# Scripting If Then Statements

# $number = 10

# if ($number -gt 5)

# {

# Write-Host "The number is greater than 5"

}

$number = 10

if ($number -gt 20)

{

Write-Host "The number is greater than 20"

}

elseif ($number -gt 5)

{

Write-Host "The number is greater than 5 but less than or equal to 20"

}

Else

{

Write-Host "The number is 5 or less"

}

$number = 10

if ($number -gt 5)

{

if ($number -lt 15)

{

Write-Host "The number is between 5 and 15"

}

}

$number = 10

if ($number -lt 5 -or $number -gt 20)

{

Write-Host "The number is either less than 5 or greater than 20"

}

$day = 'Monday'

switch ($day) {

'Monday' { Write-Host "It's Monday!" }

'Tuesday' { Write-Host "It's Tuesday!" }

default { Write-Host "It's some other day!" }

}

$color = "red"

switch ($color) {

"red", "blue", "green" { Write-Output "Primary color" }

"yellow", "orange" { Write-Output "Secondary color" }

default { Write-Output "Other color" }

-eq : Equal

-ne : Not equal

-gt : Greater than

-lt : Less than

-ge : Greater than or equal to

-le : Less than or equal to

-and : Logical AND

-or : Logical OR

-not : Logical NOT

Looping

### **1. for Loop**

The for loop is typically used when you know in advance how many times you want to repeat a block of code. It consists of three parts: initialization, condition, and increment.

for ($i = 0; $i -lt 5; $i++)

{

# Code to execute repeatedly

Write-Host "Iteration $i"

}

$i = 0 is the initialization (starting value).

$i -lt 5 is the condition (loop will run as long as this is true).

$i++ is the increment (how the loop counter changes after each iteration).

for ($i = 1; $i -le 5; $i++)

{

Write-Host "This is loop iteration $i"

}

### **2. foreach Loop**

The foreach loop is used to iterate over each item in a collection, like an array or a list. It's ideal when you want to process all elements in a collection without needing an index.

$numbers = 1, 2, 3, 4, 5

foreach ($number in $numbers)

{

# Code to execute for each item

Write-Host "Number is $number"

}

$fruits = 'Apple', 'Banana', 'Cherry'

foreach ($fruit in $fruits) {

Write-Host "I like $fruit"

### **3. while Loop**

The while loop repeatedly executes a block of code as long as a specified condition is true. The condition is checked before the code inside the loop is executed.

while ($condition)

{

# Code to execute as long as the condition is true

}

$count = 1

while ($count -le 5) {

Write-Host "Current count: $count"

$count++

}

In this example, the loop runs as long as $count is less than or equal to 5, and increments $count by 1 in each iteration.

### **do-while Loop**

The do-while loop is similar to the while loop, but it checks the condition **after** the code inside the loop is executed. This means the code will always run at least once, even if the condition is initially false.

Do

{

# Code to execute

} while ($condition)

$count = 1

do {

Write-Host "Current count: $count"

$count++

} while ($count -le 5)

In this case, the loop will run at least once, even if $count starts as a value greater than 5.

### **5. do-until Loop**

The do-until loop is similar to the do-while loop, but the loop continues **until** a condition becomes true. This means it will loop until the condition evaluates to $true.

Do

{

# Code to execute

} until ($condition)

$count = 1

do {

Write-Host "Current count: $count"

$count++

} until ($count -gt 5)

In this case, the loop continues until $count is greater than 5.

### **6. break and continue Statements**

* **break**: Exits a loop entirely.
* **continue**: Skips the current iteration of the loop and moves to the next iteration.

for ($i = 1; $i -le 10; $i++) {

if ($i -eq 5) {

Write-Host "Breaking out of loop at $i"

break

}

Write-Host "Iteration $i"

}

In this case, the loop will exit when $i reaches 5.

for ($i = 1; $i -le 5; $i++) {

if ($i -eq 3) {

Write-Host "Skipping iteration $i"

continue

}

Write-Host "Iteration $i"

}

In this case, when $i equals 3, the loop will skip that iteration and continue with the next iteration.

### **7. Nested Loops**

You can nest loops inside each other to perform more complex iterations, such as iterating over a 2D array or performing multiple operations.

for ($i = 1; $i -le 3; $i++) {

for ($j = 1; $j -le 2; $j++) {

Write-Host "i=$i, j=$j"

}

}

This will produce the following output:

i=1, j=1

i=1, j=2

i=2, j=1

i=2, j=2

i=3, j=1

i=3, j=2

**for loop**: Best for a known number of iterations.

**foreach loop**: Best for iterating over elements in a collection.

**while loop**: Continues while the condition is true, checked before each iteration.

**do-while loop**: Executes the code at least once, checks the condition after each iteration.

**do-until loop**: Similar to do-while, but continues until the condition is true.

**break**: Exits the loop.

**continue**: Skips the current iteration and continues with the next one.

Other useful commands

### **10. Boolean Logic**

You can use Boolean logic (-and, -or, -not) to combine multiple conditions and create more complex conditions:

$age = 25

$hasLicense = $true

if ($age -ge 18 -and $hasLicense) {

Write-Host "Eligible to drive"

} else {

Write-Host "Not eligible to drive"

}

This example checks both conditions (age >= 18 and hasLicense), combining them using the -and operator.

### **1. Introduction to Regular Expressions in PowerShell**

Regular expressions (regex) are patterns used to match character combinations in strings. PowerShell leverages the .NET regex engine, which provides extensive support for regular expressions. The most common use cases are:

* Searching for patterns in text.
* Extracting data.
* Validating data.
* Replacing text.

#### **Regex Pattern Basics**

Here are some of the basic regex syntax elements:

* .: Matches any single character except newline characters.
* \d: Matches any digit (0-9).
* \D: Matches any non-digit character.
* \w: Matches any word character (alphanumeric + underscore).
* \W: Matches any non-word character.
* \s: Matches any whitespace character (space, tab, newline).
* \S: Matches any non-whitespace character.
* ^: Anchors the pattern to the start of the string.
* $: Anchors the pattern to the end of the string.
* \*: Matches 0 or more of the preceding element.
* +: Matches 1 or more of the preceding element.
* ?: Matches 0 or 1 of the preceding element (optional).
* {n,m}: Matches between n and m occurrences of the preceding element.
* |: Logical OR, matches one pattern or the other.
* []: Matches any one of the characters in the brackets.
* (): Groups a pattern together (captures matched text).

### **2. Using Regex with PowerShell Cmdlets**

PowerShell has several cmdlets for regex operations. Below are some of the most important ones:

#### **-match Operator (Used for matching a pattern)**

The -match operator checks if the regex pattern matches a string and returns a Boolean value. It also populates the $matches automatic variable with the matched groups.

powershell

Copy

# Check if a string contains a pattern (e.g., "abc")

$string = "The quick brown fox jumps over abc"

if ($string -match "abc") {

Write-Host "Pattern found!"

} else {

Write-Host "Pattern not found."

}

#### **-replace Operator (Used for replacing matched patterns)**

The -replace operator allows you to replace parts of a string that match a regular expression.

powershell

Copy

# Replace the word "quick" with "fast"

$string = "The quick brown fox"

$newString = $string -replace "quick", "fast"

Write-Host $newString # Output: "The fast brown fox"

#### **-split Operator (Splitting strings by a regex pattern)**

The -split operator splits a string into an array based on a regex pattern.

powershell

Copy

# Split the string by whitespace

$string = "The quick brown fox"

$words = $string -split "\s+"

$words # Output: "The", "quick", "brown", "fox"

#### **Select-String Cmdlet (Searches for patterns in files or strings)**

Select-String is used to search for text matching a regular expression in a file or string, similar to grep in Unix.

powershell

Copy

# Search for the pattern "fox" in a string

$string = "The quick brown fox jumps over the lazy dog"

Select-String -InputObject $string -Pattern "fox"

### **3. Advanced Regex Techniques in PowerShell**

Now that you understand the basics, let’s dive deeper into regex in PowerShell.

#### **Capturing Groups**

Captured groups allow you to extract specific parts of a match. The captured groups are stored in the $matches automatic variable.

powershell

Copy

# Example with capturing groups

$string = "John:25;Doe:30;Alice:22"

if ($string -match "(\w+):(\d+)") {

Write-Host "Name: $($matches[1]), Age: $($matches[2])"

}

In this example:

* (\w+) captures the name (one or more word characters).
* (\d+) captures the age (one or more digits).

The $matches array will contain the entire match and captured groups:

* $matches[0] will contain the entire match: John:25
* $matches[1] will contain the name: John
* $matches[2] will contain the age: 25

#### **Lookahead and Lookbehind Assertions**

Lookahead and lookbehind assertions are used to match a pattern only if it’s followed or preceded by another pattern.

**Positive Lookahead (?=)**: Ensures a pattern is followed by another pattern.

powershell

Copy

# Match a digit followed by a letter

$string = "123abc 456xyz"

if ($string -match "\d(?=\w)") {

Write-Host "Match found!" # Will match the "3" in "123abc"

}

**Negative Lookahead (?!)**: Ensures a pattern is not followed by another pattern.

powershell

Copy

# Match a digit that is NOT followed by a letter

$string = "123 456x"

if ($string -match "\d(?!\w)") {

Write-Host "Match found!" # Will match the "6" in "456"

}

**Lookbehind (?<=)**: Matches a pattern only if it is preceded by another pattern.

powershell

Copy

# Match a letter preceded by a digit

$string = "a1b c2d"

if ($string -match "(?<=\d)\w") {

Write-Host "Match found!" # Will match the "b" in "a1b"

}

**Negative Lookbehind (?<!)**: Ensures a pattern is not preceded by another pattern.

powershell

Copy

# Match a letter that is NOT preceded by a digit

$string = "a1b c2d"

if ($string -match "(?<!\d)\w") {

Write-Host "Match found!" # Will match the "c" in "c2d"

}

#### **Regex Quantifiers**

Quantifiers specify how many times a character or group should be matched.

* \*: Matches 0 or more occurrences.
* +: Matches 1 or more occurrences.
* {n}: Matches exactly n occurrences.
* {n,}: Matches n or more occurrences.
* {n,m}: Matches between n and m occurrences.

powershell

Copy

# Match one or more digits

$string = "123 abc 456"

if ($string -match "\d+") {

Write-Host "Match found!" # Will match "123" and "456"

}

powershell

Copy

# Match exactly two digits

$string = "I have 12 apples."

if ($string -match "\d{2}") {

Write-Host "Match found!" # Will match "12"

}

#### **Case Insensitive Matching**

By default, regex matching in PowerShell is case-sensitive. You can add the (?i) flag to make it case-insensitive.

powershell

Copy

# Case-insensitive match

$string = "The Quick Brown Fox"

if ($string -match "(?i)quick") {

Write-Host "Match found!" # Will match "Quick" or "quick"

}

#### **Using -match with Multiple Matches**

If you want to capture multiple matches (not just the first one), you can use Select-String to return all occurrences.

powershell

Copy

# Capture multiple matches

$string = "John:25; Alice:22; Bob:30"

$matches = Select-String -InputObject $string -Pattern "(\w+):(\d+)"

foreach ($match in $matches) {

Write-Host "Name: $($match.Matches.Groups[1].Value), Age: $($match.Matches.Groups[2].Value)"

}

This will output all names and ages from the string.

### **4. Replacing with Regex in PowerShell**

You can use the -replace operator to replace portions of a string that match a regex pattern. Additionally, you can reference captured groups in the replacement string.

#### **Basic Replacement**

powershell

Copy

# Replace digits with "X"

$string = "My phone number is 123-456-7890"

$newString = $string -replace "\d", "X"

Write-Host $newString # Output: "My phone number is XXX-XXX-XXXX"

#### **Using Captured Groups in Replacement**

powershell

Copy

# Swap the order of the name and age

$string = "John:25; Alice:22"

$newString = $string -replace "(\w+):(\d+)", '$2 years old, $1'

Write-Host $newString # Output: "25 years old, John; 22 years old, Alice"

### **5. Validating Input with Regex**

Regex can be used for input validation, such as validating email addresses, phone numbers, and other patterns.

#### **Validating an Email Address**

powershell

Copy

# Email regex pattern

$emailPattern = "^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"

$email = "test@example.com"

if ($email -match $emailPattern) {

Write-Host "Valid email address"

} else {

Write-Host "Invalid email address"

}

#### **Validating a Phone Number**

powershell

Copy

# Phone number regex pattern (basic example for US phone numbers)

$phonePattern = "^\(\d{3}\) \d{3}-\d{4}$"

$phone = "(123) 456-7890"

if ($phone -match $phonePattern) {

Write-Host "Valid phone number"

} else {

Write-Host "Invalid phone number"

}

### **6. Conclusion**

PowerShell’s support for regular expressions, powered by the .NET regex engine, allows you to perform complex text manipulation, data extraction, and validation. With -match, -replace, Select-String, and other cmdlets, you can leverage regex to solve a wide range of text processing tasks.

#### **Key Points to Remember:**

* Regular expressions are powerful but can be complex, so always test your patterns.
* Use Select-String for finding all matches in a string or file.
* Regular expressions in PowerShell support all standard .NET regex syntax, including assertions, quantifiers, and modifiers.

With this guide, you should now be equipped to perform advanced text processing and manipulation tasks using regular expressions in PowerShell!

### **1. Basic File and Directory Operations**

#### **Get-ChildItem: Lists files and directories.**

powershell

Copy

# List all files and directories in the current directory

Get-ChildItem

# List files and directories recursively

Get-ChildItem -Recurse

# List files with a specific extension (e.g., .txt)

Get-ChildItem -Filter \*.txt

#### **New-Item: Creates a new file or directory.**

powershell

Copy

# Create a new directory

New-Item -Path "C:\Test" -ItemType Directory

# Create a new file

New-Item -Path "C:\Test\example.txt" -ItemType File

#### **Remove-Item: Deletes a file or directory.**

powershell

Copy

# Delete a file

Remove-Item "C:\Test\example.txt"

# Delete a directory and its contents

Remove-Item "C:\Test" -Recurse -Force

#### **Rename-Item: Renames a file or directory.**

powershell

Copy

# Rename a file

Rename-Item "C:\Test\oldname.txt" -NewName "newname.txt"

### **2. File Content Operations**

#### **Get-Content: Reads the content of a file.**

powershell

Copy

# Display the content of a file

Get-Content "C:\Test\example.txt"

# Read the first 10 lines of a file

Get-Content "C:\Test\example.txt" -TotalCount 10

#### **Set-Content: Writes content to a file.**

powershell

Copy

# Write text to a file

Set-Content "C:\Test\example.txt" -Value "Hello, World!"

# Append text to a file

Add-Content "C:\Test\example.txt" -Value "Appended text."

### **3. System Information and Process Management**

#### **Get-Process: Lists all running processes.**

powershell

Copy

# List all processes

Get-Process

# List a specific process (e.g., notepad)

Get-Process -Name "notepad"

#### **Stop-Process: Stops a running process.**

powershell

Copy

# Stop a process by its name

Stop-Process -Name "notepad"

# Stop a process by its ID

Stop-Process -Id 1234

#### **Get-Service: Lists all services on the system.**

powershell

Copy

# List all services

Get-Service

# List a specific service (e.g., the Windows Update service)

Get-Service -Name "wuauserv"

#### **Start-Service: Starts a service.**

powershell

Copy

# Start a service by its name

Start-Service -Name "wuauserv"

#### **Stop-Service: Stops a service.**

powershell

Copy

# Stop a service by its name

Stop-Service -Name "wuauserv"

### **4. User Management**

#### **Get-LocalUser: Lists local users.**

powershell

Copy

# List all local users

Get-LocalUser

#### **New-LocalUser: Creates a new local user.**

powershell

Copy

# Create a new local user

New-LocalUser -Name "JohnDoe" -Password (ConvertTo-SecureString "P@ssw0rd" -AsPlainText -Force) -FullName "John Doe" -Description "A test user"

#### **Set-LocalUser: Modifies a local user account.**

powershell

Copy

# Set the account status of a user to disabled

Set-LocalUser -Name "JohnDoe" -Enabled $false

#### **Remove-LocalUser: Deletes a local user.**

powershell

Copy

# Delete a local user

Remove-LocalUser -Name "JohnDoe"

### **5. Network and Remote Operations**

#### **Test-Connection: Checks network connectivity (similar to ping).**

powershell

Copy

# Test connectivity to a remote server

Test-Connection -ComputerName "google.com"

# Test connectivity with a specific number of pings

Test-Connection -ComputerName "google.com" -Count 5

#### **Invoke-Command: Runs commands on remote computers.**

powershell

Copy

# Run a command on a remote machine

Invoke-Command -ComputerName "RemoteComputer" -ScriptBlock { Get-Process }

# Run a script on multiple remote machines

Invoke-Command -ComputerName "RemoteComputer1", "RemoteComputer2" -ScriptBlock { Get-Process }

### **6. Conditional and Looping**

#### **If Statement: Executes a block of code if a condition is true.**

powershell

Copy

# Check if a file exists

if (Test-Path "C:\Test\example.txt") {

Write-Host "File exists"

} else {

Write-Host "File does not exist"

}

#### **ForEach Loop: Loops through a collection of items.**

powershell

Copy

# Loop through all files in a directory and display their names

$files = Get-ChildItem -Path "C:\Test"

foreach ($file in $files) {

Write-Host $file.Name

}

#### **While Loop: Loops while a condition is true.**

powershell

Copy

# Loop until a number reaches 5

$i = 1

while ($i -le 5) {

Write-Host "Current value: $i"

$i++

}

### **7. Variables and Arrays**

#### **Variables: Create and use variables.**

powershell

Copy

# Create a variable

$myVar = "Hello, PowerShell!"

# Display the variable's value

Write-Host $myVar

#### **Arrays: Create and manipulate arrays.**

powershell

Copy

# Create an array

$myArray = @("apple", "banana", "cherry")

# Access an element in the array

$myArray[1] # Outputs "banana"

# Loop through an array

foreach ($item in $myArray) {

Write-Host $item

}

### **8. Working with Dates and Times**

#### **Get-Date: Gets the current date and time.**

powershell

Copy

# Get the current date and time

Get-Date

# Get the date and time in a specific format

Get-Date -Format "yyyy-MM-dd HH:mm:ss"

#### **Add-Days: Adds days to the current date.**

powershell

Copy

# Add 5 days to the current date

(Get-Date).AddDays(5)

### **9. Export and Import Data**

#### **Export-Csv: Export data to a CSV file.**

powershell

Copy

# Export a list of processes to a CSV file

Get-Process | Export-Csv -Path "C:\Test\processes.csv" -NoTypeInformation

#### **Import-Csv: Import data from a CSV file.**

powershell

Copy

# Import data from a CSV file

$processes = Import-Csv -Path "C:\Test\processes.csv"

$processes

### **10. Error Handling**

#### **Try-Catch: Handle errors in PowerShell scripts.**

powershell

Copy

try {

# Code that may cause an error

Get-Content "C:\NonExistentFile.txt"

} catch {

# Handle the error

Write-Host "An error occurred: $\_"

}

### **11. Aliases and Shortcuts**

#### **Get-Alias: List all aliases in PowerShell.**

powershell

Copy

# List all aliases

Get-Alias

# Get the alias for a specific command

Get-Alias -Name "ls"

#### **Set-Alias: Create or modify aliases.**

powershell

Copy

# Create a new alias for the Get-Process command

Set-Alias -Name gp -Value Get-Process

### **1. PowerShell Cmdlet Basics**

#### **Get-Help: Display help for cmdlets.**

powershell

Copy

# Get help for a cmdlet

Get-Help Get-Process

# Get detailed help for a cmdlet

Get-Help Get-Process -Detailed

# Get examples of how to use a cmdlet

Get-Help Get-Process -Examples

#### **Get-Command: Discover cmdlets, functions, workflows, etc.**

powershell

Copy

# List all available cmdlets

Get-Command

# List all cmdlets for managing files

Get-Command -Noun File

#### **Get-Item: Get the properties of a specific file or object.**

powershell

Copy

# Get the properties of a file

Get-Item "C:\Test\example.txt"

### **2. Working with Objects and Properties**

#### **Select-Object: Select properties from an object.**

powershell

Copy

# Select specific properties of a process

Get-Process | Select-Object -Property Name, Id, CPU

# Select unique values from an array

$values = 1, 2, 3, 3, 4

$values | Select-Object -Unique

#### **Format-Table: Format output as a table.**

powershell

Copy

# Format the output of processes as a table

Get-Process | Format-Table -Property Name, Id, CPU

# Format output with auto-size columns

Get-Process | Format-Table -Property Name, Id, CPU -AutoSize

#### **Format-List: Format output as a list of properties.**

powershell

Copy

# Format process information as a list

Get-Process | Format-List -Property Name, Id, CPU, StartTime

#### **Where-Object: Filter objects based on conditions.**

powershell

Copy

# Filter processes where CPU usage is greater than 100

Get-Process | Where-Object { $\_.CPU -gt 100 }

#### **ForEach-Object: Perform an action on each object in a pipeline.**

powershell

Copy

# Stop all processes with a CPU usage greater than 100

Get-Process | Where-Object { $\_.CPU -gt 100 } | ForEach-Object { Stop-Process -Id $\_.Id }

### **3. Pipeline Usage**

#### **Pipeline Basics: Understanding how data flows through cmdlets.**

powershell

Copy

# List all files and pass them to the Select-Object cmdlet

Get-ChildItem | Select-Object Name

# Get processes and pass them to Format-Table

Get-Process | Format-Table Name, Id, CPU

### **4. Working with Strings**

#### **String Manipulation: Working with string properties and methods.**

powershell

Copy

# Concatenate strings

$greeting = "Hello"

$name = "John"

$greeting + " " + $name # Outputs "Hello John"

# String formatting

$formattedString = "{0} {1}" -f $greeting, $name # Outputs "Hello John"

#### **Regular Expressions: Using regex with cmdlets like Select-String.**

powershell

Copy

# Search for a pattern in a file

Select-String -Path "C:\Test\example.txt" -Pattern "Hello"

# Use regex to match specific file types

Get-ChildItem | Where-Object { $\_.Name -match "\.txt$" }

### **5. Working with Dates and Times**

#### **Working with DateTime Objects.**

powershell

Copy

# Get the current date and time

Get-Date

# Calculate a date in the future

(Get-Date).AddDays(10)

# Format date output

(Get-Date).ToString("yyyy-MM-dd HH:mm:ss")

### **6. Variables and Arrays**

#### **Creating Variables:**

powershell

Copy

# Assign a value to a variable

$myVariable = "PowerShell is great!"

# Access variable content

Write-Host $myVariable

#### **Arrays: Creating and working with arrays.**

powershell

Copy

# Create an array

$myArray = @("apple", "banana", "cherry")

# Access an element in an array

$myArray[1] # Outputs "banana"

# Loop through an array

foreach ($fruit in $myArray) {

Write-Host $fruit

}

#### **Hash Tables: Storing data in key-value pairs.**

powershell

Copy

# Create a hash table

$person = @{

Name = "John"

Age = 30

Occupation = "Engineer"

}

# Access a value from the hash table

$person["Name"] # Outputs "John"

### **7. Error Handling**

#### **Try-Catch-Finally: Exception handling in PowerShell.**

powershell

Copy

try {

# Simulating a possible error (dividing by zero)

$result = 10 / 0

} catch {

Write-Host "An error occurred: $\_"

} finally {

Write-Host "This will always run"

}

#### **Throw: Throw an exception manually.**

powershell

Copy

# Throw an exception

throw "Something went wrong!"

#### **$Error Variable: Accessing the last error.**

powershell

Copy

# Check the last error that occurred

$Error[0]

### **8. Working with Files and Directories**

#### **Copy-Item: Copy files or directories.**

powershell

Copy

# Copy a file

Copy-Item "C:\Test\example.txt" -Destination "C:\Backup"

# Copy a directory

Copy-Item "C:\Test" -Destination "C:\Backup" -Recurse

#### **Move-Item: Move files or directories.**

powershell

Copy

# Move a file

Move-Item "C:\Test\example.txt" -Destination "C:\NewFolder"

# Move a directory

Move-Item "C:\Test" -Destination "C:\NewFolder"

#### **Get-Content: Read the contents of a file.**

powershell

Copy

# Display the content of a file

Get-Content "C:\Test\example.txt"

#### **Set-Content: Write to a file.**

powershell

Copy

# Write to a file (overwrites)

Set-Content "C:\Test\example.txt" -Value "This is new content."

# Append to a file

Add-Content "C:\Test\example.txt" -Value "This is appended content."

### **9. Advanced Scripting**

#### **Functions: Defining and using functions.**

powershell

Copy

# Define a function

function Get-Square {

param ($number)

return $number \* $number

}

# Call the function

Get-Square -number 5 # Outputs 25

#### **Scripting with Parameters:**

powershell

Copy

# Function with parameters

function Get-UserInfo {

param ($userName)

Write-Host "User: $userName"

}

# Call the function with a parameter

Get-UserInfo -userName "JohnDoe"

#### **Modules: Creating and using PowerShell modules.**

powershell

Copy

# Import a module

Import-Module -Name "C:\Path\To\Module.psm1"

# List imported modules

Get-Module

# Get help for a specific module

Get-Help -Name "ModuleName"

### **10. Working with Remoting**

#### **Enable-PSRemoting: Enable PowerShell Remoting on a system.**

powershell

Copy

# Enable PowerShell Remoting on the local computer

Enable-PSRemoting -Force

#### **Enter-PSSession: Start an interactive session with a remote computer.**

powershell

Copy

# Start a remote session

Enter-PSSession -ComputerName "RemoteComputer"

#### **Invoke-Command: Run commands on remote computers.**

powershell

Copy

# Run a command remotely

Invoke-Command -ComputerName "RemoteComputer" -ScriptBlock { Get-Process }

### **11. Job Scheduling and Automation**

#### **Start-Job: Run a background job.**

powershell

Copy

# Start a background job

Start-Job -ScriptBlock { Get-Process }

# Get the job results

Get-Job

# Receive job results

Receive-Job -Id 1

#### **Scheduled Tasks: Using Task Scheduler via PowerShell.**

powershell

Copy

# Create a scheduled task

$action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument "-File C:\scripts\myScript.ps1"

$trigger = New-ScheduledTaskTrigger -At 7am -Daily

Register-ScheduledTask -Action $action -Trigger $trigger -TaskName "My Task"

### **12. Security and Permissions**

#### **Get-ACL: Get the Access Control List of a file or folder.**

powershell

Copy

# Get ACL of a file or folder

Get-Acl "C:\Test\example.txt"

#### **Set-ACL: Set the Access Control List of a file or folder.**

powershell

Copy

# Set ACL on a file or folder

$acl = Get-Acl "C:\Test\example.txt"

$rule = New-Object System.Security.AccessControl.FileSystemAccessRule("User", "Read", "Allow")

$acl.SetAccessRule($rule)

Set-Acl -Path "C:\Test\example.txt" -AclObject $acl

### **13. Working with JSON and XML**

#### **ConvertTo-Json: Convert objects to JSON format.**

powershell

Copy

# Convert an object to JSON

$person = @{ Name = "John"; Age = 30 }

$person | ConvertTo-Json

#### **ConvertFrom-Json: Convert JSON data back to objects.**

powershell

Copy

# Convert JSON string to object

$jsonString = '{"Name": "John", "Age": 30}'

$jsonString | ConvertFrom-Json

### **1. Basic Usage of Get-Help**

The most basic usage of Get-Help is simply calling it with a cmdlet name. It returns a basic description of the cmdlet, including its syntax, parameters, and a brief overview of its functionality.

#### **Example:**

powershell

Copy

Get-Help Get-Process

This will return general help information for the Get-Process cmdlet, such as:

* A short description of what Get-Process does.
* The syntax for how to use the cmdlet.
* The available parameters.

### **2. Getting Detailed Help**

If you want more detailed help information about a cmdlet, including examples and additional details about each parameter, you can use the -Detailed flag.

#### **Example:**

powershell

Copy

Get-Help Get-Process -Detailed

This will show:

* A description of the cmdlet.
* A list of parameters.
* The syntax with more in-depth explanation.
* Some examples of how to use the cmdlet in different scenarios.

### **3. Viewing Examples with Get-Help**

To see practical examples of how to use a cmdlet, you can use the -Examples flag. This is especially useful when you want to see real-world use cases without digging through detailed documentation.

#### **Example:**

powershell

Copy

Get-Help Get-Process -Examples

The output will display a series of practical examples demonstrating how Get-Process can be used, which might include:

* How to list processes.
* How to filter processes.
* How to get detailed process information.

### **4. Getting Full Help Information**

If you want **all available help** for a cmdlet, including full parameter descriptions, examples, and more, you can use the -Full flag.

#### **Example:**

powershell

Copy

Get-Help Get-Process -Full

This will provide everything:

* A detailed description of the cmdlet.
* All available parameters, with an explanation for each.
* All the examples.
* Additional notes about the cmdlet.

This is the most comprehensive form of help and will give you everything you need to understand the cmdlet.

### **5. Displaying Parameter Information**

If you want to get information **only about the parameters** for a cmdlet, you can use the -Parameter flag, followed by the parameter name.

#### **Example:**

powershell

Copy

Get-Help Get-Process -Parameter Name

This will return detailed information about the -Name parameter for the Get-Process cmdlet. It will show you what values this parameter can take and how to use it correctly.

### **6. Getting Help for Aliases**

PowerShell cmdlets may have **aliases**, which are shorter versions of the cmdlet names. For example, Get-Help is aliased to help, and Get-Process is aliased to ps.

You can use Get-Help to learn about aliases too.

#### **Example:**

powershell

Copy

Get-Help help

This command will show the help for help (which is an alias for Get-Help), and you’ll see information on how to use Get-Help itself, including available flags and options.

### **7. Searching Help with Get-Help**

If you are not sure of the exact cmdlet or keyword you are looking for, you can search for cmdlets, functions, or topics using -Name or -Topic.

#### **Example:**

powershell

Copy

Get-Help -Name \*process\*

This will return a list of all cmdlets, functions, or topics whose name contains "process."

Alternatively, you can search help topics related to a specific area:

powershell

Copy

Get-Help -Category Function

This command will return help for all **functions** available in PowerShell.

### **8. Help with Modules**

If you're working with modules in PowerShell, Get-Help can also be used to provide help on cmdlets that are part of a specific module. For example, if you want help with cmdlets related to a specific module, such as Microsoft.PowerShell.Management, you can use:

#### **Example:**

powershell

Copy

Get-Help -Module Microsoft.PowerShell.Management

This will display a list of cmdlets and their descriptions from the Microsoft.PowerShell.Management module.

### **9. Getting Help for Scripts**

You can also use Get-Help to obtain help for custom scripts or functions you’ve created. For example, if you’ve written a script called MyScript.ps1 and want to get help for it, you can run:

#### **Example:**

powershell

Copy

Get-Help C:\Scripts\MyScript.ps1

This assumes that MyScript.ps1 has been documented with comment-based help (explained later in the "Comment-based Help" section).

### **10. Comment-based Help**

In PowerShell, you can add **comment-based help** to your scripts, functions, or modules. This allows you to use Get-Help to retrieve help for custom scripts or functions. Comment-based help uses special comment blocks within the script or function to describe the functionality, parameters, examples, etc.

#### **Example of Comment-based Help:**

powershell

Copy

function Get-Square {

<#

.SYNOPSIS

Returns the square of a number.

.DESCRIPTION

This function takes a number as input and returns its square.

.PARAMETER Number

The number to square.

.EXAMPLE

Get-Square -Number 5

Returns 25.

.NOTES

Created by your name.

#>

param (

[Parameter(Mandatory=$true)]

[int]$Number

)

return $Number \* $Number

}

With the comment-based help included in the function, you can run Get-Help to retrieve the documentation:

powershell

Copy

Get-Help Get-Square

This will display the synopsis, description, parameters, examples, and notes as written in the comments.

### **11. Updating Help**

PowerShell allows you to update the help documentation from the internet. This is especially useful if you want to get the most up-to-date help information for cmdlets that might have been updated since your version of PowerShell was installed.

To update the help files, run:

powershell

Copy

Update-Help

This will download the latest help files for your version of PowerShell. Keep in mind that the Update-Help cmdlet requires administrative privileges in some cases.

### **12. Using Get-Help with the -ShowWindow Option**

You can also display the help documentation in a separate, scrollable window by using the -ShowWindow flag.

#### **Example:**

powershell

Copy

Get-Help Get-Process -ShowWindow

This opens a graphical window with detailed help for the Get-Process cmdlet, making it easier to navigate and read.

### **13. Online Help**

For cmdlets that are part of modules that have online documentation, you can access the help documentation directly from the web by using the -Online flag.

#### **Example:**

powershell

Copy

Get-Help Get-Process -Online

This command will open a web page that contains the full documentation for the Get-Process cmdlet, including detailed usage and examples.

### **Summary of Get-Help Usage**

Here’s a summary of the most common ways to use Get-Help:

| **Command** | **Description** |
| --- | --- |
| Get-Help <cmdlet> | Basic help for a cmdlet. |
| Get-Help <cmdlet> -Detailed | Detailed help with parameter explanations and more information. |
| Get-Help <cmdlet> -Examples | Show examples of how to use the cmdlet. |
| Get-Help <cmdlet> -Full | Comprehensive help including full descriptions, examples, and notes. |
| Get-Help <cmdlet> -Parameter <param> | Get help on a specific parameter of a cmdlet. |
| Get-Help -Name <pattern> | Search for cmdlets or functions by name pattern. |
| Get-Help -Category <category> | View help content based on a category (e.g., cmdlet, function, etc.). |
| Get-Help -Module <module> | Get help for a specific module. |
| Get-Help <script-path> | Get help for a custom script or function (requires comment-based help). |
| Update-Help | Update local help files from the internet. |
| Get-Help <cmdlet> -ShowWindow | Display help content in a separate, scrollable window. |
| Get-Help <cmdlet> -Online | Open online documentation in a browser. |

### **Conclusion**

Get-Help is a versatile and powerful cmdlet that allows you to find detailed information about cmdlets, functions, scripts, modules, and more. It is essential for any PowerShell user, whether you're just starting or are an advanced user. By using various flags like -Detailed, -Examples, -Full, and -Parameter, you can quickly find the information you need to understand and use PowerShell effectively.

Remember, you can always access up-to-date help with Update-Help, and if you’re working with custom functions or scripts, comment-based help is an excellent way to document your code and make it accessible via Get-Help.

### **1. List All Categories in PowerShell**

PowerShell organizes help information into categories. You can view available categories with the Get-Help cmdlet.

#### **Available Categories**

Here’s a general list of categories that you can find in PowerShell:

* **Cmdlet**: Basic cmdlets provided by PowerShell and its modules.
* **Function**: PowerShell functions created either natively or by users.
* **Alias**: Shortened names for cmdlets.
* **Provider**: Cmdlets related to file system, registry, and other providers.
* **HelpFile**: Help information related to modules or cmdlets.
* **Script**: Scripts (often user-created).
* **Workflow**: Workflows for managing more complex tasks in PowerShell.

#### **How to list all available categories:**

powershell

Copy

Get-Help -Category \*

### **2. List All Modules in PowerShell**

Modules in PowerShell are collections of cmdlets, functions, variables, and more. To list all installed modules, you can use the Get-Module cmdlet.

#### **Command to list all available modules:**

powershell

Copy

Get-Module -ListAvailable

This will return a list of all modules that are installed on your system.

#### **Example output:**

plaintext

Copy

ModuleType Version Name ExportedCommands

---------- ------- ---- ----------------

Binary 1.0.0.0 Microsoft.PowerShell.Management {Get-Command, Get-Process, Get-Service, Get-EventLog...}

Script 1.0.0.0 PSReadLine {Get-Command, Get-Process, Get-Service, Get-EventLog...}

Each module will contain cmdlets, functions, and scripts related to that module.

### **3. List All Functions in PowerShell**

Functions in PowerShell are blocks of code that perform a specific task and can be reused.

#### **Command to list all available functions:**

powershell

Copy

Get-Command -CommandType Function

This will return all functions available in the session, both native and custom functions created by users.

#### **Example output:**

plaintext

Copy

CommandType Name Version Source

----------- ---- ------- ------

Function Add-Content 7.1.3 Microsoft.PowerShell.Management

Function Clear-Host 7.1.3 Microsoft.PowerShell.Utility

Function Get-Help 7.1.3 Microsoft.PowerShell.Management

Function Set-Content 7.1.3 Microsoft.PowerShell.Management

...

### **4. List All Cmdlets in PowerShell**

Cmdlets are the core commands of PowerShell, usually written in C# or managed code. They perform various operations such as file manipulation, system management, and more.

#### **Command to list all available cmdlets:**

powershell

Copy

Get-Command -CommandType Cmdlet

This will list all cmdlets available in your PowerShell environment.

#### **Example output:**

plaintext

Copy

CommandType Name Version Source

----------- ---- ------- ------

Cmdlet Add-Content 7.1.3 Microsoft.PowerShell.Management

Cmdlet Clear-Host 7.1.3 Microsoft.PowerShell.Utility

Cmdlet Get-Help 7.1.3 Microsoft.PowerShell.Management

Cmdlet Get-Process 7.1.3 Microsoft.PowerShell.Management

Cmdlet Set-Content 7.1.3 Microsoft.PowerShell.Management

...

This command will list all cmdlets, their version, and the module (or source) they belong to.

### **5. Detailed Breakdown of Command Types in PowerShell**

To further understand the types of commands in PowerShell, here’s a breakdown of the different command types you can list with Get-Command:

* **Cmdlet**: A command implemented in PowerShell's runtime, generally implemented in a module.
* **Function**: A set of PowerShell statements grouped together to perform a task.
* **Alias**: A shortcut or nickname for another cmdlet.
* **Workflow**: A special type of cmdlet for running tasks in the background.
* **ExternalScript**: An external script file (e.g., .ps1 file) you can run in PowerShell.
* **Application**: Executable files or programs you can run from PowerShell.

#### **Command to list all types of commands:**

powershell

Copy

Get-Command

This will list **all** commands, including cmdlets, functions, workflows, aliases, external scripts, and applications.

### **6. Getting Help for Specific Categories, Cmdlets, or Functions**

You can also use the Get-Help cmdlet to get more detailed information on a specific cmdlet, function, or module.

#### **Example: Get detailed help for a cmdlet**

powershell

Copy

Get-Help Get-Process -Full

This will show the full help information for the Get-Process cmdlet, including examples, parameter descriptions, and more.

#### **Example: Get help for a specific module**

powershell

Copy

Get-Help -Module Microsoft.PowerShell.Management

This will show help for the Microsoft.PowerShell.Management module, which includes cmdlets for managing processes, services, and more.

### **7. Searching for Cmdlets, Functions, or Modules**

You can search for specific cmdlets, functions, or modules using Get-Help and Get-Command with patterns.

#### **Search cmdlets by name:**

powershell

Copy

Get-Command -Name \*process\*

This will list all cmdlets, functions, or scripts whose name contains the string "process."

#### **Search for cmdlets by category (e.g., Cmdlets related to services):**

powershell

Copy

Get-Help -Category Cmdlet | Where-Object { $\_.Name -like "\*service\*" }

This will list cmdlets related to services, based on the string "service."

### **8. Using Get-Help to Get Information About Modules**

If you are interested in getting information about a specific module and its cmdlets, you can use the Get-Help cmdlet as follows:

powershell

Copy

Get-Help -Module Microsoft.PowerShell.Management

This will show you all cmdlets and functions in the Microsoft.PowerShell.Management module, which is a standard module for managing various system-related tasks (like processes, services, etc.).

### **Conclusion**

To recap, here’s a list of commands to explore categories, modules, functions, and cmdlets in PowerShell:

* **List categories**: Get-Help -Category \*
* **List modules**: Get-Module -ListAvailable
* **List functions**: Get-Command -CommandType Function
* **List cmdlets**: Get-Command -CommandType Cmdlet
* **Get help for cmdlets**: Get-Help <Cmdlet-Name> -Full
* **Search for specific cmdlets**: Get-Command -Name \*pattern\*
* **Get help for modules**: Get-Help -Module <Module-Name>

With these commands, you can explore the PowerShell environment and access detailed help information for the various cmdlets, functions, and modules you work with.

### **How to View Properties of Objects in PowerShell**

To get a list of all the properties of any object in PowerShell, you can use the Select-Object cmdlet or Format-List cmdlet. The Select-Object cmdlet with -Property \* lists all properties of an object, while Format-List provides a formatted view.

For example:

powershell

Copy

# Get all properties of a process

Get-Process | Select-Object -Property \*

Or:

powershell

Copy

# Get a detailed list of all properties of a file

Get-Item "C:\path\to\file.txt" | Format-List \*

### **Common Properties of Common PowerShell Objects**

Here are some of the **common types of objects** you might deal with in PowerShell, along with examples of some properties they typically expose.

#### **1. Get-Process Output (Process Object)**

Each process object has several properties describing the process.

powershell

Copy

$process = Get-Process

$process | Format-List \*

**Common Properties:**

* **Id**: The process ID.
* **Name**: The name of the process.
* **CPU**: The amount of CPU time used.
* **WorkingSet**: The amount of physical memory used by the process.
* **StartTime**: The time when the process started.
* **Responding**: Whether the process is responding.

#### **2. Get-Service Output (ServiceController Object)**

Each service object represents a service on your system.

powershell

Copy

$service = Get-Service

$service | Format-List \*

**Common Properties:**

* **Status**: The current status of the service (e.g., Running, Stopped).
* **Name**: The name of the service.
* **DisplayName**: The display name of the service.
* **DependentServices**: Services that depend on this service.
* **ServicesDependedOn**: Services that this service depends on.

#### **3. Get-ChildItem Output (FileInfo or DirectoryInfo Object)**

This command provides information about files or directories.

powershell

Copy

$file = Get-Item "C:\path\to\file.txt"

$file | Format-List \*

**Common Properties:**

* **Name**: The name of the file or directory.
* **FullName**: The full path to the file or directory.
* **Extension**: The file extension (e.g., .txt, .jpg).
* **Length**: The size of the file (in bytes).
* **CreationTime**: The time the file or directory was created.
* **LastWriteTime**: The last time the file was written to.
* **Attributes**: The file attributes (e.g., read-only, hidden).
* **IsReadOnly**: A boolean indicating if the file is read-only.

#### **4. Get-EventLog Output (EventLogRecord Object)**

Represents a log entry in the Windows event log.

powershell

Copy

$eventLog = Get-EventLog -LogName Application

$eventLog | Format-List \*

**Common Properties:**

* **TimeGenerated**: The time the event was generated.
* **EntryType**: The type of event (e.g., Error, Information).
* **Message**: The message of the event.
* **Source**: The source of the event.
* **EventID**: The event ID.
* **UserName**: The user associated with the event.

#### **5. Get-ADUser Output (ADUser Object)**

If you are using the Active Directory module, this cmdlet returns a user object.

powershell

Copy

$adUser = Get-ADUser -Identity "username"

$adUser | Format-List \*

**Common Properties:**

* **SamAccountName**: The user’s SAM account name.
* **Name**: The full name of the user.
* **GivenName**: The user’s first name.
* **Surname**: The user’s last name.
* **DistinguishedName**: The distinguished name (DN) of the user.
* **Enabled**: Whether the user account is enabled.
* **LastLogonDate**: The last logon date and time.
* **EmailAddress**: The user’s email address.

#### **6. Get-ComputerInfo Output (ComputerInfo Object)**

This cmdlet gives details about the computer system.

powershell

Copy

$computerInfo = Get-ComputerInfo

$computerInfo | Format-List \*

**Common Properties:**

* **CsName**: The name of the computer.
* **OsArchitecture**: The architecture of the operating system (e.g., x64, x86).
* **OsFamily**: The family of the operating system (e.g., Windows).
* **WindowsVersion**: The version of the Windows operating system.
* **TotalPhysicalMemory**: The total physical memory (RAM) in the system.
* **Cpu**: Information about the CPU.

#### **7. Get-NetAdapter Output (NetAdapter Object)**

Provides information about network adapters.

powershell

Copy

$netAdapter = Get-NetAdapter

$netAdapter | Format-List \*

**Common Properties:**

* **Name**: The name of the network adapter.
* **InterfaceDescription**: A description of the network interface.
* **Status**: The operational status (e.g., Up, Down).
* **MacAddress**: The MAC address of the adapter.
* **LinkSpeed**: The speed of the connection.

#### **8. Get-Command Output (CommandInfo Object)**

The output of Get-Command represents the metadata about cmdlets, functions, scripts, etc.

powershell

Copy

$commandInfo = Get-Command Get-Process

$commandInfo | Format-List \*

**Common Properties:**

* **Name**: The name of the cmdlet, function, or command.
* **CommandType**: The type of command (e.g., Cmdlet, Function, Application).
* **Definition**: The script definition of the command.
* **Module**: The module that the command belongs to.

### **How to Explore Properties of Objects**

You can use several methods to explore the properties of objects:

**Format-List \***: This lists all properties of an object in a readable format.  
  
 powershell  
Copy  
Get-Process | Format-List \*

**Select-Object -Property \***: This shows all properties in the output.  
  
 powershell  
Copy  
Get-Process | Select-Object -Property \*

**Get-Member**: This cmdlet lists the members (properties, methods) of an object.  
  
 powershell  
Copy  
Get-Process | Get-Member

### **Conclusion**

PowerShell objects have various properties depending on their types. There is no single list of all properties, as they vary by object class. By using commands like Get-Member, Select-Object, and Format-List \*, you can explore properties dynamically for any object you encounter in PowerShell.

Feel free to ask for more specific examples or further details!

In PowerShell, methods are actions or operations that an object can perform. Methods are associated with specific types of objects and can be invoked to manipulate or interact with the object in various ways. PowerShell gives you the ability to work with methods just as you do with properties, and they are available to you via the objects in the pipeline.

### **How to List Methods in PowerShell**

You can view the methods of any object using the Get-Member cmdlet. This cmdlet will show you both properties and methods available for a specific object. For example:

powershell

Copy

# List methods for a process object

Get-Process | Get-Member -MemberType Method

This will return only the methods available for the process object. You can do similar things for any other object to discover its available methods.

### **Common Methods in PowerShell Objects**

Here’s a list of methods available for some of the commonly used object types in PowerShell.

### **1. Methods for System.IO.FileInfo (File Objects)**

When you use Get-Item or Get-ChildItem to get information about files, the resulting objects are FileInfo objects. These objects have several methods.

#### **Example:**

powershell

Copy

$file = Get-Item "C:\path\to\file.txt"

$file | Get-Member -MemberType Method

**Common Methods:**

**Delete()**: Deletes the file.  
  
 powershell  
Copy  
$file.Delete()

**CopyTo()**: Copies the file to another location.  
  
 powershell  
Copy  
$file.CopyTo("C:\path\to\newfile.txt")

**MoveTo()**: Moves the file to a new location.  
  
 powershell  
Copy  
$file.MoveTo("C:\newpath\file.txt")

**Open()**: Opens the file as a FileStream object.  
  
 powershell  
Copy  
$file.Open()

**OpenRead()**: Opens the file for reading.  
  
 powershell  
Copy  
$file.OpenRead()

**OpenWrite()**: Opens the file for writing.  
  
 powershell  
Copy  
$file.OpenWrite()

### **2. Methods for System.String (String Objects)**

String objects in PowerShell also have many methods that allow you to manipulate strings.

#### **Example:**

powershell

Copy

$string = "Hello, World!"

$string | Get-Member -MemberType Method

**Common Methods:**

**ToUpper()**: Converts all characters to uppercase.  
  
 powershell  
Copy  
$string.ToUpper() # Output: "HELLO, WORLD!"

**ToLower()**: Converts all characters to lowercase.  
  
 powershell  
Copy  
$string.ToLower() # Output: "hello, world!"

**Trim()**: Removes all leading and trailing white-space characters.  
  
 powershell  
Copy  
$string.Trim()

**Substring()**: Extracts a substring from the string.  
  
 powershell  
Copy  
$string.Substring(7, 5) # Output: "World"

**Replace()**: Replaces all occurrences of a specified string.  
  
 powershell  
Copy  
$string.Replace("World", "Universe") # Output: "Hello, Universe!"

**Split()**: Splits a string into an array of substrings based on a delimiter.  
  
 powershell  
Copy  
$string.Split(",") # Output: ["Hello", " World!"]

**Contains()**: Checks if a substring exists within the string.  
  
 powershell  
Copy  
$string.Contains("World") # Output: True

### **3. Methods for System.DateTime (DateTime Objects)**

The DateTime objects represent date and time, and they come with methods for manipulating dates and times.

#### **Example:**

powershell

Copy

$date = Get-Date

$date | Get-Member -MemberType Method

**Common Methods:**

**AddDays()**: Adds a specified number of days to the date.  
  
 powershell  
Copy  
$date.AddDays(5) # Adds 5 days to the current date

**AddMonths()**: Adds a specified number of months to the date.  
  
 powershell  
Copy  
$date.AddMonths(2) # Adds 2 months to the current date

**ToString()**: Converts the DateTime object to a string with a specified format.  
  
 powershell  
Copy  
$date.ToString("yyyy-MM-dd") # Output: "2025-03-19"

**ToUniversalTime()**: Converts the local time to Coordinated Universal Time (UTC).  
  
 powershell  
Copy  
$date.ToUniversalTime()

**Subtract()**: Subtracts one DateTime from another and returns the time difference.  
  
 powershell  
Copy  
$date.Subtract([datetime]"2025-01-01") # Output: 77 days

### **4. Methods for System.ServiceProcess.ServiceController (Service Objects)**

When you use Get-Service, the objects are ServiceController objects, which represent Windows services.

#### **Example:**

powershell

Copy

$service = Get-Service -Name "wuauserv"

$service | Get-Member -MemberType Method

**Common Methods:**

**Start()**: Starts the service.  
  
 powershell  
Copy  
$service.Start()

**Stop()**: Stops the service.  
  
 powershell  
Copy  
$service.Stop()

**Pause()**: Pauses the service.  
  
 powershell  
Copy  
$service.Pause()

**Continue()**: Continues the service if it was paused.  
  
 powershell  
Copy  
$service.Continue()

**Refresh()**: Refreshes the service’s properties.  
  
 powershell  
Copy  
$service.Refresh()

### **5. Methods for System.Collections.ArrayList (ArrayList Objects)**

ArrayList is a collection object that you can use to store and manipulate data in a dynamic array.

#### **Example:**

powershell

Copy

$arrayList = New-Object System.Collections.ArrayList

$arrayList | Get-Member -MemberType Method

**Common Methods:**

**Add()**: Adds an element to the array list.  
  
 powershell  
Copy  
$arrayList.Add(10)

**Remove()**: Removes the first occurrence of an element.  
  
 powershell  
Copy  
$arrayList.Remove(10)

**Insert()**: Inserts an element at a specified index.  
  
 powershell  
Copy  
$arrayList.Insert(0, 5)

**Clear()**: Removes all elements from the array list.  
  
 powershell  
Copy  
$arrayList.Clear()

**Contains()**: Checks if the array list contains a specific element.  
  
 powershell  
Copy  
$arrayList.Contains(10) # Output: True or False

### **6. Methods for System.Net.WebClient (WebClient Objects)**

WebClient is used for downloading and uploading data over HTTP/HTTPS.

#### **Example:**

powershell

Copy

$webClient = New-Object System.Net.WebClient

$webClient | Get-Member -MemberType Method

**Common Methods:**

**DownloadFile()**: Downloads a file from a specified URL to a local path.  
  
 powershell  
Copy  
$webClient.DownloadFile("http://example.com/file.zip", "C:\path\to\file.zip")

**UploadFile()**: Uploads a file to a specified URL.  
  
 powershell  
Copy  
$webClient.UploadFile("http://example.com/upload", "C:\path\to\file.zip")

**DownloadString()**: Downloads a string from a URL.  
  
 powershell  
Copy  
$webClient.DownloadString("http://example.com/data")

**UploadString()**: Uploads a string to a URL.  
  
 powershell  
Copy  
$webClient.UploadString("http://example.com/upload", "POST", "data")

### **How to Explore Methods for Any Object in PowerShell**

To explore methods for any object in PowerShell, you can use:

**Get-Member -MemberType Method**: Lists the methods of an object.  
  
 powershell  
Copy  
$object | Get-Member -MemberType Method

**$object.GetType().GetMethods()**: For more advanced exploration, you can use this to reflect all methods on an object in a .NET way.  
  
 powershell  
Copy  
$object.GetType().GetMethods()

### **Conclusion**

In PowerShell, methods are actions you can perform on objects, and they vary depending on the object type. You can discover available methods by using the Get-Member cmdlet. Each object class (such as System.IO.FileInfo, System.String, System.DateTime, etc.) will have its own set of methods that allow you to manipulate or interact with the data.

Let me know if you'd like more detailed examples on a specific class or method!

Powershell kata