PowerShell Scripting Introduction for Windows 10

What is PowerShell?

PowerShell is Microsoft's powerful command-line shell and scripting language built on the .NET Framework. It's designed specifically for system administration, automation, and configuration management in Windows environments.

Getting Started

Opening PowerShell

- **Regular PowerShell**: Press `Win + X`, select "Windows PowerShell"
- **Administrator PowerShell**: Press `Win + X`, select "Windows PowerShell (Admin)"
- **PowerShell ISE**: Search for "PowerShell ISE" in Start Menu (Integrated Scripting Environment)

Basic Concepts

Cmdlets: PowerShell commands follow a Verb-Noun pattern (e.g., `Get-Process`, `Set-Location`)

Objects: Unlike traditional shells that work with text, PowerShell works with .NET objects

Pipeline: Use `|` to pass output from one command to another

Essential Commands

Information Gathering

```powershell
Get-Help <cmdlet>           # Get help for any command
Get-Command                 # List available commands
Get-Process                 # Show running processes
Get-Service                 # Show system services
Get-Location                # Show current directory
Get-ChildItem               # List files and folders (like 'dir')
```

Navigation and File Operations

```powershell
Set-Location C:\Users      # Change directory (like 'cd')
New-Item -Type File test.txt     # Create new file
New-Item -Type Directory MyFolder    # Create new folder
Copy-Item source.txt dest.txt    # Copy file
Remove-Item file.txt       # Delete file
```

System Information

```powershell
Get-ComputerInfo          # Comprehensive system info
Get-WmiObject Win32_ComputerSystem    # Hardware info
Get-EventLog System -Newest 10    # Recent system events
```

Variables and Data Types

Creating Variables

```powershell
$name = "John Doe"        # String variable
$age = 30                 # Integer variable
$isActive = $true         # Boolean variable
$services = Get-Service   # Store command output
```

Working with Variables

```powershell
Write-Host "Hello, $name"    # Display variable
$name.Length                 # String properties
$services.Count              # Array properties
```

Control Structures

If Statements

powershell

```powershell
$diskSpace = Get-WmiObject Win32_LogicalDisk -Filter "DeviceID='C:'"
if ($diskSpace.FreeSpace -lt 1GB) {
    Write-Host "Low disk space warning!"
} else {
    Write-Host "Disk space is adequate"
}
```

Loops

powershell

```powershell
# For Loop
for ($i = 1; $i -le 5; $i++) {
    Write-Host "Count: $i"
}

# ForEach Loop
$processes = Get-Process
foreach ($process in $processes) {
    Write-Host $process.Name
}

# While Loop
$counter = 1
while ($counter -le 3) {
    Write-Host "Iteration: $counter"
    $counter++
}
```

Functions

Creating Functions

powershell

```powershell
function Get-SystemUptime {
    $bootTime = (Get-CimInstance Win32_OperatingSystem).LastBootUpTime
    $uptime = (Get-Date) - $bootTime
    return "System uptime: $($uptime.Days) days, $($uptime.Hours) hours"
}

# Call the function
Get-SystemUptime
```

Functions with Parameters

powershell

```powershell
function Get-ServiceStatus {
    param(
        [string]$ServiceName
    )
    $service = Get-Service -Name $ServiceName -ErrorAction SilentlyContinue
    if ($service) {
        return "Service '$ServiceName' is $($service.Status)"
    } else {
        return "Service '$ServiceName' not found"
    }
}

# Usage
Get-ServiceStatus -ServiceName "Spooler"
```

Working with Files and Folders

Reading Files

powershell

```powershell
Get-Content "C:\path\to\file.txt"    # Read entire file
Get-Content "file.txt" -TotalCount 5    # Read first 5 lines
```

Writing Files

```powershell
"Hello World" | Out-File "output.txt"    # Write to file
"New line" | Add-Content "output.txt"    # Append to file
```

CSV Operations

```powershell
# Import CSV
$data = Import-Csv "data.csv"

# Export to CSV
Get-Process | Export-Csv "processes.csv" -NoTypeInformation
```

Error Handling

Try-Catch Blocks

```powershell
try {
    $file = Get-Content "nonexistent.txt"
    Write-Host "File read successfully"
} catch {
    Write-Host "Error: $($_.Exception.Message)"
} finally {
    Write-Host "Cleanup operations here"
}
```

Script Execution Policy

Before running scripts, you may need to set the execution policy:

```powershell
# Check current policy
Get-ExecutionPolicy

# Set policy to allow local scripts
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Best Practices

Script Structure

1. **Comments**: Use `#` for single-line comments
2. **Help**: Include help documentation using comment-based help
3. **Parameters**: Use `param()` block for script parameters
4. **Error Handling**: Always include error handling for robust scripts

Example Script Template

```powershell
<#
.SYNOPSIS
Brief description of the script

.DESCRIPTION
Detailed description of what the script does

.PARAMETER Name
Description of the parameter

.EXAMPLE
Example of how to use the script
#>

param(
    [Parameter(Mandatory=$true)]
    [string]$Name
)

try {
    # Main script logic here
    Write-Host "Processing $Name..."

} catch {
    Write-Error "An error occurred: $($_.Exception.Message)"
} finally {
    # Cleanup code
    Write-Host "Script completed"
}
```

Useful Tips

1. **Use Tab Completion**: Press Tab to auto-complete commands and parameters
2. **Pipeline Explorer**: Use `Get-Member` to explore object properties: `Get-Process | Get-Member`
3. **Measure Performance**: Use `Measure-Command { Your-Command }` to time operations
4. **Format Output**: Use `Format-Table`, `Format-List`, or `Format-Wide` for better output formatting
5. **Save Commands**: Use `Get-History` to see command history

Common Use Cases

- **System Administration**: Managing services, processes, and system configuration
- **File Management**: Bulk file operations, log analysis, and data processing
- **Network Operations**: Testing connectivity, managing network settings
- **Automation**: Scheduled tasks, deployment scripts, and maintenance routines
- **Reporting**: Generating system reports and gathering performance metrics

Next Steps

1. Practice basic commands in PowerShell console
2. Create simple scripts for daily tasks
3. Explore PowerShell modules for extended functionality
4. Learn about PowerShell remoting for managing multiple computers
5. Study advanced topics like PowerShell classes and DSC (Desired State Configuration)

PowerShell's strength lies in its object-oriented nature and deep integration with Windows. Start with simple tasks and gradually build complexity as you become more comfortable with the syntax and concepts.