PowerShell Essential and Scripting

Overview of PowerShell

- **PowerShell** is a task automation and configuration management framework developed by **Microsoft**.
- It consists of:
  - A **command-line shell**, and
  - A **scripting language** built on the **.NET Framework / .NET Core**.
- It is designed for **system administrators**, **DevOps professionals**, and **IT automation** tasks.

History and Evolution

Version	Platform	Description
PowerShell 1.0 (2006)	Windows	Introduced with Windows Server 2008
PowerShell 2.0 (2009)	Windows	Added remoting and advanced scripting features
PowerShell 3.0–5.1	Windows	Better workflow, debugging, and integration with Windows Management Framework
PowerShell 6.0 (Core)	Cross-platform	Based on .NET Core; runs on Windows, Linux, and macOS
PowerShell 7.x	Cross-platform	Modern version with improved performance and compatibility

Features

1. Command-line Interface (CLI) – Execute commands directly.
2. Scripting Language – Write scripts to automate tasks.
3. Cmdlets – Specialized .NET commands (e.g., Get-Process, Get-Service).
4. Pipeline Support – Pass output of one command as input to another.
   Example:
   ```
   Get-Process | Where-Object {$_.CPU -gt 100}
   ```
5. Object-Oriented Output – Unlike other shells (which output text), PowerShell outputs .NET objects.
6. Remoting – Manage remote computers using Enter-PSSession or Invoke-Command.
7. Extensibility – Supports modules, snap-ins, and custom cmdlets.
8. Integration – Deep integration with Windows tools, Active Directory, Azure, etc.

PowerShell vs. Command Prompt (CMD)

Feature	Command Prompt	PowerShell
Output Type	Text	Objects
Extensibility	Limited	Highly extensible
Scripting Language	Batch (.bat)	PowerShell Script (.ps1)
Cross-platform	No	Yes (PowerShell Core & 7+)
Automation Support	Basic	Advanced automation capabilities

Common Use Cases

- Automating system administration tasks (e.g., backups, updates).
- Managing Active Directory users and groups.
- Deploying software or patches.
- Managing Azure and Microsoft 365 environments.
- Gathering system inventory reports.

Environment Setup

Installing PowerShell

Windows

- PowerShell is pre-installed on most Windows systems.
- To update or install PowerShell 7 (latest version):
  - Go to: https://github.com/PowerShell/PowerShell
  - Download the installer (MSI package).
  - Run the installer and follow the prompts.

macOS

- Use **Homebrew**:

  ```
  brew install --cask powershell
  ```

- Run PowerShell:

  ```
  pwsh
  ```

Linux (Ubuntu example)

```
sudo apt-get update
sudo apt-get install -y wget apt-transport-https software-properties-common
wget -q https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
sudo apt-get install -y powershell
```

- Run PowerShell:

  ```
  pwsh
  ```

- Checking PowerShell Version

  $PSVersionTable.PSVersion

- PowerShell File Types

File Type	Description
.ps1	Script file
.psm1	Module file
.psd1	Data file
.ps1xml	Formatting or type data file
.psc1	Console file

Running Scripts

1. Save your script with a .ps1 extension.
2. Change execution policy if needed:

   Set-ExecutionPolicy RemoteSigned
3. Run your script:

   .\myscript.ps1

Note: By default, PowerShell restricts script execution for security reasons.

Common Execution Policies

Policy	Description
Restricted	No scripts can run.
AllSigned	Only scripts signed by a trusted publisher can run.
RemoteSigned	Locally created scripts can run; remote ones must be signed.
Unrestricted	All scripts can run.
Bypass	No restrictions or warnings.

PowerShell Integrated Scripting Environment (ISE)

What is PowerShell ISE?

- **PowerShell ISE** is a **graphical user interface (GUI)** for writing, testing, and debugging PowerShell scripts.
- It comes **pre-installed** with Windows PowerShell (up to version 5.1).
- PowerShell 7 and newer versions use **Visual Studio Code** with the PowerShell extension instead of ISE.

Features of PowerShell ISE

Feature	Description
Tabbed Interface	Multiple script files can be opened simultaneously.
Syntax Highlighting	Color-codes commands, variables, and strings.
IntelliSense	Auto-completion for cmdlets, parameters, and variables.
Integrated Console	Run and debug scripts in the same window.
Command Add-On Pane	Quickly insert cmdlets and parameters.
Script Debugging Tools	Set breakpoints, step through code, and watch variables.
Multi-line Editing	Write and edit multiple lines easily.
Customizable Layout	Resize panes, change themes, and adjust fonts.

Launching PowerShell ISE

Method 1: Using Start Menu

- Type “**PowerShell ISE**” in the Windows search bar and open it.

Method 2: From PowerShell

- `powershell_ise`

Method 3: From Run Dialog

- Press **Win + R**, then type:
- `powershell_ise`

Components of PowerShell ISE Window

1. Menu Bar – Contains options like File, Edit, Debug, Tools, etc.
2. Toolbar – Quick access to Run, Stop, Save, Open, etc.
3. Script Pane – Where you write your PowerShell scripts.
4. Console Pane – Interactive execution of commands.
5. Output Pane – Displays output, errors, and messages.
6. Command Pane – Provides easy access to cmdlets and help.

Debugging in PowerShell ISE

1. Set Breakpoints – Click on the left margin or use:
   Set-PSBreakpoint -Script "myscript.ps1" -Line 5
2. Run Script in Debug Mode – Press F5.
3. Step Through Code – Use F10 (Step Over), F11 (Step Into).
4. Check Variables – Use the Watch Window or hover over variables.

Using ISE vs Visual Studio Code

Feature	PowerShell ISE	Visual Studio Code
Availability	Built into Windows PowerShell (≤5.1)	Works with PowerShell 7+
Cross-platform	No	Yes
Extensions and Themes	Limited	Extensive
Performance	Lightweight	Modern and powerful
Recommendation	For Windows PowerShell	For PowerShell 7+

Working with PowerShell Scripts

What is a PowerShell Script?

- A **PowerShell script** is a collection of PowerShell commands saved in a file with the **.ps1 extension**.
- Scripts allow **automation** of repetitive administrative tasks.
- They can include:
  - Cmdlets
  - Variables
  - Loops and conditionals
  - Functions and modules
  - Comments and parameters

Creating and Running Scripts

1. **Create a new file** using any text editor (e.g., Notepad, PowerShell ISE, or VS Code).
2. Save it as myscript.ps1.

   Example:

   ```
   # My First Script
   Write-Host "Welcome to PowerShell Scripting!"
   Get-Date
   ```

3. **Run the script** in PowerShell:

   ```
   .\myscript.ps1
   ```

4. If you get a security error, you need to modify the **Execution Policy**:

   ```
   Set-ExecutionPolicy RemoteSigned
   ```

Script Structure

Section	Description	Example
Comment Header	Explains purpose, author, and date	# Author: John Doe
Variables	Store reusable values	$name = "Alex"
Logic/Commands	Main script operations	Get-Process
Output	Display or log results	Write-Host or Out-File

Example Script

```
# Script: SystemCheck.ps1
# Author: Admin
# Purpose: Display basic system information

Write-Host "Checking system information..."
$computer = $env:COMPUTERNAME
$user = $env:USERNAME
$os = (Get-CimInstance Win32_OperatingSystem).Caption

Write-Host "Computer Name: $computer"
Write-Host "User Name: $user"
Write-Host "Operating System: $os"
```

Parameters and Arguments

Scripts can accept **parameters**, making them more flexible.

Example:

```
param(
    [string]$Name,
    [int]$Age
)
Write-Host "Hello, $Name! You are $Age years old."
```

- Run the script:

  .\myscript.ps1 -Name "Riya" -Age 25

Functions in Scripts

Functions make scripts modular and reusable.

```
function Get-DiskSpace {
    param([string]$Drive)
    Get-PSDrive -Name $Drive | Select-Object Name, Free, Used
}
Get-DiskSpace -Drive "C"
```

Error Handling

Use **Try/Catch/Finally** for controlled execution.

```
try {
    Get-Content "C:\file.txt"
}
catch {
    Write-Host "File not found!"
}
finally {
    Write-Host "Script completed."
}
```

Working with Cmdlets

What are Cmdlets?

- **Cmdlets** (pronounced “command-lets”) are **lightweight PowerShell commands** built into the environment.
- Each cmdlet follows the **Verb-Noun naming pattern**:
  - Example: Get-Process, Set-Item, New-User.

Discovering Cmdlets

- You can explore available cmdlets with:

  Get-Command
- Filter by verb or noun:

  Get-Command -Verb Get

  Get-Command -Noun Service

Getting Help

Use the built-in help system:

- Get-Help Get-Process
- Get-Help Get-Command -Full
- Update-Help

Examples of Common Cmdlets

Category	Cmdlets	Example
File Management	Get-ChildItem, Copy-Item, Remove-Item	Get-ChildItem C:\
Service Management	Get-Service, Start-Service, Stop-Service	Get-Service Spooler
Process Management	Get-Process, Stop-Process	Stop-Process -Name notepad
System Info	Get-Date, Get-EventLog, Get-ComputerInfo	Get-Date
User Interaction	Read-Host, Write-Host, Write-Output	Read-Host "Enter your name"

Pipeline Concept

- Cmdlets can **pipe output** from one to another:

  ```
  Get-Service | Where-Object {$_.Status -eq "Running"} | Sort-Object DisplayName
  ```

- Each cmdlet processes **.NET objects**, not just plain text.

Files and Folders

File System Cmdlets

- PowerShell provides comprehensive cmdlets for managing files and directories.

Action	Cmdlet	Example
List files/folders	Get-ChildItem	Get-ChildItem C:\Docs
Create a folder	New-Item	New-Item -ItemType Directory -Path "C:\NewFolder"
Copy files	Copy-Item	Copy-Item file1.txt D:\Backup\
Move files	Move-Item	Move-Item file1.txt D:\Archive\
Delete files	Remove-Item	Remove-Item C:\OldFiles* -Recurse
Rename files	Rename-Item	Rename-Item old.txt new.txt

Reading and Writing Files

- **Reading a file:**

  Get-Content "C:\notes.txt"

- **Writing to a file:**

  "PowerShell is powerful!" | Out-File "C:\output.txt"

- **Appending text:**

  Add-Content -Path "C:\output.txt" -Value "Appended line"

Checking File and Folder Existence

```
Test-Path "C:\temp\data.txt"
```

Conditional Example:

powershell

```
if (Test-Path "C:\temp\data.txt") {
    Write-Host "File exists."
} else {
    Write-Host "File not found."
}
```

- **File Metadata and Properties**

  Get-Item "C:\temp\report.docx" | Select-Object Name, Length, LastWriteTime

- **Searching for Files**

  Get-ChildItem -Path C:\ -Recurse -Filter *.log

Dates and Timers

- **Get the Current Date and Time**

  Get-Date

  - Example Output:

    Thursday, November 13, 2025 10:23:45 AM

- Formatting Dates
  - You can format dates using the -Format parameter.

    Get-Date -Format "dd-MM-yyyy"

    Get-Date -Format "dddd, MMMM dd yyyy"
  - Or use .ToString() method:

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

Working with Date Components

$date = Get-Date

$date.Year

$date.Month

$date.Day

$date.Hour

$date.Minute

Calculating Date Differences

```
$start = Get-Date "01/01/2025"
$end = Get-Date
$diff = $end - $start
$diff.Days
```

Output: Number of days between two dates.

Using Timers for Delays

- To **pause execution** for a specific number of seconds:

```
Start-Sleep -Seconds 5
Write-Host "5 seconds have passed."
```

Measuring Execution Time

- You can measure how long a command takes:

  Measure-Command { Get-Process }

Example Output:

Days : 0

Hours : 0

Minutes : 0

Seconds : 0

Milliseconds : 123

Ticks : 1234567

Automating Scheduled Tasks

PowerShell can schedule tasks via ScheduledTasks module.

- **Example:**

```
$Action = New-ScheduledTaskAction -Execute "PowerShell.exe" -Argument "C:\Scripts\Backup.ps1"
$Trigger = New-ScheduledTaskTrigger -Daily -At 3am
Register-ScheduledTask -TaskName "DailyBackup" -Action $Action -Trigger $Trigger
```

File I/O (File Input/Output)

- File I/O (Input/Output) in PowerShell refers to reading data from files and writing data to files.
- PowerShell provides multiple cmdlets to perform file operations efficiently.
- These cmdlets support text, CSV, XML, and JSON formats, as well as binary files

Basic File I/O Cmdlets

Action	**Cmdlet**	**Example**
Read file contents	Get-Content	Get-Content C:\data.txt
Write to a file	Out-File	`"Hello World"
Append to a file	Add-Content	Add-Content C:\data.txt "New line"
Overwrite file	Set-Content	Set-Content C:\data.txt "Replaced content"
Export to file	Export-Clixml, Export-Csv, ConvertTo-Json	`Get-Process

Reading Files

Example: Read a Text File

```
# Read file content line by line
$lines = Get-Content "C:\Users\Admin\Documents\data.txt"
foreach ($line in $lines) {
    Write-Host $line
}
```

- **Reading with Encoding**

```
Get-Content -Path "C:\notes.txt" -Encoding UTF8
```

Writing to Files

Example: Write Custom Output

```
$report = "System Backup Completed: " + (Get-Date)
$report | Out-File -FilePath "C:\Logs\BackupReport.txt"
```

Appending to Existing File

```
Add-Content -Path "C:\Logs\BackupReport.txt" -Value "Backup successful at $(Get-Date)"
```

- **Copying and Moving Files**

  Copy-Item "C:\data\file1.txt" "D:\backup\file1.txt"

  Move-Item "C:\data\old.txt" "C:\archive\old.txt”

- **Deleting Files**

  Remove-Item -Path "C:\Temp*.log" -Recurse -Force

Tip: Always use -WhatIf before running deletion commands to preview actions.

- Example:

  Remove-Item -Path "C:\Temp*.log" -Recurse -WhatIf

- Working with Binary Files: PowerShell can also read and write binary data using .NET methods.

```
# Read binary file
[byte[]]$bytes = Get-Content -Path "C:\image.png" -Encoding Byte
# Write binary file
Set-Content -Path "C:\copy.png" -Value $bytes -Encoding Byte
```

Handling CSV Files

- **Import CSV**

```
$users = Import-Csv "C:\Users.csv"
$users | ForEach-Object { $_.Name }
```

- **Export CSV**

```
Get-Process | Select-Object Name, CPU | Export-Csv "C:\processes.csv" -NoTypeInformation
```

Handling JSON Files

- **Convert Object to JSON**

  ```
  Get-Service | ConvertTo-Json | Out-File "C:\services.json"
  ```

- **Read JSON File**

  ```
  $json = Get-Content "C:\services.json" | ConvertFrom-Json
  $json[0].Name
  ```

- **Handling XML Files**

  ```
  [xml]$xmlData = Get-Content "C:\data.xml"
  $xmlData.Root.Node
  ```

- Export XML:

  ```
  $object | Export-Clixml "C:\object.xml"
  ```

Example: File I/O Script

```
# Script: FileLogger.ps1
$logFile = "C:\Logs\Activity.txt"

# Write log
"[$(Get-Date)] Script Started" | Out-File $logFile -Append

# Read file list
$files = Get-ChildItem "C:\Projects"
foreach ($file in $files) {
    Add-Content $logFile "File: $($file.Name)"
}

"[$(Get-Date)] Script Completed" | Out-File $logFile -Append
```

Advanced Cmdlets

- Advanced cmdlets extend the basic PowerShell capabilities with scripting enhancements, automation control, and data manipulation.
- Many of them are built into modules such as:
  - Microsoft.PowerShell.Management
  - Microsoft.PowerShell.Utility
  - Microsoft.PowerShell.Security
  - ScheduledTasks
  - NetSecurity

Commonly Used Advanced Cmdlets

Category	Cmdlet	Description
System Management	Get-WmiObject, Get-CimInstance	Access system and hardware data.
Event Logs	Get-EventLog, Get-WinEvent	Retrieve Windows logs.
Services	Start-Service, Stop-Service, Restart-Service	Manage Windows services.
Processes	Start-Process, Stop-Process, Wait-Process	Launch or stop applications.
Registry	Get-ItemProperty, Set-ItemProperty	Access and modify Windows registry.
Jobs & Background Tasks	Start-Job, Receive-Job, Remove-Job	Execute asynchronous background processes.
Remoting	Invoke-Command, Enter-PSSession	Execute commands on remote systems.
Security	Get-Acl, Set-Acl, Get-Credential	Manage file permissions and credentials.

Using WMI and CIM Cmdlets

- **Get System Information**

  Get-WmiObject Win32_OperatingSystem | Select-Object Caption, Version, BuildNumber

- **Using CIM (Modern Alternative)**

  Get-CimInstance Win32_LogicalDisk | Select-Object DeviceID, Size, FreeSpace

- **Working with Background Jobs**
- PowerShell can run tasks asynchronously in the background.

  Start-Job -ScriptBlock { Get-Process }

  Get-Job

  Receive-Job -Id 1

  Remove-Job -Id 1

Using Scheduled Tasks Cmdlets

- Create an automated task:

  ```
  $Action = New-ScheduledTaskAction -Execute "PowerShell.exe" -Argument "C:\Scripts\Backup.ps1"
  $Trigger = New-ScheduledTaskTrigger -Daily -At 2am
  Register-ScheduledTask -TaskName "NightlyBackup" -Action $Action -Trigger $Trigger
  ```

- **Networking Cmdlets**

  ```
  Test-Connection google.com
  Resolve-DnsName openai.com
  Get-NetIPAddress
  ```

Event Log Cmdlets

- **Retrieve System Events**

  ```
  Get-EventLog -LogName System -Newest 10
  ```

- **Filter by Entry Type**

  ```
  Get-EventLog -LogName Application | Where-Object {$_.EntryType -eq "Error"}
  ```

Security and Access Control

- **View File Permissions**

  ```
  Get-Acl "C:\Data"
  ```

- **Modify Permissions**

  ```
  $acl = Get-Acl "C:\Data"
  $rule = New-Object System.Security.AccessControl.FileSystemAccessRule("Domain\User","FullControl","Allow")
  $acl.AddAccessRule($rule)
  Set-Acl "C:\Data" $acl
  ```

Advanced Object Manipulation

- PowerShell allows filtering, sorting, and selecting data using cmdlets.

Task	**Cmdlet**	**Example**
Filter objects	Where-Object	`Get-Process
Sort results	Sort-Object	`Get-Service
Select properties	Select-Object	`Get-Process
Group data	Group-Object	`Get-Service
Measure data	Measure-Object	`Get-ChildItem

Example: System Health Check Script

```
# Script: HealthCheck.ps1
$Report = "C:\Reports\HealthReport.txt"
"System Health Report - $(Get-Date)" | Out-File $Report
# CPU Usage
"CPU Usage:" | Out-File $Report -Append
Get-Counter '\Processor(_Total)\% Processor Time' | Select-Object -ExpandProperty CounterSamples | Out-File $Report -Append
# Disk Space
"Disk Space:" | Out-File $Report -Append
Get-CimInstance Win32_LogicalDisk | Select-Object DeviceID, FreeSpace, Size | Out-File $Report -Append
# Running Services
"Running Services:" | Out-File $Report -Append
Get-Service | Where-Object {$_.Status -eq "Running"} | Select-Object DisplayName | Out-File $Report -Append
Write-Host "System health report generated successfully."
```

PowerShell Scripting

PowerShell Special Variables

Special variables, called automatic variables, store information related to the PowerShell environment and runtime state. Examples include:

- $? denotes the execution status of the last command, storing True if it succeeded, False otherwise.
- $_ represents the current object in the pipeline, especially used within script blocks.
- $$ holds the last token in the expression typed in the session.
- $PROFILE points to the script file that runs when PowerShell starts, allowing user customization.

Automatic variables are reserved and their values are set by the system, not user scripts.

Operators in PowerShell

PowerShell offers a rich set of operators grouped into categories:

- Arithmetic Operators: +, -, *, /, % for basic calculations.
- Assignment Operators: =, +=, -=, *=, /=, %= for assigning and updating variable values.
- Comparison Operators: -eq, -ne, -gt, -ge, -lt, -le for value comparison.
- Logical Operators: -and, -or, -not, !.
- Type Operators: -is, -as, -contains for type checks and casting. Operators are fundamental for manipulating data and controlling flow in scripts.

Operators in PowerShell

PowerShell offers a rich set of operators grouped into categories:

- Arithmetic Operators: +, -, *, /, % for basic calculations.
- Assignment Operators: =, +=, -=, *=, /=, %= for assigning and updating variable values.
- Comparison Operators: -eq, -ne, -gt, -ge, -lt, -le for value comparison.
- Logical Operators: -and, -or, -not, !.
- Type Operators: -is, -as, -contains for type checks and casting. Operators are fundamental for manipulating data and controlling flow in scripts.

Looping Constructs

PowerShell provides multiple loop structures for repeated execution:

- For Loop: Standard initialization, condition, and iteration for numerical/computational tasks.
  for ($i = 0; $i -lt 10; $i++) { *# code }*
- Foreach Loop: Iterates over collections.
  foreach ($item in $collection) { *# code }*
- While Loop: Loops while a condition is true.
  while ($condition) { *# code }*
- Do...While/Until Loop: Executes once before checking the condition, with variations for while or until.

Conditional Structures

Conditional logic enables decision-making in scripts:

- if statements execute code based on conditions.
- elseif and else provide alternate branches.

  if ($a -gt $b) { *# code } elseif ($a -eq $b) { # code } else { # code }*
- Complex conditions can be built using logical operators to combine multiple criteria.

Arrays

Arrays store ordered collections of items:

- Declaration: $array = @(1,2,3,4).
- Access by index: $array retrieves the first element.
  Arrays are dynamic and capable of holding mixed data types.

Hash Tables

- Hash tables are key-value data structures for efficient lookup:
- Declaration: $hash = @{ "name" = "Alice"; "age" = 30 }
- Access: $hash["name"] returns "Alice".
  Useful for structured data requiring fast key-based search.

Regular Expressions (Regex)

- PowerShell integrates regex for advanced text and pattern matching:
- Utilizes the -match operator, e.g., $str -match '^\d{3}-\d{2}-\d{4}$' for pattern matching.
- Supports standard .NET regex patterns.
  Regex is powerful for parsing, validation, and text extraction tasks.

Brackets

- Brackets serve syntactic and functional roles:
- () for grouping expressions and function calls.
- {} for script blocks.
- [] for arrays and indexers.
  Proper use of brackets is essential for code organization and execution.

Alias

- Command aliases offer shorthand for frequent commands, improving efficiency:
- Common examples: ls (alias for Get-ChildItem), pwd for Get-Location.
- Users can define custom aliases using Set-Alias.
  Aliases enhance productivity but should be used judiciously in formal scripts for clarity.

WMIC & PowerShell

- WMIC (Windows Management Instrumentation Command-line) is a legacy CLI tool for Windows management. PowerShell supersedes WMIC, offering:
- Comprehensive administrative automation via modules like Get-WmiObject and Get-CimInstance for querying system information.
- WMIC is depreciated in recent Windows versions; PowerShell is recommended for robust scripting and remote management tasks

PowerShell providers and drives

- PowerShell providers are a foundational concept that allows users to access and manage a variety of data stores in a uniform way, treating these data as if they were part of a file system. Providers expose data as drives that support navigation, retrieval, and manipulation using familiar cmdlets.
- PowerShell Providers
  - Providers are .NET programs that give access to specialized data stores, making them appear as drives within PowerShell (e.g., C:, HKLM:, Env:, WSMan:).
  - Each provider presents its data in a hierarchical format, similar to a file system, which allows for easy access using standard PowerShell commands and paths.

Common built-in PowerShell providers include

- FileSystem: Accesses files and folders on disks (C:\, D:\, etc.).
- Registry: Navigates the Windows registry (HKLM:, HKCU:).
- Environment: Manages environment variables as items (Env:).
- Alias: Manages PowerShell command aliases (Alias:).
- Function: Works with functions defined in the session (Function:).
- Certificate: Views digital certificates in certificate stores (Cert:).
- Variable: Exposes PowerShell variables (Variable:).
- WSMan: Manages WS-Management configuration (WSMan:).

- You can display all available providers using:

  Get-PSProvider

- This lists every provider loaded in the current session.

PowerShell Drives

- Drives are exposed by providers, allowing users to access and manipulate hierarchical data.
- These drives are not physical drives but logical representations of the data stores made available by providers.
- For example, you can move through the Windows registry or environment variables using navigation (cd, ls) and management cmdlets as if you were navigating the file system.
- To see all currently mounted drives:

  Get-PSDrive

This lists the drives, their providers, and roots for reference

Consistent Cmdlet Usage

- Providers expose data so that you can use consistent cmdlets (like Get-Item, Set-Item, Remove-Item, Get-ChildItem, Get-Content) no matter the data source.
- This abstraction allows scripts to be more portable and reusable, since the underlying data store (file system, registry, etc.) can be accessed the same way

$Args Variable

- $Args is an automatic array variable in PowerShell that stores all the unnamed or positional arguments supplied to a function or script when it is invoked.
- It is useful for handling varying numbers of arguments without explicitly naming them in a parameter block.
- Example usage:

  function Demo { **foreach** ($item in $args) { Write-Host $item } }

  Demo apple orange banana *# Output: apple\norange\nbanana*

- Each argument can be accessed by its index, starting from zero: $args, $args[10], etc. Missing arguments will result in $null or empty values in the corresponding index.

param Statement

- The param statement is a formal way to define named parameters for scripts and functions in PowerShell, enabling type checking, validation, and improved readability.
- It supports required, optional, and default values, as well as validation attributes:

```
param(
    [string]$Name,
    [int]$Age = 18
)
```

- Write-Host "Name: $Name, Age: $Age"
- Parameters declared with param can be used with their names during script/function invocation:

```
.\script.ps1 -Name "Alice" -Age 23
```

- Named parameters allow flexible order and enhanced error checking, and are preferred over $Args in most cases.

Passing Data by Value and by Reference

- Pass by Value: By default, PowerShell passes arguments by value, meaning a copy of the data is created and sent to the function or script. Changes inside the function do not affect the original variable outside.

  ```
  function ChangeValue($num) { $num = 99 }
  $x = 1; ChangeValue $x; Write-Host $x   # Output: 1
  ```

- Pass by Reference: PowerShell can pass [ref] parameters so that changes inside a function affect the original variable:

  ```
  function ChangeRef([ref]$num) { $num.Value = 99 }
  $x = 1; ChangeRef ([ref]$x); Write-Host $x   # Output: 99
  ```

- [ref] parameters are less commonly used but useful for functions that need to directly modify values in their caller's scope.

PowerShell Advance Functions

- PowerShell advanced functions are specialized script functions that behave like cmdlets, providing robust parameter handling, cmdlet binding, and enhanced pipeline integration. These functions are vital for creating complex, reusable tools in administrative scripting.

Definition and Characteristics

- An advanced function is a PowerShell function that includes the [CmdletBinding()] attribute or uses advanced parameter attributes, enabling access to common parameters (e.g., -Verbose, -ErrorAction, -Debug).
- Advanced functions feature enhanced parameter validation, pipeline input handling, and integrated error management, closely mimicking compiled cmdlets.

Syntax and Structure

- An advanced function is typically structured with three blocks for logical segmentation:
- Begin: Initialization, runs once before processing begins.
- Process: Executes for each input object passed via the pipeline.
- End: Finalization, runs once after all input is processed.
- This example accepts pipeline input, validates parameters, and uses common cmdlet parameters.
- Example:

```
function Get-Info {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$true, ValueFromPipeline=$true)]
        [string]$Name
    )
    begin { Write-Verbose "Starting function..." }
    process { Write-Host "Processing $Name" }
    end { Write-Verbose "Function completed." }
}
```

Benefits and Features

- Common Parameters: Automatically supports flags like -Verbose, -Debug, and -ErrorAction, improving usability and diagnostics.
- Parameter Validation: Offers attributes such as [ValidateSet()], [ValidatePattern()], [ValidateRange()] to enforce input rules.
- Pipeline Input: Handles objects passed through the PowerShell pipeline, allowing efficient data processing across functions and scripts.

Additional Capabilities

- Parameter Sets: Enables the definition of mutually exclusive parameter sets using [Parameter(ParameterSetName="")], optimizing function versatility and input management.
- Error Management: Allows granular control of warnings and errors using ErrorAction, WarningAction, etc., directly in custom scripts and modules.
- Help Documentation: Advanced functions integrate with PowerShell's Get-Help system, supporting custom help content and automatic help for parameters.

Advanced functions are essential for developing enterprise-level PowerShell scripts, combining the flexibility of script-based tools with the consistency and power of native cmdlets.

Using PowerShell remoting capabilities: emoting concepts Invoking remote commands

- PowerShell remoting enables administrators and users to run commands and manage multiple remote computers seamlessly from a single PowerShell session.

PowerShell Remoting Concepts

- PowerShell remoting allows executing commands and scripts on remote machines across a network using the Windows Remote Management (WinRM) protocol.
- It supports interactive one-to-one sessions via Enter-PSSession for real-time command execution, and batch or parallel one-to-many commands through Invoke-Command.
- Remoting transmits commands, processes them remotely, and serializes the output as XML to send back to the local session, which deserializes it for further local manipulation.

Invoking Remote Commands

- Invoke-Command is the primary cmdlet for remoting non-interactive commands on one or more remote computers.
- It accepts parameters such as:
  - -ComputerName for target machines
  - -ScriptBlock to specify the command or code block to run remotely
  - -Credential for authentication
- Example:

  Invoke-Command -ComputerName Server01,Server02 -ScriptBlock { Get-Process }
- You can use sessions (New-PSSession) to maintain persistent remote connections for multiple commands, improving efficiency

Processing Output

- Output from remote commands is serialized on the remote computer, transmitted as XML, and deserialized locally into PowerShell objects.
- This enables remote output to be processed just like local data, supporting pipeline use and complex manipulation.
- Errors and other streams (warning, verbose, debug) are also captured and can be handled separately if needed.
- Example handling output:

```
$results = Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Service }
$results | Where-Object {$_.Status -eq "Running"}
```

Objects in Windows PowerShell: Error handling concepts

- PowerShell provides robust error handling mechanisms, essential for creating reliable and maintainable scripts.

Objects in Windows PowerShell

- PowerShell is an object-oriented shell and scripting language. Commands output .NET objects instead of plain text (unlike traditional shells), enhancing powerful data manipulation.
- Each object has properties (data) and methods (actions) which can be accessed and invoked respectively.
- Objects pass through the pipeline, allowing complex operations without converting output to text/string, preserving type integrity for subsequent commands.

- Example:

```
$process = Get-Process -Name "notepad"
$process.Id    # Access property
$process.Kill() # Invoke method
```

- This object-based paradigm allows leveraging the richness of .NET framework and interacting with system components programmatically

PowerShell Error Handling Concepts

- PowerShell distinguishes between terminating errors (exceptions that stop execution unless handled) and non-terminating errors (warnings that allow continuation).
- The primary structured error handling uses Try, Catch, and Finally blocks:
  - Try encloses the code that may throw errors.
  - Catch handles terminating exceptions, optionally filtered by exception type.
  - Finally executes cleanup code irrespective of error occurrence.

```
try {
    # Risky code here
} catch [System.Exception] {
    # Handle error here
} finally {
    # Cleanup code here
}
```

- PowerShell cmdlets support the -ErrorAction parameter to control how non-terminating errors are treated (e.g., Continue, Stop, SilentlyContinue).
- The global variable $ErrorActionPreference sets default error handling behavior.
- Error details are collected in $Error automatic variable as a collection of error objects, giving access to properties like Exception, CategoryInfo, and InvocationInfo.
- Custom errors can be generated using throw.
- Effective error handling ensures robust script behavior, clean resource management, and meaningful feedback for troubleshooting.

This object-centric design combined with comprehensive error management makes PowerShell an advanced scripting platform suitable for enterprise-level automation and administration.

Terminating and non- terminating errors

- Terminating and non-terminating errors are the two fundamental types of errors in PowerShell, each affecting script execution differently.

Terminating Errors

- These errors cause the immediate halt of script or command execution unless specifically handled.
- They are also called exceptions.
- Trigger the catch block when using try/catch error handling.
- Examples include syntax errors, invalid operations, or explicitly using throw.
- Once a terminating error occurs, the current pipeline or script stops executing further commands.
- You can convert non-terminating errors into terminating errors by using the -ErrorAction Stop parameter.
- Terminating errors are suitable for critical failures requiring immediate attention.

Non-Terminating Errors

- These errors are reported but do not stop script execution; the script continues processing subsequent commands or pipeline elements.
- Typical for recoverable or less severe issues like file not found during a query, or issues applying a command to one item in a batch.
- They are written to the error stream and logged in the $Error automatic variable but do not invoke the catch block unless escalated.
- You can handle non-terminating errors by changing the $ErrorActionPreference or using -ErrorAction parameter.
- Useful when you want a script to continue running despite some errors, common in batch operations.

Handling errors using $?

- The $? variable in PowerShell is an automatic Boolean variable that indicates the success or failure status of the last executed command or statement.

Using $? for Error Handling

- After running a command or script statement, $? is set to:
  - True if the previous operation succeeded without errors.
  - False if the previous operation generated any errors, including both terminating and non-terminating errors.
- It allows quick, simple checks to determine if the last command was successful without detailed error information.

Characteristics

- $? is reset after every command, so it must be checked immediately after the command of interest; otherwise, its value may be overwritten.
- It doesn’t provide the nature or details of the error, just a Boolean pass/fail indicator.
- When PowerShell runs native executables, $? is set to False if the executable returns a non-zero exit code or writes to the standard error stream, though this might depend on environment specifics.
- $? is useful for conditional scripting or simple success checks.
- Example Usage

```
Get-Item "C:\file.txt"
if ($?) {
    Write-Host "File retrieved successfully."
} else {
    Write-Host "Failed to retrieve file."
}
```

Limitations

- Cannot replace structured error handling like try/catch.
- Does not provide error context or exception details.
- Not suitable for complex error management scenarios but helpful for quick success validation.
- Thus, $? serves as a quick success/failure flag immediately after commands and complements, but does not replace, more detailed PowerShell error handling techniques.

$Error and $lastExitCode variables

- In PowerShell, the automatic variables $Error and $LASTEXITCODE are crucial for error handling and understanding command execution outcomes.

$Error Variable

- $Error is an automatic array that holds all the error records generated in the current PowerShell session, with the most recent error at index [0].
- It contains rich error objects, including exception details, script position, and error category.
- You can inspect errors with $Error[0] to see the last error, or loop through $Error to analyze past errors.
- $Error persists throughout the session and accumulates errors unless cleared using $Error.Clear().
- It is commonly used for logging, debugging, and conditional error handling in scripts.

- Example:

```
Get-Item "C:\NonExistentFile"
if ($Error.Count -gt 0) {
    Write-Host "An error occurred: $($Error[0].Exception.Message)"
}
```

- Use with caution when clearing the variable, as it is global to the session and may contain relevant history.

$LASTEXITCODE Variable

- $LASTEXITCODE contains the exit code of the last native executable (non-PowerShell command) run in the session.
- PowerShell cmdlets do not update this variable; it only tracks external program exit codes.
- The variable is useful to check success or failure when running external tools or scripts.
- Conventionally, an exit code of 0 indicates success; any non-zero value signals an error or specific status.
- Example:

```
ping google.com
if ($LASTEXITCODE -eq 0) {
    Write-Host "Ping succeeded."
} else {
    Write-Host "Ping failed."
}
```

Error Record object anatomy

- The PowerShell ErrorRecord object is a rich container used to represent both terminating and non-terminating errors generated by cmdlets or scripts. It provides detailed contextual information to help diagnose and handle errors effectively.

Anatomy of the ErrorRecord Object

- **Exception:** (Property: Exception)
  Contains the actual .NET exception object thrown. This includes the error message and stack trace, providing the foundational error information.
- **CategoryInfo:** (Property: CategoryInfo)
  An object providing a categorized description of the error via several properties such as:
  - Category: One of PowerShell's predefined error categories (e.g., PermissionDenied, SyntaxError).
  - Activity: Text description of the command or operation (typically the cmdlet name) during which the error occurred.
  - Reason: A short error reason string.
  - TargetName: The name of the object the cmdlet was processing when the error was generated.
  - TargetType: The .NET type of the target object.

- **InvocationInfo**: (Property: InvocationInfo)
  Includes details about the context of the script/cmdlet invocation:
  - Script name, line number, position in line.
  - The invocation command.
  - Call stack information for troubleshooting.
- **TargetObject**: (Property: TargetObject)
  The input to the cmdlet when the error occurred; for example, a file or process object that caused the failure.
- **ErrorDetails**: (Property: ErrorDetails)
  Provides localized or additional descriptive information about the error.
- **ScriptStackTrace**: (Property: ScriptStackTrace)
  Provides a script-specific trace useful when diagnosing issues in script files.

Example Usage to Inspect an ErrorRecord Object

```
try {
    Get-Item "C:\NonExistentFile.txt"
} catch {
    $errorRecord = $_
    $errorRecord.Exception.Message      # Detailed .NET exception message
    $errorRecord.CategoryInfo.Category  # Type of error category
    $errorRecord.CategoryInfo.Activity  # Cmdlet or activity that caused error
    $errorRecord.InvocationInfo.ScriptName  # Script or command
    $errorRecord.TargetObject           # The target object involved
}
```

Working with textual files : Saving information into textual and csv files

- Working with textual and CSV files in PowerShell involves several cmdlets designed to save output and data efficiently into files for logging, reporting, or further processing.

Saving Information into Text Files

- Out-File:
  Used to send command output to a text file, converting output into formatted strings.
- powershell
- Get-Process | Out-File -FilePath "C:\temp\processes.txt"
  - Use -Append to add to existing file instead of overwriting.
  - Use -NoClobber to prevent overwriting a file.

Set-Content / Add-Content:

- Set-Content writes content to a file, overwriting it.
- Add-Content appends content to an existing file or creates a new file if it doesn’t exist.

```
"Log started at $(Get-Date)" | Set-Content -Path "C:\temp\log.txt"
"New entry" | Add-Content -Path "C:\temp\log.txt"
```

StreamWriter (Advanced):

For large or complex output, using .NET System.IO.StreamWriter allows manual control over file writes.

Example:

```
$stream = [System.IO.StreamWriter]::new("C:\temp\data.txt")
try {
    $data | ForEach-Object { $stream.WriteLine($_) }
} finally {
    $stream.Close()
}
```

Saving Information into CSV Files

- Export-Csv:
  Designed for structured data, especially collections of objects. Converts properties to columns in a CSV.

```
$data = @(
    [pscustomobject]@{ Name = "Anne"; Age = 40; Dept = "HR" },
    [pscustomobject]@{ Name = "Leo"; Age = 35; Dept = "IT" },
    [pscustomobject]@{ Name = "Sarah"; Age = 28; Dept = "Finance" }
)
$data | Export-Csv -Path "C:\temp\data.csv" -NoTypeInformation
```

- Parameters:
  - -NoTypeInformation excludes the type metadata line at the top.
  - Supports custom delimiters if needed via -Delimiter.

Reading Files Back

- Use Get-Content to read lines from text or CSV files:

```
Get-Content -Path "C:\temp\data.csv"
```

Reading information from textual and csv files

Reading information from text and CSV files in PowerShell involves specialized cmdlets designed for efficient and versatile data access.

Reading Text Files

- The primary cmdlet to read text files is Get-Content. It reads content line-by-line by default.
- Example:

```
$lines = Get-Content -Path "C:\temp\file.txt"
foreach ($line in $lines) {
  Write-Host $line
}
```

- You can read specific portions:
  - First n lines: Get-Content -Path "file.txt" -TotalCount 10
  - Last n lines: Get-Content -Path "file.txt" -Tail 5
- For large files, use the -ReadCount parameter to process chunks instead of loading entire content at once for better memory management.
- Advanced reading with .NET’s System.IO.StreamReader allows line-by-line streaming and is useful for huge files.

Reading CSV Files

- PowerShell provides the Import-Csv cmdlet to read CSV files and convert each row into a custom object, with headers as properties.
- Example:

```
$records = Import-Csv -Path "C:\temp\data.csv"
foreach ($record in $records) {
  Write-Host "Name: $($record.Name), Age: $($record.Age)"
}
```

- This cmdlet automatically handles parsing, making it convenient for structured data processing.
- Supports custom delimiters through -Delimiter parameter for non-comma-separated values.

Example implementation of error handling code

```
try {
    # Code that might throw an exception
    Write-Host "Attempting to get content of a file..."

    # Attempt to get content of a file that might not exist
    $content = Get-Content -Path "C:\temp\nonexistentfile.txt" -ErrorAction Stop
    # Process the content
    Write-Host "File content loaded successfully."
    Write-Host $content
}
```

```
catch [System.Management.Automation.ItemNotFoundException] {
    # Handle file not found error specifically
    Write-Warning "File was not found. Please check the path and try again."
}
catch {
    # General catch for other errors
    Write-Error "An unexpected error occurred: $($_.Exception.Message)"
}
finally {
    # Cleanup code that runs regardless of error occurrence
    Write-Host "Execution completed. Cleaning up resources if any..."
}
```

Explanation

- try block: Contains the risky operation (reading a file) with -ErrorAction Stop to ensure the cmdlet treats non-terminating errors as terminating.
- catch blocks:
  - The first catch is specific to the file-not-found exception, providing targeted handling and user-friendly messaging.
  - The second, general catch handles any other exceptions, outputting the exception message for diagnostics.
- finally block: Runs always after the try/catch, useful to release resources or perform any final logging or cleanup.
- This pattern can be adapted for other operations like network calls, registry access, or database queries, providing clear paths for error detection and graceful recovery in PowerShell scripts.
- This comprehensive structure is the recommended approach for writing maintainable and reliable PowerShell scripts, ensuring that errors do not cause unhandled crashes and give meaningful feedback.