

ELLYPUNK

Foundation

WEEKLY DIGEST / E101

3-9 August 2026



This week

3-9 August 2026

Available archived editions, in their original date order.

MONDAY / 3 August 2026

PowerShell 101: Meet the Shell

4

[Open original online](#)

TUESDAY / 4 August 2026

PowerShell 102: Objects Power the Pipeline

9

[Open original online](#)

WEDNESDAY / 5 August 2026

PowerShell 103: Files and Structured Data

14

[Open original online](#)

THURSDAY / 6 August 2026

PowerShell 104: Variables, Decisions, Loops

19

[Open original online](#)

FRIDAY / 7 August 2026

PowerShell 105: Scripts, Functions, Parameters

24

[Open original online](#)

Only retained editions are included; gaps have not been filled.

Copyright Ellypunk. Contains AI-generated artwork.

ellypunk.com

This week

3-9 August 2026

Available archived editions, in their original date order.

SATURDAY / 8 August 2026

PowerShell 106: Errors, Safety, Debugging

29

[Open original online](#)

SUNDAY / 9 August 2026

PowerShell 107: Build a Daily System Report

34

[Open original online](#)

POWERSHELL 101: MEET THE SHELL



Hi! I'm **Elly**.
Welcome to **PowerShell** 101!
PowerShell is a command shell and scripting language built for Windows, Linux, and beyond.
It helps you **automate work**, **save time**, and **get more done**.



WHAT IS POWERSHELL?



PowerShell is both a **command shell** and a **scripting language**.

-  **COMMAND SHELL**
Run commands one at a time to get things done.
-  **SCRIPTING LANGUAGE**
Write scripts to automate repetitive tasks.

 Works across Windows, Linux, and macOS.
Consistent. Powerful. Productive.

THE POWER OF VERB-NOUN



PowerShell commands follow a simple pattern:
Verb-Noun.
The **verb** is the action.
The **noun** is what the action applies to.



 This makes commands **easy to guess**, **learn**, and **discover**!

LET'S SEE IT IN ACTION!



Here are some common examples.

Get-Date



Get the current date and time.

PS> **Get-Date**

Get-Process



Get the list of running processes.

PS> **Get-Process**

Get-Help



Get help about commands.

PS> **Get-Help**



That's the beauty of **PowerShell**!
You can discover powerful commands just by thinking in **Verb-Noun**.

KEY TAKEAWAY

VERB-NOUN

MAKES COMMANDS EASY TO DISCOVER

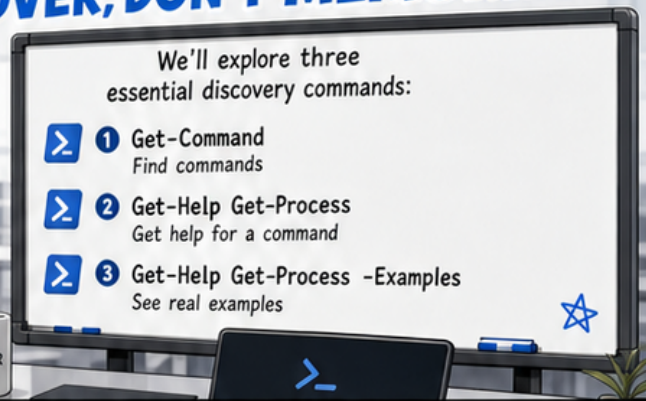


POWERSHELL 101: MEET THE SHELL

DISCOVER, DON'T MEMORIZE



In PowerShell, you don't have to memorize everything. You can **DISCOVER** what's available and learn as you go!



1 GET-COMMAND: FIND COMMANDS

Want to see what commands are available? Use **Get-Command**!



```
PS> Get-Command -Module Microsoft.PowerShell.Management
```

CommandType	Name	Version	Source
Cmdlet	Get-Process	7.4.1	Microsoft.PowerShell.Management
Cmdlet	Stop-Process	7.4.1	Microsoft.PowerShell.Management
Cmdlet	Start-Process	7.4.1	Microsoft.PowerShell.Management
Cmdlet	Wait-Process	7.4.1	Microsoft.PowerShell.Management
...			



Get-Command shows you what exists—so you can **explore** and learn.

2 GET-HELP GET-PROCESS: GET HELP

Found a command? Get detailed help with **Get-Help <command>**.



```
PS> Get-Help Get-Process
```

NAME
Get-Process

SYNOPSIS
Gets the processes that are running on the local computer or a remote computer. *What it does*

SYNTAX
Get-Process [[-Name] <string[]>] [-Id <int[]>] [-ComputerName <string[]>] [<CommonParameters>] *How to use it (syntax)*

PARAMETERS

- Name <string[]>
Specifies the process names.
- Id <int[]>
Specifies the process IDs. *What you can customize*
- ComputerName <string[]>
Specifies the remote computers.
- <CommonParameters>
This cmdlet supports the common parameters.

3 GET-HELP GET-PROCESS -EXAMPLES: SEE EXAMPLES

Examples show you how the command is used in **real situations**.



```
PS> Get-Help Get-Process -Examples
```

EXAMPLE 1
Get-Process
Gets all the processes running on the local computer.

EXAMPLE 2
Get-Process -Name notepad
Gets the process named Notepad.

EXAMPLE 3
Get-Process | Where-Object { \$_.CPU -gt 100 }
Gets the processes that are using more than 100 CPU seconds.

EXAMPLE 4
Get-Process -Id 1234 -ComputerName Server01
Gets the process with the ID 1234 on the remote computer Server01.



Examples turn knowledge into **confident action**.

KEY TAKEAWAY

HELP IS BUILT IN



1 DISCOVER

Use Get-Command to find commands.



2 LEARN

Use Get-Help to learn syntax and parameters.



3 APPLY

Use -Examples to see real-world usage.



Explore, read the help, try the examples—that's how you become a **confident PowerShell user!**



Explore more. Learn more. Do more with PowerShell.

NEXT: YOUR FIRST COMMAND



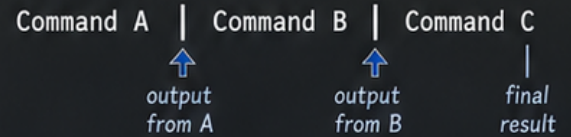
POWERSHELL 101: MEET THE SHELL

THE PIPELINE CONNECTS COMMANDS



In PowerShell, the **pipeline** (|) sends the **output** of one command as the **input** to the next command.

PowerShell



EXAMPLE

Let's look at a simple pipeline:

Get-Process | **Sort-Object CPU -Descending** | **Select-Object -First 5**

THREE BEATS, ONE FLOW

1 GET PROCESSES

Get-Process gets a list of all running processes.

Id	Process	CPU
4320	Code	1.2
2104	PowerShell	0.7
812	chrome	4.5
1296	Teams	0.3
5220	spotify	2.1
...		

2 SORT BY CPU

Sort-Object CPU -Descending orders the list from highest CPU to lowest.

Id	Process	CPU
812	chrome	4.5
5220	spotify	2.1
4320	Code	1.2
2104	PowerShell	0.7
1296	Teams	0.3
...		

3 KEEP THE FIRST FIVE

Select-Object -First 5 keeps only the top 5 results.

Id	Process	CPU
812	chrome	4.5
5220	spotify	2.1
4320	Code	1.2
2104	PowerShell	0.7
1296	Teams	0.3

WHAT HAPPENS

Data flows from left to right. Each command works on the output it receives.



Small commands.
Connected together.
Big results.



KEY TAKEAWAY

SMALL COMMANDS, ONE USEFUL FLOW

The pipeline (|) lets you chain commands so the output of one becomes the input of the next. That's how you do more with less.



REMEMBER

- ✓ | connects commands
- ✓ Output becomes input
- ✓ Left to right
- ✓ Clear, focused steps



Learn the shell. Use the pipeline. Get things done.

NEXT: Seeing the Pipeline in Action! →

POWERSHELL 101: MEET THE SHELL

PAGE 4

FIND YOUR WAY AROUND

EVERY
COMMAND
IS A STEP
FORWARD
➤

Let's learn three simple navigation commands to move around your file system safely.

OUR TERMINAL JOURNEY



1. Get-Location — shows where you are

Run **Get-Location** to see your current location (folder path).

```

PowerShell
PS C:\Users\Elly> Get-Location

Path
----
C:\Users\Elly\Documents
    
```

Think of it as
"Where am I
right now?"

2. Get-ChildItem — lists items in the current folder

Run **Get-ChildItem** (aka **dir**) to see what's in the current folder.

```

PowerShell
PS C:\Users\Elly\Documents> Get-ChildItem

Mode                LastWriteTime         Length Name
----                -
d-----          5/12/2025 10:15 AM             Projects
d-----          5/10/2025  9:02 AM             Notes
-a----          5/11/2025  3:45 PM           1234 todo.txt
-a----          5/11/2025  4:05 PM           2048 ideas.md
-a----          5/10/2025  8:30 PM             512 plan.csv
    
```

It lists files
and folders
here.

3. Set-Location — changes folder

Use **Set-Location** (aka **cd**) to move to a different folder.

```

PowerShell
PS C:\Users\Elly\Documents> Set-Location Projects
PS C:\Users\Elly\Documents\Projects> Get-Location

Path
----
C:\Users\Elly\Documents\Projects
    
```

```

PowerShell
PS C:\Users\Elly\Documents\Projects> Get-ChildItem

Mode                LastWriteTime         Length Name
----                -
-a----          5/12/2025  9:10 AM           1024 readme.md
-a----          5/12/2025  9:12 AM           2048 app.ps1
d-----          5/12/2025  9:11 AM             src
    
```

Now you're
in the
Projects
folder!

✓ READ FIRST. RUN SECOND.

Not sure what a command does? **READ** it first.
Use **Get-Help <command>** for details.

E
X
A
M
P
L
E

```

PS> Get-Help Set-Location -Full
NAME
    Set-Location
SYNOPSIS
    Changes the current location.
...
    
```

A little reading
keeps your
system safe
and your
journey smooth!

★ Know where you are. See what's there. Go where you want.

NEXT: Working with Files ➡

YOUR FIRST POWERSHELL MISSION

Great job learning the basics! Let's put it together with a quick **mission**. You'll run a command, discover more, read its help, and build your first **pipeline**!

1. RUN Get-Date

First, let's check the date and time.

```
PS C:\> Get-Date

Sunday, May 18, 2025
10:15:30 AM
```

One simple command, instant result.

2. DISCOVER WITH Get-Command

Now, let's discover a command for looking at running processes.

```
PS C:\> Get-Command *Process*

CommandType Name
-----
Cmdlet      Get-Process
Cmdlet      Stop-Process
Cmdlet      Start-Process
Cmdlet      Wait-Process
```

PowerShell shows you what's available. **Explore** and get **curious**!

3. LEARN WITH Get-Help

Let's open the help for **Get-Process**.

```
PS C:\> Get-Help Get-Process -Examples

EXAMPLE 1
Get-Process
Gets the processes on the local computer.

EXAMPLE 2
Get-Process -Name notepad
Gets the Notepad process.

EXAMPLE 3
Get-Process | Sort-Object CPU -Descending
Sorts processes by CPU usage.
```

Help is your built-in **guide**. Read the examples and learn by doing!

4. CONNECT WITH |

Finally, build your first **pipeline** to see the top 5 processes by memory usage.

```
PS C:\> Get-Process |
Sort-Object PM -Descending |
Select-Object -First 5

Handles NPM(K) PM(M) WS(M) CPU(s) Id ProcessName
-----
1234    90,512 512.3 600.1 12.34 4568 chrome
890     45,320 310.6 400.2 6.21 2340 powershell
650     25,104 220.4 300.0 2.11 7892 code
432     18,220 150.2 210.1 1.25 1112 explorer
321     12,340 110.7 150.3 0.75 210  notepad
```

Pipelines let commands work together. This is where PowerShell **shines**!

POWERSHELL FOUNDATIONS CHECKLIST



1

RUN **Get-Date**

You ran your first command.

2

DISCOVER WITH **Get-Command**

You explored what's possible.

3

LEARN WITH **Get-Help**

You read help and examples.

4

CONNECT WITH |

You built your first pipeline.

★ **DISCOVER → READ → RUN → COMBINE**

You've got the foundation. Keep practicing and have fun! ★

POWERSHELL 102: OBJECTS POWER THE PIPELINE



In PowerShell,
the pipeline doesn't
pass plain text...
it passes **rich OBJECTS!**

NOT JUST LINES OF TEXT

Other tools send strings—just
lines of text.

```
chrome 1234 12.3
code 5678 1.1
explorer 9012 5.4
...
```

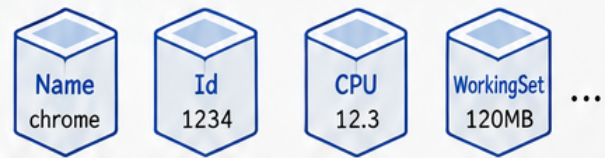
Harder to work with.
Easy to break.



POWERSHELL SENDS OBJECTS

PowerShell sends structured objects
with named properties.

PS> **Get-Process**



Easy to filter, sort, format,
and build on.

SEE IT WITH **Get-Process**

Let's run
Get-Process
and look at
the objects it
returns!

PS> **Get-Process**

Name	Id	CPU	WorkingSet
chrome	1234	12.34	120,456,192
code	5678	1.12	98,345,984
explorer	9012	5.41	150,781,952
powershell	11234	0.95	75,643,904
...			

These are
**named
properties.**

Each row is an **object**.
Each column is a **property** with a value.

WHY IT MATTERS

- Filter by object data
- Sort and measure
- Automate with confidence

Because
objects carry
meaning—
PowerShell
can do more
for you!

THE BIG TAKEAWAY

OBJECTS CARRY DATA AND MEANING



Get-Process returns objects.
The pipeline keeps them rich.
You stay **in control**.



Remember: **Objects** > Text → More **power**. Less pain.

NEXT: Filtering Objects →

POWERSHELL 102: OBJECTS POWER THE PIPELINE



STEP 1: INSPECT AN OBJECT

Use **Get-Member** to see everything an object has: its **properties** (data) and **methods** (actions).

Get-Process | **Get-Member**

```

TypeName: System.Diagnostics.Process

Name      MemberType Definition
-----
Handles   Property    int Handles {get;}
Id         Property    int Id {get;}
ProcessName Property    string ProcessName {get;}
CPU        Property    double CPU {get;}
StartTime  Property    datetime StartTime {get;}
Kill       Method       void Kill()
Refresh    Method       void Refresh()
...
    
```



✧ ✧ This shows all the parts of the object.

Some are **PROPERTIES**
(the facts it knows).

Some are **METHODS**
(the things it can do).

PROPERTIES = WHAT IT KNOWS

Properties are pieces of **data** about the object.

Id = 1234

ProcessName = chrome

CPU = 1.25

StartTime = 9:15:03 AM

They describe the object.

METHODS = WHAT IT CAN DO

Methods are **actions** the object can perform.

Kill() → Stops the process

Refresh() → Updates the info

CloseMainWindow() → Closes the window

...

They do something.

STEP 2: CHOOSE WHAT TO SEE

Use **Select-Object** to pick the properties you want to see.

Get-Process | **Select-Object Name, Id, CPU**

Name	Id	CPU
chrome	1234	1.25
Code	5678	0.42
explorer	4321	0.18
powershell	678	0.05
...		



Now you're looking at just the info you care about!

Inspect to learn.
Select to focus.
That's how **objects** power your pipeline!



Inspect to understand → Select to focus → Act with confidence.

NEXT: Filter with Where-Object! →

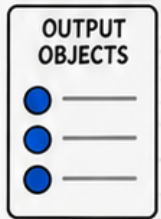
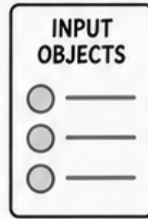
POWERSHELL 102:

OBJECTS POWER THE PIPELINE

FILTER WITH A CONDITION



Use **Where-Object** to filter objects. It keeps only the ones that match your condition.



Only objects where the condition is **TRUE** continue down the pipeline.



Inside the condition script block, **\$_** is the current object being checked.

THE CURRENT ITEM TOKEN

CURRENT ITEM

CONDITION EXAMPLE

{ \$_.CPU -gt 100 }

Check the CPU property of the current item.

EXAMPLE: FILTER PROCESSES USING CPU THRESHOLD

```
PS> Get-Process | Where-Object { $_.CPU -gt 100 }
```

ALL PROCESSES (INPUT)

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

PROCESSES WITH CPU > 100 (OUTPUT)

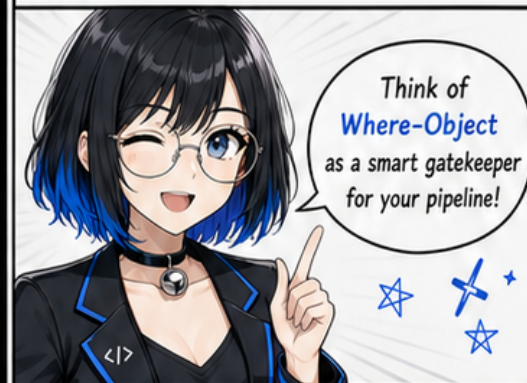
Id	Process Name	CPU
1234	Code	215.6
5678	Game	187.2
9012	BuildAgent	132.4

This keeps only processes whose CPU is **greater than 100**.



Id	Process Name	CPU
1234	Code	215.6
5678	Game	187.2
9012	BuildAgent	132.4

Each process object flows in. If **\$_.CPU -gt 100** is **TRUE**, it passes through. If **FALSE**, it stops here.



Think of **Where-Object** as a smart gatekeeper for your pipeline!

KEY TAKEAWAY

TRUE PASSES. FALSE STOPS.



Where-Object filters objects using your condition.



\$_ represents the current object in the pipeline.



Only objects where the condition is **TRUE** continue.



Remember: Objects in, filter, objects out. → Better filters → Better results.

NEXT: Sorting Objects →

OBJECTS POWER THE PIPELINE

OBJECTS
IN,
ANSWERS
OUT
</>

SORT, GROUP, MEASURE

Objects give you more than just data—they give you questions to answer.

Here are three powerful ways to understand a collection.

TODAY'S LESSON:

1. [Sort-Object CPU -Descending](#)
2. [Group-Object Status](#)
3. [Measure-Object](#)



CODE
CREATE
REPEAT

1) SORT-OBJECT CPU -DESCENDING

Sort objects by a property to quickly see what's at the top.

Get-Process |
[Sort-Object CPU -Descending](#)

BEFORE (Unsorted)		
Name	CPU	Status
Code	12.4	Running
chrome	48.7	Running
pwsh	2.1	Running
explorer	5.2	Running
dotnet	25.3	Running

AFTER (Sorted by CPU -Descending)		
Name	CPU	Status
chrome	48.7	Running
dotnet	25.3	Running
Code	12.4	Running
explorer	5.2	Running
pwsh	2.1	Running



Find the highest, lowest, or just see what matters most.

2) GROUP-OBJECT STATUS

Group by a property to see how items are categorized.

Get-Process |
[Group-Object Status](#)

BEFORE (Raw List)		
Name	CPU	Status
Code	12.4	Running
chrome	48.7	Running
pwsh	2.1	Running
svchost	1.3	Stopped
wslservice	0.7	Stopped
OneDrive	0.9	Running

AFTER (Grouped by Status)		
Count	Name	(Samples)
Running	4	Code, chrome, pwsh, OneDrive

Stopped	2	svchost, wslservice



Instantly see patterns and balance.

3) MEASURE-OBJECT

Get summary statistics about your collection.

Get-Process |
[Measure-Object CPU](#)

BEFORE (Raw List)		
Name	CPU	Status
Code	12.4	Running
chrome	48.7	Running
pwsh	2.1	Running
explorer	5.2	Running
dotnet	25.3	Running

AFTER (Summary)	
Count	: 5
Average	: 18.74
Sum	: 93.7
Maximum	: 48.7
Minimum	: 2.1
Property	: CPU



Summaries turn a long list into clear insight.



PowerShell gives you tools to explore, compare, and understand like a pro.

KEY TAKEAWAY

ASK BETTER QUESTIONS OF YOUR OBJECTS



SORT
to prioritize

+



GROUP
to find patterns

+



MEASURE
to summarize



REMEMBER: The pipeline is your microscope. Objects are the sample. You choose the lens.

NEXT: Filtering with [Where-Object](#) →

BUILD AN OBJECT PIPELINE

PAGE 5



THE FULL PIPELINE



`Get-Service | Where-Object { $_.Status -eq 'Running' } | Sort-Object DisplayName | Select-Object -First 10 Name, Status`

1 GET: Get all services.

```
PS> Get-Service
```

Status	Name	DisplayName
Running	WinDefend	Windows Defender Antivirus Service
Running	Spooler	Print Spooler
Stopped	WSearch	Windows Search
...		

We start by getting all services. This gives us many **service objects**.



2 FILTER: Keep only running services.

```
PS> Get-Service | Where-Object { $_.Status -eq 'Running' }
```

Status	Name	DisplayName
Running	WinDefend	Windows Defender Antivirus Service
Running	Spooler	Print Spooler
Running	LanmanServer	Server
...		

We filter to keep only the services that are **currently running**.



3 SORT: Sort by DisplayName.

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

Status	Name	DisplayName
Running	AppIDSvc	Application Identity
Running	BITS	Background Intelligent Transfer Service
Running	CeramicService	Windows Camera Frame Server
...		

Next, we sort alphabetically by **DisplayName** so it's easy to scan.



4 SELECT: Take the first 10 and show Name and Status.

```
PS> Get-Service | Where-Object { $_.Status -eq 'Running' } | Sort-Object DisplayName | Select-Object -First 10 Name, Status
```

Name	Status
AppIDSvc	Running
BITS	Running
CeramicService	Running
DcomLaunch	Running
... (up to 10 services total)	

Finally, we take the **first 10** results and show just the **Name** and **Status** properties.



Objects move through the **pipeline**, transformed at each step.

GET



FILTER



SORT



SELECT



GET → FILTER → SORT → SELECT

POWERSHELL 103:

FILES AND STRUCTURED DATA



Welcome to
PowerShell 103!

Today, we'll explore how PowerShell helps you work with **files** and **structured data** using **providers**.

Today's Plan:

1. What are Providers?
2. The FileSystem Provider
3. Navigate Like a Pro



1. WHAT ARE PROVIDERS?



Providers are PowerShell's way of exposing different data stores as drives.

Once a provider is available, you can use familiar **navigation commands** to work with its data.

Examples of Providers:

- FileSystem (working with files and folders)
- Registry (Windows Registry)
- SQLServer (databases)
- And more!

2. THE FILESYSTEM PROVIDER



The FileSystem provider lets you work with files and folders on your computer—just like you do in File Explorer!

Provider: FileSystem
Drives: C:, D:, \\server\share, ...

3. NAVIGATE LIKE A PRO

These PowerShell commands help you explore the FileSystem provider.

Get-PSDrive

Shows available drives from all providers.

```
PS> Get-PSDrive
```

Name	Used (GB)	Free (GB)	Provider	Root
C	120.52	379.48	FileSystem	C:\
D	200.11	299.89	FileSystem	D:\
Env			Environment	
HKCU			Registry	HKCU_CURRENT_USER
Temp			FileSystem	C:\temp\Elly\local\temp
WSMan			WSMan	
Alias			Alias	

➔ Look for **Provider** to see where the data comes from.

Get-Location

Shows your current location in the active provider.

```
PS> Get-Location
```

Path
C:\Users\Elly\Documents

C: > Users > Elly > Documents

➔ Think of it as "where am I right now?"

Get-ChildItem

Lists items in the current location.

```
PS> Get-ChildItem
```

Directory: C:\Users\Elly\Documents

Mode	LastWriteTime	Length	Name
d----	5/12/2025 9:15 AM		Projects
d----	5/10/2025 1:02 PM		Notes
-a---	5/12/2025 9:14 AM	1536	todo.txt
-a---	5/11/2025 4:47 PM	2048	budget.csv
-a---	5/10/2025 11:08 AM	4096	report.docx

➔ See folders (d----) and files (-a----) in your current location.

PUTTING IT TOGETHER



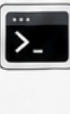
You can **move** around and **explore** just like in the terminal!

Get-PSDrive



See available drives

Set-Location C:\Data



Move to a location

Get-ChildItem



List items in the location

These same commands work with other **providers** too—not just files!



★ KEY TAKEAWAY

NAVIGATE DATA LIKE FOLDERS

PowerShell providers let you use familiar commands to explore and manage many kinds of data.



READ, WRITE, APPEND



1. READ A FILE

Use **Get-Content** to read what's inside a text file.



```
PS> Get-Content .\notes.txt
Line one
Line two
Line three
```

It shows the file's current contents.



2. WRITE (REPLACE) A FILE

Use **Set-Content** to write new content.

WARNING: It replaces everything in the file!

```
PS> Set-Content .\notes.txt 'Hello'
(No output)
```

```
notes.txt
Hello
```

Set-Content replaces the entire file.



3. APPEND TO A FILE

Use **Add-Content** to add text to the end of the file without removing what's there.

```
PS> Add-Content .\notes.txt 'Next line'
(No output)
```

```
notes.txt
Hello
Next line
```

Add-Content appends a new line to the end.



SET-CONTENT REPLACES.
ADD-CONTENT APPENDS.

Use **Set-Content** when you mean to **replace** everything.
Use **Add-Content** when you mean to **add** more.



TIP: Always double-check before using **Set-Content** on important files!

CSV ROWS BECOME OBJECTS

PAGE 3



CSV files are simple tables.
In PowerShell, each **row** becomes a custom **object**!

people.csv			
Name	Department	Title	Years
Ava	Engineering	Developer	4
Ben	Marketing	Analyst	3
Cleo	Engineering	DevOps	5
Diego	Sales	Rep	2
Eli	Engineering	QA	1



PowerShell Objects	
Name	: Ava
Department	: Engineering
Title	: Developer
Years	: 4
...	
Name	: Eli
Department	: Engineering
Title	: QA
Years	: 1

Each **row** becomes an **object** with properties you can use!

1 IMPORT: LOAD CSV INTO OBJECTS



Import the CSV.
Now **\$people** is a collection of objects.

```
PS> $people = Import-Csv .\people.csv
```

\$people

Name	Department	Title	Years
Ava	Engineering	Developer	4
Ben	Marketing	Analyst	3
Cleo	Engineering	DevOps	5
Diego	Sales	Rep	2
Eli	Engineering	QA	1



people.csv



2 FILTER: SELECT WHAT YOU NEED

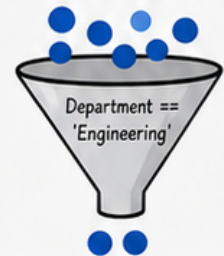


Filter like a pro.
This keeps only **Engineering** team members.

```
PS> $people | Where-Object { $_.Department -eq 'Engineering' }
```

Filtered Results

Name	Department	Title	Years
Ava	Engineering	Developer	4
Cleo	Engineering	DevOps	5
Eli	Engineering	QA	1



3 EXPORT: SAVE THE RESULTS



Export the filtered objects back to a CSV.

```
PS> $people | Where-Object { $_.Department -eq 'Engineering' } |  
Export-Csv .\engineering.csv -NoTypeInformation
```

Filtered Results

Name	Department	Title	Years
Ava	Engineering	Developer	4
Cleo	Engineering	DevOps	5
Eli	Engineering	QA	1



engineering.csv

Name	Department	Title	Years
Ava	Engineering	Developer	4
Cleo	Engineering	DevOps	5
Eli	Engineering	QA	1

-NoTypeInformation keeps the file clean (without extra PowerShell metadata). ★

KEY TAKEAWAY

CSV is more than text—it's **data** you can work with.

IMPORT



FILTER



EXPORT



Bring CSV in as objects.



Use PowerShell commands to find what you need.



Save clean results to a new CSV.

Small steps,
powerful results.

That's the **PowerShell** way! ★



Remember: **Data in** → **Insight out.**

NEXT: JSON in PowerShell! →

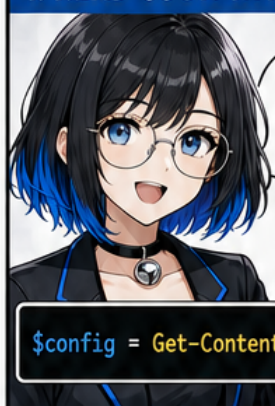
JSON BECOMES AN OBJECT



JSON (JavaScript Object Notation) is a popular format for configuration and data exchange. In PowerShell, JSON becomes an **object** you can use and update!

POWERSHELL 103:
FILES AND STRUCTURED DATA ★

1. READ JSON INTO AN OBJECT



Read the JSON file and convert it into a PowerShell object.

```
config.json
{
  "Theme": "Dark",
  "Window": {
    "Width": 1200,
    "Height": 800
  },
  "Features": ["Search", "Export"]
}
```

```
$config = Get-Content .\config.json -Raw | ConvertFrom-Json
```

2. ACCESS DATA EASILY



Once it's an object, you can access properties using dot notation.

```
PS> $config.Theme
Dark
```

You can also explore nested values, like `$config.Window.Width` or `$config.Features[0]`.

3. CHANGE THE OBJECT



Update any values in the object just like working with a hashtable or custom object.

```
$config.Theme = "Light"
$config.Window.Width = 1400
$config.Features += "Logs"
```

```
{
  "Theme": "Light",
  "Window": {
    "Width": 1400,
    "Height": 800
  },
  "Features": ["Search", "Export", "Logs"]
}
```

4. SAVE BACK TO JSON



Convert the object back to JSON and write it to the file.

```
$config | ConvertTo-Json | Set-Content .\config.json
```



config.json

Your changes are saved!

KEY TAKEAWAY

PARSE → USE → SAVE



Read JSON into an object.



Access and change what you need.



Convert and save back to JSON.



JSON makes configuration powerful, flexible, and easy to manage! ★



Remember: JSON in → Object → Power → JSON out

NEXT: CSV LIKE A PRO →

POWERSHELL 103: FILES AND STRUCTURED DATA

PAGE 5

BUILD A PROCESS REPORT



Let's build a useful process report.

We'll **collect** the data, **shape** it, **save** it to CSV, and then **verify** it.



1) COLLECT, SHAPE, AND SAVE

This one-liner gets processes, keeps the columns we want, sorts by CPU (high to low), and saves a **clean CSV**.



```
PS> Get-Process | Select-Object Name, Id, CPU, WorkingSet |
Sort-Object CPU -Descending |
Export-Csv .\process-report.csv -NoTypeInformation
```

Get-Process
Collect the process data.

Select-Object
Keep only the columns we need.

Sort-Object CPU -Descending
Sort from highest CPU to lowest.

Export-Csv ... -NoTypeInformation
Save to CSV without type information.

2) THE REPORT (process-report.csv)

Here's what the CSV looks like when we open it.



Name	Id	CPU	WorkingSet
PowerShell	12348	152.34	128450560
devenv	21560	87.61	512450560
chrome	18876	64.21	942080768
MsMpEng	4324	23.44	325058560
explorer	6840	15.36	205520896

Clean, focused, and **sorted** by CPU usage.

3) VERIFY THE FILE

Always **verify** what you write. We'll read the CSV back and preview the **first 5 rows**.



```
PS> Import-Csv .\process-report.csv |
Select-Object -First 5
```

Name	Id	CPU	WorkingSet
PowerShell	12348	152.34	128450560
devenv	21560	87.61	512450560
chrome	18876	64.21	942080768
MsMpEng	4324	23.44	325058560
explorer	6840	15.36	205520896



Looks good! The file saved correctly and the data is **exactly** what we need.

COLLECT



SHAPE



SAVE



VERIFY



Reliable Reports!



COLLECT → SHAPE → SAVE → VERIFY

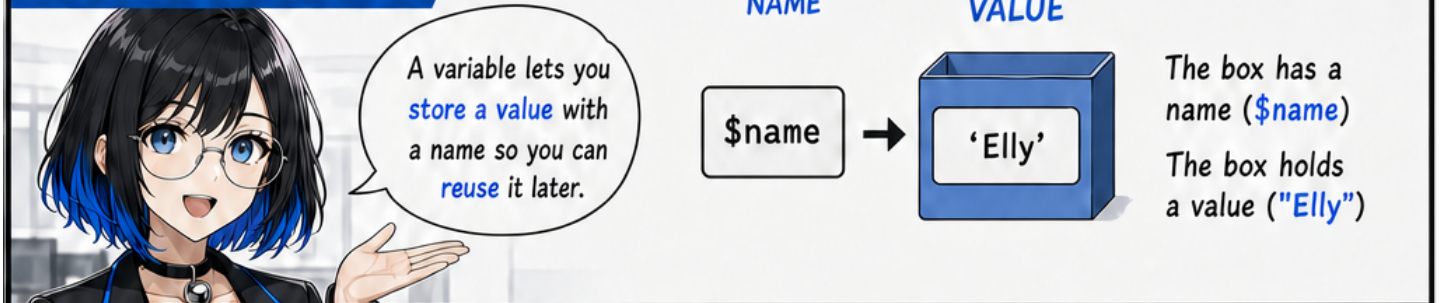
POWERSHELL 104:

VARIABLES, DECISIONS, LOOPS

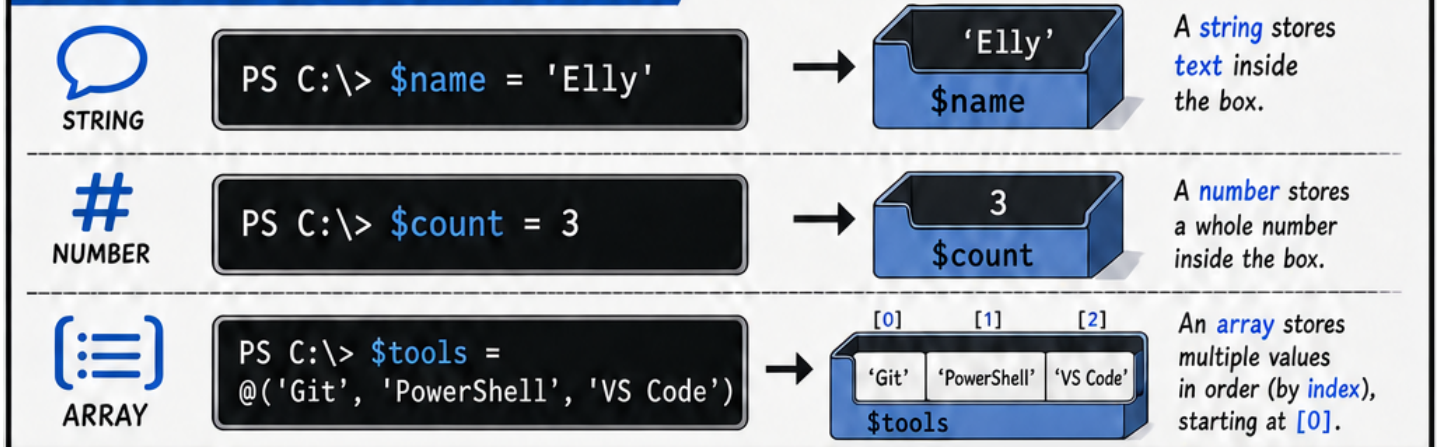
PAGE 1



WHAT IS A VARIABLE?



EXAMPLES: STRINGS, NUMBERS, ARRAYS



POWERHELL 104:

VARIABLES, DECISIONS, LOOPS

MAKE A DECISION WITH IF

COMPARISON OPERATORS

-eq	EQUAL	"A" -eq "A" → True
-ne	NOT EQUAL	5 -ne 3 → True
-gt	GREATER	10 -gt 3 → True
-lt	LESS	3 -lt 10 → True
-ge	AT LEAST	10 -ge 10 → True
-le	AT MOST	8 -le 10 → True

In PowerShell, we use **comparisons** in conditions to make decisions.

IF, ELSEIF, ELSE

Use **if** to test a condition.
Use **elseif** for more checks.
Use **else** for anything else.

```
if (condition) {  
    # do something  
}  
elseif (another condition) {  
    # do something else  
}  
else {  
    # default action  
}
```

Run this block if the first condition is **True**.

If the first is not **True**, check this condition.

If none are **True**, run this block.

EXAMPLE: CHECK DISK SPACE

Let's check free disk space (simplified example).

We have:
\$freeGB = 8

```
$freeGB = 8  
if ($freeGB -lt 10) {  
    Write-Host "⚠ Low disk space: $freeGB GB free"  
}  
elseif ($freeGB -lt 20) {  
    Write-Host "OK: $freeGB GB free"  
}  
else {  
    Write-Host "Plenty of space: $freeGB GB free"  
}
```

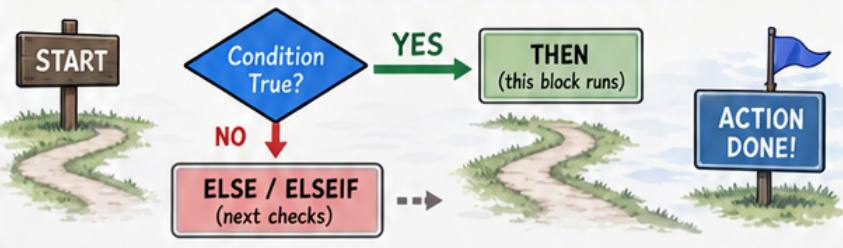
```
PS C:\> .\Check-Disk.ps1  
⚠ Low disk space: 8 GB free
```

Since **\$freeGB = 8**, the first condition (**\$freeGB -lt 10**) is **True**!

This is the output we get.

A CONDITION CHOOSES THE PATH

Use **comparisons** in conditions to guide your scripts and make smart choices!



★ Remember: Comparisons test values. → Conditions choose the path. → Better scripts! ★

POWERSHELL 104:

VARIABLES, DECISIONS, LOOPS

TODAY'S TOPIC:

REPEAT FOR EACH ITEM

Sometimes you need to run the same action for every item in a list.

In PowerShell, use **foreach**!

BUILD
THINK
ITERATE
</>

CODE
CREATE
REPEAT

WHAT IS FOREACH?

foreach loops through each **item** in a collection. For each item, it runs the code inside the braces, then moves to the next.



Repeat for each item



Clear and safe



Easy to read

OUR EXAMPLE LIST

Here's our list of servers:

```
$servers = @('web-01', 'web-02', 'db-01')
```



web-01



web-02



db-01

Stored in a variable called **\$servers**.

FOREACH SYNTAX

```
1 foreach ($server in $servers) {
2     Write-Output "Checking $server"
3 }
```

CURRENT ITEM

\$server holds one item from **\$servers** for this pass.

BODY

The code inside the braces runs once for the current item.

NEXT ITEM

When done, PowerShell moves to the next item and repeats.

WHAT HAPPENS?



SEE IT RUN

```
PS> $servers = @('web-01', 'web-02', 'db-01')
foreach ($server in $servers) {
    Write-Output "Checking $server"
}
```

Checking web-01
Checking web-02
Checking db-01

Clean,
readable,
and easy
to trust!

QUICK TIPS



Perfect for **lists** of names, files, servers, or any items.



Use **clear variable names** so your code is easy to read.



foreach is safe and predictable—it doesn't skip or duplicate items.

KEY TAKEAWAY



ONE LIST. ONE ACTION. EVERY ITEM.

foreach helps you repeat work for each item in a list—clearly, safely, and powerfully.

POWERSHELL 104: VARIABLES, DECISIONS, LOOPS FOR COUNTS. WHILE WAITS.

Two powerful loops in PowerShell:
FOR counts ahead.
WHILE waits for a condition to change.

TODAY'S TOPIC:
FOR VS. WHILE

FOR LOOP: COUNTED

Use a **FOR** loop when you know how many times to run.

PowerShell

```
for ($i = 1; $i -le 3; $i++) {  
}
```

↑ start at 1 ↑ keep going while \$i is less than or equal to 3 ↑ add 1 each time

OUTPUT:

1
2
3

Counts from 1 to 3, then stops. Exactly 3 times.

WHILE LOOP: CONDITION-BASED

Use a **WHILE** loop when you don't know how many times in advance—just when to stop.

PowerShell

```
$n = 3  
while ($n -gt 0) {  
  Write-Output $n  
  $n--  
}
```

← start the counter ← keep going while \$n is greater than 0 ← subtract 1 each time

OUTPUT:

3
2
1

Starts at 3 and counts down to 1. Stops when the condition is false.



MAKE SURE WHILE CAN STOP

GOOD: WILL STOP

```
$n = 3  
while ($n -gt 0) {  
  Write-Output $n  
  $n--  
}
```



\$n gets smaller and will eventually be 0. The loop stops.

RISKY: MAY NEVER STOP

```
$n = 3  
while ($n -gt 0) {  
  Write-Output $n  
  # Oops! Forgot to change $n  
}
```



\$n never changes. Condition stays TRUE. This loop will never stop!

TAKEAWAY

CHOOSE THE LOOP THAT MATCHES THE JOB



USE FOR WHEN YOU KNOW:

- ✓ How many times to run
- ✓ There is a clear start, stop, and step
- ✓ You want a counted loop



USE WHILE WHEN YOU KNOW:

- ✓ The condition to continue
- ✓ The loop should stop when a condition becomes false
- ✓ You don't know how many times

Pick **FOR** for counts.
Pick **WHILE** for conditions.

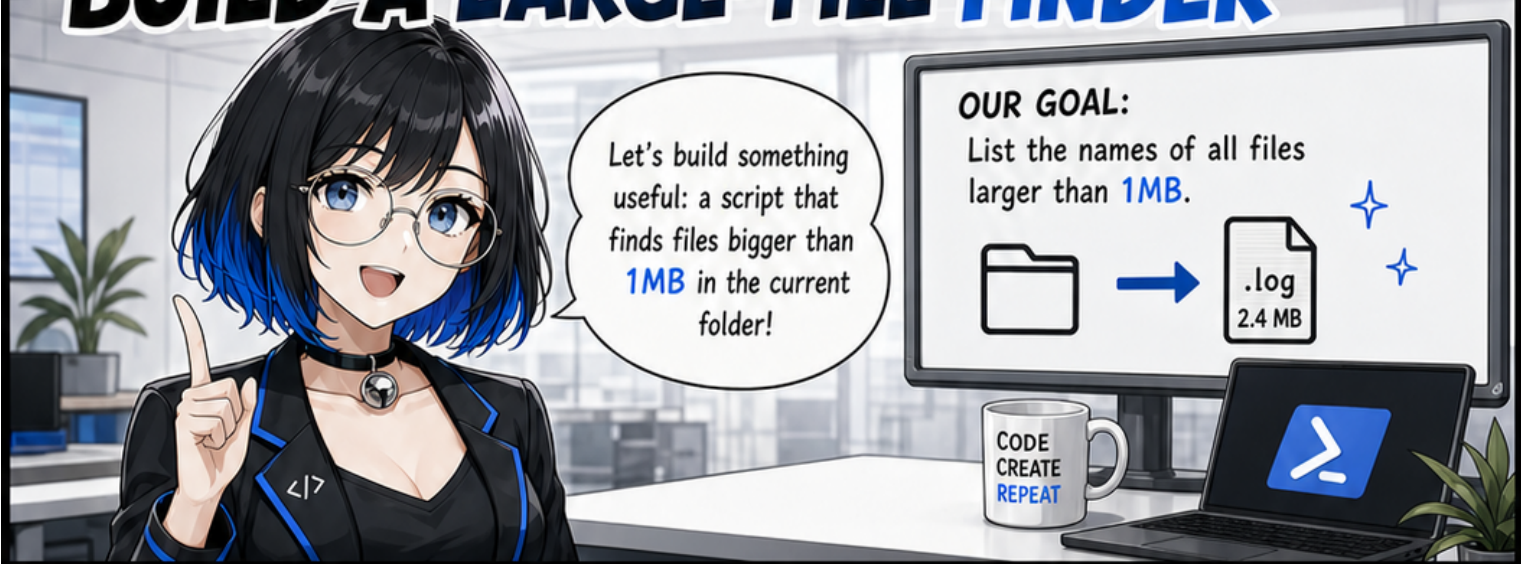


Remember: **FOR** counts. **WHILE** waits. Make sure **WHILE** can stop!

NEXT: Putting Loops to Work! →

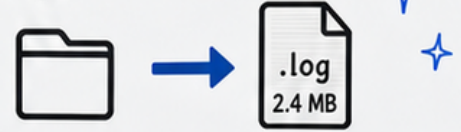
BUILD A LARGE-FILE FINDER

PAGE 5



OUR GOAL:

List the names of all files larger than **1MB**.



THE SCRIPT (ONE VARIABLE, ONE LOOP, ONE IF CONDITION)

```
1 $limit = 1MB
2 foreach ($file in Get-ChildItem . -File) {
3     if ($file.Length -gt $limit) { $file.Name }
4 }
```



HOW IT WORKS (STEP BY STEP)

1 VARIABLE

```
$limit = 1MB
```

Set the size limit. Anything bigger than 1MB is a match.

2 LOOP

```
foreach ($file in
Get-ChildItem . -File) {
```

Look at every file in the current folder, one by one.

3 CONDITION

```
if ($file.Length -gt $limit) {
```

If the file's size (**Length**) is greater than **\$limit**...

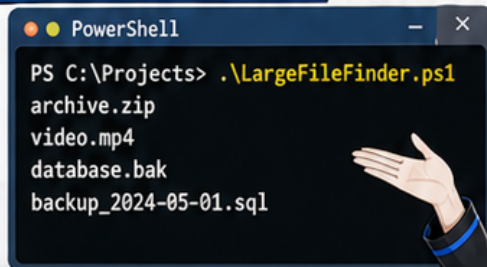
4 RESULT

```
$file.Name
```

...show the file's name.



EXAMPLE OUTPUT



TIPS

- ✓ Change the limit easily:
`$limit = 5MB`
- ✓ Search subfolders too:
`Get-ChildItem . -Recurse -File`
- ✓ Sort by size (largest first):
... | `Sort-Object Length -Descending`

CONTROL FLOW RECAP



VARIABLE
Set the rule.



LOOP
Go through each item.



CONDITION
Check if it meets the rule.



RESULT
Show the matches.



**VARIABLE → LOOP →
CONDITION → RESULT**



Think in flow. Build with **PowerShell**.

NEXT: Putting it all together! →

POWERSHELL 105:

SCRIPTS, FUNCTIONS, PARAMETERS



BUILD
THINK
AUTOMATE

>

Hi! I'm **Elly**.
Today we'll learn how
to turn a few tested
commands into a
reusable **PowerShell**
script.

TODAY'S TOPIC:
Turn your tested
commands into a
reusable **.ps1** script!

★ ★

CODE
CREATE
REPEAT

>

1. WRITE – START WITH WORKING COMMANDS



Start with a tiny
task. These two
commands show
a **greeting**.

hello.ps1

```
1 $name = 'Elly'  
2 Write-Output "Hello, $name!"
```

💡 A script is just PowerShell commands
saved in a **.ps1** file.

2. SAVE – GIVE IT A NAME



Save it as
hello.ps1.
The **.ps1** extension
tells PowerShell
it's a script.



hello.ps1

3. RUN – EXECUTE YOUR SCRIPT



Run it from the
same folder with:
.\hello.ps1

```
PS C:\Scripts> .\hello.ps1  
  
Hello, Elly!  
  
PS C:\Scripts> █
```

> WHAT HAPPENED?
PowerShell read
hello.ps1 and ran
the commands
inside it.

4. IMPROVE – EDIT, TEST, REPEAT



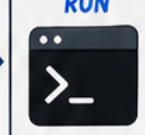
Improve your script
over time.
Edit, test, and make
it **better**!

EDIT



Make changes
in **hello.ps1**.

RUN



Run **.\hello.ps1**
to test it.

IMPROVE



Refactor, add features,
and keep **improving**.





Turn good steps
into a script,
then **reuse** it
anytime!

KEY TAKEAWAY

SAVE GOOD STEPS AS A **SCRIPT**

★

THE SCRIPT JOURNEY:  **WRITE** →  **SAVE** →  **RUN** →  **IMPROVE**

NEXT:
FUNCTIONS
IN ACTION!

→

POWERSHELL 105: SCRIPTS, FUNCTIONS, PARAMETERS

BUILD
THINK
ITERATE
</>

Hard-coded values lock your script to one thing. Parameters make it flexible.

Let's replace a hard-coded name with a parameter.

FROM HARD-CODED...

```
Write-Output "Hello, Sam!"
```

...TO PARAMETERIZED!

```
param([Parameter(Mandatory)][string]$Name)  
Write-Output "Hello, $Name!"
```

Now the name can change when we run the script.

THE PARAM BLOCK AT THE TOP OF YOUR SCRIPT

```
1 param([Parameter(Mandatory)][string]$Name)  
2  
3 Write-Output "Hello, $Name!"
```

NAME

\$Name is the parameter name. It's a placeholder for the value we'll provide.

T TYPE

[string] means this parameter expects text.

! MANDATORY

Mandatory means you must provide a value when running the script.

PERSON SUPPLIED VALUE

When you run the script, you'll supply the value for -Name. The script uses that value.

This line tells PowerShell what input your script needs.

RUNNING THE SCRIPT WITH A VALUE

We pass a value for -Name when we run the script.

Run this in PowerShell:

```
PS C:\Scripts> .\hello.ps1 -Name 'Sam'
```

Output:

```
Hello, Sam!
```

WHAT HAPPENED?

- ☒ -Name tells the script what to use for \$Name.
- ☒ The script uses that value in the message.
- ☒ Change the value, change the output!

WHY THIS MATTERS



No more hard-coded values.



Easier to reuse your script in different situations.



Flexible, clean, and professional scripts!

KEY TAKEAWAY

INPUTS CHANGE. THE SCRIPT STAYS.

Use parameters to make your scripts powerful and reusable!

Remember: Parameters make scripts flexible → Reusable → Professional

NEXT: Functions = Reusable Blocks! →

FUNCTIONS

PACKAGE REUSABLE WORK



Functions are **named**, **reusable** blocks of code. Define them once, and **call** them whenever you need them!

WHY FUNCTIONS?

-  Avoid repetition
-  Make scripts easier to read
-  Simplify updates
-  Boost productivity!

CODE
CREATE
REPEAT

DEFINE ONCE...

Create a function with a clear, Verb-Noun name.

```
> function Get-Greeting {
    param(
        [Parameter(Mandatory=$true)]
        [string]$Name
    )
    "Hello, $Name!"
}
```

Get-Greeting takes a parameter called **Name** and returns a greeting.

...CALL MANY TIMES

Use the function wherever you need it.

```
PS> Get-Greeting -Name 'Sam'
Hello, Sam!
```

```
PS> Get-Greeting -Name 'Ava'
Hello, Ava!
```

Different input, same logic—**no copy-paste!**

HOW IT WORKS

1 DEFINE THE FUNCTION



Write the logic once and give it a name.

2 CALL WITH INPUT

```
Get-Greeting -Name 'Sam'
Get-Greeting -Name 'Ava'
```

Pass different values when you call it.

3 GET RESULTS

```
Hello, Sam!
Hello, Ava!
```

The function handles the work and returns the result.

Change the function, improve it **everywhere!**

PRO TIP: USE APPROVED NAMES

Follow Verb-Noun for function names. It makes your code clear and consistent.

Get-Greeting

↑ ↑
Verb Noun



KEY TAKEAWAY

DEFINE ONCE.
CALL WHEN NEEDED.



Functions turn repetition into **reusable power!**★



Remember: Write once → Reuse everywhere → Save time

NEXT: Output and formatting! →

POWERSHELL 105:
SCRIPTS, FUNCTIONS,
PARAMETERS

MAKE CODE
EASY TO TRUST

PAGE 4

Great code isn't just working code—it's **trustworthy** code.

People (and future you) should be able to read it and know what it does, how to use it, and what to expect.

YOUR TRUST CHECKLIST

☒

CLEAR NAMES

☒

SAFE DEFAULTS

☒

SMALL OUTPUTS

☒

TEST EXAMPLES

★

CODE
CREATE
REPEAT

</>

MYSTERY SCRIPT vs SELF-EXPLAINING SCRIPT

MYSTERY SCRIPT

```
1 param($p,$d=7)
2 $x = Get-Process
3 $y = $x | Where-Object {$_.CPU -gt $p}
4 $y | Select-Object Name,CPU
5 Start-Sleep -Seconds $d
6 $y.Count
```

What is \$p?
What is \$d?
Why these steps?
What's the result?

SELF-EXPLAINING SCRIPT

```
1 ## Get-HighCpuProcesses.ps1
2 . DESCRIPTION
3 Returns processes using more than a CPU% threshold.
4 . PARAMETER MinCpu
5 Minimum CPU percent (default: 10)
6 . EXAMPLE
7 .\Get-HighCpuProcesses.ps1 -MinCpu 15 -Top 5
8
9 param(
10     [int]$MinCpu = 10,
11     [int]$Top = 5
12 )
13 Get-Process | Where-Object CPU -ge $MinCpu |
14 Sort-Object CPU -Descending |
15 Select-Object -First $Top Name, CPU
```

Now I know what it does, what I can change, and what I'll get!

1. CLEAR NAMES

Use names that say what they mean.

MinCpu is better than \$p!

HELLO MY NAME IS

MinCpu ✓
Top ✓
Path ✓
\$p, \$x, \$d ✗

2. SAFE DEFAULTS

Pick sensible defaults so the script works safely out of the box.

Defaults should be helpful, not risky.

Example: MinCpu = 10
Top = 5

3. SMALL OUTPUTS

Return only what's needed. Small outputs are easier to read and use.

Less noise, more signal!

4. TEST EXAMPLES

Include examples and test with sample values.

Try it with a few values. Trust grows with evidence!

TEST IT! EXAMPLES BUILD CONFIDENCE

Run a few examples and check the results.

```
PS> .\Get-HighCpuProcesses.ps1 -MinCpu 15 -Top 3
Name      CPU
----
chrome    27.1
Code      18.4
Teams     16.2
PS> .\Get-HighCpuProcesses.ps1
```

Different values, predictable results. That's trustworthy.

QUICK TIP

Add comment-based help (.DESCRIPTION, .PARAMETER, .EXAMPLE) so others can use your script without having to ask you.

CODE
CREATE
REPEAT

★ READABLE CODE IS MAINTAINABLE CODE ☆

BUILD *Get-Large File*



STEP 1: CREATE THE FUNCTION



```
function Get-LargeFile {  
    param([string]$Path = '.', [long]$MinimumBytes = 1MB)  
    Get-ChildItem $Path -File |  
    Where-Object Length -gt $MinimumBytes  
}
```

PARAMETERS

- Path: folder to search (default '.')
- MinimumBytes: minimum size in bytes (default 1MB)

LOGIC

Get-ChildItem lists files. Where-Object filters files where Length is greater than MinimumBytes.

STEP 2: CALL THE FUNCTION



```
PS C:\> Get-LargeFile -Path . -MinimumBytes 5MB
```

Mode	LastWriteTime	Length	Name
-a----	5/01/2025 9:42 AM	12.45 MB	backup.zip
-a----	4/28/2025 3:17 PM	8.31 MB	data-export.csv
-a----	4/27/2025 11:05 AM	6.72 MB	video.mp4

STEP 3: REUSE IT ANYWHERE

Different folders?
Change -Path.

Different size threshold?
Change -MinimumBytes.

Same logic, anywhere you need it!

One function. Many uses. Less typing. More power!

EXAMPLES

```
Get-LargeFile -Path 'D:\Projects' -MinimumBytes 100MB  
Find files larger than 100MB in D:\Projects
```

```
Get-LargeFile -MinimumBytes 50MB  
Find files larger than 50MB in the current folder
```

```
Get-LargeFile -Path 'C:\Logs' -MinimumBytes 10MB  
Find files larger than 10MB in C:\Logs
```

PARAMETERS → FUNCTION → REUSABLE TOOL



POWERSHELL 106: ERRORS, SAFETY, DEBUGGING

BUILD
THINK
ITERATE
</>

Errors aren't the enemy—
they're *useful evidence*.

PowerShell tells you what
went wrong and often how
to fix it. Let's learn how
to read it.

EXAMPLE: A HARMLESS MISTAKE

I'll run a command
with a small typo.

Watch what
PowerShell shows us.

MESSAGE

CATEGORY

LINE

COMMAND

```
PS C:\Demo> Get-Proces
Get-Proces : The term 'Get-Proces' is not recognized as the
name of a cmdlet, function, script file, or operable program.
Check the spelling of the name, or if a path was included,
verify that the path is correct and try again.
At line:1 char:1
+ Get-Proces
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Get-Proces:String)
  [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

PowerShell gives us four key pieces of information.
Let's break them down.

1 COMMAND

The command you ran.
Start here.

```
PS C:\Demo> Get-Proces
```

Check spelling and
every character.

2 MESSAGE

What PowerShell is telling you.

```
The term 'Get-Proces' is
not recognized as the name
of a cmdlet, function, script
file, or operable program.
```

Read this carefully—
it often gives the clue
for the fix.

3 CATEGORY

The type of problem.

```
+ CategoryInfo
: ObjectNotFound:
(Get-Proces:String)
[], CommandNotFoundException
```

This one means PowerShell
can't find a command with
that name.

4 LINE

Where the error happened.

```
At line:1 char:1
+ Get-Proces
+ ~~~~~
```

Tells you the exact line
and position to look at.

FOLLOW THE CLUES

- ✓ The message says the term
isn't recognized.
- ✓ Category: CommandNotFoundException
(confirms it's not a real command).
- ✓ The command looks almost right...
Get-Proces is probably
meant to be *Get-Process*.
- ✓ So the fix is likely a spelling
correction, not a big rewrite.

MAKE A SMALL, SAFE CHANGE

Fix the spelling
and try again.

Before (error):

```
PS C:\Demo> Get-Proces ❌
```

After (success):

```
PS C:\Demo> Get-Process ✅
```

One small change.
Big difference. ★

★ KEY TAKEAWAY

READ THE ERROR BEFORE CHANGING THE CODE



Read the command



Read the message



Check the category



1:1 Note the line



Understand the problem.
Make the right fix.

DECIDE HOW ERRORS BEHAVE



Not all PowerShell errors are equal. Some are **non-terminating**.

They show an error message, but your script **keeps running**.

EXAMPLE: NON-TERMINATING ERROR

```
PS> Get-Content .\missing.txt
Get-Content : Cannot find path
'.\missing.txt' because it
does not exist.
PS> Write-Host "Still running"
Still running
```

The command failed, but the script kept going.

MAKE FAILURES CATCHABLE



Add **-ErrorAction Stop** to convert that failure into a terminating error.

Now **try/catch** can **catch** it.

USING -ERRORACTION STOP

```
PS> Get-Content .\missing.txt -ErrorAction Stop
```

```
try {
    Get-Content .\missing.txt -ErrorAction Stop
    "This will not print if it fails."
}
catch {
    "Caught it! File is missing."
}
```

With **-ErrorAction Stop**, the error jumps to **catch**.

THE BIG PICTURE

PowerShell has a preference for how it treats errors. By default, many cmdlets use:

```
$ErrorActionPreference = 'Continue'
```



Continue
(default)
Non-terminating:
show error and
keep going.



Stop
Terminating:
stop and allow
try/catch to
handle it.

SET IT FOR YOUR SCRIPT (OPTIONAL)

```
$ErrorActionPreference = 'Stop'
```

Now most errors behave like **-ErrorAction Stop** without adding it to every command.

Use this at the top of a script when you need safety by default.



You can still override per command with **-ErrorAction Continue** or **-ErrorAction Stop**.

ELLY'S SUMMARY



- ✓ Some errors are non-terminating.
- ✓ **-ErrorAction Stop** turns a failure into a terminating error that **try/catch** can catch.
- ✓ **\$ErrorActionPreference** controls the default behavior.

KEY TAKEAWAY

USE STOP WHEN FAILURE MUST STOP THE STEP



Fail fast. Catch it. Handle it.
That's safer, cleaner scripts.

★ **USE STOP WHEN FAILURE MUST STOP THE STEP**



POWERSHELL 106:

ERRORS, SAFETY, DEBUGGING

TRY. CATCH. FINALLY.

When things can go wrong, use **try, catch, finally**. It keeps your scripts safe, clear, and easy to debug!

THE BIG IDEA

try → do the work.
catch → handle failure and inspect `$_`.
finally → always run cleanup.



1 TRY: DO THE WORK



Put the risky operation in **try { ... }**. If it works, we continue as normal.

try { ... }

```
try {
    # risky work here
    Get-Content 'data.txt' -Raw
    -ErrorAction Stop
    Write-Host "File read!"
}
```



If everything works, this block completes.

EXAMPLE: READ A FILE SAFELY

```
try {
    $content = Get-Content 'data.txt' -Raw
    -ErrorAction Stop
    Write-Host "Read success!"
}
catch {
    Write-Warning $_.Exception.Message
}
finally {
    Write-Host "Cleanup: closing stuff."
}
```

2 CATCH: HANDLE FAILURE



If something fails in **try**, control jumps to **catch { ... }**. Use `$_` to inspect the error.

catch { ... }

```
catch {
    # $_ is the current error
    Write-Warning $_.Exception.Message
}
```



Catch can log it, show a friendly message, or take alternative action.

SUCCESS PATH



FAILURE PATH



3 FINALLY: ALWAYS CLEAN UP



Finally runs no matter what. Use it to clean up resources and leave things in a good state.

finally { ... }

```
finally {
    # runs on success or failure
    Write-Host "Cleanup: closing stuff."
}
```



Close files, connections, timers, handles—always.

KEY TAKEAWAY

These three blocks are a powerful pattern for reliable PowerShell scripts!



TRY THE WORK.



CATCH THE FAILURE.



ALWAYS CLEAN UP.



Safe scripts are smart scripts!



Remember: Anticipate errors → Handle them → Leave things clean

NEXT: Debug with Write-Host →

PREVIEW BEFORE YOU CHANGE



Before you make changes, **preview** them, get **approval**, **narrow** your target, **verify** the result, and keep **backups**. That's how you stay safe and build trust.

SAFE CHANGE CHECKLIST

- ✓ **PREVIEW**
See what would happen.
- ✓ **CONFIRM**
Ask before making the change.
- ✓ **NARROW**
Target only what you mean.
- ✓ **VERIFY**
Check the result.

1) PREVIEW WITH -WhatIf



Use **-WhatIf** to see what PowerShell would do—without actually doing it.

EXAMPLE: PREVIEW A SAFE DELETE

Remove-Item .\temp -Recurse -WhatIf

```
PS C:\Demo> Remove-Item .\temp -Recurse -WhatIf
What if: Performing the operation "Remove Directory" on target
"C:\Demo\temp\logs".
What if: Performing the operation "Remove File" on target
"C:\Demo\temp\logs\old.log".
What if: Performing the operation "Remove Directory" on target
"C:\Demo\temp\cache".
PS C:\Demo>
```

WHATIF SHOWS THE ACTION WITHOUT DOING IT 

Your files and folders stay exactly as they are.

2) CONFIRM WITH -Confirm



Add **-Confirm** to ask before each change is made.

```
PS C:\Demo> Remove-Item .\temp -Recurse -Confirm
Confirm
Are you sure you want to perform this action?
Performing the operation "Remove Directory"
on target "C:\Demo\temp\logs".
[Y] Yes [A] Yes to All [N] No
[L] No to All [?] Help (default is "N"): _
```

3) NARROW THE TARGET

Be specific! Target only what you mean—use paths, filters, and **-Include** / **-Exclude** when needed.

 **.\temp\logs**
(only logs) ✓

 **.\temp\cache**
(skip cache) ✗

```
Remove-Item .\temp\logs -Recurse -WhatIf
Get-ChildItem .\temp\*.log -WhatIf
Remove-Item .\temp -Recurse `
-Exclude cache -WhatIf
```

4) VERIFY THE RESULT



After the change, verify that the result matches what you expected.

```
PS C:\Demo> Get-ChildItem .\temp

Directory: C:\Demo\temp

Mode                LastWriteTime         Length Name
----                -
d-----          5/20/2025  9:15 AM              cache
d-----          5/20/2025  9:15 AM              logs
```

Check the output. Trust, but verify! 

5) KEEP BACKUPS



If it's important, make a backup first. That's your safety net.

QUICK BACKUP EXAMPLE

```
# Backup before making changes
Copy-Item .\temp .\temp_backup -Recurse
Get-ChildItem .\temp_backup

Directory: C:\Demo\temp_backup

Mode                LastWriteTime         Length Name
----                -
d-----          5/20/2025  9:00 AM              cache
d-----          5/20/2025  9:00 AM              logs
```



Preview.
Confirm.
Narrow.
Verify.
Backup.

Safe changes,
confident results!

POWERSHELL 106:

ERRORS, SAFETY, DEBUGGING

SAFE
SCRIPTS
SAVE
TIME
>_



Before deleting anything, let's build a safe cleanup preview. We'll find old files and use `-WhatIf` so nothing is actually removed.

TODAY'S LESSON:

BUILD A SAFE CLEANUP PREVIEW

THINK BEFORE YOU DELETE

1. SET INPUTS

Path `$Path`

Older Than (Days) `$OlderThanDays`

```
# Example inputs
$Path = "C:\Temp"
$OlderThanDays = 30
```

We'll search a folder (`$Path`) for files older than a number of days (`$OlderThanDays`).



2. CALCULATE CUTOFF DATE

Anything older than this will be a match.

```
# Compute the cutoff date
$cutoff = (Get-Date).AddDays(-$OlderThanDays)
```

This date is the boundary. Files with a `LastWriteTime` earlier than `$cutoff` are considered old.

3. FIND OLD FILES

Filter by `LastWriteTime`

```
# Find candidate files
Get-ChildItem $Path -File |
Where-Object LastWriteTime -lt $cutoff
```

We list files in `$Path` and keep only those whose `LastWriteTime` is earlier than `$cutoff`.



4. PREVIEW DELETION

`-WhatIf`
= Preview Only (No Changes)

```
# Preview what would be deleted
ForEach-Object { Remove-Item $_.FullName -WhatIf }
```

This shows what would be deleted—but *doesn't delete anything*. It's the *safest* way to preview actions.

EXAMPLE PREVIEW OUTPUT (NOTHING IS DELETED)

What if: Performing the operation "Remove File" on target "C:\Temp\old.log".
What if: Performing the operation "Remove File" on target "C:\Temp\debug.txt".
What if: Performing the operation "Remove File" on target "C:\Temp\old_report.csv".
...

Read every line. If something looks wrong, *stop* and *adjust* before you approve.



REVIEW BEFORE ANY REAL CHANGE

- ✓ Is the `$Path` correct?
- ✓ Is `$OlderThanDays` the right value?
- ✓ Does the preview list what you expect?
- ✓ Could something important be affected?
- ✓ If unsure—don't run it for real.



THE SAFE CLEANUP MINDSET



Remember: *Preview* first. *Review* carefully. *Approve* only when you're sure.

FIND → PREVIEW → REVIEW → APPROVE

POWERSHELL 107:

BUILD A DAILY SYSTEM REPORT



This week's capstone:
we'll build a **safe**, **reusable** script
that captures a daily system
snapshot you can **trust**.

Our script:
New-SystemReport.ps1



WHAT OUR SCRIPT CREATES

Each run produces two useful files
in the output folder.

system-summary.json

A JSON summary with
key system facts.

top-processes.csv

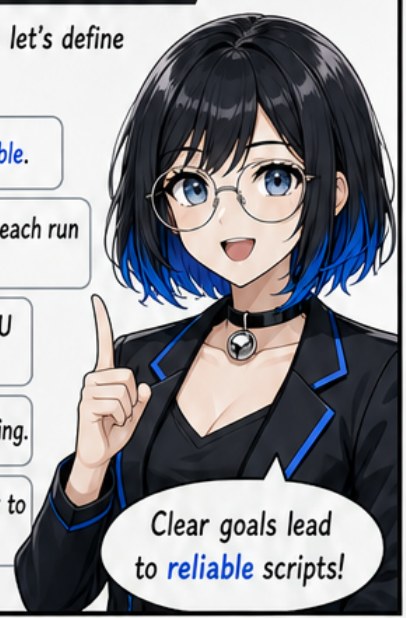
A CSV list of the top ten
processes by CPU.

Safe by design: **read-only** data collection.
No changes. No risk.

DEFINE SUCCESS BEFORE YOU CODE

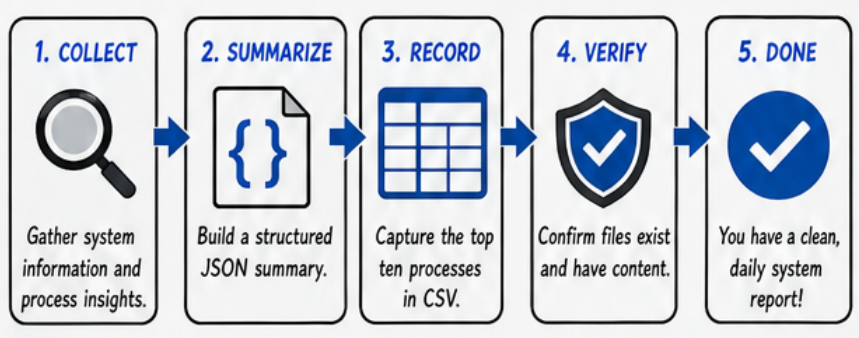
Before we write a single line, let's define
what "done" looks like.

- ✓ Output folder is **configurable**.
- ✓ Facts are **timestamped** so each run
is unique and traceable.
- ✓ **Top ten** processes by CPU
are recorded.
- ✓ Files are **verified** after writing.
- ✓ Errors are **useful** and easy to
understand.



Clear goals lead
to **reliable** scripts!

HOW IT WILL WORK



Run it daily with Task Scheduler for **hands-free insights**.

KEY TAKEAWAY

DEFINE SUCCESS BEFORE YOU SCRIPT.



A clear plan
today makes
automation
tomorrow
reliable!



Plan clearly. Script confidently. Report daily. **Repeat.**

NEXT: Plan the Script →

STEP 1: PREPARE THE OUTPUT FOLDER

Every good script starts by knowing where its output will go. Let's create a **parameter** for the output folder and make sure it **exists**.



GOOD OUTPUT
STARTS WITH
A GREAT
DESTINATION



```
1 param([string]$OutputPath = '.\reports')
2 New-Item -ItemType Directory -Path $OutputPath -Force | Out-Null
```

WHAT IS THIS PARAMETER?



`param([string]$OutputPath = '.\reports')` defines a parameter named `$OutputPath`.

This lets you **choose where** the script puts its files when you run it.

```
PS> .\DailyReport.ps1
      (uses default)
PS> .\DailyReport.ps1
    -OutputPath 'D:\Reports'
```

WHAT IS THE DEFAULT?



If you don't pass anything, the script uses the **default**: `.\reports`

That means it will create (or use) a folder named **reports** in the **same folder** as the script.



WHAT DOES THE SECOND LINE DO?



```
New-Item -ItemType Directory
-Path $OutputPath -Force | Out-Null
```

This checks if the folder exists. If it doesn't, it **creates** it. **-Force** makes sure it won't error if it already exists, and **| Out-Null** keeps the output **clean**.



WHY GIVE OUTPUT ITS OWN FOLDER?

Keeping output in its own folder keeps things **organized** and **safe**.

- ✓ Easy to find your reports
- ✓ No clutter in your script folder
- ✓ Safe to run again and again
- ✓ Simple to clean up or archive



KEY TAKEAWAY

One parameter gives you **control**.
One folder keeps everything **organized**.



ONE PARAMETER
MAKES THE SCRIPT
FLEXIBLE



STEP 2: COLLECT AND SHAPE OBJECTS

OBJECTS
KEEP
DATA
STRUCTURED
AND
USEFUL.



Now we'll collect useful info into a summary object, then create a top-process list.

1) COLLECT A SUMMARY OBJECT



Let's gather key system info into a single object.

```
$summary = [pscustomobject]@{  
    GeneratedAt      = (Get-Date)  
    ComputerName     = $env:COMPUTERNAME  
    PowerShellVersion = $PSVersionTable.PSVersion.ToString()  
    ProcessCount     = (Get-Process).Count  
}
```

This object keeps related data together with **clear names**.
Easy to read, easy to export, easy to reuse!

\$summary

	GeneratedAt	5/26/2025 9:15:00 AM
	ComputerName	DEV-LAPTOP
	PowerShellVersion	7.4.2
	ProcessCount	142

2) SHAPE A TOP-PROCESS LIST



Now we'll get the processes, sort by CPU (highest first), and keep the top 10 with just the fields we need.

**Get-Process | Sort-Object CPU -Descending |
Select-Object -First 10 Name, Id, CPU**

Name	Id	CPU
Code	18420	32.45
chrome	9124	18.21
PowerShell	6520	10.37
Teams	7316	7.84
...	...	...
svchost	1024	1.23

This gives us a clean, focused list of the top 10 CPU users.



WHY OBJECTS?



Structured

Data stays organized with clear properties.



Reusable

Easy to filter, sort, export, or combine.



Reliable

Consistent shape makes automation easier.

KEY TAKEAWAY

**COLLECT OBJECTS,
THEN SHAPE THEM**



Good data in. Good structure. Great reports out! ★



Objects keep data structured. → Structure makes automation powerful.

NEXT: Put it together: the full script! →

STEP 3: SAVE, HANDLE, VERIFY

PAGE 4

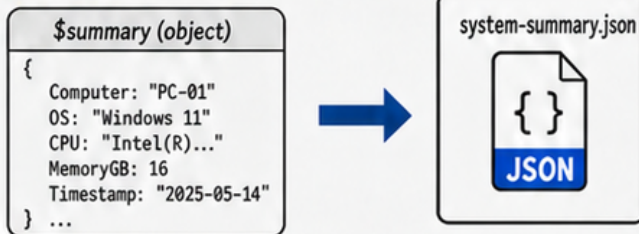
Now we'll save the report safely, handle any errors, and verify the files were created.

Wrap everything in try/catch.

```
1 try {
2     # Build paths
3     # Save summary (JSON)
4     # Save processes (CSV)
5     # Verify outputs
6 }
7 catch { Write-Error $_.Exception.Message }
```

1) SAVE SUMMARY AS JSON

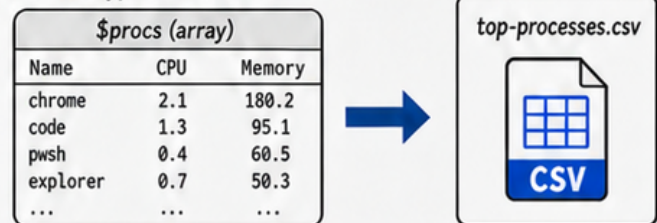
Convert the summary object to JSON and save it.



```
1 $json = $summary | ConvertTo-Json -Depth 5
2 Set-Content -Path $jsonPath -Value $json -Encoding UTF8
```

2) SAVE PROCESSES AS CSV

Export the processes list to CSV without type information.



```
1 $procs | Export-Csv -Path $csvPath -NoTypeInfo -Encoding UTF8
```

3) BUILD PATHS WITH JOIN-PATH

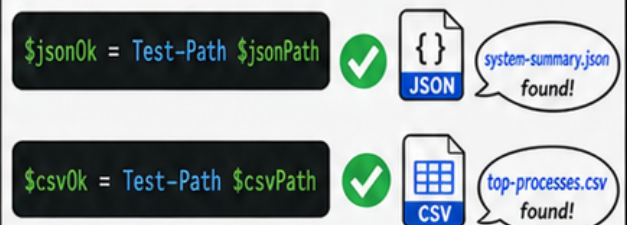
Compose clean, reliable file paths.



```
1 $subDir = Join-Path $reportDir $dateStamp
2 $jsonPath = Join-Path $subDir 'system-summary.json'
3 $csvPath = Join-Path $subDir 'top-processes.csv'
```

4) VERIFY BOTH FILES

Use Test-Path to confirm both outputs exist.



IF SOMETHING GOES WRONG...

Catch the error and show a clear message.

```
1 try {
2     # ... saving and verification steps ...
3 }
4 catch { Write-Error $_.Exception.Message }
```

Exactly this in the catch block:
catch { Write-Error \$_.Exception.Message }

KEY TAKEAWAY

A REPORT IS DONE WHEN ITS OUTPUTS ARE VERIFIED



Save it. Verify it. Trust it.
That's how you build reliable systems!

Save with care. Handle errors. Verify every time.

NEXT: PAGE 5 – PUT IT ON A SCHEDULE

RUN THE REPORT— YOU BUILT IT!



Time to **run** the script you built this week!

Open **PowerShell** and run:

```
PS C:\PS107> .\New-SystemReport.ps1 -OutputPath .\reports
```

It **gathers** the data, **creates** the report files, and **saves** them for you.



CHECK THE OUTPUTS

Your script creates two files in the **.\reports** folder.



system-summary.json



top-processes.csv

```
{
  "ComputerName": "DESKTOP-7X1Q22",
  "User": "Elly",
  "OS": "Windows 11 Pro",
  "Uptime": "2.03:14:27",
  "CPU": "Intel(R) Core(TM) i7",
  "TotalMemoryGB": 16,
  "FreeMemoryGB": 6.1,
  "DiskFreeGB": 237.4,
  "Timestamp": "2025-05-18T10:30:00"
}
```

ProcessName	Id	CPU	MemoryMB
Code	24468	5.8	512.3
powershell	12344	2.1	210.7
chrome	16780	1.9	385.4
MsMpEng	4420	1.3	190.2
explorer	5632	1.1	145.6
...			

LOOKS GOOD!



Two clear files. Ready to read, share, or store. **Nice work!**

★ TIP

Open the folder in File Explorer and peek inside. That's your **automated system report!**



WHAT WE BUILT THIS WEEK

GOAL



Define what the report should collect and **why** it matters.

PARAMETERS



Make the script flexible with parameters like **-OutputPath**.

OBJECTS



Collect data as PowerShell **objects** (not just text).

PIPELINE



Use the **pipeline** to process and shape the data.

FILES



Export clean **files** (JSON and CSV) for easy use.

ERROR HANDLING



Handle problems with **try, catch**, and helpful messages.

VERIFICATION



Check the **outputs** so you know it worked end to end.



You turned an idea into a tool that runs, creates useful files, and is ready to reuse every day!

PowerShell week:
COMPLETE!

KEEP GOING!

- ☒ Add more data
- ☒ Email the report
- ☒ Schedule it daily
- ☒ Make it even better!



GOAL → PLAN → BUILD → **RUN** → VERIFY → IMPROVE