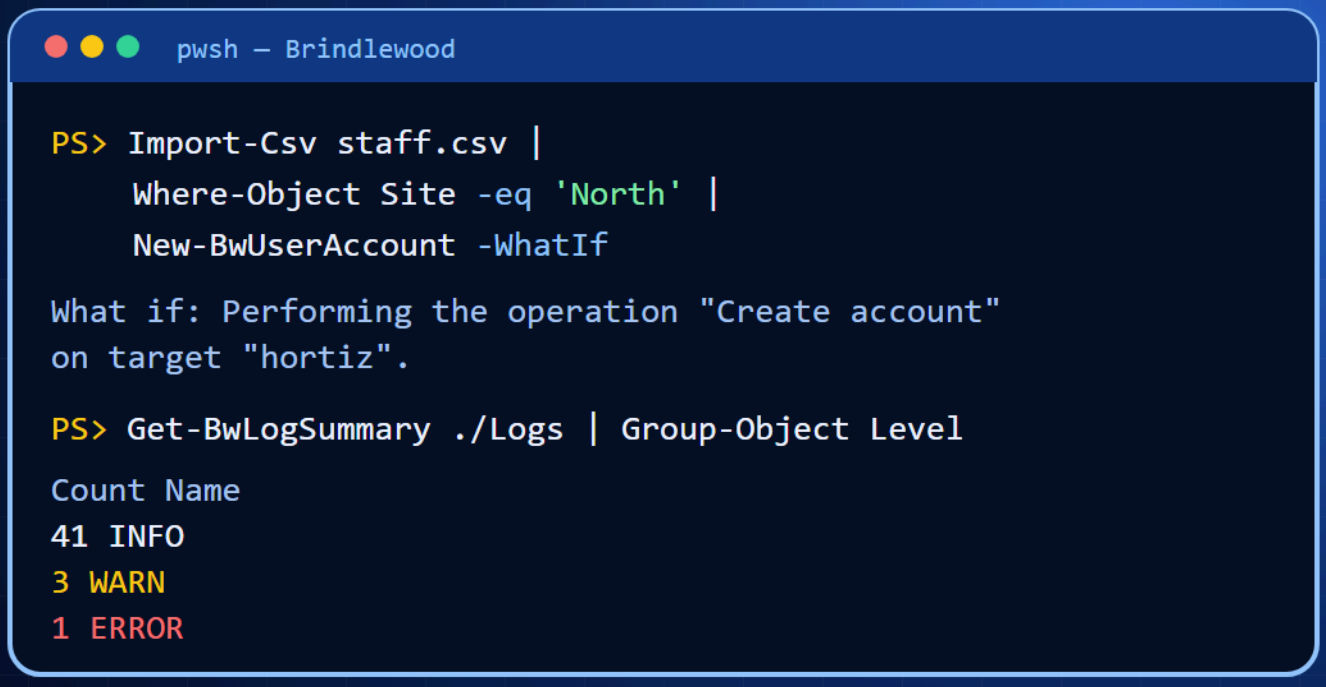


pipeline · modules · Pester · REST APIs · remoting · -Parallel · scheduling · secrets

A COMPLETE, PRACTICAL GUIDE

PowerShell from Zero

Learn PowerShell 7 scripting and automate real IT work on Windows, Linux, and macOS

objects · pipeline · advanced functions · CSV/JSON · regex
modules · Pester · REST · remoting · scheduling

SHELL · LANGUAGE · AUTOMATION · PROJECTS

THENEXBOOKS

PowerShell from Zero

A Complete, Practical Guide to PowerShell 7 Scripting and Automation
From your first command to tested modules and scheduled automation

First Edition

THENEXBOOKS

PowerShell from Zero
A Complete, Practical Guide to PowerShell 7 Scripting and Automation

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Automation scripts can change or delete data at scale: test with -WhatIf, on non-production systems, before running anything for real. PowerShell and its modules evolve; always consult the official documentation for the versions you use.

Microsoft, Windows, PowerShell, Visual Studio Code, Microsoft Entra ID, and Azure are trademarks of the Microsoft group of companies. Linux is a registered trademark of Linus Torvalds. macOS is a trademark of Apple Inc. GitHub is a trademark of GitHub, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Microsoft.

Brindlewood Veterinary Group, the organization used as the running example in this book, is fictional, as are its clinics, people, pets, computers, and data. Domains such as brindlewood.example are reserved for examples.

Preface

Every IT team has the same list of chores. Create accounts for the new starters. Check that the disks are not filling up. Clean out old log files. Produce the Monday report. Each one takes ten minutes by hand, and every week they come back. PowerShell exists so that you can write those ten minutes down once, as a script, and let the computer do them from then on — reliably, the same way every time.

What makes PowerShell different from other shells is that it passes *objects* between commands, not text. When you list files, you get file objects with sizes and dates you can sort, filter, and measure directly. You don't have to cut columns out of text output. Add a consistent Verb-Noun command style, built-in help, and the whole .NET platform underneath, and you get a shell that is easy to start with and powerful enough for serious automation on Windows, Linux, and macOS.

This book teaches PowerShell from the ground up, one idea at a time, and always explains *why* it behaves the way it does. Every chapter adds a piece to real automation for a small organization. By the end you will have written a tested PowerShell module, an onboarding toolkit, and a scheduled daily health report — and you will know how to keep learning on your own.

Who This Book Is For

- **IT support staff and system administrators** who want to stop clicking through the same tasks and start automating them.
- **Beginners** who want a practical first scripting language with immediate, useful results.
- **Developers** who want a powerful cross-platform shell for build, deployment, and data chores.
- **Windows PowerShell users** who want to move to PowerShell 7 and understand what changed.

What You Will Learn

- How to install PowerShell 7 and VS Code, find any command, and read help and errors with confidence.
- The object pipeline: filtering, sorting, grouping, measuring, and formatting objects instead of parsing text.
- The language: variables, types, operators, strings, arrays, hashtables, PSCustomObject, control flow, and functions.
- Advanced functions with parameter validation, pipeline input, -WhatIf and -Confirm, and built-in help.
- Files and folders, CSV, JSON, and XML, regular expressions, processes, services, event logs, and the registry.
- Error handling, scripts, profiles, execution policy, modules, Pester tests, debugging, and logging.
- REST APIs, remoting over WinRM and SSH, jobs and parallel execution, scheduling, and handling secrets safely.

- Classes, .NET interop, performance, and cross-platform tips.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) installs PowerShell and covers cmdlets, help, the pipeline, and variables. **Part II — Language Essentials** (Chapters 6–10) covers strings, collections, control flow, and functions up to full advanced functions. **Part III — Working with Data and the System** (Chapters 11–15) covers working with objects, files, data formats, text, and the operating system. **Part IV — Robust Scripts** (Chapters 16–20) covers error handling, scripts, modules, testing, and debugging. **Part V — Automation in Practice** (Chapters 21–25) covers REST APIs, remoting, parallelism, scheduling, and security. **Part VI — Projects and Next Steps** (Chapters 26–28) builds two complete automation projects and looks beyond. The appendices provide a cheat sheet, a Windows PowerShell 5.1 vs PowerShell 7 reference, an error and gotchas field guide, and a glossary.

Each chapter ends with practice exercises, selected solutions, and a **Key Takeaways** box for quick review.

The Running Example: Brindlewood Veterinary Group

Throughout the book you join the small IT team of **Brindlewood Veterinary Group**, a fictional group of three veterinary clinics — North, Central, and South. Nadia leads IT, Owen manages the practice and keeps asking for reports, and Kai, a junior admin, is learning PowerShell alongside you. Starting from a CSV of staff, you build `Brindlewood.Admin`, a PowerShell module that onboards new starters, summarizes clinic logs, checks disk space, calls the appointments API, cleans up old files, and produces a daily health report on a schedule.

Versions and Conventions

Examples target **PowerShell 7.4 (LTS) and later**. Where Windows PowerShell 5.1 behaves differently, we say so, and Appendix B summarizes the differences. Features that exist only on Windows — services, event logs, the registry, WinRM, and Task Scheduler — are labeled, with cross-platform alternatives where they exist.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause bugs, data loss, or security problems.

Code appears in `monospace`. Interactive commands are shown after a `PS>` prompt, followed by their output. File names above code blocks, such as `Brindlewood.Admin/Public/Get-BwStaff.ps1`, tell you where the code lives. Long output is trimmed to fit the page.

Contents

Preface

PART I — GETTING STARTED

1 Why PowerShell? Objects, Automation, and Where It Runs

2 Installing PowerShell 7, VS Code, and Your First Commands

3 Cmdlets, Parameters, and Discovering Help

4 The Pipeline: Objects, Not Text

5 Variables, Types, and Operators

PART II — LANGUAGE ESSENTIALS

6 Strings, Here-Strings, and Formatting Output

7 Arrays, Hashtables, and PSCustomObject

8 Control Flow: if, switch, Loops, and ForEach-Object

9 Functions and Script Blocks

10 Advanced Functions: CmdletBinding, Parameter Validation, and Pipeline Input

PART III — WORKING WITH DATA AND THE SYSTEM

11 Filtering, Sorting, Grouping, and Measuring Objects

12 Files, Folders, and Paths: The Provider Model

13 Working with CSV, JSON, and XML

14 Text Processing and Regular Expressions

15 Processes, Services, Event Logs, and the Registry

PART IV — ROBUST SCRIPTS

16 Error Handling: Terminating Errors, try/catch, and $ErrorActionPreference

17 Scripts, Execution Policy, Profiles, and Script Signing

18 Modules: Using, Writing, and Publishing

19 Testing with Pester

20 Debugging, Logging, and Transcripts

PART V — AUTOMATION IN PRACTICE

21 Calling REST APIs with Invoke-RestMethod

22 Remoting: PowerShell over WinRM and SSH

23 Jobs, ForEach-Object -Parallel, and Thread Jobs

24 Scheduling and Running Automation

25 Security: Credentials, SecretManagement, and Safe Scripting

PART VI — PROJECTS AND NEXT STEPS

26 Project: The Brindlewood Onboarding Toolkit

27 Project: Daily Health Report and Log Cleanup

28 Classes, .NET Interop, Performance, Cross-Platform Tips, and Where to Go Next

APPENDICES

A PowerShell Syntax and Cmdlet Cheat Sheet

B Windows PowerShell 5.1 vs PowerShell 7 and Tooling Reference

C Common Errors and Gotchas Field Guide

D Glossary

PART I

Getting Started

What PowerShell is and why its object pipeline changes everything, installing PowerShell 7 and VS Code, finding and understanding any command, and the variables, types, and operators every script is built from.

1 · Why PowerShell?

2 · Installing PowerShell 7

3 · Cmdlets, Parameters, and Help

4 · The Pipeline

5 · Variables, Types, and Operators

CHAPTER 1 · PART I — GETTING STARTED

Why PowerShell? Objects, Automation, and Where It Runs

Every IT team has a pile of small, repetitive jobs that eat the week: creating accounts, checking disks, clearing old logs, answering "can you send me a list of…". PowerShell exists to turn those jobs into commands you type once and scripts you run forever — and it does it with one idea that sets it apart from every shell that came before it.

What a shell is, and why you would want one

A **shell** is a program that reads commands you type, runs them, and shows you the result. The word comes from the idea of a layer wrapped around the operating system: you speak to the shell, and the shell speaks to the system for you. Windows has shipped the Command Prompt (`cmd.exe`) for decades; Linux and macOS users usually meet Bash or Zsh. PowerShell is a shell too — but, as you will see, a rather unusual one.

Three words often get mixed up, so let us separate them now:

- **Terminal** (or terminal emulator) — the window that draws text and passes your keystrokes along. Windows Terminal, macOS Terminal, and the terminal panel inside VS Code are terminals.
- **Shell** — the program running *inside* the terminal that interprets what you type. PowerShell, Bash, Zsh, and `cmd.exe` are shells.
- **Console** — an older word for the same idea as a terminal; you will still see it in names such as "the PowerShell console".

You can run PowerShell inside many different terminals, and one terminal can host many different shells in separate tabs. Keeping that picture in your head saves a lot of confusion when a guide tells you to "open a terminal and start PowerShell".

Why bother with a shell when Windows, macOS, and most admin tools have perfectly good graphical interfaces? Because clicking does not scale. Resetting one password in a web console takes a minute. Resetting forty takes an afternoon and invites mistakes. A command, on the other hand, can be repeated, saved in a file, reviewed by a colleague, run on a schedule at 2 a.m., and pointed at one machine or a hundred. Graphical tools are wonderful for exploring; shells are wonderful for *repeating*. Most working administrators use both — the GUI to discover what is possible, the shell to do it reliably.

There is a second, quieter benefit: a shell command is a precise record of what was done. "I clicked around in the settings until it worked" cannot be reviewed or repeated. A script can be read line by line, tested on a spare machine first, and kept in version control so that six months from now you can

see exactly what changed and why.

PowerShell's big idea: objects, not text

Traditional shells pass **text** between commands. When you pipe the output of one Bash command into another, the second command receives a stream of characters and has to work out where the columns are. That is why Bash users become expert in tools like `grep`, `awk`, `cut`, and `sed`: so much of the work is slicing text back into fields that the first command had in perfectly good shape before it printed them.

Here is a Bash-style way (on Linux with GNU `ps`) to list the three processes using the most memory:

```
$ ps -eo comm,rss --sort=-rss | head -4
```

That works, but notice what it depends on: you must know that `rss` is the column name, that the output has a header line (hence `head -4` rather than 3), and that the numbers are in kilobytes. If a process name contains a space, or a future version changes the column layout, a script that cuts the text by position quietly breaks.

PowerShell takes a different route. Commands pass **objects** — structured pieces of data with named **properties** — from one to the next. Text is only produced at the very end, when the results land on your screen. The equivalent PowerShell command reads almost like a sentence:

```
PS> Get-Process | Sort-Object -Property WS -Descending |
    Select-Object -First 3 -Property Name, Id, WS
```

`Get-Process` does not print a table and hand the printout to `Sort-Object`. It hands over process objects, each with a `Name`, an `Id`, a `WS` (its working set — the memory it is using — in bytes), a `StartTime`, and dozens more properties. `Sort-Object` sorts by a property *name*, not a column position. `Select-Object` keeps the first three and only the properties you asked for. Nothing has to be parsed, because nothing was ever flattened into text in the middle.

Reading the output

Run the command above and you will see something like this (your process names and numbers will differ — these are illustrative):

```
Name       Id          WS
----       --          --
msedge  14212   412516352
Code     9876   298057728
pwsh    11840   102371328
```

Let us read it slowly, because this pattern will appear thousands of times in your PowerShell life:

1. **The header row** (`Name`, `Id`, `WS`) shows the property names of the objects that reached the end of the pipeline. Those are the exact names you can use in later commands — `Sort-Object Id`, `Where-Object Name -eq 'pwsh'`, and so on.
2. **The dashed row** is just decoration added by PowerShell's formatting system. It is not data.
3. **Each data row** is one object. There are three rows because `Select-Object -First 3` kept three objects.
4. **The numbers** under `WS` are plain bytes, not "400 MB". Because they are real numbers rather than text, you can do arithmetic with them — dividing by `1MB`, for instance, which Chapter 5 shows.

That last point is the heart of it. In a text shell, "412516352" on the screen is just nine characters. In PowerShell it is a number attached to a process object, which also knows when it started, which file it was launched from, and how to stop itself. Chapter 4 is devoted entirely to the pipeline; for now, the takeaway is that you manipulate *data*, and PowerShell turns it into text only when it has to show you something.

> **NOTE**
>
> The table you see is a *view* of the objects, not the objects themselves. PowerShell picks a few useful properties to display by default and hides the rest. A process object has far more properties than the handful shown — you will learn to reveal them with `Get-Member` and `Select-Object -Property *` in Chapters 3 and 4.

Verb-noun cmdlets: names you can guess

PowerShell's built-in commands are called **cmdlets** (pronounced "command-lets"). Nearly every cmdlet follows the same naming pattern: a **verb**, a hyphen, and a singular **noun**. The verb says what you are doing; the noun says what you are doing it to.

```
Get-Process      # read information about processes
Stop-Process     # stop a process
Get-Service      # read information about Windows services
Start-Service    # start a Windows service
Get-ChildItem    # list the items inside a folder
New-Item         # create a file or folder
Remove-Item      # delete a file or folder
Import-Csv       # read a CSV file into objects
Export-Csv       # write objects out to a CSV file
```

The verbs come from a short, published list of **approved verbs**. You can see that list yourself by running `Get-Verb`. Because everyone — Microsoft, other vendors, and eventually you — is encouraged to use the same verbs, you can often *guess* a command before you look it up. If `Get-Service` reads services, then `Stop-Service` and `Restart-Service` almost certainly exist, and they do. If you find a cmdlet called `Get-Something`, you can be confident it only reads and does not change anything.

Verb	Meaning	Example
Get	Read, never change	`Get-Date`
Set	Change an existing thing	`Set-Location`
New	Create something	`New-Item`
Remove	Delete something	`Remove-Item`
Start / Stop	Begin or end an activity	`Start-Process`
Import / Export	Bring data in or out	`Export-Csv`
ConvertTo / ConvertFrom	Change format	`ConvertTo-Json`

Later in this book you will write your own commands, and you will follow the same convention. Our Brindlewood functions will be called things like `Get-BwStaff` and `Remove-BwOldLog` — the `Bw` prefix on the noun marks them as ours, so they never collide with somebody else's `Get-Staff`.

> **TIP**
> When you do not know a command's name, search by noun or verb with wildcards: `Get-Command -Noun *Item*` or `Get-Command -Verb Export`. Chapter 3 turns this into a full discovery routine with `Get-Help`.

Consistent parameters

Commands take **parameters** — named options that start with a hyphen, such as `-Name` or `-Path`. In older shells, every tool invents its own switches: one program uses `/s` for "include subfolders", another uses `-r`, a third uses `--recursive`. PowerShell pushes hard toward consistency:

- Commands that work with files and folders use `-Path`; commands that work with named things use `-Name`; recursing into subfolders is `-Recurse`.
- Every cmdlet supports a set of **common parameters** such as `-Verbose`, `-ErrorAction`, and `-ErrorVariable`, whether its author thought about them or not.
- Commands that change things are encouraged to support `-WhatIf` (show what *would* happen, without doing it) and `-Confirm` (ask before each change).

That `-WhatIf` switch deserves a moment of appreciation, because it makes PowerShell unusually safe for learners. Before you delete a batch of files, you can rehearse:

```
PS> Remove-Item -Path ./old-reports/*.csv -WhatIf
What if: Performing the operation "Remove File" on target ".../old-reports/q1.csv".
What if: Performing the operation "Remove File" on target ".../old-reports/q2.csv".
```

Nothing was deleted. PowerShell simply told you what it would have done. When the list looks right, you remove `-WhatIf` and run it for real. Our own `Remove-BwOldLog` and `New-BwUserAccount` functions will support `-WhatIf` too — Chapter 10 shows how little code that takes.

Parameter names can also be shortened, as long as the short form is unambiguous: `-Recurse` can be typed as `-Rec`. That is handy at the prompt, but in scripts we always write parameters out in full so that the next reader — often you, months later — does not have to decode them.

A short history

Knowing where PowerShell came from explains why two versions sit side by side on most Windows machines today, and why so many web articles disagree with each other.

Monad and Windows PowerShell

In the early 2000s, a Microsoft architect named Jeffrey Snover described a new kind of management shell, code-named **Monad**. Its central argument was the one you just read: administrators should work with structured objects from the .NET Framework rather than with text. Monad was renamed **Windows PowerShell**, and version 1.0 was released in **2006**.

Over the following versions Windows PowerShell gained the features that made it the standard tool for Windows administration: remoting (running commands on other computers), background jobs, modules, a scripting editor called the PowerShell ISE, and Desired State Configuration. The line ended with **Windows PowerShell 5.1**, which still ships built into Windows 10, Windows 11, and Windows Server. It is finished software: it receives security fixes, but no new features.

Open source and cross-platform

Windows PowerShell was tied to the Windows-only .NET Framework. As Microsoft moved toward the open-source, cross-platform .NET Core, PowerShell followed. In **2016** the project was open-sourced on GitHub and began running on Linux and macOS as well as Windows. The first releases of this new line were called **PowerShell Core 6**.

With version 7 the "Core" was dropped, and the product became simply **PowerShell** — usually written **PowerShell 7** to make the distinction clear. PowerShell 7 is built on modern .NET and has its own release rhythm, with some releases designated **Long-Term Support (LTS)**:

- **PowerShell 7.4** is an LTS release (November 2023, built on .NET 8). This book targets 7.4 and later.
- **PowerShell 7.5** (2025, built on .NET 9) is a stable, non-LTS release.
- The next LTS release is planned to follow the next LTS version of .NET. Release plans change, so check the current release notes before you standardize on a version.

> **NOTE**
>
> When you read an older blog post that says "PowerShell can't do X on Linux" or "PowerShell Core doesn't support Y", check its date and the version it is talking about. A great deal changed between PowerShell Core 6 and PowerShell 7, and many early gaps have since been closed.

PowerShell vs cmd.exe vs Bash

You do not have to abandon other shells to learn PowerShell, and this book is not here to start a shell war. Each one is good at something. It helps, though, to know where PowerShell sits among the shells you may already have met.

	cmd.exe	Bash	PowerShell 7
Platforms	Windows	Linux, macOS, WSL	Windows, Linux, macOS
Pipeline carries	Text	Text	.NET objects
Command naming	Varies by tool	Varies by tool	Verb-Noun
Scripting language	Batch (limited)	Full, text-centric	Full, object-centric
Built-in data formats	None	Via extra tools	CSV, JSON, XML
Error handling	Exit codes	Exit codes, traps	Exceptions, try/catch

cmd.exe is the oldest of the three. It still runs batch files from decades ago, and some installers and build tools still assume it. As a scripting language it is limited: there are no real data structures, error handling is awkward, and string manipulation is painful. If you come from cmd.exe, the good news is that PowerShell runs almost all the same external programs — `ipconfig`, `ping`, `robocopy`, `git` — exactly as before. You lose nothing, and gain a real language.

Bash is a mature, powerful shell and the default habitat of Linux administrators. Its pipeline of small text tools is elegant, and on Linux you will keep using it. PowerShell's advantages show up when data has structure: reading a CSV and filtering by a column, calling a web API and picking fields out of JSON, or producing a report with consistent columns. If you come from Bash, expect two adjustments: comparison operators are words (`-eq`, `-gt`) rather than symbols, and you will reach for `Where-Object` and `Select-Object` where you used to reach for `grep` and `awk`.

And if you come from **Python**: PowerShell will feel like a scripting language with a very good interactive prompt bolted on. Variables start with `$`, functions are called without parentheses or commas (a classic early stumble — Chapter 9 explains it), and the pipeline replaces a lot of explicit loops.

Where PowerShell is used

PowerShell started life as a Windows administration tool, and that is still its home ground, but it has spread well beyond it.

- **System administration.** Managing users, groups, files, services, scheduled work, and event logs on Windows servers and desktops. Many Microsoft products ship PowerShell modules as their primary automation interface, and some settings are easier to reach from PowerShell than from any graphical console.
- **Cloud.** Microsoft Azure has the `Az` PowerShell modules and offers PowerShell in Azure Cloud Shell; AWS publishes the AWS Tools for PowerShell; Microsoft 365 and Microsoft Entra ID are managed through the Microsoft Graph PowerShell SDK. The same skills you learn on a laptop carry straight into these environments.
- **DevOps and CI/CD.** Build and release pipelines — GitHub Actions, Azure Pipelines, and others — can run steps in PowerShell on Windows, Linux, and macOS runners. A single cross-platform build script is a common reason teams adopt PowerShell 7.
- **Developers.** As an everyday shell with excellent tab completion, an interactive way to explore .NET libraries, and a glue language for small tools: renaming files, transforming JSON, calling an internal API, or preparing test data.

What ties these together is the object pipeline. Whether the objects describe a Windows service, an Azure virtual machine, or a row from a CSV file, the same handful of commands — `Where-Object`, `Sort-Object`, `Select-Object`, `Group-Object`, `Export-Csv` — work on all of them. Learn those well and you can work with almost anything that has a PowerShell module.

Windows PowerShell 5.1 and PowerShell 7, side by side

On a Windows machine you will very likely have *both* versions installed at once. They do not replace each other; they live side by side, with different executables and different folders. Here is how they compare:

	Windows PowerShell 5.1	PowerShell 7
Executable	`powershell.exe`	`pwsh` (`pwsh.exe`)
Runs on	Windows only	Windows, macOS, Linux
Built on	.NET Framework	Modern .NET
How you get it	Built into Windows	Installed separately
Development	Security fixes only	Active, open source
`$PSVersionTable.PSEdition`	`Desktop`	`Core`
Default file encoding	Mixed (UTF-16LE / ANSI)	UTF-8 without BOM
Default error display	Multi-line, verbose	Concise, one line

PowerShell 7 adds a number of language conveniences that 5.1 lacks, such as the ternary operator (`$a ? 'yes' : 'no'`), the null-coalescing operators `??` and `??=`, the pipeline chain operators `&&` and `||`, running pipeline work in parallel with `ForEach-Object -Parallel`, and the `Get-Error` cmdlet for inspecting failures. You will meet all of them in later chapters, each clearly marked as PowerShell 7 only.

What about older Windows-only modules that were written for 5.1? Most popular modules now work in PowerShell 7 directly. For those that do not, PowerShell 7 on Windows has a compatibility feature — `Import-Module -UseWindowsPowerShell` — that loads the module in a background 5.1 session and lets you call its commands from 7. It is a bridge, not a destination, but it means you rarely have to choose one version for everything.

> **WARNING**
>
> Do not try to "upgrade" or remove Windows PowerShell 5.1. It is part of Windows, and other software depends on it. Install PowerShell 7 *alongside* it, as Chapter 2 shows, and choose which one to start. When a script must also run on machines that only have 5.1, test it there — and avoid the 7-only syntax listed above.

Throughout this book, examples are written and shown for PowerShell 7. Where Windows PowerShell 5.1 behaves differently in a way that could trip you up, a note says so.

Meet Brindlewood Veterinary Group

Learning a shell from isolated one-liners is like learning a language from a phrasebook: you can order coffee, but you cannot hold a conversation. So in this book you will join a (fictional) organization and solve its real-feeling problems from Chapter 1 to the end.

Brindlewood Veterinary Group runs three veterinary clinics, called simply **North**, **Central**, and **South**. Its email domain is `brindlewood.example` — a reserved example domain, so nothing you do in this book can accidentally reach a real company. The IT team is small, which is exactly why automation matters:

- **Nadia** is the IT lead. She has been scripting for years and plays the mentor: she sets the tasks, reviews scripts, and explains why things are done the way they are.
- **Owen** is the practice manager. He is not technical, but he constantly needs information — who started this month, which clinic logged the most errors, how much disk space is left — and he likes it in a spreadsheet or a web page.
- **Kai** is a junior IT admin who, like you, is learning PowerShell. Kai makes the mistakes that everyone makes, so you can learn from them without having to make them yourself.

And you? You are the newest member of the IT team. On your first morning Nadia sums up the job: "Every Monday somebody hand-creates accounts for new hires. Every morning somebody checks the servers by clicking around. Once a month somebody remembers that the log folder is full. I want all of

that to be a command — and I want it to be safe to run."

What we will build

Over the course of the book, those requests turn into four pieces of working automation:

1. **An onboarding toolkit.** Brindlewood keeps its staff list in a file called `staff.csv`, with new hires arriving in `newhires.csv`. You will read those files, generate usernames (first initial plus last name, lowercase — Hana Ortiz becomes `hortiz`) and email addresses, and create accounts. Because we do not want to touch a real directory while learning, accounts are recorded in a JSON file that stands in for Active Directory or Microsoft Entra ID; the book points out where the real modules would slot in.
2. **A daily health report.** Free disk space, a summary of the clinic application's log files, and a few other checks, gathered into an HTML page that Owen can open in a browser.
3. **Log and backup housekeeping.** The clinic application writes one log file per day. Brindlewood's retention policy is to delete logs older than 30 days and keep backups for 14 days — a perfect job for a careful script with `-WhatIf` support.
4. **A module, `Brindlewood.Admin`.** In Chapter 18, the functions you have written in scripts — `Get-BwStaff`, `New-BwUserAccount`, `Get-BwLogSummary`, and others — are packaged into a proper module that anyone on the team can import, joined by housekeeping commands such as `Remove-BwOldLog`; later chapters add more, such as `Export-BwHealthReport`.

Here is a first glimpse of the data you will be working with. This is the beginning of `staff.csv`:

Brindlewood/Data/staff.csv

```
EmployeeId,FirstName,LastName,Role,Site,StartDate
E1001,Hana,Ortiz,Veterinarian,North,2024-03-04
E1002,Liam,Becker,VetTech,Central,2025-01-13
E1003,Ama,Mensah,Receptionist,South,2025-06-02
```

And a few lines from one day's application log:

Brindlewood/Logs/app-2026-04-14.log

```
2026-04-14T09:30:12 INFO North Appointment booked A-20417
2026-04-14T09:41:03 WARN Central Printer offline PRN-C2
2026-04-14T10:02:55 ERROR South Payment terminal timeout POS-S1
```

As a small preview of where the book is heading, here is how PowerShell turns that CSV file into objects you can query — no text slicing required. Do not worry about typing this yet; Chapter 13 covers CSV in depth.

```
PS> Import-Csv -Path ./staff.csv | Where-Object Site -eq 'North' |
    Select-Object FirstName, LastName, Role

FirstName LastName Role
```

```
--------- -------- ----
Hana      Ortiz    Veterinarian
Priti     Rao      PracticeManager
```

Notice that `Where-Object` filtered on the `Site` *column name*. `Import-Csv` turned each row into an object whose properties are the header names, so the rest of the pipeline works with names, not positions. Add a column to the CSV next year and this command still works.

All Brindlewood files live under one folder in your home directory, which we will refer to as `$BwRoot`. It has four subfolders — `Data`, `Logs`, `Reports`, and `Backups` — and in Chapter 2 you will create it with your very first lines of PowerShell. (Later a `Scripts` folder joins them to hold your script files.) Because it sits under your home folder, the same instructions work on Windows, macOS, and Linux.

> **NOTE**
>
> Some topics in this book are Windows-specific: Windows services, the event log, the registry, WinRM remoting, and Task Scheduler. Those chapters say so up front and offer cross-platform alternatives where they exist. Everything in the Brindlewood running example is designed to work on all three operating systems.

Common mistakes and misconceptions

- **"PowerShell is just a fancier Command Prompt."** It can run the same programs, but it is a full programming language with an object pipeline. Treating it like cmd.exe — parsing text output with string tricks — throws away its main advantage.
- **Thinking the screen shows everything.** The default table shows a few chosen properties. When a value you need seems to be missing, it is usually hiding, not absent. `Select-Object -Property *` and `Get-Member` reveal it.
- **Following a 5.1 tutorial in PowerShell 7 (or the reverse).** Most commands behave the same, but encodings, error displays, and a few cmdlets differ. Check which one you are running — Chapter 2 shows how with `$PSVersionTable`.
- **Assuming PowerShell means Windows.** PowerShell 7 runs on macOS and Linux too. What does *not* travel is Windows-only functionality such as services and the registry.
- **Running a downloaded script without reading it.** Automation is powerful in both directions. A script that can create forty accounts can also delete forty. Read first, rehearse with `-WhatIf`, then run.

Practice

You do not need PowerShell installed for these exercises — they check your understanding of the ideas in this chapter. Chapter 2 gets you typing.

1. **Name that cmdlet.** Using only the verb-noun convention and the verb table above, guess the names of the cmdlets that would (a) read the contents of a file, (b) create a new folder, (c) convert objects to JSON, and (d) restart a Windows service.
2. **Text or objects?** Owen asks for "everyone at the Central clinic, sorted by start date". Describe in one or two sentences how you might do this in a text-based shell, and why the PowerShell approach is less fragile.
3. **Which PowerShell?** For each statement, say whether it describes Windows PowerShell 5.1, PowerShell 7, or both: (a) runs on Linux, (b) started with `powershell.exe`, (c) supports the `??` operator, (d) can run `Get-Process`, (e) receives new features.
4. **Brindlewood usernames.** Applying Brindlewood's username rule, what are the usernames and email addresses for Liam Becker and Jonas Lindqvist?

Selected Solutions

1. (a) `Get-Content`, (b) `New-Item` (with a parameter saying it is a directory — you will see it in Chapter 2), (c) `ConvertTo-Json`, (d) `Restart-Service`. If you guessed most of these, the naming convention is already doing its job.
2. In a text shell you would filter lines containing "Central" (risking false matches, such as a person whose name contains that word) and sort on a column number. In PowerShell, `Import-Csv` produces objects, so you filter on the `Site` property and sort on the `StartDate` property by name. Column order and stray text cannot confuse it.
3. (a) PowerShell 7. (b) Windows PowerShell 5.1 — PowerShell 7 is `pwsh`. (c) PowerShell 7. (d) Both. (e) PowerShell 7; 5.1 receives security fixes only.
4. Liam Becker is `lbecker` (`lbecker@brindlewood.example`). Jonas Lindqvist is `jlindqvist` (`jlindqvist@brindlewood.example`).