3 Using the help system

In chapter 1, we mentioned that discoverability is a key feature that makes graphical user interfaces (GUIs) easier to learn and use and that command-line interfaces (CLIs) like PowerShell are often more difficult because they lack those discoverability features. In fact, PowerShell has fantastic discoverability features—but they're not that obvious. One of the main discoverability features is its help system.

3.1 The help system: How you discover commands

Bear with us for a minute as we climb up on a soapbox and preach to you. We work in an industry that doesn't place a lot of emphasis on reading, although we do have an acronym that we cleverly pass along to users when we wish *they* would *read the friendly manual*—RTFM. Most administrators tend to dive right in, relying on things like tooltips, context menus, and so forth—those *GUI* discoverability tools—to figure out how to do something. That's how we often work, and we imagine you do the same thing. But let's be clear about one point:

> If you aren't willing to read PowerShell's help files, you won't be effective with PowerShell. You won't learn how to use it; you won't learn how to administer other services like Azure, AWS, Microsoft 365, and so on, with it; and you might as well stick with the GUI.

That's about as clear as we can be. It's a blunt statement, but it's absolutely true. Imagine trying to figure out Azure Virtual Machines, or any other administrative portal, without the help of tooltips, menus, and context menus. Trying to learn to use PowerShell without taking the time to read and understand the help files is the same thing. It's like trying to assemble that do-it-yourself furniture from the department store without reading the manual. Your experience will be frustrating, confusing, and ineffective. So why do it?

If you need to perform a task and don't know what command to use, the help system is how you'll find that command. Start with the help system before going to your favorite search engine.

If you run a command and get an error, the help system is what will show you how to properly run the command so you don't get errors. If you want to link multiple commands together to perform a complex task, the help system is where you'll find out how each command is able to connect to others. You don't need to search for examples on Google or Bing; you need to learn how to use the commands themselves so that you can create your own examples and solutions.

We realize our preaching is a little heavy handed, but 90% of the problems we see users struggling with in forums could be solved if those folks would find a few minutes to sit back, take some deep breaths, and read the help. And then read this chapter, which is all about helping you understand the help you're reading in PowerShell.

From here on out, we encourage you to read the help for several more reasons:

- Although we show many commands in our examples, we almost never expose the complete functionality, options, and capabilities of each command to make understanding the concept easier to grasp. You should read the help for each and every command we show you so that you'll be familiar with the additional actions each command can accomplish.
- In the labs, we may give you a hint about which command to use for a task, but we won't give you hints about the syntax. You'll need to use the help system to discover that syntax on your own in order to complete the labs.
- We promise you that mastering the help system is the key to becoming a PowerShell expert. No, you won't find every little detail in there, and a lot of super-advanced material isn't documented in the help system, but in terms of being an effective day-to-day administrator, you need to master the help system. This book will make that system understandable, and it will teach you the concepts that the help skips over, but it'll do this only in conjunction with the built-in help.

Stepping off the soapbox now.

Command vs. cmdlet

PowerShell contains many types of executable commands. Some are called *cmdlets* (we will cover cmdlets in the next chapter), some are called *functions*, and so on. Collectively, they're all *commands*, and the help system works with all of them. A cmdlet is something unique to PowerShell, and many of the commands you run will be cmdlets. But we'll try to consistently use *command* whenever we're talking about the more general class of executable utility.

3.2 Updatable help

You may be surprised the first time you fire up help in PowerShell, because, well, there isn't much there. But wait, we can explain.

Microsoft included a new feature beginning with PowerShell v3 called *updatable help*. PowerShell can download updated, corrected, and expanded help right from the internet. Initially when you ask for help on a command, you get an abbreviated, auto-generated version of help, along with a message on how to update the help files, which may look like the following:

```
PS /User/travisp> help Get-Process
NAME
    Get-Process

SYNTAX
    Get-Process [[-Name] <string[]>] [-Module] [-FileVersionInfo]
    [<CommonParameters>]

    Get-Process [[-Name] <string[]>] -IncludeUserName [<CommonParameters>]

    Get-Process -Id <int[]> -IncludeUserName [<CommonParameters>]

    Get-Process -Id <int[]> [-Module] [-FileVersionInfo] [<CommonParameters>]

    Get-Process -InputObject <Process[]> [-Module] [-FileVersionInfo]
    [<CommonParameters>]

    Get-Process -InputObject <Process[]> -IncludeUserName
    ➥ [<CommonParameters>]


ALIASES
    gps


REMARKS
    Get-Help cannot find the Help files for this cmdlet on this computer. It
    is displaying only partial help.
        -- To download and install Help files for the module that includes
    this cmdlet, use Update-Help.
        -- To view the Help topic for this cmdlet online, type: "Get-Help
    Get-Process -Online" or
           go to https://go.microsoft.com/fwlink/?LinkID=113324.
```

TIP It's impossible to miss the fact that you don't have local help installed. The first time you ask for help, PowerShell will prompt you to update the help content.

Updating PowerShell's help should be your first task. In Windows PowerShell, you were required to update help as "Administrator" or the equivalent of "root." In PowerShell 6 and higher, you can now update help as the current user. Open PowerShell and run `Update-Help`, and you'll be good to go in a few minutes.

TIP If you are not running the `en-us` culture, you may have to specify `-UICulture en-US` to get `Update-Help` to work.

It's important to get in the habit of updating the help every month or so. PowerShell can even download updated help for non-Microsoft commands, provided the commands' modules are located in the proper spot and that they've been coded to include the online location for updated help (modules are how commands are added to PowerShell, and they are explained in chapter 7).

Do you have computers that aren't connected to the internet? No problem: go to one that's connected, and use `Save-Help` to get a local copy of the help. Put it on a file server or somewhere that's accessible to the rest of your network. Then run `Update-Help` with its `-Source` parameter, pointing it to the downloaded copy of the help. That'll let any computer on your network grab the updated help from that central spot, rather than from the internet.

Help is open sourced

Microsoft's PowerShell help files are open source materials that are available at https://github.com/MicrosoftDocs/PowerShell-Docs. This can be a good place to see the latest source, which might not yet be compiled into help files that PowerShell can download and display.

3.3 Asking for help

PowerShell provides a cmdlet, `Get-Help`, that accesses the help system. You may see examples (especially on the internet) that show people using the `Help` keyword instead. The `Help` keyword isn't a native cmdlet at all; it is a *function*, which is a wrapper around the core `Get-Help` cmdlet.

Help on macOS/Linux

The help files, when viewed on macOS and Linux, are displayed using the operating system's traditional *man* (manual) feature, which usually "takes over" the screen to display the help, returning you to your normal screen when you're finished.

`Help` works much like the base `Get-Help`, but it pipes the `Help` output to `less`, allowing you to have a nice paged view instead of seeing all the help scroll by at once. Running `Help Get-Content` and `Get-Help Get-Content` produces the same results, but the former has a page-at-a-time display. You could run `Get-Help Get-Content | less` to produce that paged display, but that requires a lot more typing. We typically use only `Help`, but we want you to understand that some trickery is going on under the hood.

By the way, sometimes that paginated display can be annoying, because you have the information you need, but it still wants you to press the spacebar to display the

remaining information. If you encounter this, press `q` to cancel the command and return to the shell prompt. When using `less`, `q` always means *quit*.

The help system has two main goals: to help you find commands to perform specific tasks and to help you learn how to use those commands after you've found them.

3.4 Using help to find commands

Technically speaking, the help system has no idea what commands are present in the shell. All it knows is what help topics are available, and it's possible for commands to not have a help file, in which case the help system won't know that the commands exist. Fortunately, Microsoft ships a help topic for nearly every cmdlet it produces, which means you usually won't find a difference. In addition, the help system can access information that isn't related to a specific cmdlet, including background concepts and other general information.

Like most commands, `Get-Help` (and, therefore, `Help`) has several parameters. One of those—perhaps the most important one—is `-Name`. This parameter specifies the name of the help topic you'd like to access, and it's a positional parameter, so you don't have to type `-Name`; you can just provide the name you're looking for. It also accepts wildcards, which makes the help system useful for discovering commands.

For example, suppose you want to do something with events on .NET objects. You don't know what commands might be available, and you decide to search for help topics that cover events. You might run either of these two commands:

```
Help *event*
Help *object*
```

The first command returns a list like the following on your computer:

```
Name                  Category ModuleName

Get-Event             Cmdlet   Microsoft.PowerShell.Utility
Get-EventSubscriber   Cmdlet   Microsoft.PowerShell.Utility
New-Event             Cmdlet   Microsoft.PowerShell.Utility
Register-EngineEvent  Cmdlet   Microsoft.PowerShell.Utility
Register-ObjectEvent  Cmdlet   Microsoft.PowerShell.Utility
Remove-Event          Cmdlet   Microsoft.PowerShell.Utility
Unregister-Event      Cmdlet   Microsoft.PowerShell.Utility
Wait-Event            Cmdlet   Microsoft.PowerShell.Utility
```

NOTE After you have installed some modules from other sources, you may notice that the command help list includes commands (and functions) from modules such as `Az.EventGrid` and `Az.EventHub`. The help system displays all of these even though you haven't loaded those modules into memory yet, which helps you discover commands on your computer that you might otherwise have overlooked. It'll discover commands from any modules that are installed in the proper location, which we'll discuss in chapter 7.

Many of the cmdlets in the previous list seem to have something to do with events. In your environment, you may also have additional commands that are unrelated or you may have "about" topics, which provide background information (discussed in detail in section 3.6). When you are using the help system to find a PowerShell command, try to search using the broadest term possible—`*event*` or `*object*` as opposed to `*objectevent*`—because you'll get the most results possible.

When you have a cmdlet that you think will do the job (`Register-ObjectEvent` looks like a good candidate for what you're after in the example), you can ask for help on that specific topic:

```
Help Register-ObjectEvent
```

Don't forget about tab completion! As a reminder, it lets you type a portion of a command name and press Tab, and the shell completes what you've typed with the closest match. You can continue pressing Tab to get a list of alternative matches.

> **TRY IT NOW** Type `Help Register-` and press Tab. This will match a few commands and not complete. On a Windows machine, when you press Tab a second time, it will keep scrolling through the available commands. On a non-Windows machine, if you press Tab, a second tab will display a list of the available commands.

You can also use wildcards with `Help`—mainly the `*` wildcard, which stands in for zero or more characters. If PowerShell finds only one match to whatever you've typed, it won't display a list of topics for that one item. Instead, it'll display the content for that item.

> **TRY IT NOW** Run `Help Get-EventS*`, and you should see the help file for `Get-EventSubscriber`, rather than a list of matching help topics.

If you've been following along in the shell, you should now be looking at the help file for `Get-EventSubscriber`. This file, called the *summary help*, is meant to be a short description of the command and a reminder of the syntax. This information is useful when you need to quickly refresh your memory of a command's usage, and it's where we'll begin interpreting the help file itself.

Above and beyond

Sometimes we want to share information that, although nice, isn't essential to your understanding of the shell. We put that information into an "Above and beyond" sidebar, like this one. If you skip these, you'll be fine; if you read them, you'll often learn about an alternative way of doing something or get additional insight into PowerShell.

We mentioned that the `Help` command doesn't search for cmdlets; it searches for help topics. Because every cmdlet has a help file, we could say that this search retrieves the same results. But you can also directly search for cmdlets by using the `Get-Command` cmdlet (or its alias, `gcm`).

Like the `Help` cmdlet, `Get-Command` accepts wildcards—so you can, for example, run `gcm *get*` to see all of the commands that contain *get* in their name. For better or worse, that list will include not only cmdlets, but also external commands such as `wget`, which may not be useful.

A better approach is to use the `-Noun` or `-Verb` parameters. Because only command names have nouns and verbs, the results will be limited to cmdlets. `Get-Command -Noun *event*` returns a list of cmdlets dealing with events; `Get-Command -Verb Get` returns all cmdlets capable of retrieving things. You can also use the `-CommandType` parameter, specifying a type of cmdlet: `Get-Command *event* -Type cmdlet` shows a list of all cmdlets that include *event* in their names, and the list won't include any external applications or commands.

3.5 Interpreting the help

PowerShell's cmdlet help files have a particular set of conventions. Learning to understand what you're looking at is the key to extracting the maximum amount of information from these files and to learning to use the cmdlets themselves more effectively.

3.5.1 Parameter sets and common parameters

Most commands can work in a variety of ways, depending on what you need them to do. For example, here's the syntax section for the `Get-Item` help:

```
SYNTAX
    Get-Item [-Stream <String[]>] [-Credential <PSCredential>] [-Exclude
    ➥ <String[]>] [-Filter <String>] [-Force] [-Include
    <String[]>] -LiteralPath <String[]> [<CommonParameters>]

    Get-Item [-Path] <String[]> [-Stream <String[]>] [-Credential
    ➥ <PSCredential>] [-Exclude <String[]>] [-Filter <String>] [-Force]
    [-Include <String[]>] [<CommonParameters>]
```

Notice that the command in the previous syntax is listed twice, which indicates that the command supports two *parameter sets*; you can use the command in two distinct ways. Some of the parameters will be shared between the two sets. You'll notice, for example, that both parameter sets include the `-Filter` parameter. But the two parameter sets will always have at least one unique parameter that exists only in that parameter set. In this case, the first set supports `-LiteralPath`, which is not included in the second set; the second set contains the `-Path` parameter, which isn't included in the first set, but both could contain additional unshared parameters.

Here's how this works: If you use a parameter that's included in only one set, you're locked into that set and can use only additional parameters that appear within that same set. If you choose to use `-LiteralPath`, you cannot use parameters from the other set, in this case `-Path`, because it doesn't live in the first parameter set. That means `-Path` and `-LiteralPath` are *mutually exclusive*—you'll never use both of them at the same time because they live in different parameter sets.

Sometimes it's possible to run a command with only parameters that are shared between multiple sets. In those cases, the shell will usually select the first-listed parameter set. Because each parameter set implies different behavior, it's important to understand which parameter set you're running.

You'll notice that every parameter set for every PowerShell cmdlet ends with `[<CommonParameters>]`. This refers to a set of 11 (at the time this was written) parameters that are available on every single cmdlet, no matter how you're using that cmdlet. We'll discuss some of those common parameters later in this book, when we'll use them for a real task. Later in this chapter, though, we'll show you where to learn more about those common parameters, if you're interested.

> **NOTE** Astute readers will by now have recognized variations in some of our examples. Readers will notice a different help layout for `Get-Item` depending on their version of PowerShell. You might even see a few new parameters. But the fundamentals and concepts we're explaining haven't changed. Don't get hung up on the fact that the help you see might be different from what we show in the book.

3.5.2 Optional and mandatory parameters

You don't need every single parameter in order to make a cmdlet run. PowerShell's help lists optional parameters in square brackets. For example, `[-Credential <PSCredential>]` indicates that the entire `-Credential` parameter is optional. You don't have to use it at all; the cmdlet will probably default to the current user's default credentials if you don't specify an alternative credential using this parameter. That's also why `[<-Common-Parameters>]` is in square brackets: you can run the command without using any of the common parameters.

Almost every cmdlet has at least one optional parameter. You may never need to use some of these parameters and may use others daily. Keep in mind that when you choose to use a parameter, you must type only enough of the parameter name so that PowerShell can unambiguously figure out which parameter you mean. For example, `-F` wouldn't be sufficient for `-Force`, because `-F` could also mean `-Filter`. But `-Fo` would be a legal abbreviation for `-Force`, because no other parameter starts with `-Fo`.

What if you try to run a command and forget one of the mandatory parameters? Take a look at the help for `Get-Item`, for example, and you'll see that the `-Path` parameter is mandatory. You can tell because the entire parameter—its name and its value—isn't surrounded by square brackets. This means that an optional parameter can be identified because the entire parameter and its value will be surrounded by square brackets. Try running `Get-Item` without specifying a file path (figure 3.1).

```
Get-Item [-Path] <System.String[]> [-Stream <System.String[]>] [-Credential
<System.Management.Automation.PSCredential>] [-Exclude <System.String[]>] [-Filter <System.String>] [-Force]
[-Include <System.String[]>] [<CommonParameters>]
```

Figure 3.1 This is help from Get-Item showing that the path variable accepts an array of strings indicated by the square brackets [ ].

TRY IT NOW Follow along in this example by running `Get-Item` without any parameters.

PowerShell should have prompted you for the mandatory `-Path` parameter. If you type something like `~` or `./` and press Enter, the command will run correctly. You could also press Ctrl-C to abort the command.

3.5.3 Positional parameters

PowerShell's designers knew that some parameters would be used so frequently that you wouldn't want to continually type the parameter names. Those commonly used parameters are often *positional*: you can provide a value without typing the parameter's name, provided you put that value in the correct position. You can identify positional parameters in two ways: via the syntax summary or the full help.

FINDING POSITIONAL PARAMETERS IN THE SYNTAX SUMMARY

You'll find the first way in the syntax summary: the parameter name—only the name—will be surrounded by square brackets. For example, look at the first two parameters in the second parameter set of `Get-Item`:

```
[-Path] <String[]> [-Stream <String[]>]…[-Filter <String>]
```

The first parameter, `-Path`, isn't optional. You can tell because the entire parameter—its name and its value isn't surrounded by square brackets. But the parameter name is enclosed in square brackets, making it a positional parameter—you could provide the log name without having to type `-Path`. And because this parameter appears in the first position within the help file, you know that the log name is the first parameter you have to provide.

The second parameter, `-Stream`, is optional; both it and its value are enclosed in square brackets. Within those, `-Stream` itself is not contained in a second set of square brackets, indicating that this is not a positional parameter. If it were positional, it would look like `[[-Stream] <string[]>]`. So, you'd need to use the parameter name to provide the value.

The `-Filter` parameter (which comes later in the syntax; run `Help Get-Item` and find it for yourself) is optional, because it's entirely enclosed within square brackets. The `-Filter` name is in square brackets, which tells you that if you choose to use that parameter, you must type the parameter name (or at least a portion of it). There are some tricks to using positional parameters:

- It's okay to mix and match positional parameters with those that require their names. Positional parameters must always be in the correct positions. For example, `Get-Item ~ -Filter *` is legal: `~` will be fed to the `-Path` parameter because that value is in the first position, and `*` will go with the `-Filter` parameter because the parameter name was used.
- It's always legal to specify parameter names, and when you do so, the order in which you type them isn't important. `Get-Item -Filter * -Pa *` is legal because we've used parameter names (in the case of `-Path`, we abbreviated it).

> **NOTE** Some commands, such as `Get-ChildItem`, have multiple positional parameters. The first is `-Path` and then `-Filter`. If you use multiple positional parameters, don't lose track of their positions. `Get-ChildItem ~ Down*` will work, with `~` being attached to `-Path` and `Down*` being attached to `-Filter`. `Get-ChildItem Down* ~` won't get anything, because `~` will be attached to `-Filter`, and most likely no items will match.

We'll offer a best practice: Use parameter names until you become comfortable with a particular cmdlet and get tired of typing a commonly used parameter name over and over. After that, use positional parameters to save yourself typing. When the time comes to paste a command into a text file for easier reuse, always use the full cmdlet name and type out the complete parameter name—no positional parameters and no abbreviated parameter names. Doing so makes that file easier to read and understand in the future, and because you won't have to type the parameter names repeatedly (that's why you pasted the command into a file, after all), you won't be creating extra typing work for yourself.

FINDING POSITIONAL PARAMETERS IN THE FULL HELP

We said that you can locate positional parameters in two ways. The second requires that you open the help file by using the `-Full` parameter of the `Help` command.

> **TRY IT NOW** Run `Help Get-Item -Full`. Remember to use the spacebar to view the help file one page at a time and to press Ctrl-C if you want to stop viewing the file before reaching the end. For now, page through the entire file, which lets you scroll back and review it all. Also, instead of `-Full`, try using the `-Online` parameter, which should work on any client computer or server with a browser. Be aware that the success of using `-Online` depends on the quality of the underlying help file. If the file is malformed, you might not see everything.

Page down until you see the help entry for the `-Path` parameter. It should look something like figure 3.2.

```
-Path <System.String[]>
    Specifies the path to an item. This cmdlet gets the item at the specified location. Wildcard characters are
    permitted. This parameter is required, but the parameter name Path is optional.

    Use a dot (`.`) to specify the current location. Use the wildcard character (`*`) to specify all the items in
    the current location.

    Required?                    true
    Position?                    0
    Default value                None
    Accept pipeline input?       True (ByPropertyName, ByValue)
    Accept wildcard characters?  true
```

Figure 3.2 Segment from help for `Get-Item` showing that the `-path` variable is required

In the preceding example, you can see that this is a positional parameter, and it occurs in the first position, right after the cmdlet name, based on the 0 position index.

We always encourage students to focus on reading the full help when they're getting started with a cmdlet, rather than only the abbreviated syntax reminder. Reading the help reveals more details, including that description of the parameter's use. You can also see that this parameter does accept wildcards, which means you can provide a value like `Down*`. You don't need to type out the name of an item, such as the Downloads folder.

3.5.4 Parameter values

The help files also give you clues about the kind of input that each parameter accepts. Most parameters expect some kind of input value, which will always follow the parameter name and be separated from the parameter name by a space (not by a colon, equal sign, or any other character, although you might encounter exceptions from time to time). In the abbreviated syntax, the type of input expected is shown in angle brackets, like `< >`:

```
-Filter <String>
```

It's shown the same way in the full syntax:

```
-Filter <String>
    Specifies a filter in the format or language of the provider. The
 value of this parameter qualifies the Path parameter.

    The syntax of the filter, including the use of wildcard characters,
 depends on the provider. Filters are more efficient than
    other parameters, because the provider applies them when the cmdlet
 gets the objects rather than having PowerShell filter
    the objects after they are retrieved.

    Required?                    false
    Position?                    named
    Default value                None
    Accept pipeline input?       False
    Accept wildcard characters?  true
```

Let's look at some common types of input:

- `String`—A series of letters and numbers. These can sometimes include spaces, but when they do, the entire string must be contained within quotation marks. For example, a string value such as `/usr/bin` doesn't need to be enclosed in quotes, but `~/book samples` does, because it has that space in the middle. For now, you can use single or double quotation marks interchangeably, but it's best to stick with single quotes.
- `Int`, `Int32`, *or* `Int64`—An integer number (a whole number with no decimal portion).
- `DateTime`—Generally, a string that can be interpreted as a date based on your computer's regional settings. In the United States, that's usually something like `10-10-2010`, with the month, day, and year.

We'll discuss other, more specialized types as we come to them. You'll also notice some values that have more square brackets:

```
-Path <String[]>
```

The side-by-side brackets after `String` don't indicate that something is optional. Instead, `String[]` indicates that the parameter can accept an *array*, a *collection*, or a *list* of strings. In these cases, it's always legal to provide a single value:

```
Get-Item -Path ~
```

But it's also legal to specify multiple values. A simple way to do so is to provide a comma-separated list. PowerShell treats all comma-separated lists as arrays of values:

```
Get-Item -Path ~, ~/Downloads
```

Once again, any individual value that contains a space must be enclosed in quotation marks. But the entire list doesn't get enclosed in quotation marks; it's important that only individual values be in quotes. The following is legal:

```
Get-Item -Path '~', '~/Downloads'
```

Even though neither of those values needs to be in quotation marks, it's okay to use the quotes if you want to. But the following is wrong:

```
Get-Item -Path '~, ~/Downloads'
```

In this case, the cmdlet will look for a file named `~, ~/Downloads`, which probably isn't what you want.

You can also feed a list of values to a parameter in a few other ways, including reading computer names from a file, or other cmdlets. Those techniques are a bit more complex, though, so we'll get to them in later chapters, after you learn some of the cmdlets you need to make the trick work.

Another way you can specify multiple values for a parameter (provided it's a mandatory parameter) is to not specify the parameter at all. As with all mandatory parameters, PowerShell will prompt you for the parameter value. For parameters that accept multiple values, you can type the first value and press Enter. PowerShell will then prompt for a second value, which you can type and finish by pressing Enter. Keep doing that until you're finished, and press Enter on a blank prompt to let PowerShell know you're finished. As always, you can press Ctrl-C to abort the command if you don't want to be prompted for entries.

Other parameters, referred to as *switches*, don't require any input value at all. In the abbreviated syntax, they look like the following:

```
[-Force]
```

And in the full syntax, they look like this:

```
-Force [<SwitchParameter>]
    Indicates that this cmdlet gets items that cannot otherwise be
 accessed, such as hidden items. Implementation varies from
    provider to provider. For more information, see about_Providers
 (../Microsoft.PowerShell.Core/About/about_Providers.md).
    Even using the Force parameter, the cmdlet cannot override security
 restrictions.

    Required?                    false
    Position?                    named
    Default value                False
    Accept pipeline input?       False
    Accept wildcard characters?  false
```

The `[<SwitchParameter>]` part confirms that this is a switch and that it doesn't expect an input value. Switches are never positional; you always have to type the parameter name (or at least an abbreviated version of it). Switches are always optional, which gives you the choice to use them or not.

For example, `Get-Item .*` will not show you any files, but `Get-Item .* -Force` will give you a list of files starting with . because files starting with . are considered hidden, and `-Force` tells the command to include hidden files.

3.5.5 Finding command examples

We tend to learn by example, which is why we try to squeeze as many examples into this book as possible. PowerShell's designers know that most administrators enjoy having examples, so they built a lot of them into the help files. If you scrolled to the end of the help file for `Get-Item`, you probably noticed almost a dozen examples of how to use the cmdlet.

Let's look at an easier way to get to those examples, if they're all you want to see. Use the `-Example` parameter of the `Help` command, rather than the `-Full` parameter:

```
Help Get-Item -Example
```

TRY IT NOW Go ahead and pull up the examples for a cmdlet by using this new parameter.

NOTE Because of PowerShell's origins on Windows, many of the examples use Windows paths. You are expected to know that you can use macOS or Linux paths. In fact, PowerShell does not care if you use / or \ as the directory separator on either platform.

We love having these examples, even though some of them can be complicated. If an example looks too complicated for you, ignore it and examine the others for now. Or experiment a bit (always on a nonproduction computer) to see if you can figure out what the example does and why.

3.6 Accessing "about" topics

Earlier in this chapter, we mentioned that PowerShell's help system includes background topics as well as help for specific cmdlets. These background topics are often called "about" topics, because their filenames all start with `about_`. You may also recall from earlier in this chapter that all cmdlets support a set of common parameters. How do you think you could learn more about those common parameters?

> **TRY IT NOW** Before you read ahead, see if you can list the common parameters by using the help system.

You start by using wildcards. Because the word *common* has been used repeatedly here in the book, that's probably a good keyword to start with:

```
Help *common*
```

It's such a good keyword, in fact, that it matches only one help topic: `About _common_parameters`. That topic displays automatically because it's the only match. Paging through the file a bit, you'll find the following list of the 11 (when this was written) common parameters:

```
-Verbose
-Debug
-WarningAction
-WarningVariable
-ErrorAction
-ErrorVariable
-OutVariable
-OutBuffer
-InformationAction
-InformationVariable
-PipelineVaribale
```

The file says that PowerShell has two additional *risk mitigation* parameters, but those aren't supported by every single cmdlet. The about topics in the help system are tremendously important, but because they're not related to a specific cmdlet, they can be easy to overlook. If you run `help about*` for a list of all of them, you might be surprised at how much extra documentation is hidden away inside the shell.

> **TRY IT NOW** Run the command `get-help about_*` and look at all the about topics. Now run `get-help about_Updateable_Help`

3.7 Accessing online help

Mere human beings wrote PowerShell's help files, which means they're not error free. In addition to updating the help files (which you can do by running `Update-Help`), Microsoft publishes help on its website. The `-Online` parameter of PowerShell's help command will attempt to open the web-based help—even on macOS or Linux!—for a given command:

```
Help Get-Item -Online
```

The Microsoft Docs website hosts the help, and it's often more up to date than what's installed with PowerShell itself. If you think you've spotted an error in an example or in the syntax, try viewing the online version of the help. Not every single cmdlet in the universe has online help; it's up to each product team (such as the Azure compute team, which provides VM functionality, Azure Storage team, and so forth) to provide that help. But when it's available, it's a nice companion to what's built in.

We like the online help because it lets us read the text in one window (the web browser, where the help is also nicely formatted) as we're typing in PowerShell.

It's important for us to point out that the PowerShell team at Microsoft has open sourced all of their help files as of April 2016. Anyone can add examples, correct errors, and generally help improve the help files. The online, open source project is at https://github.com/MicrosoftDocs/Powershell-Docs and typically includes only the documentation owned by the PowerShell team; it doesn't necessarily include documentation from other teams who are producing PowerShell commands. You can bug those teams directly about open sourcing their docs!

3.8 Lab

NOTE For this lab, you need any computer running PowerShell 7.

We hope this chapter has conveyed the importance of mastering the help system in PowerShell. Now it's time to hone your skills by completing the following tasks. Keep in mind that sample answers follow. Look for *italicized* words in these tasks, and use them as clues to complete the tasks:

1. Run `Update-Help`, and ensure that it completes without errors so that you have a copy of the help on your local computer. You need an internet connection.
2. Can you find any cmdlets capable of converting other cmdlets' output into *HTML*?
3. Are there any cmdlets that can redirect output into a *file*?
4. How many cmdlets are available for working with *processes*? (Hint: Remember that cmdlets all use a singular noun.)
5. What cmdlet might you use to *set* to a PowerShell breakpoint? (Hint: PowerShell-specific nouns are often prefixed with *PS*.)
6. You've learned that aliases are nicknames for cmdlets. What cmdlets are available to create, modify, export, or import *aliases*?
7. Is there a way to keep a *transcript* of everything you type in the shell, and save that transcript to a text file?
8. Getting all processes can be overwhelming. How can you get processes by the name of the process?
9. Is there a way to tell `Get-Process` to tell you the user who started the process?
10. Is there a way to run a *command* on a remote host? (Hint: *Invoke* is the verb for running something now.)

11 Examine the help file for the `Out-File` cmdlet. The files created by this cmdlet default to a width of how many characters? Is there a parameter that would enable you to change that width?

12 By default, `Out-File` overwrites any existing file that has the same filename as what you specify. Is there a parameter that would prevent the cmdlet from overwriting an existing file?

13 How could you see a list of all *aliases* defined in PowerShell?

14 Using both an alias and abbreviated parameter names, what is the shortest command line you could type to retrieve a list of *commands* with the word *process* in the name?

15 How many cmdlets are available that can deal with generic objects? (Hint: Remember to use a singular noun like *object* rather than a plural one like *objects*.)

16 This chapter briefly mentioned *arrays*. What help topic could tell you more about them?

3.9 Lab answers

1 `Update-Help`

Or if you run it more than once in a single day, use the following:

```
Update-Help -force
```

2 `help html`

Or you could try using the `Get-Command`:

```
Get-Command -Noun html
```

3 `Get-Command -Noun file,printer`

4 `Get-Command -Noun process`

or

```
Help *Process
```

5 `Get-Command -Verb set -Noun psbreakpoint`

Or if you aren't sure about the noun, use a wildcard:

```
help *breakpoint
```

or

```
help *break*
```

6 `help *alias`

or

```
Get-Command -Noun alias
```

7 `help transcript`

8 `help Get-Process -Parameter Name`

9 `help Get-Process -Parameter IncludeUserName`

10 The command to execute over SSH is

```
help Invoke-Command -Parameter hostname
```

Or the command to execute over the legacy Windows protocol is

```
help Invoke-Command -Parameter computername
```

11 `Help Out-File -Full`

or

```
Help Out-File -Parameter Width
```

should show you 80 characters as the default for the PowerShell console. You would use this parameter to change it as well.

12 If you run `Help Out-File -Full` and look at parameters, you should see `-NoClobber`.

13 `Get-Alias`

14 `Gcm -na *process*`

15 `Get-Command -Noun object`

16 `help about_arrays`

Or you could use wildcards:

```
help *array*
```