**Today Assignment – Date: August 9,2025**

**Name: 34733-Prasang Kumar Singh**

**Gmail:** [**thakurprasang23@gmail.com**](mailto:thakurprasang23@gmail.com)

PowerShell Cmdlets, Scripts, and Commands — Detailed Expanded Version

1. Get-NetIPConfiguration | Out-File D:\powershellcommands\datafiles.txt

This PowerShell command is designed to retrieve comprehensive network configuration information from your computer.  
When you run Get-NetIPConfiguration, it fetches a detailed summary of your active network settings, which typically includes:

* IP Addresses assigned to each network adapter.
* Subnet Masks that define the network portion of the IP address.
* Default Gateways which act as the route your device uses to connect to other networks, including the internet.
* Network Adapter Information such as adapter names and types (Ethernet, Wi-Fi, etc.).
* DNS Server Addresses your system uses for resolving domain names into IP addresses.

By using the Out-File cmdlet with the absolute file path D:\powershellcommands\datafiles.txt, all of this output is redirected and stored inside a file rather than appearing on the screen.  
This ensures:

* No on-screen clutter — output is silent.
* The file can be shared for troubleshooting without running the command again.
* You can archive network settings at a particular time for future comparison.

This is especially useful for system administrators and IT support to keep official records of network configurations.

2. Get-NetIPConfiguration | Out-File D:\powershellcommands\netfileipconfigs.txt

While similar to the first command, this one serves the purpose of saving the same type of network configuration details to a different file named netfileipconfigs.txt.  
This can be especially helpful when:

* You want to keep multiple network configuration snapshots taken at different times.
* You’re troubleshooting intermittent network problems and wish to compare configuration changes.
* You want a record file separate from other system logs for quicker access.

The kind of details stored will still include IP addresses, adapter names, subnet masks, gateways, DNS servers, and other related configurations.  
By regularly running a variation of this command, it is possible to track network changes over time.

3. Get-Service | Out-File D:\powershellcommands\processfile -Append

The Get-Service cmdlet lists all the services on your Windows operating system—both running and stopped ones.  
For each service, the output includes:

* Service Name (technical name used by the OS)
* Display Name (human-friendly label)
* Status (Running, Stopped, Paused, etc.)

Using the -Append parameter in Out-File ensures that new results are added at the end of the file processfile without deleting or replacing existing entries.  
Why is this useful?

* You can track changes to service states over time, making it easier to diagnose unstable service problems.
* You can preserve a historical log of service information for compliance or auditing.
* Great for monitoring automated services (like web servers or database engines) to ensure they remain active.

4. Get-ChildItem | Out-File D:\powershellcommands\allfiles

This command lists all files and directories within your current working directory.  
It will display:

* File Names
* Folder Names
* File Sizes
* Modification Dates
* Attributes (Read-Only, Hidden, etc.)

The output is then stored in allfiles.  
Possible uses:

* Creating a permanent record of a directory's contents at a particular moment (good for backups or audits).
* Sharing directory inventory reports with colleagues or supervisors.
* Quickly checking folder changes between two given times.

5. Get-Process | Sort-Object CPU -Descending | Select-Object -First 5 | Select-Object Name, CPU

This command is a resource monitoring tool within PowerShell. Here’s what it does step by step:

1. Get-Process – lists every active process along with details like CPU time, memory usage, and process ID.
2. Sort-Object CPU -Descending – sorts those processes starting with the one using the most CPU resources.
3. Select-Object -First 5 – narrows the list to only the top 5 CPU-heavy processes.
4. Select-Object Name, CPU – further limits the output to only display the process name and total CPU time used.

This is a quick troubleshooting helper when:

* Your computer is slow and you suspect a process is overusing resources.
* You need immediate visibility into the most CPU-intensive tasks.
* You’re trying to optimize performance by identifying bottlenecks.

6. Get-EventLog -LogName System -Newest 5

This command fetches the five most recent events from your Windows System event log.  
The System log contains messages logged by the operating system and drivers, such as:

* Hardware errors (like a failing disk).
* Important driver load events.
* Critical system warnings.
* Restart or shutdown records.

Why this is important:

* You can quickly diagnose errors right after a problem occurs.
* It helps in locating the exact time and cause of certain crashes or restarts.
* Great for post-mortem analysis after unexpected shutdowns or hardware malfunctions.

7. Get-PSDrive -PSProvider FileSystem | Out-File D:\powershellcommands\datafiles.txt -Append

This lists all file system drives recognized by PowerShell, such as:

* Physical drives (C:, D:, E:)
* Mapped network drives
* Temporary virtual drives created during scripts

Appending this to datafiles.txt ensures you maintain a continuous log of available drives across multiple runs.  
Practical uses:

* Tracking removable drive activity (USB sticks, external HDDs).
* Monitoring network storage availability in corporate environments.
* Documenting storage changes for compliance and audits.

8. Get-Command | Out-File D:\powershellcommands\datafiles.txt -Append

This command outputs a complete inventory of all usable commands in your PowerShell session—which includes:

* Built-in cmdlets (Get-Process, Get-Service, etc.)
* Aliases (shorter names for commands)
* Functions (pre-defined sets of instructions)
* Modules currently loaded

Appending this to the file datafiles.txt creates a growing historical record of what commands were available at different points in time.  
This is particularly important when:

* Comparing command availability across different systems.
* Identifying missing cmdlets/modules after a system update.
* Tracking changes after software installations.

9. Get-Process

The basic version of Get-Process lists every running process on your machine along with performance and resource usage details.  
It provides:

* Process name
* Process ID (PID)
* CPU usage
* Memory consumption
* Start time (in some views)

Since it displays directly in the console (without Out-File), it’s ideal for real-time monitoring.  
You can pair it with additional PowerShell filters or sort functions for focused trouble

10. Get-Service | Get-Member | Out-File D:\powershellcommands\opfilecommands.txt

Purpose

This command is a three-step pipeline that helps you inspect the structure and capabilities of objects returned by Get-Service and then store that metadata into a text file for documentation, learning, or troubleshooting purposes.

Step-by-Step Breakdown

1. Get-Service
   * Retrieves a list of all services on your Windows system.
   * Each service object contains information like:
     + Service name (technical system name)
     + Display name (human-readable)
     + Status (e.g., Running, Stopped)
     + Start type (Automatic, Manual, Disabled)
   * These are rich objects of type System.ServiceProcess.ServiceController.
2. | Get-Member
   * Sends each ServiceController object to Get-Member.
   * Lists:
     + Properties – such as ServiceName, Status, StartType.
     + Methods – actions like Start(), Stop(), Pause().
     + Events – triggers related to the service.
     + Aliases – alternate names for certain members.
   * Knowing these members helps in precise scripting and automation.
3. | Out-File D:\powershellcommands\opfilecommands.txt
   * Redirects the analyzed structure to a file instead of the screen.
   * The resulting document contains:
     + Type name
     + Full property/method list
   * Useful for:
     + Offline reference
     + Documentation for class projects or IT work
     + Troubleshooting scripts

Example Output (Summary)

The file will start with:

text

TypeName: System.ServiceProcess.ServiceController

Followed by a complete list of:

* Methods like .Start() / .Stop()
* Properties like ServiceName, Status, StartType

This is essentially a reference guide for ServiceController objects.

Uses

* Learning full object details for better PowerShell scripting.
* Documenting service properties and actions for IT automation.
* Teaching object-member concepts in a B.Tech lab session.

11. Get-Date | Get-Member | Out-File D:\powershellcommands\opfilecommands.txt -Append

Purpose

This command documents the full structure and capabilities of the object returned by Get-Date (a System.DateTime object) and adds it to the same file created earlier, building a personal PowerShell object capability reference.

Step-by-Step Breakdown

1. Get-Date
   * Retrieves the current date and time.
   * Returns a .NET System.DateTime object, not plain text.
   * Contains multiple details:
     + Date (day, month, year)
     + Time (hour, minute, seconds)
     + Time zone context (Kind property)
2. | Get-Member
   * Shows all methods to manipulate or format dates:
     + .AddDays(), .AddHours(), .AddMonths()
     + .ToShortDateString(), .ToUniversalTime()
     + .Subtract() for calculating differences.
   * Shows properties like:
     + Year, Month, Day
     + DayOfWeek, Ticks, TimeOfDay
3. | Out-File … -Append
   * Appends the output to the opfilecommands.txt file (does not overwrite previous data from Step 10).
   * Builds a combined reference file—now containing both ServiceController and DateTime object structures.

Example Output (Summary)

The file will now also contain:

text

TypeName: System.DateTime

And over 60 properties and methods such as:

* .AddYears()
* .ToLongDateString()
* .DayOfWeek
* TimeOfDay

Why Useful

* Learning the depth of DateTime manipulation in PowerShell.
* Enables precise scheduling, logging, and time calculations in scripts.
* Creates a ready-to-use study and reference document for B.Tech coursework.

12. Get-Date | Select-Object -Property Second

Purpose

This command retrieves the current system date and time, but instead of showing every detail, it filters and displays only the “Second” component — a value from 0 to 59 that represents the seconds within the current minute.  
It’s very useful when only the precise second portion of the current time is needed, such as in interval-based logging, performance benchmarking, or time-sensitive automation.

How it Works

1. Get-Date — Fetches the system’s present date and time as a rich .NET System.DateTime object.
2. | Select-Object -Property Second — Sends that object through the pipeline to Select-Object, which keeps only the Second property, hiding all other date/time parts.

Example Output

If your system time is:

text

12 August 2025 14:25:47

The output will be:

text

Second

------

47

Common Uses

* Adding precise second markers in monitoring or diagnostic logs.
* Measuring elapsed time within the same minute.
* Testing automated scripts with second-based triggers.

13. Get-Date | Select-Object -Property TimeOfDay

Purpose

This command provides the exact time elapsed since midnight as a TimeSpan value.  
The TimeOfDay property includes hours, minutes, seconds, and fractions of a second, which is particularly valuable for timing performance, calculating durations, or high-precision logging.

How it Works

1. Get-Date — Returns the current system date/time object.
2. Select-Object -Property TimeOfDay — Extracts only the TimeOfDay property from the object, showing how far the system has progressed into the current day.

Example Output

text

TimeOfDay

---------

14:25:47.1530000

Here, the system has been up 14 hours, 25 minutes, and 47 seconds since midnight (local time).

Key Uses

* Measuring uptime since midnight for applications or servers.
* Calculating script execution windows during the day.
* Comparing time intervals in performance tests.

14. Get-Command \*hotfix\* | Out-File D:\powershellcommands\opfilecommands.txt -Append

Purpose

This command searches for all PowerShell commands whose names contain the keyword "hotfix", then appends the results to a text file for documentation purposes rather than showing them only on screen.

How it Works

1. Get-Command \*hotfix\* — Queries available commands and filters them by name pattern (\*hotfix\*), which includes both native cmdlets and functions from modules.
2. | Out-File … -Append — Writes this list to the file without deleting its existing content, ensuring that new entries are simply added.

Practical Uses

* Maintaining a command availability log for certain features.
* Tracking module or feature installation changes over time.
* Quickly reviewing update-related commands after installing new Windows components.

15. Get-HotFix | Get-Member

Purpose

This is an object inspection command. It lists all installed Windows Hotfixes (updates, patches, security fixes) and then passes them to Get-Member to display their complete properties, methods, and object type.

How it Works

1. Get-HotFix — Retrieves a list of updates installed on the system, each as an object containing fields like:
   * HotFixID — Identifier of the hotfix.
   * InstalledOn — Date of installation.
   * InstalledBy — User/system that installed it.
2. | Get-Member — Shows all members (properties & methods) for the update object type:
   * TypeName: System.Management.ManagementObject#root\cimv2\Win32\_QuickFixEngineering
   * Properties: Caption, Description, InstalledOn, etc.
   * Methods: Conversion utilities related to date/time.

Why This is Important

* Helps scripters learn which properties are available for filtering or reporting.
* Essential when formatting update reports in scripts.
* Assists in creating custom update compliance checks.

16. Get-Hotfix | Select-Object –Property InstalledBy | Out-File D:\powershellcommands\netipconfigs -Append

Purpose

This extracts the InstalledBy property from all installed hotfixes — showing who installed each update — and logs it into a file for audit/documentation purposes.

Example Output

text

InstalledBy

-----------

NT AUTHORITY\SYSTEM

NT AUTHORITY\SYSTEM

Why Useful

* Important for security audits (knowing which account installed updates).
* Helps in administrative accountability in multi-admin systems.
* Allows quick filtering of updates applied via system accounts vs. human administrators.

17. Get-Hotfix | Select-Object –Property HotFixID, InstalledOn, InstalledBy | Out-File D:\powershellcommands\netipconfigs -Append

Purpose

Generates a detailed update report by listing:

* Hotfix ID
* Installation date
* Installer  
  And adds it to the tracking file.

Key Benefits

* Facilitates historical tracking of patches.
* Useful for system change documentation.
* Supports troubleshooting by identifying when a system started having issues relative to updates.

18. Get-Hotfix | Select-Object –Property HotFixID, @{n='HotFixAge'; e={(New-TimeSpan –Start $PSItem.InstalledOn).Days}}, InstalledBy

Purpose

Enhances the standard hotfix list by calculating the age (in days) since installation, alongside ID and installer.

Why Important

* Quickly identifies older updates.
* Helps risk assess outdated fixes that might require reapplication.
* Useful in compliance reports for patch lifecycle management.

19. help \*rule\*

Purpose

Searches the PowerShell help index for all topics whose names contain the word "rule".

Uses

* Discovering all firewall rule-related cmdlets quickly.
* Learning cmdlet syntax and examples before using them.
* Useful for exploring capabilities in networking scripts.

20. Get-NetFirewallRule | Out-File D:\powershellcommands\netipconfigs -Append

Purpose

Retrieves every currently configured Windows Firewall Rule, including inbound/outbound rules, and sends this list to a file for backup and auditing.

Why Useful

* Security audits.
* Comparing before/after firewall configurations.
* Backup for disaster recovery planning.

21. Get-NetIPConfiguration | Out-File D:\powershellcommands\netipconfigs -Append

Purpose

Retrieves current network configuration details (IP addresses, subnet, default gateway, DNS servers) and appends them to a file.

Practical Use Cases

* Tracking network changes over time.
* Assisting in troubleshooting connectivity issues.
* Creating documentation for IT handovers.

22. Saving Help Information to a File

powershell

CopyEdit

Help Get-NetFirewallRule | Out-File D:\powershellcommands\netipconfigs -Append

* What it does:  
  This command takes the full help guide for the Get-NetFirewallRule cmdlet — including what the command does, all its parameters, and usage examples — and saves it into a text file named netipconfigs inside D:\powershellcommands.
* Why it’s useful:  
  You can read the help later without using PowerShell or the internet. The -Append part ensures the file keeps old content and simply adds new text at the bottom.
* Example use case:  
  Creating your own offline PowerShell “manual” with all command references in one place.

23. Viewing Help Directly

powershell

CopyEdit

Help Get-NetFirewallRule

* What it does:  
  Displays the help for Get-NetFirewallRule directly in PowerShell. It includes explanations, syntax, parameters, and examples.
* Why it’s useful:  
  A quick way to learn what a command does without saving the output to a file.

24. Listing Only Enabled Firewall Rules

powershell

CopyEdit

Get-NetFirewallRule –Enabled True | Out-File D:\powershellcommands\a.txt -Append

* What it does:  
  Shows only the firewall rules that are currently active and saves them to a file named a.txt, adding to the file instead of replacing it.
* Why it’s useful:  
  This gives a clean list of all active firewall rules for checking or archiving.

25. Enabled Firewall Rules in Table Format

powershell

CopyEdit

Get-NetFirewallRule –Enabled True | Format-Table -wrap | Out-File D:\powershellcommands\b.txt -Append

* What it does:  
  Shows only enabled firewall rules but displays them neatly in a table with long lines wrapped for easier reading. Saves to b.txt.

26. Listing All Firewall Rules

powershell

CopyEdit

Get-NetFirewallRule | Format-Table -wrap | Out-File D:\powershellcommands\abc.txt -Append

* What it does:  
  Lists every firewall rule — both enabled and disabled — formats it into a table, and saves to a file named abc.txt.

27. Sorted Firewall Rules with Specific Details

powershell

CopyEdit

Get-NetFirewallRule –Enabled True |

Select-Object –Property DisplayName,Profile,Direction,Action |

Sort-Object –Property Profile, DisplayName |

Format-Table -wrap |

Out-File D:\powershellcommands\ad.txt -Append

* What it does:  
  Filters only enabled rules, shows key details (Name, Profile, Direction, Action), sorts them nicely, and saves to a file.

28. Viewing ARP Cache

powershell

CopyEdit

Get-NetNeighbor

* What it does:  
  Shows the list of IP addresses and MAC addresses of devices on your local network.

29. Grouping Network Neighbors by State

powershell

CopyEdit

Get-NetNeighbor | Sort-Object –Property State |

Select-Object –Property IPAddress, State |

Format-Wide -GroupBy State |

Out-File D:\powershellcommands\cd.txt -Append

* What it does:  
  Lists ARP cache entries, groups them by connection state, and saves the output to a file.

30. Comparison Operator Examples

powershell

CopyEdit

100 -gt 10 # True

500 -le 100 # False

'hello' -eq 'HELLO' # True

'hello' -ceq 'HELLO' # False

* What it does:  
  Shows how to compare numbers and text in PowerShell.

31. Listing Services That Are Not Running

powershell

CopyEdit

Get-Service | Where status -ne Running | Out-File D:\powershellcommands\a.txt -Append

* What it does:  
  Finds all stopped or paused services and saves them to a file.

32. Listing Non-Running Services Without Saving

powershell

CopyEdit

Get-Service | Where status -ne Running

* What it does:  
  Shows stopped or paused services on the screen without saving.

33. Finding High CPU Usage Processes

powershell

CopyEdit

Get-process | Where CPU -gt 100

* What it does:  
  Shows processes that have used more than 100 seconds of CPU time.

34. Closing Notepad

powershell

CopyEdit

Get-process -Name notepad | Stop-Process

* What it does:  
  Finds Notepad processes and closes them.

35. Showing Only Service Status

powershell

CopyEdit

Get-Service | ForEach Status

* What it does:  
  Displays only the status of each service.

36. Showing Only Service Display Names

powershell

CopyEdit

Get-Service | ForEach Displayname

* What it does:  
  Displays the display names of all services.

37. Viewing All Event Logs

powershell

CopyEdit

Get-EventLog –List

* What it does:  
  Lists all available event logs with their sizes and settings.

38. Viewing Only the System Event Log

powershell

CopyEdit

Get-EventLog –List | Where Log –eq 'System'

* What it does:  
  Filters the event log list to show only the “System” log.

39. Converting Process List to HTML

powershell

CopyEdit

Get-Process | ConvertTo-HTML

* What it does:  
  Converts process information into HTML format for browser viewing.

40. Saving HTML Output to a File

powershell

CopyEdit

Get-Process | ConvertTo-HTML | Out-File D:\powershellcommands\htmldata.txt

* What it does:  
  Saves the HTML-formatted process list into a file.

41. Listing Network CIM Classes

powershell

CopyEdit

Get-CimClass \*network\* | Sort-Object CimClassName

* What it does:  
  Lists all CIM classes related to networking and sorts them alphabetically.

**42. Getting Keyboard Hardware Information**

powershell

CopyEdit

Get-WmiObject -Class Win32\_Keyboard

* **What it does:**  
  This command asks Windows for detailed information about your computer’s keyboard hardware using WMI (Windows Management Instrumentation).
* **What you’ll see:**  
  Things like:
  + Device ID
  + Description (e.g., “Standard PS/2 Keyboard”)
  + Status (OK, Error, etc.)
* **Why it’s useful:**  
  Helpful for hardware inventory, troubleshooting issues where multiple keyboards might be connected, or confirming the system is detecting the keyboard correctly.

**43. Listing All Available WMI Classes**

powershell

CopyEdit

Get-WmiObject

* **What it does:**  
  If you run Get-WmiObject without any extra parameters, PowerShell shows you *all* WMI classes available in the default namespace (root\cimv2).
* **Why it’s useful:**  
  You can discover which system, hardware, and configuration classes you can query — like a “menu” of WMI data sources.

**44. Saving Root Namespace Classes to a File**

powershell

CopyEdit

Get-WmiObject -Namespace root -List | Out-File D:\powershellcommands\Namespace.txt -Append

* **What it does:**  
  Lists all WMI classes from the **root** namespace (the very top level of WMI) and appends them to Namespace.txt.
* **Why it’s useful:**  
  Great for creating documentation of the base WMI namespace without overwriting your existing notes.

**45. Listing CIMv2 Namespace Classes in Alphabetical Order**

powershell

CopyEdit

Get-WmiObject -Namespace root\cimv2 -List | Sort Name

* **What it does:**  
  Shows all WMI classes inside the root\cimv2 namespace, sorted alphabetically by name.
* **Why it’s useful:**  
  Makes it much easier to find a particular class quickly instead of scanning a random order list.

**46. Saving CIMv2 Namespace Classes to a File**

powershell

CopyEdit

Get-WmiObject -Namespace root\CIMv2 -List | Out-File D:\powershellcommands\cimv2.txt -Append

* **What it does:**  
  Gets all WMI classes from the root\CIMv2 namespace and adds them to cimv2.txt.
* **Why it’s useful:**  
  Good for keeping an offline reference of all available system information classes.

**47. Saving and Sorting CIMv2 Classes to a File**

powershell

CopyEdit

Get-WmiObject -Namespace root\cimv2 -List | Sort Name | Out-File D:\powershellcommands\sortcimv2.txt -Append

* **What it does:**  
  Retrieves all classes from root\cimv2, sorts them alphabetically, and saves the sorted list into sortcimv2.txt.
* **Why it’s useful:**  
  Creates a neat, organized reference that’s easy to read and search.

**48. Listing CIM Classes with CIM Cmdlets**

powershell

CopyEdit

Get-CimClass -Namespace root\CIMv2 | Sort CimClassName

* **What it does:**  
  Retrieves all CIM classes from the root\CIMv2 namespace and sorts them alphabetically.
* **Why it’s useful:**  
  This is the newer CIM way to explore what information your system can provide, often preferred over Get-WmiObject for newer scripts.

**49. Getting Chip Hardware Information**

powershell

CopyEdit

Get-WmiObject -Class CIM\_Chip

* **What it does:**  
  Retrieves details about hardware chips inside your computer — such as processors, chipsets, or other integrated chips.
* **Why it’s useful:**  
  Handy for low-level hardware diagnostics, system inventory, or support ticket reporting.

**50. Getting Installed Operating System Details**

powershell

CopyEdit

Get-WmiObject -ClassName CIM\_InstalledOS

* **What it does:**  
  Fetches details about the installed operating system using the CIM\_InstalledOS class, like version, manufacturer, and install date.
* **Why it’s useful:**  
  Good for confirming OS details on remote machines or documenting system environments.

**51. Listing All Logical Disks (CIM)**

powershell

CopyEdit

Get-CimInstance -ClassName Win32\_LogicalDisk

* **What it does:**  
  Lists all logical disks on the computer — these are things like:
  + Hard drive partitions (C:, D:)
  + CD/DVD drives
  + USB drives
* **Why it’s useful:**  
  Shows size, free space, and drive type for storage management.

**52. Listing All Logical Disks (WMI)**

powershell

CopyEdit

Get-WmiObject -Class Win32\_LogicalDisk

* **What it does:**  
  Same as the CIM command but using WMI instead.

**53. Showing Only Local Fixed Drives (WMI)**

powershell

CopyEdit

Get-WmiObject -Class Win32\_LogicalDisk -Filter "DriveType=3"

* **What it does:**  
  Filters the output so only internal hard drives (DriveType 3) are shown, excluding USB drives or network drives.

**54. Showing Only Local Fixed Drives (CIM)**

powershell

CopyEdit

Get-CimInstance -ClassName Win32\_LogicalDisk -Filter "DriveType=3"

* **What it does:**  
  Same as above, but using the CIM cmdlet syntax.

**55. Filtering Local Drives with a WMI Query**

powershell

CopyEdit

Get-WmiObject -Query "SELECT \* FROM Win32\_LogicalDisk WHERE DriveType = 3"

* **What it does:**  
  Runs a SQL-like query to pull only local fixed drives.

**56. Filtering Local Drives with a CIM Query**

powershell

CopyEdit

Get-CimInstance -Query "SELECT \* FROM Win32\_LogicalDisk WHERE DriveType = 3"

* **What it does:**  
  Same as the WMI query, but using the newer CIM cmdlet interface.