameters

Run the example command on your computer and review the output. Information is grouped into:

- **NAME**
- **SYNOPSIS**
- **SYNTAX**
- **DESCRIPTION**

- **RELATED LINKS**
- **REMARKS**

Help topics can contain significant amounts of information—this is just part of the output.

Parameters in Detail

- **Parameter**: A way to provide input to a command.
- `Get-Help` has many parameters to return the entire help topic or subsets of it.

Parameter Sets:

- Multiple blocks in the syntax section represent different parameter sets.
- Each set has at least one unique parameter.
- Example: The `Full` and `Detailed` parameters are in different parameter sets and cannot be used together.

Parameter Sets

- Parameter sets are **mutually exclusive**.
- Once a parameter unique to one set is used, only other parameters in that set can be used.

Examples of Parameters in Different Sets:

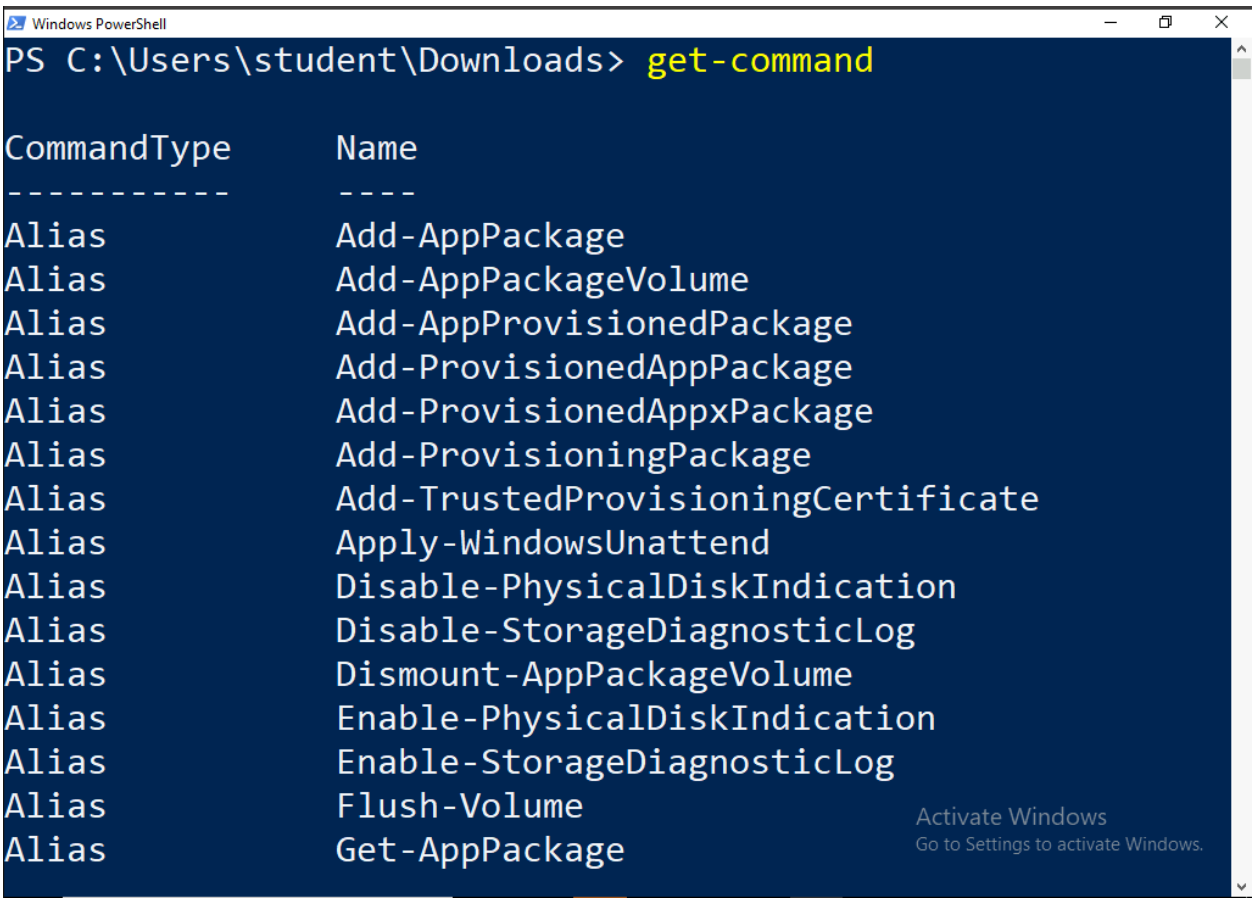
- `Full`
- `Detailed`
- `Examples`
- `Online`
- `Parameter`
- `ShowWindow`

Get-Command

- `Get-Command` retrieves all commands installed on the computer.
- Includes: cmdlets, aliases, functions, filters, scripts, and applications.

Syntax:

```
Get-Command
```



Get-ChildItem

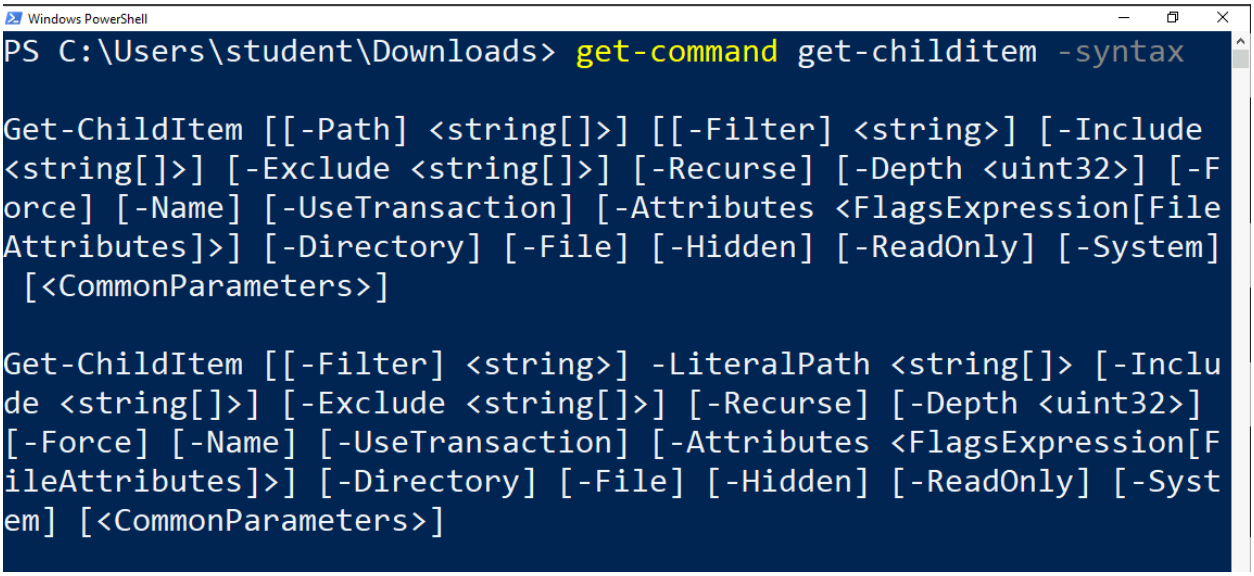
- Retrieves the parameter sets of the `Get-ChildItem` command.

Syntax:

```
Get-Command -Name Get-ChildItem -Syntax
```

You can replace `Get-ChildItem` with any command to retrieve its syntax.

We are calling `Get-Command` on the cmdlet named `Get-ChildItem` to see its accepted syntax.



Wildcard *

- The wildcard character `*` matches any and all string filters.
- **Syntax**:

  We are calling
  `Get-Command` and filtering for commands that start with `Get`.

  ```
  Get-Command Get*
  ```

Get-Member

- The `Get-Member` cmdlet retrieves the **members**, **properties**, and **methods** of objects.
  - It will show you all of the possible properties of a particular command as well as all the methods you can call on it.

- **Syntax**:

  Takes `Get-Process` and pipes it into `Get-Member` to display all the methods and properties of that object.

  ```
  Get-Process | Get-Member
  ```

  - **Methods**: Things we can do with these objects.
  - **Properties**: Things these objects are.

Check on Learning

1. What does the `Get-Member` cmdlet do?
   a. Tells you about a command! Retrieves the **members, properties,** and **methods.**
2. True or False: Parameter sets are not mutually exclusive.
   a. False! If you were to run `Examples` and `Full` it would not know what to do!

PowerShell Aliases

- An **alias** is an alternate name for a cmdlet, function, executable file, or script.
- PowerShell includes built-in aliases (e.g., `cls` for `Clear-Host` or `ls` for `Get-ChildItem`).
- Users can create their own aliases for the current session or save them to the PowerShell profile.

To create an alias:

```
Set-Alias -Name list -Value Get-ChildItem
```

Verifying Aliases

- Use the `Get-Alias` cmdlet to verify that an alias was created.
- **Syntax**:

  This checks that the alias `list` functions exactly like `Get-ChildItem`.

  ```
  Get-Alias -Name list
  ```

To find all aliases for a command:

```
Get-Alias -Definition Get-ChildItem
```

Check on Learning

1. What is the full command to create an alias?
   a. `Set-Alias -name <name> -value <cmdlet>`
2. True or False: The `Get-Alias` cmdlet can verify an alias was created.
   a. True!

Variables

- A **variable** is a unit of memory where values are stored.
- A variable stores **one** item.
- In PowerShell, variables are represented by text strings starting with `$` (e.g., `$a`, `$process`, `$my_var`).
- Variable names are not case-sensitive and can include spaces or special characters.
- PowerShell will make an educated guess at the data type of a variable if you give it an int, float, etc.

User-Created Variables

- Created and maintained by the user.
- **By default, user-created variables exist only while the PowerShell window is open.**
- To save variables:
  - Add them to your PowerShell profile.
  - Use `global`, `script`, or `local` scope in scripts.

```
#User-created variable
$my_var = 1
echo $my_var

$my_var += 1
echo $my_var
```

Automatic Variables

- **Automatic variables** store the state of PowerShell.
- Created and updated automatically by PowerShell; users cannot modify their values.
- PowerShell changes their values as required to maintain their accuracy.
- **Example**:
  - `$PSHOME`: Stores the path to the PowerShell installation directory.

```
#Automatic variable
echo $PSVersionTable
echo $PSHome
```

Preference Variables

- **Preference variables** store user preferences for PowerShell.
- Created by PowerShell but users can modify their values.

- **Example**:
  - `$MaximumHistoryCount`: Determines the maximum number of entries in the session history

```
#Preference variable
echo $MaximumAliasCount
echo $MaximumHistoryCount
```

Check on Learning

1. List some of the different types of variables discussed in PowerShell.
   a. User Created
   b. Automatic
   c. Preference
2. True or False: Users can change preference variables.
   a. True!

Arrays

- An **array** is a data structure designed to store a collection of items.
- Items in an array can be of the same or different types.
- Arrays are organized by **zero-based index**.
- Remember how a variable stores **one** item? An array is a kind of variable that stores many values.

To create and initialize an array:

- Assign multiple values to a variable.
- Delimit the values with a comma and separate them from the variable name using the assignment operator (`=`).

```
$A = 1,5,10,15,10,"squirrel"
#In this example we are taking the values 22,5,10,8,12,9 and 80 and storing them into the array named $A

echo $A
```

Arrays Example

- The data is associated in memory with the array named `$A`.
- Anytime we want to access this information, we can interact with the array.
- Example: The value `5` is stored at index `1`.

```
#We can access the value at a specific position by refering to its index.

echo $A[1]
```

Array Sub-Expression

- The **array sub-expression operator** creates an array from the statements inside it.
- Whatever the statement inside the operator produces, it will be placed in an array, even if there is zero or one object.

Sub Expression Syntax

```
@( commands )
```

Sub Expression Syntax

```
$services = @(get-service | where {$_.Status -eq "Running"} | Select-Object Name )
```

Example:

An array named `$services` contains a collection of all service names with a status of `"Running"`.

Usage:

- We can now interact with service names as if they were a regular array.

```
echo $services[12]
```

- This organizes output from a cmdlet into an array for further data processing or analysis.
    - One of the properties of an array in length. This is simply how many objects are in an array.

Array Length:

- Access the number of objects in an array using the `.Length` method:

```
$array.Length
```

How would you access the last value in an array of unknown length?

```
echo $VarName[$_.length-1]

# OR, more simply...

echo $VarName[-1]
```

Hash Tables

- A **hash table** (also known as a dictionary or associative array) is a compact data structure that stores one or more **key/value pairs**.
- Example:
    - Keys: IP addresses.
    - Values: Computer names (or vice versa).

Key-Based Indexing:

- Unlike regular arrays, hash tables use **keys** to access values, not numerical positions.

Hash Tables Example

To create a hash table:

- Follow this syntax:

```
$phonebook = @{ Bob = "706-123-4567"; Alice = "803-123-4567"; Steve = "555-123-4567"}
```

```
$hashTable = @{
    Key1 = "Value1";
    Key2 = "Value2";
    Key3 = "Value3"
}
```

- Keys are associated with their corresponding values.
- Reference a specific key to access its value:

```
echo $phonebook["Bob"]
echo $phonebook["Alice"]
```

Check on Learning

1. What is an array?
    a. An **array** is a data structure designed to store a collection of items. Items may be alike or different. It is zero-indexed and comma delimited.
2. What are the values stored in an array delimited with?
    a. Commas (,)!

PowerShell Features: Profiles

- A **PowerShell profile** customizes your environment and adds session-specific elements to every session you start. (Just like .bashrc in Linux! (Bash Run Configuration))

- Profiles are scripts that run when PowerShell starts.
- You can use profiles as logon scripts to configure the environment.

Elements you can add to a profile:

- Commands.
- Aliases.
- Functions.
- Variables.
- Snap-ins.
- Modules.
- PowerShell drives.
- Other session-specific elements.

PowerShell Profile Files

- PowerShell supports several profiles for users and host programs.

Description	Path
All Users, All Hosts	**Windows** $PSHOME\Profile.ps1 **Linux** /usr/local/microsoft/powershell/7/profile.ps1 **macOS** /usr/local/microsoft/powershell/7/profile.ps1
All Users, Current Host	**Windows** $PSHOME\Microsoft.PowerShell_profile.ps1 **Linux** /usr/local/microsoft/powershell/7/Microsoft.Powershell_profile.ps1 **macOS** /usr/local/microsoft/powershell/7/Microsoft.Powershell_profile.ps1
Current User, All Hosts	**Windows** $Home\Documents\PowerShell\Profile.ps1 **Linux** ~/.config/powershell/profile.ps1 **macOS** ~/.config/powershell/profile.ps1
Current user, Current Host	**Windows** $Home\Documents\PowerShell\Microsoft.PowerShell_profile.ps1 **Linux** ~/.config/powershell/Microsoft.Powershell_profile.ps1 **macOS** ~/.config/powershell/Microsoft.Powershell_profile.ps1

- **Important**: PowerShell does not create profiles automatically; you need to create them.

Editing a Profile

- You can open any PowerShell profile in a text editor (e.g., Notepad).
- **Automatic Variable**: Use `$PROFILE` to reference the path of your PowerShell profile.

```
notepad $PROFILE

#To specify a specific profile.
notepad $PROFILE.AllUsersAllHosts
```

Test the path of the $PROFILE to see if it exists:

```
Test-Path $profile
```

Creating a Profile

- If `Test-Path $PROFILE` returns `false`, it means the file doesn't exist.
- To create the profile:

  ```
  New-Item -Path $PROFILE -ItemType File -Force
  ```

- Test it again:

  ```
  Test-Path $PROFILE
  ```

  It should now return `true`.

To find the profile path:

```
$PROFILE
```

- Navigate to the path and open the file.
- In this file, you can make changes that will be applied every time you open PowerShell.

Editing (Adding to) the Profile

- To modify the profile, add desired commands, aliases, or configurations directly to the file.

- We are going to add the following to our profile:

```
function prompt{
        "CYBERMAN: $($executionContext.SessionState.Path.CurrentLocation) $('= ')"
}
set-alias -name list -value get-childitem
```

- **After editing**:
  - Save the profile file.
  - Restart PowerShell to apply the changes.

Check on Learning

1. True or False: A PowerShell profile is a script that runs when PowerShell starts.
2. List two of the PowerShell profiles.

PowerShell Objects

PowerShell Objects

- PowerShell is an **object-oriented language and shell**.
- Unlike traditional shells (e.g., cmd, Bash) that focus on text (strings), PowerShell focuses on **objects**.
- Nearly everything in PowerShell is an object.
- **Key Advantage**: Leverage its object-oriented structure to accomplish complex goals efficiently.

PowerShell Pipeline

- A **pipeline** is a series of commands connected by the pipeline operator (`|`).

- **How it works**:
  - Each pipeline operator sends the results of the preceding command to the next command.
  - The output of one command becomes the input for the next command in the pipeline.

Example:

- Start an instance of `notepad.exe`.
- Use `Get-Process` to find the `notepad` process.
- Pipe the process object into `Stop-Process` to stop the notepad process.

```
notepad.exe
Get-Process notepad | stop-process
```

```
Get-Process -Name notepad | Stop-Process
```

- The pipeline takes the output of the command on the left and pipes it as input to the command on the right.

PowerShell One-Liner

- A PowerShell **one-liner** is a continuous pipeline.
- **Note**: A one-liner doesn't have to be on a single physical line.

Example (a multi-line one-liner):

```
Get-Service |
   Where-Object StartType -eq Automatic |
     Select-Object -Property Name
```

```
Get-Service |
Where-Object {$_.StartType -eq "Automatic"} |
```

```
Select-Object -Property Name
```

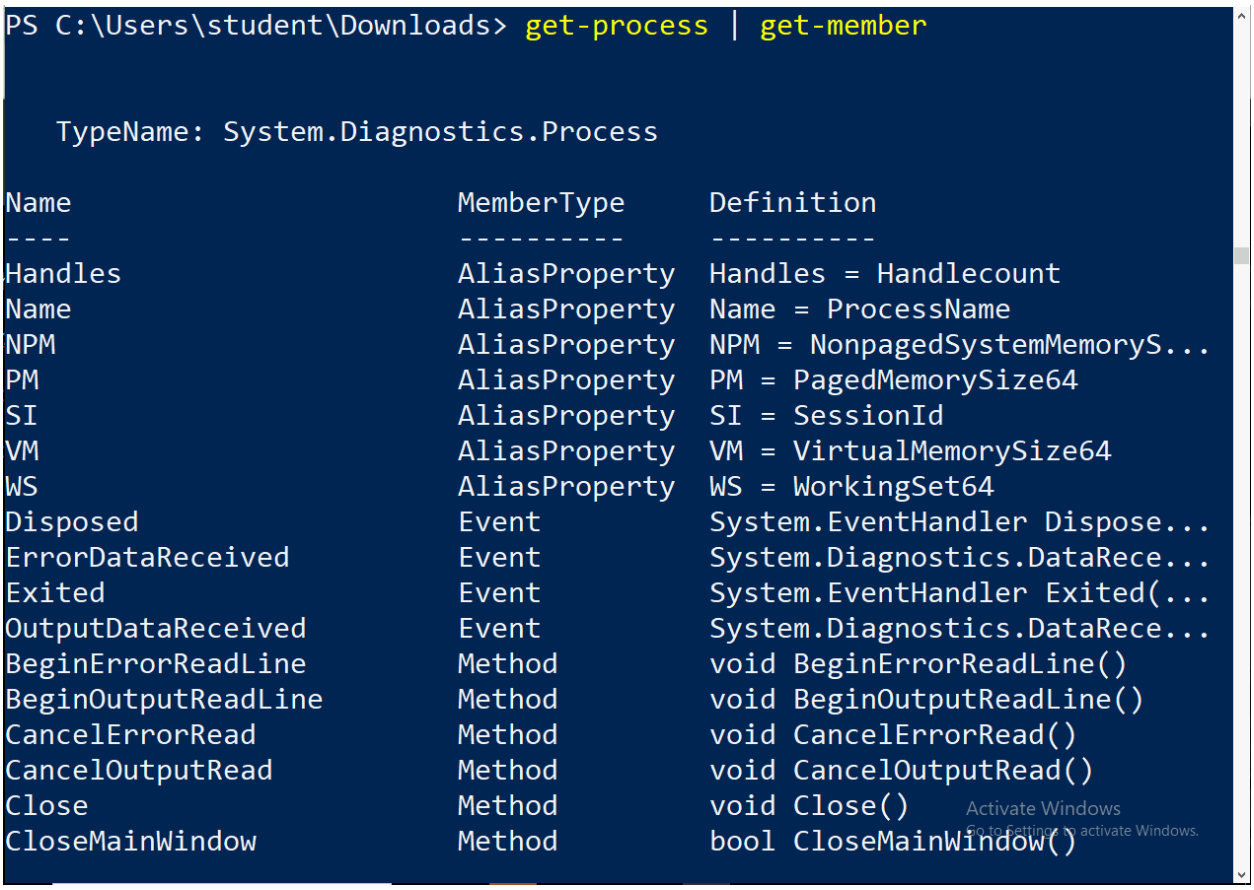
Breaking Down the One-Liner

```
Get-Service |
   Where-Object StartType -eq Automatic |
     Select-Object -Property Name
```

1. **Step 1**:
   - `Get-Service` retrieves all services.
   - Output is piped to the next command.
2. **Step 2**:
   - `Where-Object` filters the collection of services.
   - Selects services where the `StartType` property equals `"Automatic"`.
3. **Step 3**:
   - `Select-Object` retrieves the `Name` property of the filtered services.
   - Since it is the last pipe, the output is displayed on the screen.

PowerShell Objects

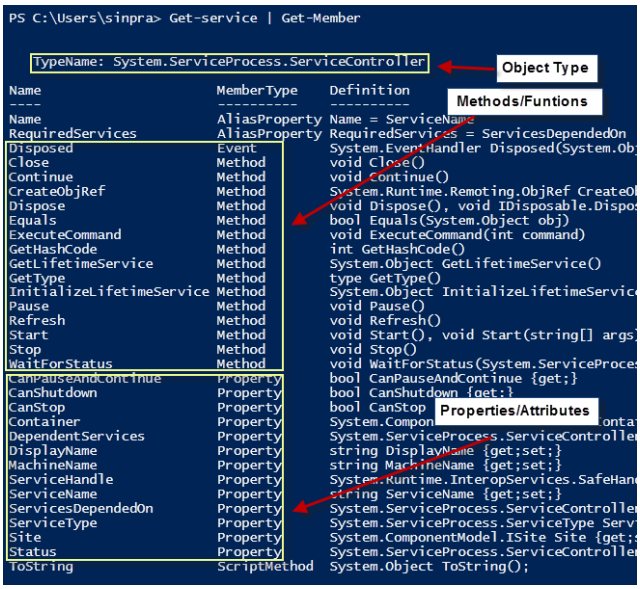
- Objects have multiple types of information, referred to as **members**.
- **Members**:
  - Generic term for all information associated with an object.
  - Includes **properties**, **methods**, and more.
- An object can have multiple members, but each member belongs to a single object.



Discovering Object Information

- Use the `Get-Member` cmdlet to discover object information.
- **Example**:

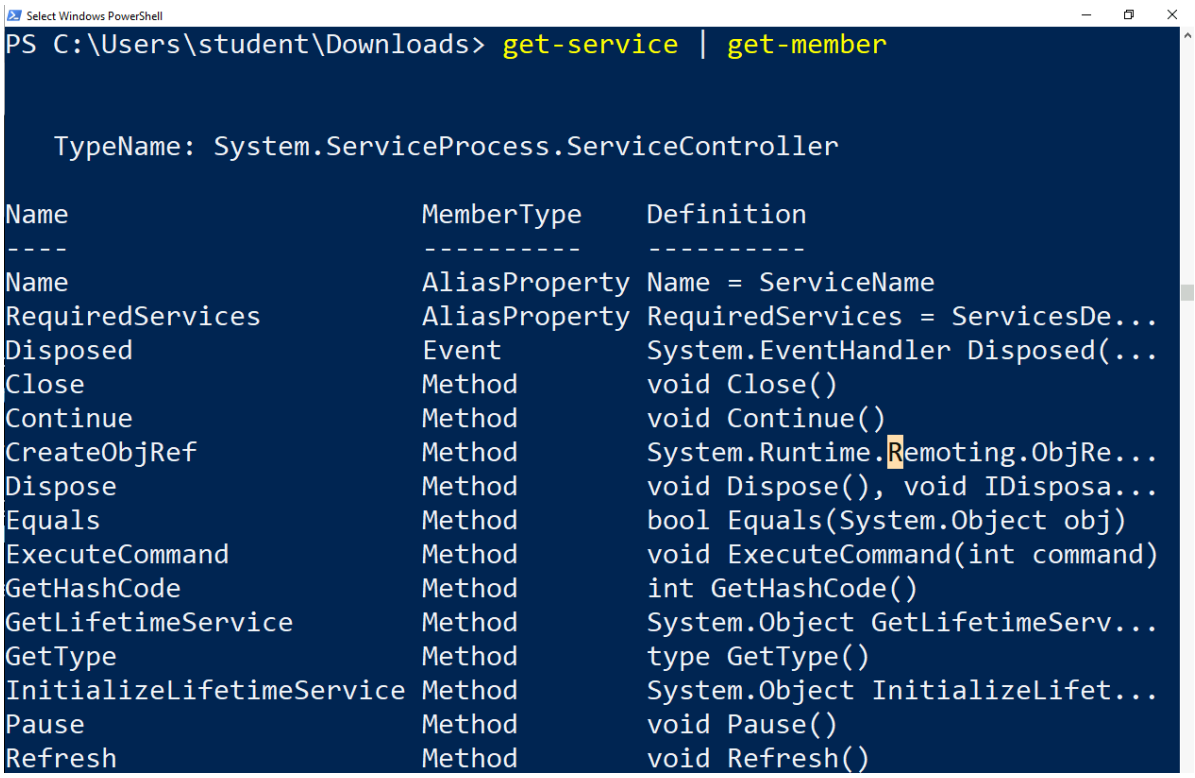
```
Get-Service | Get-Member
```



- This command reveals:
  - The type of object returned (`System.ServiceProcess.ServiceController`).
  - The available members of the object (e.g., properties, methods).

PowerShell Objects

- **Methods**: Actions that can be performed on an object.
  - Examples: `Pause`, `Stop`, `Start` (used to manage services).
- **Properties**: Attributes that describe an object.
  - Examples: `DisplayName`, `ServiceName`, `Status`.



PowerShell Objects: Methods and Properties

- **Method Members**: Actions taken on an object.
  - Example: For a "Human" object, methods might include `Walk`, `Run`, `Breathe`.
- **Property Members**: Attributes that describe an object.
  - Example: A "Human" object might have properties like `Height`, `Weight`, `Age`.
- **To search for properties**:

  Use the `Where-Object` cmdlet to filter objects based on their properties.

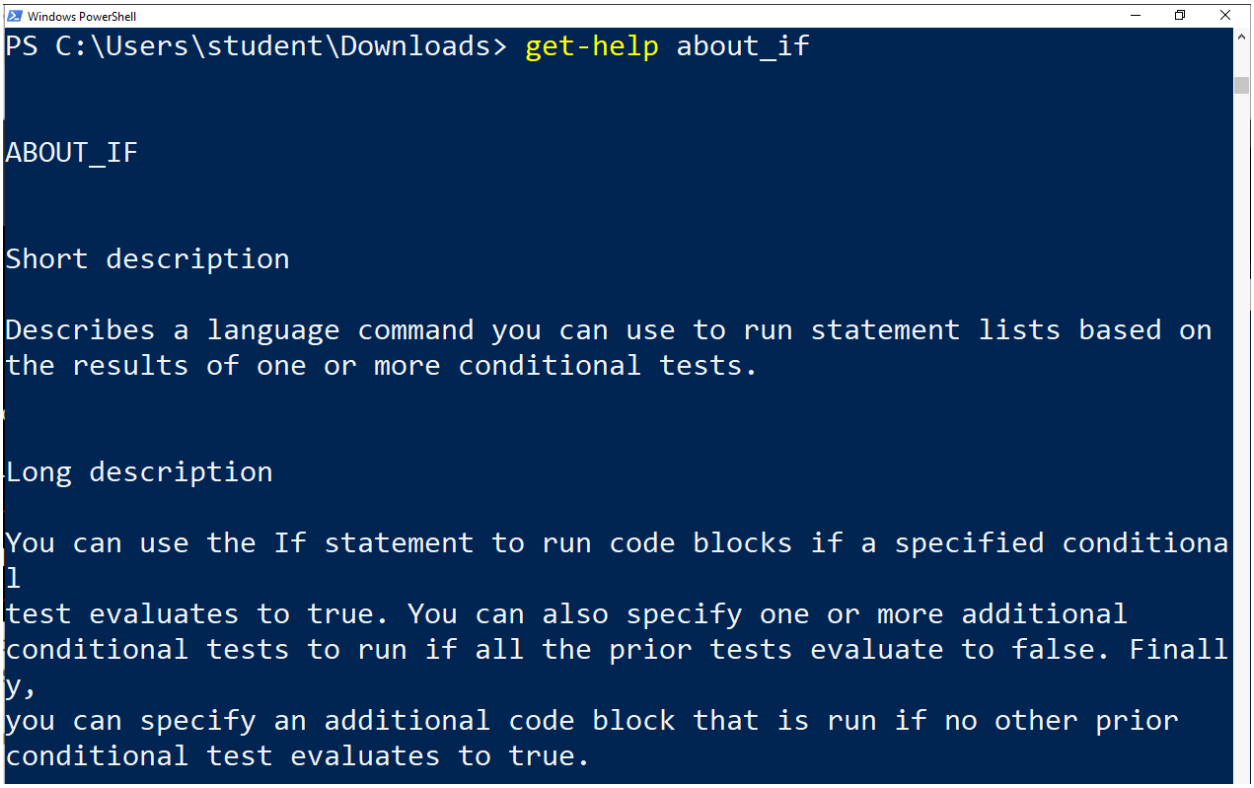
PowerShell Conditionals

- **Procedural Programming**:
  - Derived from imperative programming.
  - Based on the concept of the **procedure call** (a series of computational steps).

- All procedural languages, including PowerShell, require constructs for condition-based execution of instructions.

If Statement

- PowerShell, being procedural, includes built-in **conditionals** like the `If` statement.



- **How it works**:
  - Evaluates the condition inside parentheses.
  - If the condition is `$true`, executes the script block inside the braces `{}`.
  - If the condition is `$false`, skips the script block.

```powershell
$condition = $true
if ( $condition )
{
    Write-Output "The condition was true"
}
```

Example:

```powershell
$x = 1
if ( $x -eq 1 )
{
    Write-Output "The condition was true"
}
```

Else Statement

- The `Else` statement does not accept any conditions.
- It runs **only if all prior conditions are** `$false`.
- Acts as the **"last resort" condition**.

```powershell
$condition = $false
if ( $condition )
{
    Write-Output "The condition was true"
}
else
{
    Write-Output "The condition was false"
}
```

Elself Statement

- Use the `ElseIf` statement to add additional conditions.
- You can include multiple `ElseIf` statements for different conditions.
- PowerShell evaluates the conditions **sequentially**.

```
$condition = $false
if ( $condition )
{
    Write-Output "The condition was true"
}
elseif (!$condition)
{
    Write-Output "The condition was false"
}
else
{
    Write-Output "I dont know how you got here."
}
```

Comparison Operators

- The `If` statement is often used to **compare two items**.
- **PowerShell Comparison Operators**:
  - Compare the value on the **left-hand side** with the value on the **right-hand side**.
  - Examples include:
    - `eq` (equals).
    - `ne` (not equals).
    - `lt` (less than).
    - `gt` (greater than).

```
Get-Help about_Comparison_Operators
```

NOT (`!`) Operator

- The `!` operator inverts a Boolean value:

- A `$false` statement becomes `$true`.
- A `$true` statement becomes `$false`.

Examples:

```
$condition = $false
if ( !$condition )
{
    Write-Output "The condition was true"
}
Write-Output (!$condition)
```

```
if (-not $condition) {
    Write-Output "Condition was false."
}
```

Other Operators

- PowerShell includes a variety of operators for tasks like arithmetic, logical comparisons, and more.
- Common examples:
  - **Logical Operators**: `and`, `or`, `not`.
  - **Arithmetic Operators**: `+`, ` `, ` `, `/`.
  - **Assignment Operators**: `=`, `+=`, `=`.

eq - Equal

```
$value = 5
if ( 5 -eq $value )
{
    Write-Output "The value is $value"
}
```

ne - Not Equal

```
$value = 4
if ( 5 -ne $value )
{
    Write-Output "The value $value is not 5"
}
```

gt - Greater Than

```
$value = 4
if ( 5 -gt $value )
{
    Write-Output "The value 5 is greater than $value"
}
```

ge - Greater Than or Equal

```
$value = 5
if ( 5 -ge $value )
{
    Write-Output "The value 5 is greater than or equal to $value"
}
```

lt - Less Than

```
$value = 6
if ( 5 -lt $value )
{
    Write-Output "The value 5 is less than $value"
}
```

le - Less Than or Equal

```
$value = 5
if ( 5 -le $value )
{
    Write-Output "The value 5 is less than or equal to $value"
}
```

is - Is the same type

```
$value = "test"
if ( $value -is [string] )
{
    Write-Output "The value $value is a string"
}
```

For Loop

```
for (<Init>; <Condition>; <Repeat>)
{
    <Statement list>
}
```

- The `For` loop has four key components:
  1. **Init**: Commands executed before the loop begins.
  2. **Condition**: Evaluates to `$true` or `$false` each time the loop runs.
     - If `$true`, the commands in the loop's block are executed.
     - If `$false`, the loop ends.
  3. **Repeat**: Commands executed each time the loop repeats.
  4. **Statement**: Commands run each time the loop is entered or repeated, enclosed in braces `{}`.

Example:

```
for ($i;$i -lt 15; $i++)
{
    Write-Output "Loop Number: $i"
}
$i = 0
```

```
for ($i = 0; $i -lt 5; $i++) {
    Write-Output "Iteration $i"
}
```

ForEach Loop

- A `foreach` loop iterates through a collection of objects and runs commands for each item.
- **Typical use case**: Traversing an array.
- **Key differences from `For` loop**:
  - `For`: Loops until a condition is met, potentially infinite.
  - `ForEach`: Loops a set number of times based on the collection's size.

Syntax:

```
foreach ($<item> in $<collection>)
{
    <statement list>
}
```

```
foreach ($item in $collection) {
    Write-Output "Processing $item"
}
```

- The variable `$item` is local to the loop and named by the user.

PowerShell Scripts

- A **script** is a plain text file containing PowerShell commands with a `.ps1` file extension.
- **How to run a script**:
  - Type the script's path and file name.
  - Use parameters to submit data and set options.

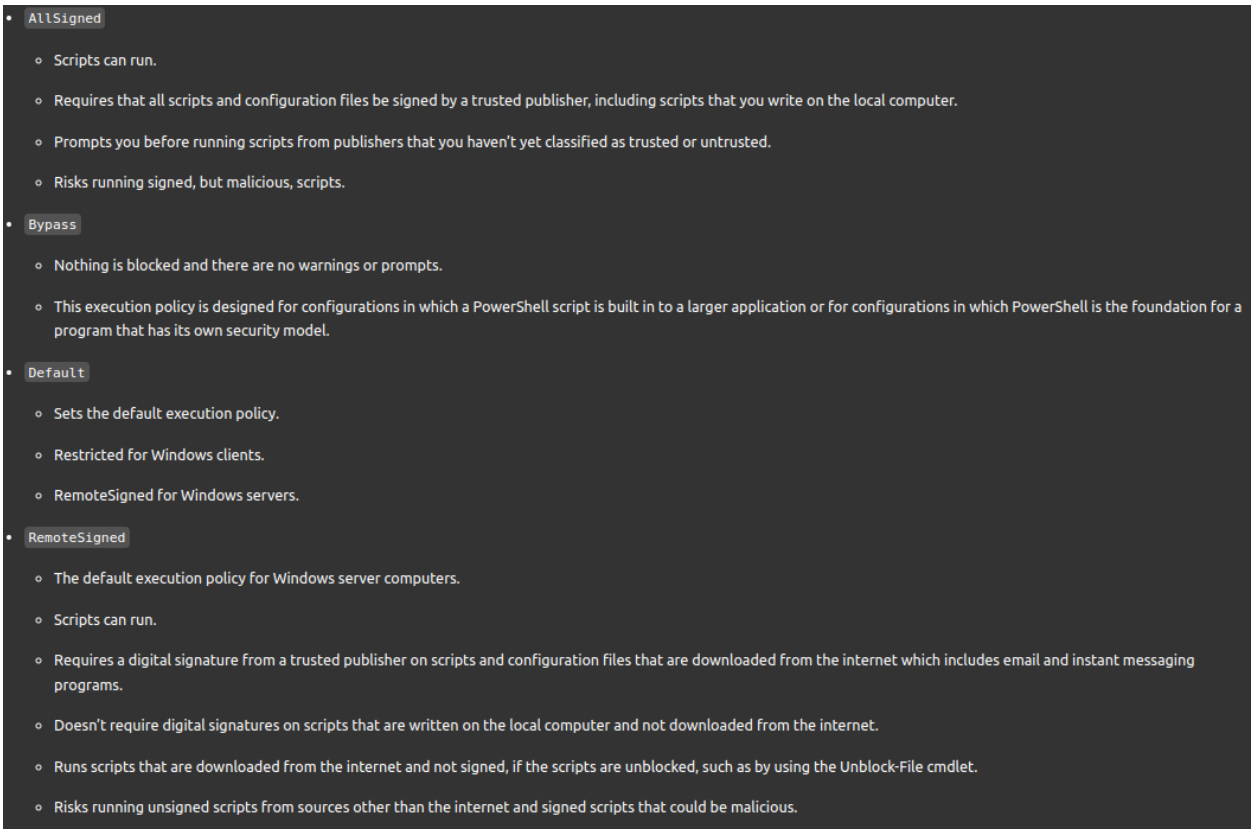
Advantages of Scripts:

- Save commands for later use.
- Simplify sharing commands.
- Execute multiple commands with a single script file.

Additional Features:

- `#Requires` special comment.
- Use of parameters and data sections.
- Digital signing for security.
- Ability to write Help topics for scripts and functions.

Execution Policy



- **Execution Policy**: Controls the conditions under which PowerShell loads configuration files and runs scripts.
- By default, Windows blocks scripts from running until the policy is changed.

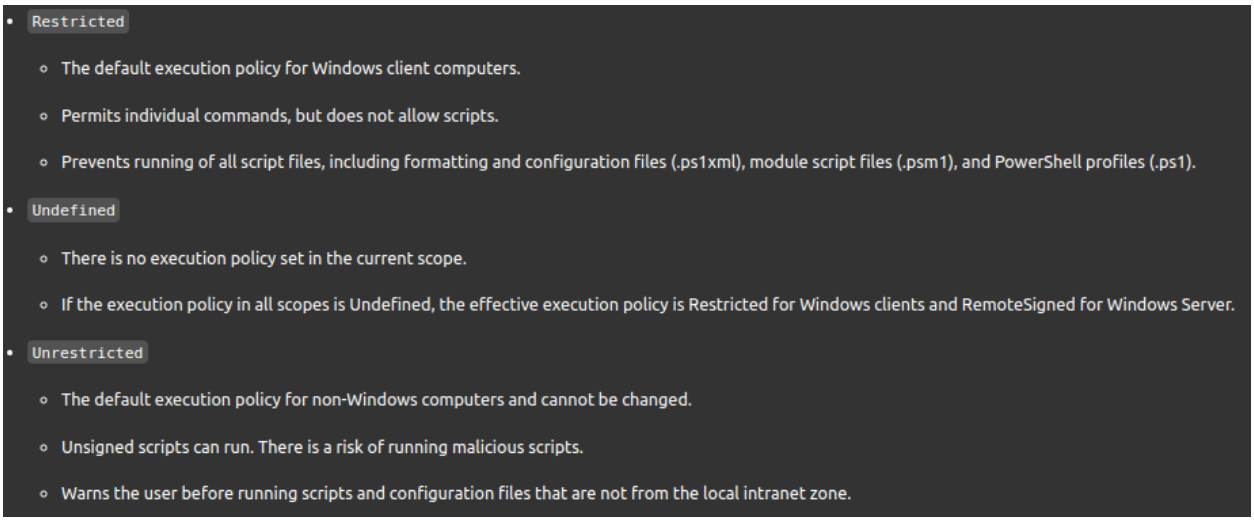
To change the execution policy:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Remot
eSigned
```

Notes:

- Execution policy only applies to PowerShell on Windows.
- Non-Windows platforms do not enforce execution policies.

Execution Policy Documentation



- For a complete description of execution policies and their differences, refer to the help page:

```
Get-Help about_Execution_Policies
```

PowerShell Functions

PowerShell Functions

- A **function** in PowerShell is a grouping of code with optional input and output.
- Functions allow for reusing code by calling it instead of duplicating it multiple times.
- Functions can be thought of as **methods** in object-oriented programming.
- Any part of your script can call a defined function.
- Functions declared will NOT remain if the terminal exits and reopens
    - Functions you want to remain persistant should be defined in the PowerShell `$PROFILE`

```
Function Write-Something {

            Param($item)

            Write-Host “You passed the parameter $item into the function”

}
```

Creating a PowerShell Function

- A function consists of a list of PowerShell statements with an assigned name.
- To run a function, type the function's name.
- The statements in the function execute as if they were typed directly into the command prompt.

Example:

```
function Get-Hello {
    Write-Output "Hello, World!"
}
```

- To call the function:

```
Get-Hello
# to call a function multiple times use a loop
for(i; i -le 2;i++){Get-Hello}
```

💡 Something is wrong with this loop. Fix it later.

Advanced Function Example

- Functions can be used to simplify repetitive tasks by consolidating reused code.
- Example: Creating a function to retrieve the DHCP service.

```
Write-Something "Hello!"
```

```
function Get-MultiplicativeResult {

Param ([int]$a,[int]$b)

$c = $a * $b

Write-Output $c

}
```

```
Get-MultiplicativeResult 5 10
```

```
function Get-DHCPService { Get-Service -Name DHCP }
```

Example:

```
function Get-DHCPService {
    Get-Service -Name DHCP
}
```

- Call the function like a cmdlet:

  ```
  Get-DHCPService
  ```

We can then call this function `Get-DHCPService` like a cmdlet.

- **Why use functions?**
  - They streamline code.
  - They make scripts easier to maintain and reuse.

Apply Your Learning

- Open PowerShell and build a custom function:
  - Define the function.
  - Use it to simplify a repetitive task.
  - Test your function by calling it.

```
function Get-FileLoc {
    Param([string]$filter)
    gci -recurse -filter "*$filter*"

 }

Get-FileLoc host
```

```
function Stop-PowerShell {
$procs=(Get-WmiObject Win32_Process | Where {$_.Name -eq 'powershell.exe'} `
        | ?{$_.ProcessId -ne $pid} | Select-Object -ExpandProperty ProcessId)

Do {
  ForEach ($i in $procs) {
        Get-Process powershell | ?{$_.Id -ne $pid} | Stop-Process | Out-Null
        $procs=(Get-WmiObject Win32_Process | Where {$_.Name -eq 'powershell.exe'} `
          | ?{$_.ProcessId -ne $pid} | Select-Object -ExpandProperty ProcessId)
  }
} While ($procs)
}
```

String Manipulation in PowerShell

- Strings in PowerShell are always **objects**, whether they are literal strings or variables.
- String objects have methods that provide powerful manipulation capabilities.
- Use the `Get-Member` cmdlet to explore available methods:

```
"this is a string" | Get-Member
#also
$string = "a string'
```

```
$string | Get-Member
#you may also call a method on a variable like...
Write-Host $string.ToUpper()
#or
Write-Host $string.Length
```

String Manipulation

- A **variable** is a name that represents an object.
- String literals and string variables both represent **string objects**.
- Use built-in **string methods** to manipulate and interact with string objects.

Select-Object

- The `Select-Object -Property` cmdlet creates a **collection** of user object properties.
- To convert the results into a string for manipulation, further processing is required.

Example:

Concatenation

```
$string = "This is a "

$string + "string"

Write-Output "$string string"
```

```
#The command below selects the collection of Names of users
$users = Get-LocalUser | Select-Object -Property Name

#The command below selects the strings within the collection of users.
$users = Get-LocalUser | Select-Object -ExpandProperty Name
```

```
Get-ADUser -Filter * | Select-Object -Property Name
```

ExpandProperty

- The `Select-Object -ExpandProperty` cmdlet directly converts the property argument (e.g., `Name`) into a string, ready for manipulation.

Example:

```
#The command below selects the strings within the collection of users.
$users = Get-LocalUser | Select-Object -ExpandProperty Name

#The script block of this foreach loop is where concatenation happens.
foreach ($user in $users){
    'User Name : ' + $user
}
```

```
Get-ADUser -Filter * | Select-Object -ExpandProperty Name
```

Replace Operator

- The **replace operator** replaces instances of a string (old values) with a specified string (new values).

```
"Old String" -replace "Old","New"
```

```
$ourstring = "Powershell is dumb and I hate it"

#The replace operator can be stacked multiple times to replace many items
$ourstring -replace "dumb","awesome" -replace "hate","love"
```

Syntax:

```
$string.Replace("oldValue", "newValue")
# To view tha avialable methods to use on a string:
"abc" | Get-Member | Where-Object MemberType -eq Method
```

Example:

```
"This is old text".Replace("old", "new")
```

Join Operator

- The **join operator** concatenates a set of strings into a single string.
- Strings are appended in the order they appear in the command.

```
-join "a", "b", "c"

#====================#

-join ("a", "b", "c")

#====================#

$z = "a", "b", "c"
-join $z

#====================#

"Windows", "PowerShell", "2.0" -join " "
```

Syntax:

```
-join "string1", "string2", "string3"
```

Example:

```
-join ("Power", "Shell")
```

Split Method

- The **split()** method splits a string into an array based on a specified delimiter (non-regex character).

```
"This.is.an.example.of.string.to.split.." -split "\."
```

Syntax:

```
$string.Split("delimiter")
```

Example:

```
"This,is,a,test".Split(",")
```

Regular Expressions (Regex)

- **Regex** is used across many programming languages to filter strings in large data outputs.
- Regex is a powerful tool, often likened to `[CTRL]+F` on adrenaline.
- Regex can perform tasks ranging from the mundane to the highly complex.

```
#Declare our string to apply our regex filter to.

$string = "HelloMyNameIS Nick123_55-4-11111-asdf_//123-45-6789-hELLOmYnAME Nick"
```

Syntax	Example	Purpose
`+`	`[a-z]+`	Match previous character or character set at least once
`?`	`[c-e]?`	Match at most once; previous character or character set may or may not exist
`*`	`App* and Bana*`	Any character; match any number of times
`{min,max}`	`[0-9]{1,3}`	Match at least `min` times, and at most `max` times
`{n}`	`[0-9]{3}`	Match exactly `n` times

Match and Pattern

- In Regex:
  - `\d` matches any number (0-9).
  - `[0-9]` achieves the same as `\d`.
  - `-` is escaped (`\-`) when used as a literal, to avoid being interpreted as a range.
  - `{2,3}` is a **repetition quantifier**, denoting that the preceding pattern must:
    - Appear at least 2 times.
    - Appear up to 3 times.

Example:

```
#Example 1

$string -Match "(\d{2,3}\-){2}\d{2,4}"
```

```
$string -match "\d{2,3}"
```

Repetition Quantifiers

- `{1,}` denotes **one or more** occurrences of a designated pattern.
- The `match` parameter can match a regex pattern against any provided string.

Example:

```
$string -match "\d{1,}"
```

Match and Pattern

- To match a string using the contents of a file(s), use the `Select-String` cmdlet.
- `Pattern` **vs** `Match`:
  - `Match` returns a Boolean indication of a match (`$true` or `$false`).
  - `Pattern` (used with `Select-String`) returns the **entire line** containing the matched string.

Example:

```
#Example 1

$string -Match "(\d{2,3}\-){2}\d{2,4}"



#Example 2

$string | Select-String -Pattern "(\d{2,3}\-){2}\d{2,4}"

#Example 3

#Foreach loops through all the matches and prints the collection to the screen.

$string | Select-String -Pattern "(\d{2,3}\-){2}\d{2,4}" | ForEach {$_.Matches.Value}
```

```
Get-Content file.txt | Select-String -Pattern "example"
```

Passing Matches to a ForEach Loop

- Use a `ForEach` loop in conjunction with `.Matches()` and `.Value()` methods to extract only the string that matches the `Pattern` parameter.

Example:

```
Get-Content file.txt | Select-String -Pattern "example" | For
Each-Object { $_.Matches.Value }
```

Regex Example: Pentesting Guide

- Download and store content from a webpage:

  ```
  Invoke-WebRequest -Uri "http://www.pentest-standard.org/in
  dex.php/PTES_Technical_Guidelines" -UseBasicParsing |
  Select-Object -ExpandProperty Content |
  Out-File ptes.txt
  ```

- Check the file size:

  ```
  (Get-Content .\ptes.txt).Length
  ```

- Use regex to search for the string `"nmap"`:

  ```
  Get-Content .\ptes.txt | Select-String -Pattern "nmap" | F
  orEach-Object { $_.Matches.Value } | Select -First 2
  ```

- Extract contextual information:

  ```
  Get-Content .\ptes.txt | Select-String -Pattern "nmap.+" |
  ForEach-Object { $_.Matches.Value } | Select -First 30 | S
  ```

```
elect -Last 10
```

- Filter for unique matches with a specific pattern:

```
Get-Content .\ptes.txt | Select-String -Pattern "nmap\s\-.
+" | ForEach-Object { $_.Matches.Value } | Sort-Object -Un
ique
```

- **Key Takeaway**: Regex allows for efficient filtering and analysis of large files without manual searching. Reinforce to students how powerful regex is for extracting meaningful information.

Anchors

- Anchors are regex characters used to match patterns at the **beginning** or **end** of a line.
- **Anchor Characters**:
  - `^`: Searches for patterns at the **beginning** of a line.
  - `$`: Searches for patterns at the **end** of a line.

Examples:

- Match "start" at the beginning of a line:

```
Select-String -Pattern "^start"
```

- Match "end" at the end of a line:

```
Select-String -Pattern "end$"
```

PowerShell Scripts

- A script is a text file containing one or more PowerShell commands, saved with a `.ps1` extension.

- Scripts allow for automation, reuse, and sharing of commands.

```
function Get-UserInfo {
    param($shadowfile)
    $shadow = get-content $shadowfile


    foreach ($shadowline in $shadow) {

    $username = ($shadowline -split ":")[0]
    $userid = ($shadowline -split ":")[2]
    $groupid = ($shadowline -split ":")[3]
    $fullname = ($shadowline -split ":")[4]
    $userhomedir = ($shadowline -split ":")[5]
    $defaultShell = ($shadowline -split ":")[6]

    Write-Output ("
    User Name: `t $username
    User Id: `t $userid
    GroupId: `t $groupid
    Name: `t $fullname
    Home : `t $userhomedir
    Shell: `t $defaultShell `n")
    }
}

Get-UserInfo `path-to-shadow-file`
```

```
#Gets all the process on a local machine and stores the output into a file name processes.txt in the current directory.
#===
get-process | Out-File -FilePath .\processes.txt

#open the file to see the contents.
#===
cat .\processes.txt

#Changes the occurences of the string "svchost" to XXXX and overwrites the file with the changes.
#===
$content = (Get-Content -Path .\processes.txt)
$content -replace "svchost","XXXX" | Set-Content -path .\processes.txt
#===
#open the file to see the changed contents.
cat .\processes.txt
```

Search and Replace Strings in a File

- Use PowerShell to search and replace strings within a file.

Example:

```
(Get-Content file.txt).Replace("oldString", "newString") | Se
t-Content file.txt
```

💡

Final Challenge Problem

From 1-100

Every # divisible by 3, print #-skibbidi

Every # divisible by 5, print #-toilet

Every # divisible by 3 AND 5, print #-rizzler

Every # not in the above, print #

Solution Algorithms for PE Problems

Count the number of occurrences of a pattern in a file:

```
(Get-Content \Path\to\file | Select-String -Pattern "Regular
Expression").Matches.Count
```

- You may wish to substitute -Pattern for `-AllMatches`, `-CaseSensitive`, ...

💡

When to use `-AllMatches`: when you may want to capture more than one instance of a pattern on a line. Other options tend to select the first instance.

To troubleshoot your solution and consider edge cases, you can view each item in the count printed to the console using:

```
(Get-Content \Path\to\file | Select-String -Pattern "Regular
Expression").Matches | Select-Object -ExpandProperty Value
```

Count the number of Unique occurrences of a pattern in a file:

```
(Get-Content \Path\to\file | Select-String -Pattern "Regular
Expression" -AllMatches | Select-Object -ExpandProperty Match
es | ForEach-Object { $_.Value } | Sort-Object -Unique).Count
```

1. **Define the pattern**
2. **Search for the pattern in the file**: Using PowerShell's `Select-String` cmdlet with regex.
3. **Filter for unique matches**: Using `Sort-Object -Unique`. You *must* sort before you can look for unique entries
4. **Count the unique matches**: Using `.Count`.

Which of the following is not a [member] available in [cmdlet] -type questions

```
#search_term is an array containing all the possible multiple
choice answers
$search_term = @("Possible", "answers")
#search_block is an array containing all of the possible memb
ers, created via pipelining. The command can be substituted f
or any other, as can the property filtered for.
$search_block = @(Get-Help | Get-Member | Where-Object -Prope
rty MemberType -eq Method)
```

```
#search_block names extracts the name data specifically from
search block.
$search_block_names = $search_block.Name

#this loop iterates over each term in search_term and checks
for it in the search_block_names, then identifies the missing
term.
foreach ($term in $search_term){
    if (-not($search_block_names -contains $term)) {
        Write-Output "Not found: $term"
    }
}
```

Composable Resources

RegEx Patterns

1. Email Addresses

Pattern: An email address typically consists of a username, an `@` symbol, and a domain.

Regex:

```
\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b
```

Explanation:

- `\b`: Word boundary.
- `[A-Za-z0-9._%+-]+`: Matches the username (letters, numbers, and special characters).
- `@`: Matches the `@` symbol.
- `[A-Za-z0-9.-]+`: Matches the domain name.
- `\.[A-Za-z]{2,}`: Matches the top-level domain (e.g., `.com`, `.org`).

2. Dates (MM/DD/YYYY or DD-MM-YYYY)

Pattern: Matches common date formats like `12/31/2023` or `31-12-2023`.

Regex:

```
\b(0[1-9]|1[0-2])[-\/](0[1-9]|[12][0-9]|3[01])[-\/](\d{4})\b
```

Explanation:

- `(0[1-9]|1[0-2])`: Matches months (`01` to `12`).
- `[-\/]`: Matches either a dash (`-`) or a slash (`/`) as a separator.
- `(0[1-9]|[12][0-9]|3[01])`: Matches days (`01` to `31`).
- `(\d{4})`: Matches a four-digit year.

---

3. Times (HH:MM, 24-Hour Format)

Pattern: Matches time in the 24-hour format like `23:59`.

Regex:

```
\b([01][0-9]|2[0-3]):[0-5][0-9]\b
```

Explanation:

- `([01][0-9]|2[0-3])`: Matches hours (`00` to `23`).
- `:[0-5][0-9]`: Matches minutes (`00` to `59`).

---

4. URLs

Pattern: Matches valid URLs, including `http`, `https`, and `www`.

Regex:

```
\b(https?:\/\/)?(www\.)?[A-Za-z0-9.-]+\.[A-Za-z]{2,}(\S*)\b
```

Explanation:

- `(https?:\/\/)?`: Matches optional `http://` or `https://`.

- `(www\.)?`: Matches optional `www.`.
- `[A-Za-z0-9.-]+`: Matches the domain name.
- `\.[A-Za-z]{2,}`: Matches the top-level domain.
- `(\S*)`: Matches the rest of the URL.

---

5. Social Security Numbers (SSNs)

Pattern: Matches US SSNs in the format `123-45-6789`.

Regex:

```
\b\d{3}-\d{2}-\d{4}\b
```

Explanation:

- `\d{3}`: Matches the first three digits.
- `-`: Matches the first dash.
- `\d{2}`: Matches the middle two digits.
- `-`: Matches the second dash.
- `\d{4}`: Matches the last four digits.

---

6. MAC Addresses

Pattern: Matches MAC addresses like `00:1A:2B:3C:4D:5E`.

Regex:

```
\b([A-Fa-f0-9]{2}[:-]){5}[A-Fa-f0-9]{2}\b
```

Explanation:

- `([A-Fa-f0-9]{2}[:-]){5}`: Matches the first five groups of two hexadecimal characters separated by `:` or `-`.
- `[A-Fa-f0-9]{2}`: Matches the last two hexadecimal characters.

---

7. Credit Card Numbers

Pattern: Matches 16-digit credit card numbers with optional spaces or dashes.

Regex:

```
\b\d{4}[-\s]?\d{4}[-\s]?\d{4}[-\s]?\d{4}\b
```

Explanation:

- `\d{4}`: Matches a group of four digits.
- `[-\s]?`: Matches an optional dash (`-`) or space.
- Repeated 4 times for 16 digits.

---

8. US ZIP Codes

Pattern: Matches ZIP codes in the formats `12345` or `12345-6789`.

Regex:

```
\b\d{5}(-\d{4})?\b
```

Explanation:

- `\d{5}`: Matches the first five digits.
- `(-\d{4})?`: Matches an optional hyphen followed by four digits.

---

9. Hexadecimal Numbers

Pattern: Matches hexadecimal numbers, e.g., `0x1A3F`.

Regex:

```
\b0x[A-Fa-f0-9]+\b
```

Explanation:

- `0x`: Matches the prefix for hexadecimal numbers.
- `[A-Fa-f0-9]+`: Matches one or more hexadecimal digits.

---

10. Floating-Point Numbers

Pattern: Matches numbers with optional decimals, e.g., `123.45` or `0.5`.

Regex:

```
\b\d+(\.\d+)?\b
```

Explanation:

- `\d+`: Matches one or more digits.
- `(\.\d+)?`: Matches an optional decimal point followed by digits.

---

11. RegEx for an Area Code Optionally Enclosed in Parentheses

Pattern: Matches a three-digit area code, optionally enclosed in parentheses.

Regex:

```
\(?\d{3}\)?
```

Instructions:

- `\(?`: Matches an optional opening parenthesis (`(`).
- `\d{3}`: Matches exactly three digits.
- `\)?`: Matches an optional closing parenthesis (`)`).

Example Matches:

- `(123)`
- `456`
- `(987)`

---

12. RegEx for a Telephone Number

Pattern: Matches a standard telephone number, with flexibility for separators (spaces or dashes) and optional parentheses around the area code.

Regex:

```
\(?\d{3}\)?[-\s]?\d{3}[-\s]?\d{4}
```

Instructions:

- `\(?\d{3}\)?`: Matches a three-digit area code, optionally enclosed in parentheses.
- `[-\s]?`: Matches an optional separator (dash or space).
- `\d{3}`: Matches the next three digits.
- `[-\s]?`: Matches another optional separator.
- `\d{4}`: Matches the final four digits of the phone number.

Example Matches:

- `(123) 456-7890`
- `123-456-7890`
- `123 456 7890`
- `(987)-654-3210`

---

13. RegEx for an IP Address

Pattern: Matches a valid IPv4 address (four groups of numbers between 0 and 255, separated by dots).

Regex:

```
\b((25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])\.){3}(25
[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])\b
```

Instructions:

- `\b`: Ensures the match starts and ends on a word boundary.
- `(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])`:
  - Matches a number from 0 to 255.
  - `25[0-5]`: Matches 250–255.

    - `2[0-4][0-9]`: Matches 200–249.
    - `1[0-9]{2}`: Matches 100–199.
    - `[1-9][0-9]`: Matches 10–99.
    - `[0-9]`: Matches 0–9.
- `\.`: Matches the literal dot separating the octets.
- `{3}`: Repeats the pattern for the first three octets.
- `\b`: Ensures the match ends cleanly.

Example Matches:

- `192.168.0.1`
- `10.0.0.1`
- `255.255.255.255`
- `127.0.0.1`

RegEx Cheat Sheet:

regular-expressions.pdf

Show Data-type

```
(Get-Process ssh | Select-Object -Property Handles).GetType()
# returns PSCustomObject
# To better understand the output of a pipeline Use -ExpandPrope
(Get-Process ssh | Select-Object -ExpandProperty Handles).GetTyp
#Also use -ExpandProperty to get more info about an object.
```