PowerShell Cmdlets and Scripting Overview

# Introducing to Cmdlets

PowerShell’s building blocks are called **cmdlets** (short for “command-lets”). Think of them as tiny, purpose-built commands—each named with a clear “Verb-Noun” structure, like **Get-Process**, **Set-Item**, or **Remove-Item**. Behind the scenes, every cmdlet is simply a .NET class that can handle everything from file manipulation and service management to tweaking registry settings.

# The PowerShell Pipeline

What really sets PowerShell apart is its **pipeline**. By placing a vertical bar (|) between cmdlets, you can pass objects from one command directly into the next. For example, if you want to find all processes chewing up more than 100 seconds of CPU time, you’d write:

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

This approach makes it easy to chain simple commands into powerful, custom workflows.

# Key Cmdlets

While there are hundreds of cmdlets at your disposal, a few are indispensable when you’re getting started:

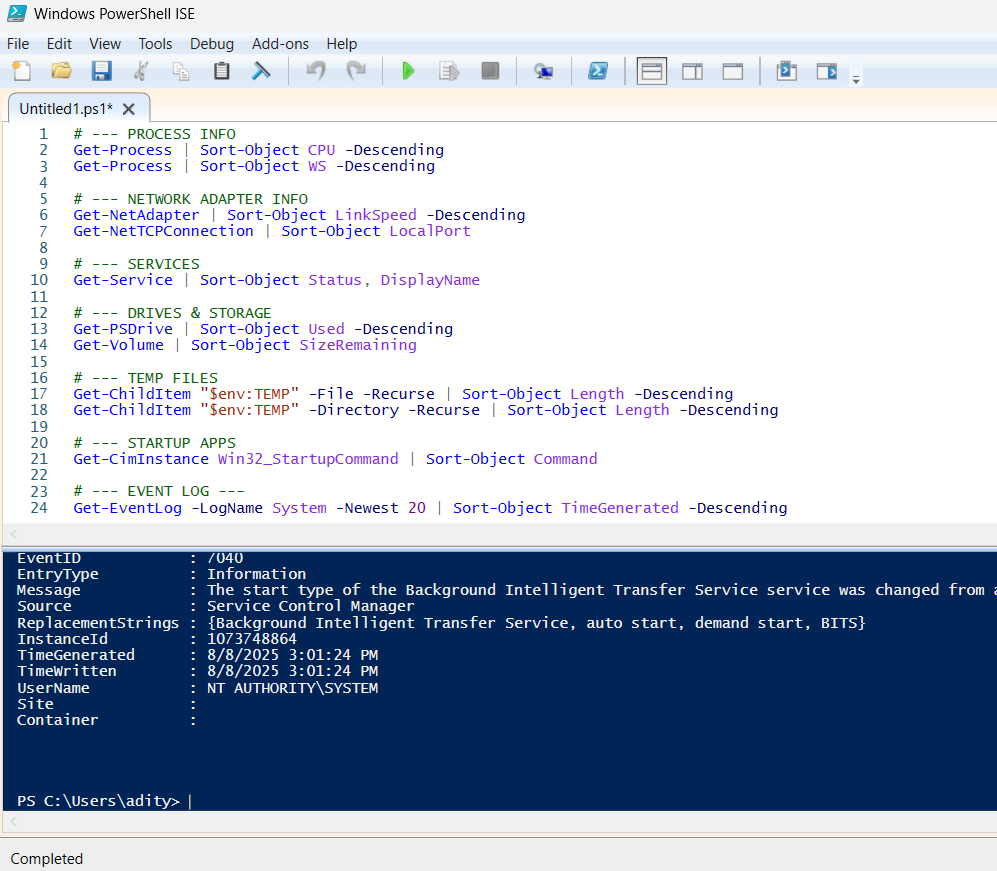
* **Get-Help** – Brings up documentation on any cmdlet or concept.
* **Get-Command** – Shows you every available cmdlet, function, and alias.
* **Get-Service** – Lets you check which Windows services are running, stopped, or paused.
* **Get-Process** – Lists active processes on your machine.
* **Set-ExecutionPolicy** – Adjusts your system’s policy for running scripts, so you can safely automate tasks.

# WMI & PowerShell

For deeper system insights, PowerShell hooks into **Windows Management Instrumentation (WMI)**. With commands like **Get-WmiObject** (or its newer cousin, **Get-CimInstance**), you can pull detailed information on hardware and configuration—everything from BIOS versions to disk health and network adapters.

# Pipeline Filtering & Operators

Filtering and selecting data in the pipeline is just as straightforward. **Where-Object** filters objects by conditions (-eq, -gt, -lt, -like, etc.), and **Select-Object** lets you pick which properties you care about. Together, they turn raw command output into precisely the data you need.



# Input, Output & Formatting

PowerShell also gives you fine-grained control over **input, output, and formatting**. Whether you’re reading from the keyboard, a file, or another command, you can pipe data through **Format-Table**, **Format-List**, or write it out with **Out-File** for later analysis.

# Scripting Overview

Finally, when your tasks grow beyond one-liners, you can bundle everything into a .ps1 script. PowerShell scripts support variables, loops, conditionals, functions, and robust error handling—making it possible to automate complex, repeatable workflows. Just remember to check or adjust your execution policy so your scripts have the permissions they need to run.

# PowerShell — Key Concepts and Practical Notes

Introduction

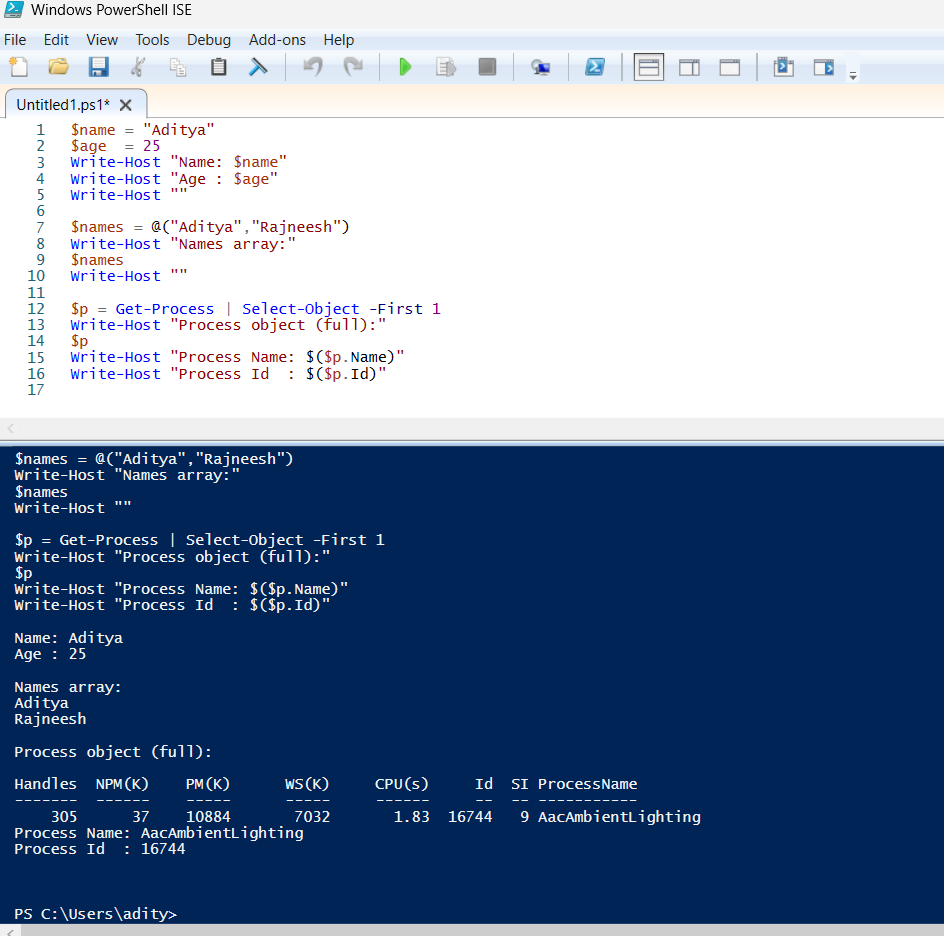
This short guide covers the PowerShell essentials you’ll want to know when automating tasks or writing scripts: objects, arrays, variables, scope, useful operators, common scripting constructs, and how to modularize your work. I’ve also added where to drop screenshots so the document looks complete and easy to follow.

# Objects, Arrays, Variables

PowerShell works with **objects**, not plain text. When a cmdlet returns something (for example Get-Process), you get an object with properties and methods you can inspect and use. That makes chaining and transforming data much cleaner.

Examples:

* Variables: $name = "Aditya", $age = 25
* Arrays: $names = @("Aditya","Rajneesh")
* Objects: $p = Get-Process | Select-Object -First 1 then use $p.Name, $p.Id.



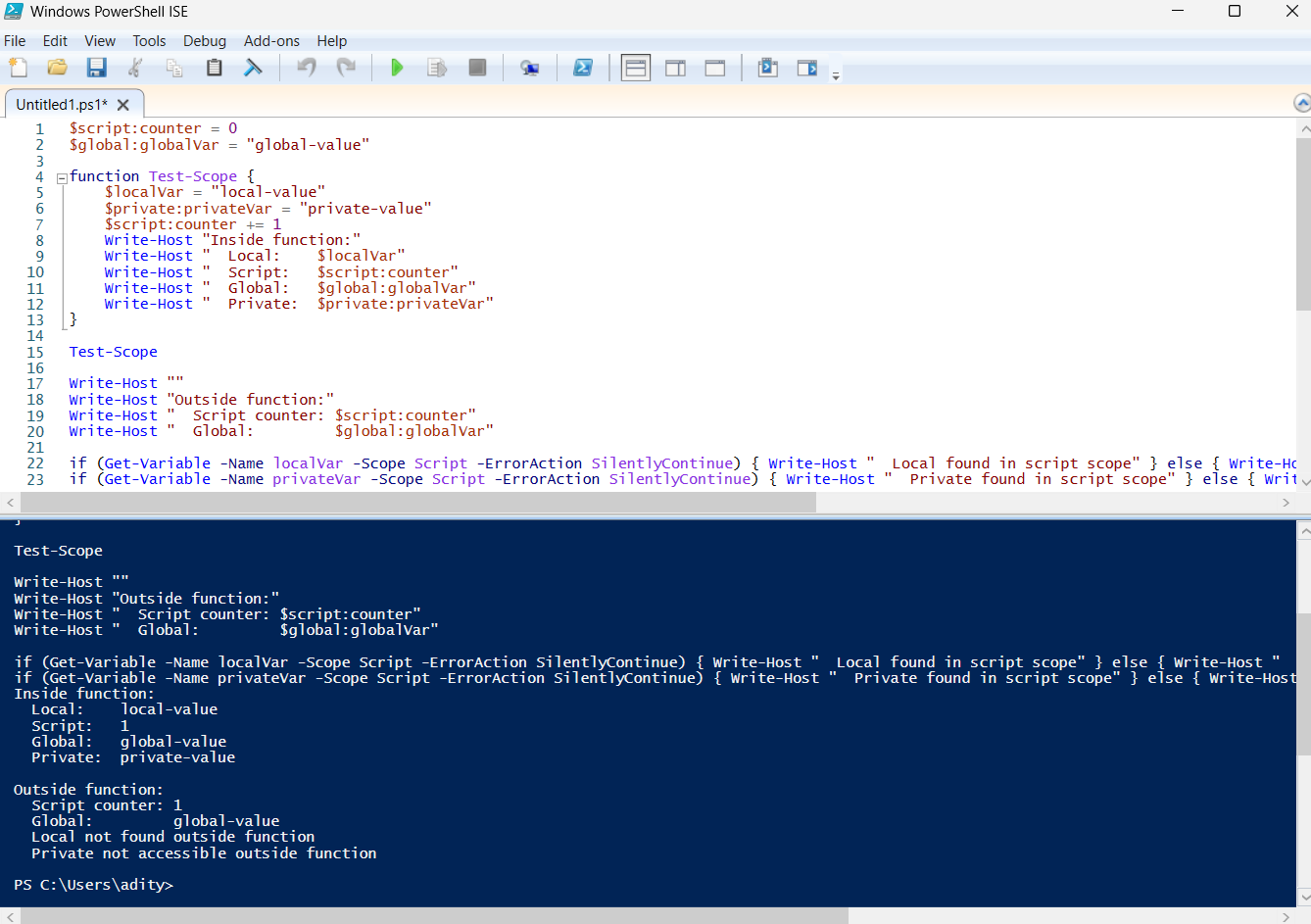
# Scope

Scope controls where a variable or function is visible. The common scopes are:

* **Local** — default inside the current block or function.
* **Script** — visible anywhere in the script file.
* **Global** — visible in the entire session.
* **Private** — explicit local visibility inside a module or function.

Example usage:

* $script:counter = 0 to make a variable available across the whole script.
* Avoid overusing global variables — they make code harder to test and reuse.



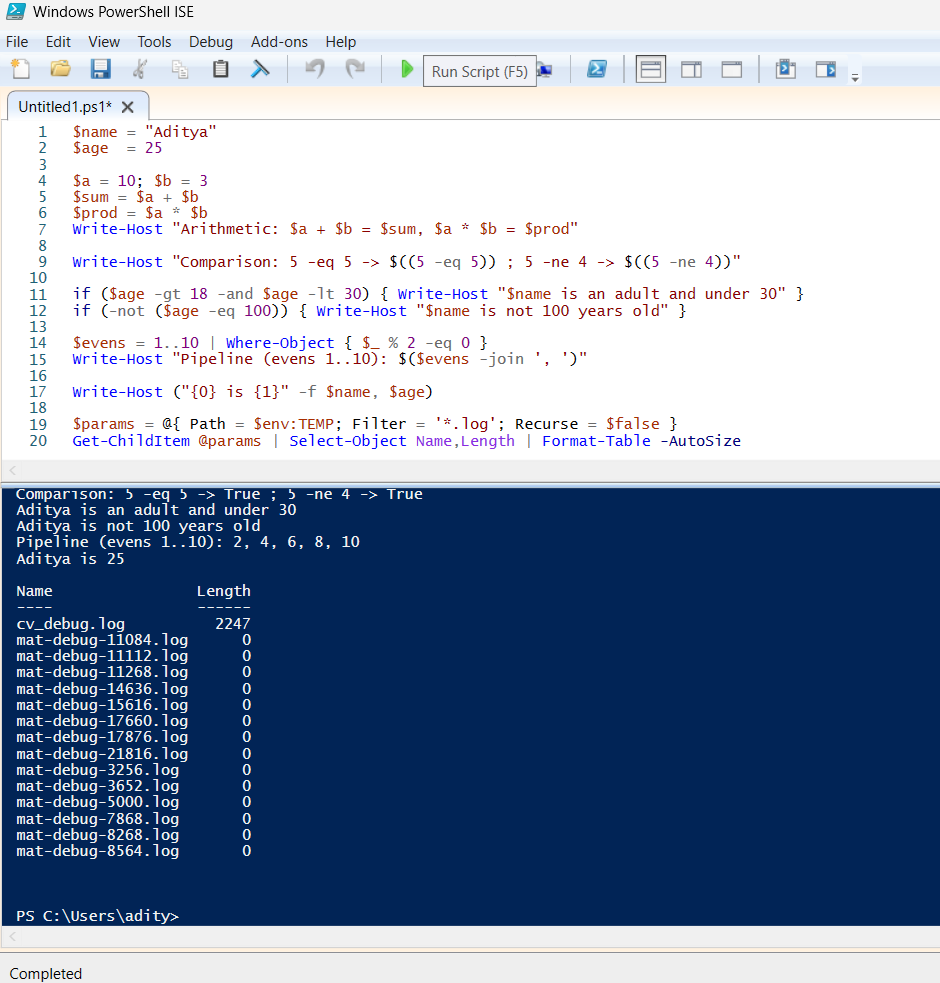
# More Operators

PowerShell’s operators are powerful and readable:

* **Arithmetic:** +, -, \*, /
* **Comparison:** -eq, -ne, -gt, -lt
* **Logical:** -and, -or, -not
* **Pipeline:** | (passes objects from one cmdlet to the next)
* **Format operator:** -f (string formatting)
* **Splatting:** @params (clean way to pass many parameters)

Quick examples:

* 1..10 | Where-Object { $\_ % 2 -eq 0 } — returns even numbers.
* "{0} is {1}" -f $name, $age — formatted string.

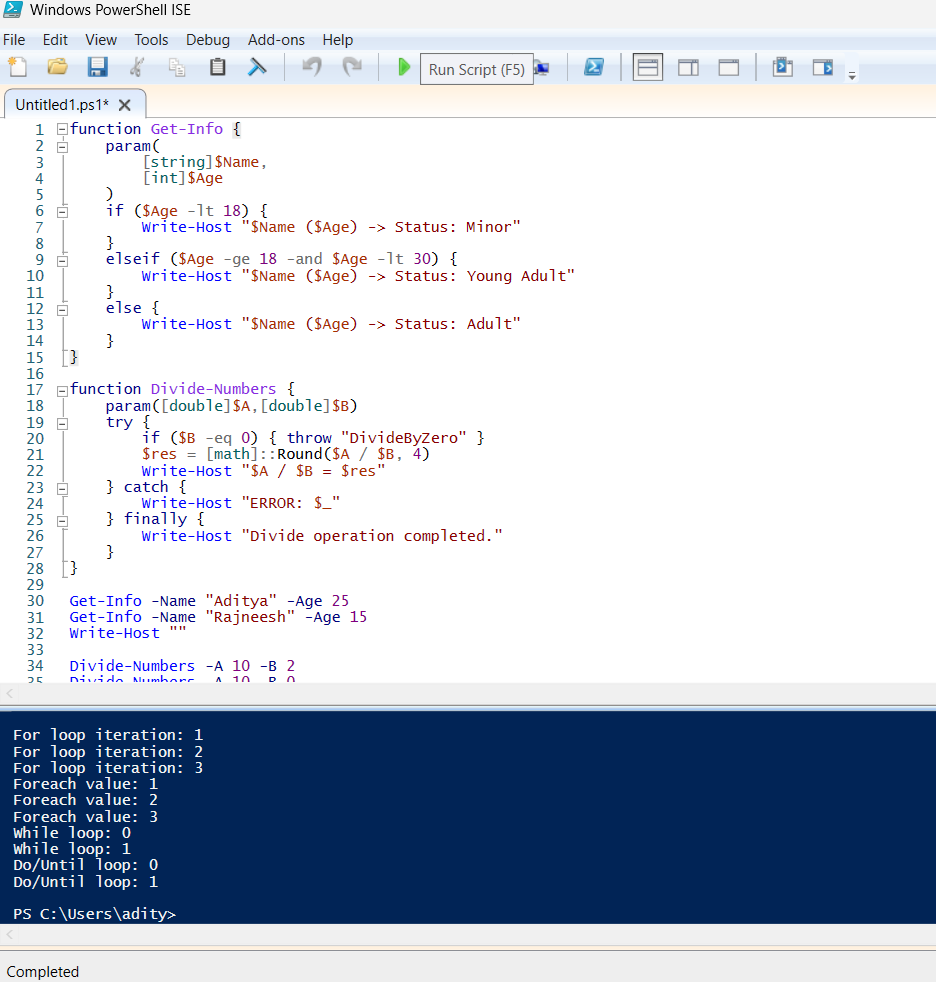


## **Scripting Constructs**

PowerShell gives you the usual building blocks for scripts:

* **Functions**: function Get-Info { param($name) ... }
* **Conditionals**: if / elseif / else
* **Loops**: for, foreach, while, do { } until
* **Error handling**: try / catch / finally
* **Parameters**: param() at the top of scripts or functions for inputs

Best practice: Use param() for inputs, validate parameters when needed, and use try/catch for predictable error handling.



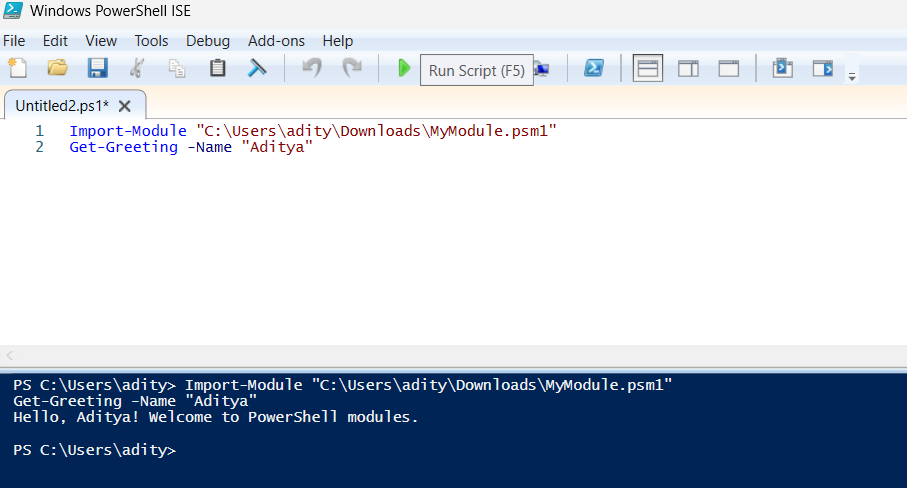
## **Modularization**

Turn repeated code into modules so it’s easy to reuse:

* Script modules: .psm1 file containing functions.
* Manifest files: .psd1 describe the module (author, version, exported functions).
* Installable modules: publish or save under $env:USERPROFILE\Documents\WindowsPowerShell\Modules\ModuleName\ModuleName.psm1 so PowerShell auto-discovers it.

Typical workflow:

1. Create MyModule.psm1 with your functions.
2. Create MyModule.psd1 with New-ModuleManifest (optional but useful).
3. Import-Module MyModule to use it in a session.
4. Use Export-ModuleMember to control what your module exposes.



## **Understand basic platform value, concepts and usage**

Why PowerShell matters:

* It’s designed for system automation and administration.
* The pipeline passes rich objects (not text), which reduces parsing pain.
* Deep integration with .NET gives you immediate access to powerful APIs.
* Works cross-platform (Windows PowerShell on Windows; PowerShell Core / pwsh on macOS/Linux).
* Good for repeatable tasks: system setup, reporting, scheduled jobs, and remote management (WinRM/SSH).

Practical usage tips:

* Start by learning common cmdlets (Get-Process, Get-Service, Get-ChildItem).
* Prefer explicit cmdlet names in scripts (don’t rely on aliases).
* Put reusable code into modules and version them.
* Keep scripts idempotent and safe: test before Remove-Item or destructive ops.

