

# Running commands



When you start looking at PowerShell examples on the internet, it's easy to get the impression that PowerShell is some kind of .NET-based scripting or programming language. Our fellow Microsoft most valuable professional (MVP) award recipients, and hundreds of other PowerShell users, are pretty serious geeks who like to dig deep into the shell to see what we can make it do. But almost all of us began right where this chapter starts: running commands. That's what you'll be doing in this chapter: not scripting, not programming, but running commands and command-line utilities.

## 4.1 **Let's talk security**

Okay, it's time to talk about the elephant in the room. PowerShell is great, and PowerShell is awesome. But the bad guys like PowerShell just as much as we do. Securing your production environment is at the top of everyone's priority list. By now, you're probably starting to get a feel for how powerful PowerShell can be—and you're wondering whether all that power might be a security problem. It *might* be. Our goal in this section is to help you understand exactly how PowerShell can impact security in your environment and how to configure PowerShell to provide exactly the balance of security and power you require.

First and foremost, PowerShell doesn't apply any additional layers of permissions on anything it touches. PowerShell enables you to do only what you already have permission to do. If you can't create new users in Active Directory by using the graphical console, you won't be able to do so in PowerShell either. PowerShell is another means of exercising whatever permissions you already have.

PowerShell is also not a way to bypass any existing permissions. Let's say you want to deploy a script to your users, and you want that script to do something that your users don't normally have permission to do. That script isn't going to work for them. If you want your users to do something, you need to give them permission to

do it. PowerShell can accomplish only those things that the person running a command or script already has permission to do.

PowerShell's security system isn't designed to prevent anyone from typing in, and running, whatever commands they have permission to execute. The idea is that it's somewhat difficult to trick a user into typing a long, complicated command, so PowerShell doesn't apply any security beyond the user's existing permissions. But we know from past experience that it's easy to trick users into running a script, which might well contain malicious commands. This is why most of PowerShell's security is designed with the goal of preventing users from unintentionally running scripts. The *unintentionally* part is important: nothing in PowerShell's security is intended to prevent a determined user from running a script. The idea is to prevent users only from being *tricked* into running scripts from untrusted sources.

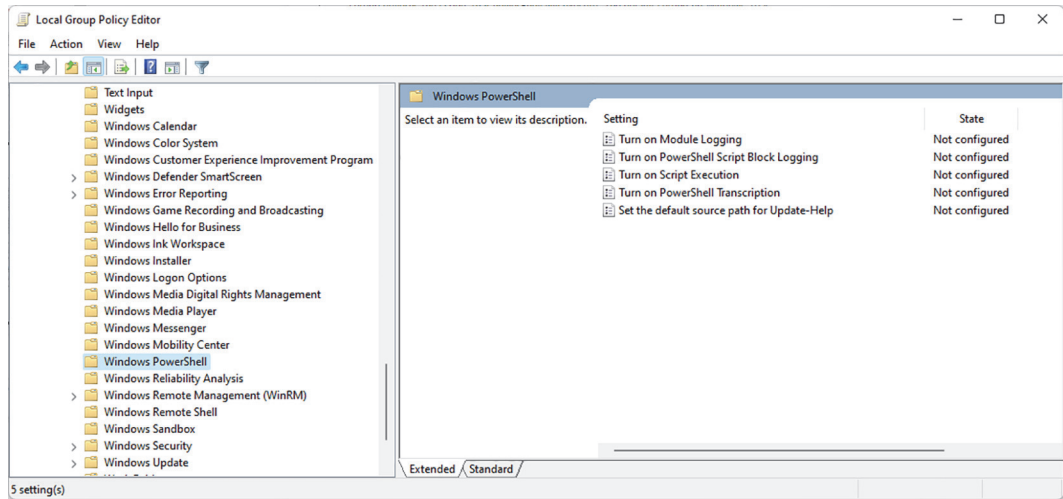
#### 4.1.1 Execution policy

The first security measure PowerShell includes is an *execution policy*. This machine-wide setting governs the scripts that PowerShell will execute. The default setting on Windows 10 is Restricted. On Windows Servers, the default is RemotedSigned and the execution policy on non-Windows devices is not enforced. The Restricted setting on Windows 10 devices prevents scripts from being executed at all. That's right: by default, you can use PowerShell to interactively run commands, but you can't use it to run scripts. Let's pretend that you downloaded a script off the internet. If you try to run it, you'll get the following error message:

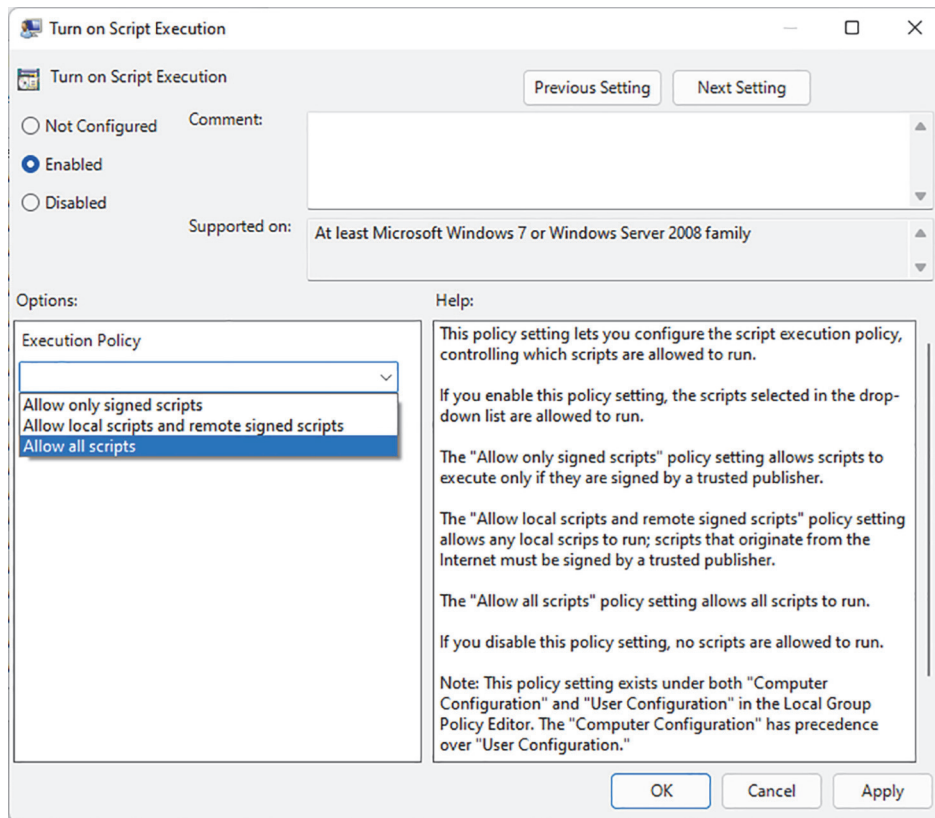
```
File C:\Scripts\Get-DiskInventory.ps1 cannot be loaded because the execution
➤ of scripts is disabled on this system. Please see "get-help about_signing"
➤ for more details.
```

View the current execution policy by running `Get-ExecutionPolicy`. You can change the execution policy in one of three ways:

- *By running the Set-ExecutionPolicy command*—This changes the setting in the `HKEY_LOCAL_MACHINE` portion of the Windows Registry and usually must be run by an administrator, because regular users don't have permission to write to that portion of the Registry.
- *By using a Group Policy Object (GPO)*—Starting with Windows Server 2008 R2, support for PowerShell-related settings is included. The PowerShell settings shown in figure 4.1 are located under Computer Configuration > Policies > Administrative Templates > Windows Components > Windows PowerShell. Figure 4.2 displays the policy setting as enabled. When configured via a GPO, the setting in the Group Policy overrides any local setting. In fact, if you try to run `Set-ExecutionPolicy`, it'll work, but a warning message will tell you that your new setting had no effect because of a Group Policy override.
- *By manually running PowerShell.exe and using its -ExecutionPolicy command-line switch*—When you run it in this fashion, the specified execution policy overrides any local setting as well as any Group Policy-defined setting. Which we can see in figure 4.1.



**Figure 4.1** Finding the Windows PowerShell settings in a Group Policy Object



**Figure 4.2** Changing the Windows PowerShell execution policy in a Group Policy Object

You can set the execution policy to one of five settings (note that the Group Policy Object provides access to only the middle three of the following list):

- **Restricted**—This is the default, and scripts aren't executed. The only exceptions are a few Microsoft-supplied scripts that set up PowerShell's default configuration settings. Those scripts carry a Microsoft digital signature and won't execute if modified.
- **AllSigned**—PowerShell will execute any script that has been digitally signed by using a code-signing certificate issued by a trusted certification authority (CA).
- **RemoteSigned**—PowerShell will execute any local script and will execute remote scripts if they've been digitally signed by using a code-signing certificate issued by a trusted CA. *Remote scripts* are those that exist on a remote computer, usually accessed by a Universal Naming Convention (UNC) path. Scripts marked as having come from the internet are also considered remote. Edge, Chrome, Firefox, and Outlook all mark downloads as having come from the internet.
- **Unrestricted**—All scripts will run.
- **Bypass**—This special setting is intended for use by application developers who are embedding PowerShell within their application. This setting bypasses the configured execution policy and should be used only when the hosting application is providing its own layer of script security. You're essentially telling PowerShell, "Don't worry. I have security covered."

### Wait, what?

Did you notice that you could set the execution policy in a Group Policy Object, but also override it by using a parameter of `PowerShell.exe`? What good is a GPO-controlled setting that people can easily override? This emphasizes that the execution policy is intended to protect only *uninformed* users from *unintentionally* running *anonymous* scripts.

The execution policy isn't intended to stop an informed user from doing anything intentional. It's not that kind of security setting.

In fact, a smart malware coder could as easily access the .NET Framework functionality directly, without going to the trouble of using PowerShell as a middleman. Or to put it another way, if an unauthorized user has admin rights to your computer and can run arbitrary code, you're already in trouble.

Microsoft recommends that you use `RemoteSigned` when you want to run scripts and that you use it only on computers where scripts must be executed. According to Microsoft, all other computers should be left at `Restricted`. They say that `RemoteSigned` provides a good balance between security and convenience. `AllSigned` is stricter but requires all of your scripts to be digitally signed. The PowerShell community as a whole is more divided, with a range of opinions on what a good execution policy is. For now,

we'll go with Microsoft's recommendation and let you explore the topic more on your own, if you wish.

**NOTE** Plenty of experts, including Microsoft's own "Scripting Guy," suggest using the `Unrestricted` setting for `ExecutionPolicy`. Their feeling is that the feature doesn't provide a layer of security, and you shouldn't give yourself false confidence that it's protecting you from anything.

## 4.2 *Not scripting, but running commands*

PowerShell, as its name indicates, is a *shell*. Other shells you've probably used or at least heard of include `cmd.exe`, `Bash`, `Zsh`, `fish`, and `ksh`. PowerShell is not only a shell but also a *scripting language*—but not in the way JavaScript or Python are.

With those languages, as with most programming languages, you sit down in front of a text editor or integrated development environment (IDE) and type a series of keywords to form a script. You save that file, and perhaps double-click it to test it. PowerShell can work like that, but that's not necessarily the main usage pattern for PowerShell, particularly when you're getting started. With PowerShell, you type a command, add a few parameters to customize the command's behavior, press Enter, and immediately see your results.

Eventually, you'll get tired of typing the same command (and its parameters) over and over again, so you'll copy and paste it all into a text file. Give that file a `.ps1` file-name extension, and you suddenly have a *PowerShell script*. Now, instead of typing the command over and over, you run that script, and it executes whatever commands are inside. It's typically far less complex than writing a program in a full programming language. In fact, it's a pattern similar to that used by UNIX administrators for years. Common UNIX/Linux shells, such as `Bash`, have a similar approach: run commands until you get them right, and then paste them into a text file and call it a *script*.

Don't get us wrong: You can get as complex as you need to with PowerShell. It does support the same kind of usage patterns as Python and other scripting or programming languages. PowerShell gives you access to the full underlying power of .NET Core, and we've seen PowerShell "scripts" that were practically indistinguishable from a C# program written in Visual Studio. PowerShell supports these different usage patterns because it's intended to be useful to a wide range of audiences. The point is that just because it *supports* that level of complexity doesn't mean you *have* to use it at that level, and it doesn't mean you can't be extremely effective with less complexity.

Here's an analogy. You probably drive a car. If you're like us, changing the oil is the most complex mechanical task you'll ever do with your car. We're not car geeks and can't rebuild an engine. We also can't do those cool, high-speed J-turns that you see in the movies. You'll never see us driving a car on a closed course in a car commercial. But the fact that we're not professional stunt drivers doesn't stop us from being extremely effective drivers at a less complex level. Someday we might decide to take up stunt driving for a hobby (our insurance companies will be thrilled), and at that point we'll need to learn a bit more about how our cars work, master some new skills,

and so on. The option is always there for us to grow into. But for now, we're happy with what we can accomplish as normal drivers.

For now, we'll stick with being normal "PowerShell drivers," operating the shell at a lower level of complexity. Believe it or not, users at this level are the primary target audience for PowerShell, and you'll find that you can do a lot of incredible stuff without going beyond this level. All you need to do is master the ability to run commands within the shell, and you're on your way.

### 4.3 The anatomy of a command

Figure 4.3 shows the basic anatomy of a complex PowerShell command. We call this the *full-form* syntax of a command. We're showing a somewhat complex command here, so you can see all of the things that might show up.

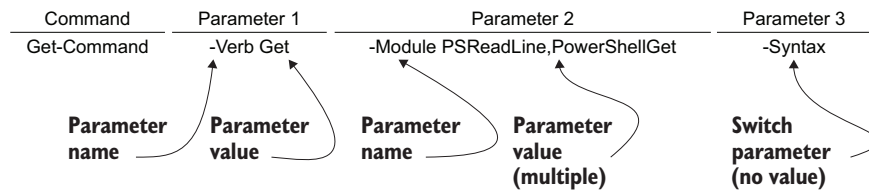


Figure 4.3 The anatomy of a PowerShell command

To make sure you're completely familiar with PowerShell's rules, let's cover each of the elements in the previous figure in more detail:

- The cmdlet name is `Get-Command`. PowerShell cmdlets always have this verb-noun naming format. We explain more about cmdlets in the next section.
- The first parameter name is `-Verb`, and it's being given the value `Get`. Because the value doesn't contain any spaces or punctuation, it doesn't need to be in quotation marks.
- The second parameter name is `-Module`, and it's being given two values: `PSReadLine` and `PowerShellGet`. These are in a comma-separated list, and because neither value contains spaces or punctuation, neither value needs to be inside quotation marks.
- The final parameter, `-Syntax`, is a switch parameter. That means it doesn't get a value; specifying the parameter is sufficient.
- Note that there's a mandatory space between the command name and the first parameter.
- Parameter names always start with a dash (`-`).
- There's a mandatory space after the parameter name and between the parameter's value and the next parameter name.
- There's no space between the dash (`-`) that precedes a parameter name and the parameter name itself.
- Nothing here is case sensitive.

Get used to these rules. Start being sensitive about accurate, neat typing. Paying attention to spaces and dashes and other rules will minimize the silly errors that PowerShell throws at you.

## 4.4 The *cmdlet* naming convention

First, let's discuss some terminology. As far as we know, we're the only ones who use this terminology in everyday conversation, but we do it consistently, so we may as well explain:

- A *cmdlet* is a native PowerShell command-line utility. These exist only inside PowerShell and are written in a .NET Core language such as C#. The word *cmdlet* is unique to PowerShell, so if you add it to your search keywords on your favorite search engine, the results you get back will be mainly PowerShell related. The word is pronounced *command-let*.
- A *function* can be similar to a cmdlet, but rather than being written in a .NET language, functions are written in PowerShell's own scripting language.
- An *application* is any kind of external executable, including command-line utilities such as ping and ipconfig.
- *Command* is the generic term that we use to refer to any or all of the preceding terms.

Microsoft has established a naming convention for cmdlets. That same naming convention *should* be used for functions, too, although Microsoft can't force anyone but its own employees to follow that rule.

The rule is this: Names start with a standard verb, such as Get or Set or New or Pause. You can run Get-Verb to see a list of approved verbs (you'll see about 100, although only about a dozen are common). After the verb is a dash, followed by a singular noun, such as Job or Process or Item. Developers get to make up their own nouns, so there's no *Get-Noun* cmdlet to display them all.

What's the big deal about this rule? Well, suppose we told you that there were cmdlets named Start-Job, Get-Job, Get-Process, Stop-Process, and so forth. Could you guess what command would start a new process on your machine? Could you guess what command would modify an Azure virtual machine (VM)? If you guessed Start-Process, you got the first one right. If you guessed Set-VM, you were close: it's Set-AzVM, and you'll find the command on Azure VMs in the Az.Compute module (we will go over modules in chapter 7). All the Azure commands use that same prefix of Az, followed by the noun that the command messes with. The point is that by having this consistent naming convention with a limited set of verbs, it becomes possible for you to guess at command names, and you could then use Help or Get-Command, along with wildcards, to validate your guess. It becomes easier for you to figure out the names of the commands you need, without having to run to Google or Bing every time.

**NOTE** Not all of the so-called verbs are really verbs. Although Microsoft officially uses the term *verb-noun naming convention*, you'll see "verbs" like New, Where, and so forth. You'll get used to it.

## 4.5 Aliases: Nicknames for commands

Although PowerShell command names can be nice and consistent, they can also be long. A command name such as `Remove-AzStorageTableStoredAccessPolicy` is a lot to type, even with tab completion. Although the command name is clear—looking at it, you can probably guess what it does—it's an *awful* lot to type.

That's where PowerShell aliases come in. An *alias* is nothing more than a nickname for a command. Tired of typing `Get-Process`? Try this:

```
PS /Users/james> Get-Alias -Definition "Get-Process"
Capability      Name
-----
Cmdlet          gps -> Get-Process
```

Now you know that `gps` is an alias for `Get-Process`.

When using an alias, the command works in the same way. Parameters are the same; everything is the same—the command name is just shorter.

If you're staring at an alias (folks on the internet tend to use them as if we've all memorized the hundreds of built-in aliases) and can't figure out what it is, ask help:

```
PS /Users/james> help gps
NAME
    Get-Process
SYNOPSIS
    Gets the processes that are running on the local computer.
SYNTAX
    Get-Process [[-Name] <String[]>] [-FileVersionInfo] [-Module]
    [<CommonParameters>]
    Get-Process [-FileVersionInfo] -Id <Int32[]> [-Module]
    [<CommonParameters>]
    Get-Process [-FileVersionInfo] -InputObject <Process[]> [-Module]
    [<CommonParameters>]
    Get-Process -Id <Int32[]> -IncludeUserName [<CommonParameters>]
    Get-Process [[-Name] <String[]>] -IncludeUserName [<CommonParameters>]
    Get-Process -IncludeUserName -InputObject <Process[]>
    ➡ [<CommonParameters>]
```

When asked for help about an alias, the help system will always display the help for the full command, which includes the command's complete name.

### Above and beyond

You can create your own aliases by using `New-Alias`, export a list of aliases by using `Export-Alias`, or even import a list of previously created aliases by using `Import-Alias`. When you create an alias, it lasts only as long as your current shell session. Once you close the window, it's gone. That's why you might want to export them, so that you can use them in another PowerShell session.



**(continued)**

We tend to avoid creating and using custom aliases, though, because they're not available to anyone but the person who made them. If someone can't look up what `xtcd` does, we're creating confusion and incompatibility.

And `xtcd` doesn't do anything. It's a fake alias we made up.

We must point out, because PowerShell is now available on non-Windows operating systems, that its concept of *alias* is a little different from an alias in, say, Bash. In Bash, an alias can be a kind of shortcut for running a command *that includes a bunch of parameters*. PowerShell doesn't behave that way. An alias is *only* a nickname for the command name, and the alias can't include any predetermined parameters.

## 4.6 **Taking shortcuts**

Here's where PowerShell gets tricky. We'd love to tell you that everything we've shown you so far is the only way to do things, but we'd be lying. And, unfortunately, you're going to be out on the internet stealing (well, repurposing) other people's examples, and you'll need to know what you're looking at.

In addition to aliases, which are shorter versions of command names, you can also take *shortcuts* with parameters. You have three ways to do this, each potentially more confusing than the last.

### 4.6.1 **Truncating parameter names**

PowerShell doesn't force you to type out entire parameter names. As you might recall from chapter 3, instead of typing `-ComputerName`, for example, you could go with `-comp`. The rule is that you have to type enough of the name for PowerShell to be able to distinguish it. If there's the `-ComputerName` parameter, the `-Common` parameter, and the `-Composite` parameter, you'd have to type at least `-compu`, `-comm`, and `-compo`, because that's the minimum number of letters necessary to uniquely identify each.

If you must take shortcuts, this isn't a bad one to take, if you can remember to press Tab after typing that minimum-length parameter so that PowerShell can finish typing the rest of it for you.

### 4.6.2 **Using parameter name aliases**

Parameters can also have their own aliases, although they can be terribly difficult to find, as they aren't displayed in the help files or anyplace else convenient. For example, the `Get-Process` command has the `-ErrorAction` parameter. To discover its aliases, you run this command:

```
PS /Users/james> (get-command get-process | select -Expand
➡ parameters).erroraction.aliases
```

We've boldfaced the command and parameter names; replace these with whatever command and parameter you're curious about. In this case, the output reveals that `-ea` is an alias for `-ErrorAction`, so you could run this:

```
PS /Users/james> Get-Process -ea Stop
```

Tab completion will show you the `-ea` alias; if you typed `Get-Process -e` and started pressing Tab, it'd show up. But the help for the command doesn't display `-ea` at all, and tab completion doesn't indicate that `-ea` and `-ErrorAction` are the same thing.

**NOTE** These are called *common parameters*. You can run the command `Get-Help about_CommonParameters` to read more about them.

### 4.6.3 Using positional parameters

When you're looking at a command's syntax in its help file, you can spot positional parameters easily:

```
SYNTAX
Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [-Include
    <string[]>] [-Exclude <string[]>] [-Recurse] [-De
    pth <uint>] [-Force] [-Name] [-Attributes {ReadOnly | Hidden | System |
    Directory | Archive | Device | Normal | Tem
    porary | SparseFile | ReparsePoint | Compressed | Offline |
    NotContentIndexed | Encrypted | IntegrityStream | NoScr
    ubData}] [-FollowSymlink] [-Directory] [-File] [-Hidden] [-ReadOnly]
[-System] [<CommonParameters>]
```

Here, both `-Path` and `-Filter` are positional, and you know that because the parameter name and the accepted input is contained within square brackets. A clearer explanation is available in the full help (`help Get-ChildItem -Full`, in this case), which looks like this:

```
-Path <String[]>
    Specifies a path to one or more locations. Wildcards are
    permitted. The default location is the current directory (.).
Required?                                false
Position?                                0
Default value                             Current directory
Accept pipeline input?                    true (ByValue, ByPropertyName)
Accept wildcard characters?               True
```

That's a clear indication that the `-Path` parameter is in position 0, which means it's the first parameter after the cmdlet. For positional parameters, you don't have to type the parameter name; you can provide its value in the correct position. For example:

```
PS /Users/james> Get-ChildItem /Users
Directory: /Users

Mode                LastWriteTime         Length Name
----                -
d-----          3/27/2016  11:20 AM              james
d-r--          2/18/2016   2:06 AM              Shared
```

That's the same as this:

```
PS /Users/james> Get-ChildItem -Path /Users
    Directory: /Users
Mode                LastWriteTime         Length Name
----                -
d-----          3/27/2019  11:20 AM             james
d-----          2/18/2019   2:06 AM             Shared
```

The problem with positional parameters is that you're taking on the responsibility of remembering what goes where. You must type all positional parameters first, in the correct order, before you can add any named (nonpositional) parameters. If you mix up the parameter order, the command fails. For simple commands such as `Dir`, which you've probably used for years, typing `-Path` feels weird, and almost nobody does it. But for more-complex commands, which might have three or four positional parameters in a row, it can be tough to remember what goes where.

For example, this is a bit difficult to read and interpret:

```
PS /Users/james> move file.txt /Users/james/
```

This version, which uses full cmdlet name and parameter names, is easier to follow:

```
PS /Users/james> move-item -Path /tmp/file.txt -Destination /Users/james/
```

This version, which puts the parameters in a different order, is allowed when you use the parameter names:

```
PS /Users/james> move -Destination /Users/james/ -Path /tmp/file.txt
```

We tend to recommend against using positional (unnamed) parameters unless you're banging out something quick and dirty at the command line. In anything that will persist, such as a PowerShell script file or a blog post, include all of the parameter names. We do that as much as possible in this book, except in a few instances where we have to shorten the command line to make it fit within the printed page.

## 4.7 *Support for external commands*

So far, all of the commands you've run in the shell (at least the ones we've suggested that you run) have been built-in cmdlets. Over 2,900 cmdlets come built into PowerShell on your Windows machine and over 200 on your Linux or macOS machine. You can add more—products such as Azure PowerShell, AWS PowerShell, and SQL Server all come with add-ins that include hundreds of additional cmdlets.

But you're not limited to PowerShell cmdlets. You can also use the same external command-line utilities that you've probably been using for years, including `ping`, `nslookup`, `ifconfig` or `ipconfig`, and so forth. Because these aren't native PowerShell cmdlets, you use them the same way that you always have. Go ahead and try a few old favorites right now.

It's the same story on non-Windows operating systems. You can use `grep`, `bash`, `sed`, `awk`, `ping`, and whatever other existing command-line tools you may have. They'll run normally, and PowerShell will display their results the same way that your old shell (e.g., `Bash`) would have.

**TRY IT NOW** Try running some external command-line utilities that you've used previously. Do they work the same? Do any of them fail?

This section illustrates an important lesson: with PowerShell, Microsoft (perhaps for the first time ever) isn't saying, "You have to start over and learn everything all over again." Instead, Microsoft is saying, "If you already know how to do something, keep doing it that way. We'll try to provide you with better and more complete tools going forward, but what you already know will still work."

In some instances, Microsoft has provided better tools than some of the existing, older ones. For example, the native `Test-Connection` cmdlet provides more options and more-flexible output than the old, external `ping` command. But if you know how to use `ping`, and it's solving whatever need you have, go right on using it. It'll work fine from within PowerShell.

All that said, we do have to deliver a harsh truth: not every external command will work flawlessly from within PowerShell, at least not without a little tweaking on your part. That's because PowerShell's parser—the bit of the shell that reads what you've typed and tries to figure out what you want the shell to do—doesn't always guess correctly. Sometimes you'll type an external command, and PowerShell will mess up, start spitting out errors, and generally not work.

For example, things can get tricky when an external command has a lot of parameters—that's where you'll see PowerShell break the most. We won't dive into the details of why it works, but here's a way to run a command that ensures that its parameters will work properly:

```
$exe = "func"
$action = "new"
$language = "powershell"
$template = "HttpTrigger"
$name = "myFunc"
& $exe $action -l $language -t $template -n $name
```

This supposes that you have an external command named `func`. (This real-life command-line utility is used to interact with Azure Functions.) If you've never used it or don't have it, that's fine; most old-school command-line utilities work the same way, so this is still a good teaching example. It accepts several parameters:

- `"new"` here is the action you want to take, and `-new`, `init`, `start`, and `logs` are the options.
- `-l` is for the language you want the function to be.
- `-t` is for the template you want to use.
- `-n` is for the name of the function.

What we've done is put all the various elements—the executable path and name, as well as all of the parameter values—into placeholders, which start with the `$` character. That forces PowerShell to treat those values as single units, rather than trying to parse them to see whether any contain commands or special characters. Then we used the invocation operator (`&`), passing it the executable name, all of the parameters, and the parameters' values. That pattern will work for almost any command-line utility that's being grumpy about running within PowerShell.

Sound complicated? Well, here's some good news: In PowerShell v3 and later, you don't have to mess around so much. Just add two dashes and a percent symbol in front of anything, and PowerShell won't even try to parse it; it'll just pass it right to the command-line utility that you're using. To be absolutely clear, this means you won't be able to pass variables as parameter values.

Here's a quick example of what will fail:

```
PS /Users/james> $name = "MyFunctionApp"
PS /Users/james> func azure functionapp list-functions --% $name

Can't find app with name "$name"
```

We tried to run the command-line utility `func` to list all of our Azure functions with the name "MyFunctionApp", but if we explicitly state what we want, PowerShell will pass all the parameters to the underlying command without trying to do anything with them:

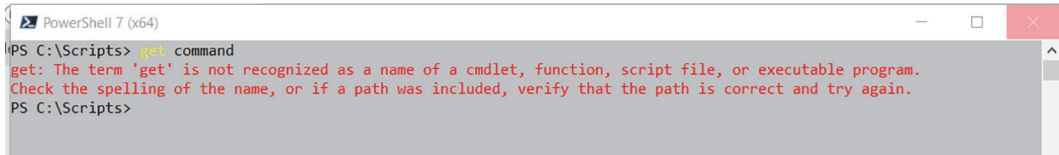
```
PS /Users/james> func new -t HttpTrigger -n --% "MyFunc"
Select a template: HttpTrigger
Function name: [HttpTrigger] Writing /Users/tyler/MyFuncApp/MyFunc/run.ps1
Writing /Users/tyler/MyFuncApp/MyFunc/function.json
The function "MyFunc" was created successfully from the "HttpTrigger"
➡ template.
PS /Users/james>
```

Hopefully this isn't something you'll need to do often.

## 4.8 *Dealing with errors*

It's inevitable that you'll see some ugly red text as you start working with PowerShell—and probably from time to time even after you're an expert-level shell user. Happens to us all. But don't let the red text stress you out. (Personally, it takes us back to high school English class and poorly written essays, so *stress* is putting it mildly.)

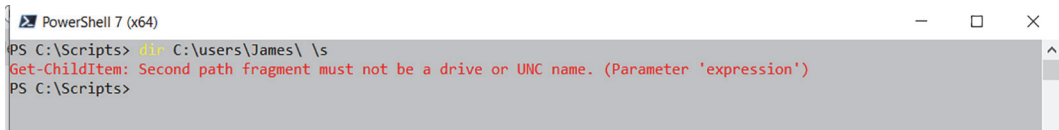
The alarming red text aside, PowerShell's error messages have improved greatly over the years (a lot of this has to do with the fact that the error messages are also open sourced). For example, as shown in figure 4.4, they try to show you exactly where PowerShell ran into trouble.



```
PowerShell 7 (x64)
PS C:\Scripts> get command
get: The term 'get' is not recognized as a name of a cmdlet, function, script file, or executable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
PS C:\Scripts>
```

Figure 4.4 Interpreting a PowerShell error message

Most error messages are easy to understand. In figure 4.4, right at the beginning, it's saying, "You typed `get`, and I have no idea what that means." That's because we typed the command name wrong: it's supposed to be `Get-Command`, not `Get Command`. Oops. What about figure 4.5?



```
PowerShell 7 (x64)
PS C:\Scripts> dir C:\users\James\ \s
Get-ChildItem: Second path fragment must not be a drive or UNC name. (Parameter 'expression')
PS C:\Scripts>
```

Figure 4.5 What's a "second path fragment"?

The error message in figure 4.5, "Second path fragment must not be a drive or UNC name," is confusing. What second path? We didn't type a second path. We typed one path, `C:\Users\James` and a command-line parameter, `\s`. Right?

Well, no. One of the easiest ways to solve this kind of problem is to read the help and to type the command out completely. If we'd typed `Get-ChildItem -Path C:\Users\james`, we'd have realized that `\s` isn't the correct syntax. We meant `-Recurse`. Sometimes the error message might not seem helpful—and if it seems like you and PowerShell are speaking different languages, you are. PowerShell obviously isn't going to change its language, so you're probably the one in the wrong, and consulting the help and spelling out the entire command, parameters and all, is often the quickest way to solve the problem.

## 4.9 Common points of confusion

Whenever it seems appropriate, we wrap up each chapter with a brief section that covers some of the common mistakes we see. The idea is to help you see what most often confuses other administrators like yourself and to avoid those problems—or at least to be able to find a solution for them—as you start working with the shell.

### 4.9.1 Typing cmdlet names

First up is the typing of cmdlet names. It's always verb-noun, like `Get-Content`. All of these are options we'll see newcomers try, but they won't work:

- `Get Content`
- `GetContent`
- `Get=Content`
- `Get_Content`

Part of the problem comes from typos (e.g., `=` instead of `-`) and part from verbal laziness. We all pronounce the command as *Get Content*, verbally omitting the dash. But you have to type the dash.

#### 4.9.2 *Typing parameters*

Parameters are also consistently written. Parameters, such as `-Recurse`, get a dash before their name. If the parameter has a value, a space will come between the parameter name and its value. You need to have spaces separating the cmdlet name from its parameters, and the parameters from each other. The following are correct:

- `123`
- `Dir -rec` (the shortened parameter name is fine)
- `New-PSDrive -name DEMO -psprovider FileSystem -root \\Server\Share`

But these examples are all incorrect:

- `Dir-rec` (no space between alias and parameter)
- `New-PSDrive -nameDEMO` (no space between parameter name and value)
- `New-PSDrive -name DEMO-psprovider FileSystem` (no space between the first parameter's value and the second parameter's name)

PowerShell isn't normally picky about upper- and lowercase, meaning that `dir` and `DIR` are the same, as are `-RECURSE` and `-recurse` and `-Recurse`. But the shell sure is picky about those spaces and dashes.

### 4.10 *Lab*

**NOTE** For this lab, you need PowerShell v7 or later running on Windows, macOS, or Linux.

Using what you learned in this chapter, and in the previous chapter on using the help system, complete the following tasks in PowerShell:

- 1 Display a list of running processes.
- 2 Test the connection to `google.com` or `bing.com` without using an external command like `ping`.
- 3 Display a list of all commands that are of the cmdlet type. (This is tricky—we've shown you `Get-Command`, but you need to read the help to find out how to narrow down the list as we've asked.)
- 4 Display a list of all aliases.
- 5 Make a new alias, so you can run `ntst` to run `netstat` from a PowerShell prompt.

- 6 Display a list of processes that begin with the letter *p*. Again, read the help for the necessary command—and don't forget that the asterisk (\*) is a near-universal wildcard in PowerShell.
- 7 Make a new folder (aka directory) using the `New-Item` cmdlet with the name of `MyFolder1`. Then do it again and call it `MyFolder2`. Use `Help` if you're not familiar with `New-Item`.
- 8 Remove the folders from step 7 in a single command. Use `Get-Command` to find a similar cmdlet to what we used in step 7—and don't forget that the asterisk (\*) is a near-universal wildcard in PowerShell.

We hope these tasks seem straightforward to you. If so—excellent. You're taking advantage of your existing command-line skills to make PowerShell perform a few practical tasks for you. If you're new to the command-line world, these tasks are a good introduction to what you'll be doing in the rest of this book.

## 4.11 Lab answers

- 1 `Get-Process`
- 2 `Test-Connection google.com`
- 3 `Get-Command -Type cmdlet`
- 4 `Get-Alias`
- 5 `New-Alias -Name ntst -Value netstat`
- 6 `Get-Process -Name p*`
- 7 `New-Item -Name MyFolder1 -Path c:\scripts -Type Directory;`  
`New-Item -Name MyFolder2 -Path c:\scripts -Type Directory`
- 8 `Remove-item C:\Scripts\MyFolder*`