PowerShell Documents

Introducing to Cmdlets

Cmdlets are lightweight, built-in commands in the Windows PowerShell environment. They follow a **"Verb-Noun" naming convention** (e.g., Get-Process, Start-Service) to clearly indicate their function. Unlike traditional executables, cmdlets are tightly integrated into PowerShell's environment and are **not standalone files**.

Key Characteristics:

- **Verb-Noun Format:** Names reflect the action and target (e.g., New-Item).
- **Pipeline-Oriented:** Cmdlets can be chained to perform complex tasks.
- **Object-Oriented:** Input and output are .NET objects, not just plain text.
- **Built-in and Custom:** PowerShell includes many pre-defined cmdlets, with the option to create custom ones.
- **Integrated:** Cmdlets are part of PowerShell's core functionality, not separate programs.

Cmdlet Workflow:

1. **Input:** Via parameters or pipeline data.
2. **Processing:** Action is performed based on the cmdlet's logic.
3. **Output:** Returns objects, which can be passed down the pipeline.

Example:

Get-Process | Sort-Object CPU
Retrieves running processes and sorts them by CPU usage.

Cmdlets provide a powerful, consistent, and object-based way to manage tasks in PowerShell.

Key Cmdlets

PowerShell cmdlets are native, task-specific commands identified by their **verb-noun** structure (e.g., Get-Process, New-Item). These are essential tools for system management, automation, and scripting.

Core Cmdlets and Their Uses:

Discovery & Documentation

- Get-Command: Lists all available commands (cmdlets, functions, aliases).

- Get-Help: Provides detailed help and examples for any cmdlet.

Process & Service Management

- Get-Process: Displays current system processes.
- Start-Service / Stop-Service: Starts or stops Windows services.

File System & Registry

- New-Item: Creates files, folders, or registry keys.
- Get-Item: Retrieves information about items.
- Set-Item: Modifies item properties.
- Remove-Item: Deletes items.
- Test-Path: Checks if a path exists.
- Get-ChildItem: Lists contents of a directory.

Modules & Execution Policy

- Get-Module: Lists loaded modules.
- Import-Module / Remove-Module: Adds or removes modules from memory.
- Get-ExecutionPolicy / Set-ExecutionPolicy: Views or changes script execution policies.

Networking & Web

- Test-Connection: Tests network connectivity (like ping).
- Invoke-WebRequest: Sends web requests.
- Invoke-RestMethod: Sends REST API calls.

Scripting & Input/Output

- Invoke-Expression: Executes code from a string.
- Write-Output: Outputs data to the console.
- Out-File: Writes output to a file.
- Read-Host: Gets user input.

Drives & Objects

- New-PSDrive / Get-PSDrive / Remove-PSDrive: Manages PowerShell drives.
- New-Object: Creates new .NET objects.
- Get-WmiObject: Accesses system information from WMI.

Example:

- Get-ChildItem C:\Program Files: Lists files and directories under *Program Files*.

These cmdlets provide the foundational tools to effectively automate and manage Windows environments using PowerShell.

The PowerShell Pipeline

PipThe **PowerShell pipeline** uses the | operator to connect a sequence of commands, passing the **output of one command as input to the next**. Unlike traditional shells that pass plain text, PowerShell pipelines transfer **.NET objects**, allowing for more powerful and flexible data manipulation.

Key Features:

- **Object-Based:** Data passed between commands is structured as objects, enabling property-based filtering and manipulation.
- **Efficient Processing:** Streams data in real time, improving performance and reducing memory usage.
- **Command Chaining:** Allows complex operations to be performed in a single command line.
- **Conditional Operators:** Supports && (execute if previous succeeds) and || (execute if previous fails).

Basic Syntax:

```
Command1 | Command2 | Command3
```

Example:

```
Get-Process | Where-Object {$_.CPU -gt 1} | Sort-Object CPU -Descending | Select-Object Name, CPU
```

What it does:

1. Get-Process: Gets all running processes.
2. Where-Object: Filters to include only processes using more than 1 second of CPU time.
3. Sort-Object: Sorts the results by CPU usage (highest first).
4. Select-Object: Displays only the Name and CPU properties.

The PowerShell pipeline is a core feature that enables efficient, readable, and powerful command chaining for automation and system management.

Pipeline Filtering & Operators in PowerShell

PowerShell's pipeline enables **sequential execution** of commands, where output from one command becomes input to the next. Efficient data handling in the pipeline is achieved using **filtering, comparison, logical, and control operators**.

Filtering with Where-Object:

- Filters objects based on conditions inside a script block.
- Example: Get-Process | Where-Object {$_.CPU -gt 1}

Filters processes to show only those using more than 1 second of CPU time.

Comparison Operators:

Used within filters to compare values:

- -eq: Equal to
- -ne: Not equal to
- -gt: Greater than
- -ge: Greater than or equal to
- -lt: Less than
- -le: Less than or equal to
- -like, -notlike: Wildcard matching
- -match, -notmatch: Regex matching

Logical Operators:

Combine multiple conditions:

- -and: Both must be true
- -or: At least one must be true
- -not or !: Negates the condition

Example:

Get-Process | Where-Object {$_.CPU -gt 1 -and $_.WorkingSet -gt 10MB}

Pipeline Chain Operators (PowerShell 7+):

Control command execution based on success or failure:

- &&: Run next command **only if previous succeeds**
- ||: Run next command **only if previous fails**

Example:

Get-Process non_existent_process && Write-Host "Found" || Write-Host "Not found"

Background Execution (& Operator):

Runs a command block as a background job.

Example:

```
Get-Process | & { Start-Sleep -Seconds 5; Write-Host "Done after 5 seconds" }
```

Best Practice – Order of Operations:

Filter early in the pipeline to reduce data passed to subsequent commands.

Example:

```
Get-ChildItem | Where-Object {$_.LastWriteTime -gt (Get-Date).AddDays(-7)} | Select-Object Name, Length
```

Filters recent files before selecting relevant properties.

This approach ensures **performance, clarity, and control** in PowerShell scripting

Input, Output & Formatting in PowerShell

PowerShell provides several cmdlets and techniques to manage **user input, output display**, and **data formatting**, enhancing its capabilities for scripting and automation.

Input Handling:

- **Read-Host**: Prompts user for input and stores it as a string.

  ```
  $name = Read-Host "Please enter your name"
  Write-Host "Hello, $name!"
  ```

- **Get-Content**: Reads content from a file.

  ```
  $content = Get-Content -Path "C:\example.log"
  ```

Output Display:

- **Write-Host**: Prints output directly to the console.

  ```
  Write-Host "This is a message."
  ```

- **Out-File**: Sends output to a text file.

  ```
  Get-Process | Out-File -FilePath "processes.txt"
  ```

Formatting Output:

- **Format-Table**: Tabular display of object properties.

  ```
  Get-Process | Format-Table -Property Name, CPU, StartTime
  ```

- **Format-List**: Displays each property on its own line.

  ```
  Get-Service | Format-List -Property Name, Status, DisplayName
  ```

- **Format-Wide**: Shows a single property in wide-column layout.

  ```
  Get-ChildItem | Format-Wide -Column 3
  ```

String Formatting:

- **-f Operator**: Allows positional formatting of strings.

  ```
  $name = "John"
  $age = 30
  "My name is {0} and I am {1} years old" -f $name, $age
  ```

Pipelines for Data Processing:

- PowerShell pipelines (|) pass objects between commands for processing and formatting.

  ```
  Get-Process | Sort-Object CPU -Descending | Select-Object -First 5 | Format-Table
  Name, CPU
  ```

These features make PowerShell highly effective for **interactive use, script automation, and data presentation**.

Cmdlet	Description
Format-Table	Outputs properties in table format
Format-List	Lists properties vertically
Format-Wide	Displays only one property per line
-f Operator	String formatting ("Name: {0}" -f $name)

PowerShell Scripting Overview: Objects, Arrays, Variables

PowerShell is a powerful scripting language and command-line shell from Microsoft, designed for **automation and configuration management**. It is built on the **.NET framework** and uses **objects** instead of plain text for data processing.

Objects:

- PowerShell commands return structured **objects** with **properties** and **methods**.
- Example:
  Get-Process returns process objects with properties like Name, ID, and CPU.

Arrays:

- Used to store **collections** of items (of same or different types).
- Defined using commas:

```
$myArray = 10, 20, 30, "apple", "banana"
```

- Access elements using **indexing**:

```
$myArray[0]    # First element (10)
$myArray[-1]   # Last element ("banana")
```

- Arrays are **dynamic**:

```
$myArray += "orange"
```

Variables:

- Begin with $ and are **case-insensitive**:

```
$name = "John Doe"
$age = 30
$isValid = $true
```

- PowerShell is **dynamically typed**, but you can enforce types:

```
[int]$number = "123"
```

- **Variable scopes** include:
  - **Local** (default)
  - **Script** (for use within scripts)

- Global (accessible everywhere)

PowerShell's use of objects, dynamic arrays, and flexible variables makes it a **powerful and efficient** tool for scripting, automation, and system management.

Input, Output & Formatting in PowerShell

PowerShell provides several cmdlets and techniques to manage **user input, output display**, and **data formatting**, enhancing its capabilities for scripting and automation.

Input Handling:

- **Read-Host**: Prompts user for input and stores it as a string.

  ```
  $name = Read-Host "Please enter your name"
  Write-Host "Hello, $name!"
  ```

- **Get-Content**: Reads content from a file.

  ```
  $content = Get-Content -Path "C:\example.log"
  ```

Output Display:

- **Write-Host**: Prints output directly to the console.

  ```
  Write-Host "This is a message."
  ```

- **Out-File**: Sends output to a text file.

  ```
  Get-Process | Out-File -FilePath "processes.txt"
  ```

Formatting Output:

- **Format-Table**: Tabular display of object properties.

  ```
  Get-Process | Format-Table -Property Name, CPU, StartTime
  ```

- **Format-List**: Displays each property on its own line.

  ```
  Get-Service | Format-List -Property Name, Status, DisplayName
  ```

- **Format-Wide**: Shows a single property in wide-column layout.

  ```
  Get-ChildItem | Format-Wide -Column 3
  ```

String Formatting:

- **-f Operator**: Allows positional formatting of strings.

  ```
  $name = "John"
  $age = 30
  "My name is {0} and I am {1} years old" -f $name, $age
  ```

Pipelines for Data Processing:

- PowerShell pipelines (|) pass objects between commands for processing and formatting.

  ```
  Get-Process | Sort-Object CPU -Descending | Select-Object -First 5 | Format-Table Name, CPU
  ```

These features make PowerShell highly effective for **interactive use, script automation, and data presentation**.

Cmdlet	Description
`Format-Table`	Outputs properties in table format
`Format-List`	Lists properties vertically
`Format-Wide`	Displays only one property per line
`-f` Operator	String formatting (`"Name: {0}" -f $name`)

PowerShell Scripting Overview: Objects, Arrays, Variables

PowerShell is a powerful scripting language and command-line shell from Microsoft, designed for **automation and configuration management**. It is built on the **.NET framework** and uses **objects** instead of plain text for data processing.

Objects:

- PowerShell commands return structured **objects** with **properties** and **methods**.

- Example:
  Get-Process returns process objects with properties like Name, ID, and CPU.

Arrays:

- Used to store **collections** of items (of same or different types).
- Defined using commas:

  ```
  powershell
  CopyEdit
  $myArray = 10, 20, 30, "apple", "banana"
  ```

- Access elements using **indexing**:

  ```
  powershell
  CopyEdit
  $myArray[0]    # First element (10)
  $myArray[-1]   # Last element ("banana")
  ```

- Arrays are **dynamic**:

  ```
  powershell
  CopyEdit
  $myArray += "orange"
  ```

Variables:

- Begin with $ and are **case-insensitive**:

  ```
  powershell
  CopyEdit
  $name = "John Doe"
  $age = 30
  $isValid = $true
  ```

- PowerShell is **dynamically typed**, but you can enforce types:

  ```
  powershell
  CopyEdit
  [int]$number = "123"
  ```

- **Variable scopes** include:
  - **Local** (default)
  - **Script** (for use within scripts)
  - **Global** (accessible everywhere)

PowerShell’s use of objects, dynamic arrays, and flexible variables makes it a **powerful and efficient** tool for scripting, automation, and system management.

Scripting Constructs in PowerShell

PowerShell provides a robust set of **scripting constructs** that support automation, control flow, logic, and error handling. Here's a concise breakdown:

Core Elements

- **Variables**: Store data using $, e.g., $name = "John".
- **Arrays**: Hold collections, e.g., $fruits = @("apple", "banana").
- **Operators**:
  - *Arithmetic*: +, -, *, /, %
  - *Comparison*: -eq, -ne, -gt, -lt, -ge, -le
  - *Logical*: -and, -or, -not

Conditional Statements

- **if / elseif / else**: Choose actions based on conditions.

  $age = 25 if ($age -ge 18) { Write-Host "Adult" } elseif ($age -ge 13) { Write-Host "Teenager" } else { Write-Host "Child" }

- **switch**: Handle multiple values efficiently.

  $day = "Monday" switch ($day) { "Monday" { Write-Host "Start of the week" } "Friday" { Write-Host "End of the week" } default { Write-Host "Mid-week" } }

Loops

- **for**: Loop a set number of times.

  for ($i = 1; $i -le 5; $i++) { Write-Host "Iteration: $i" }

- **foreach**: Iterate through items in a collection.

  $colors = @("red", "green", "blue") foreach ($color in $colors) { Write-Host "Color: $color" }

- **while**: Loop while a condition is true.

  ```
  $count = 0 while ($count -lt 3) { Write-Host "Count: $count" $count++ }
  ```

- **do-while / do-until**: Execute block at least once before condition check.(similar to while)

Functions

- Reusable code blocks with optional parameters.
  Example:

  ```
  function Greet { param($name); Write-Host "Hello, $name!" }
  ```

  ```
  function Greet { param($name) Write-Host "Hello, $name!" } Greet -name "Alice"
  ```

Script Blocks

- Code enclosed in {} used as single executable units (e.g., in ForEach-Object).

Modules

- Packages of functions, cmdlets, and scripts that can be imported with Import-Module.

Error Handling

- **try / catch / finally**: Manage exceptions and define fallback actions.

  ```
  try { # Code that might throw an error Get-Content "nonexistent_file.txt" -
  ErrorAction Stop }
  ```

  ```
  catch { Write-Host "Error: $($_.Exception.Message)" }
  ```

  ```
  finally { Write-Host "Cleanup actions" }
  ```

Comments

- Use # to explain code or disable lines temporarily.

These constructs are essential for writing clear, efficient, and reusable PowerShell scripts for automation and system management.