

# Windows PowerShell: Batch Files on Steroids

*Doug Hennig*

*Stonefield Software Inc.*

*Email: [dhennig@stonefieldquery.com](mailto:dhennig@stonefieldquery.com)*

*Corporate Web sites: [www.stonefieldquery.com](http://www.stonefieldquery.com)*

*and [www.stonefieldsoftware.com](http://www.stonefieldsoftware.com)*

*Personal Web site : [www.DougHennig.com](http://www.DougHennig.com)*

*Blog: [DougHennig.BlogSpot.com](http://DougHennig.BlogSpot.com)*

*Twitter: [DougHennig](https://twitter.com/DougHennig)*

*Windows PowerShell has been included with the operating system since Windows 7. What is PowerShell? It's Microsoft's task automation scripting framework. PowerShell isn't just a replacement for batch files; it can do a lot more than batch files ever could. This session looks at PowerShell, including why you should start using it, and how to create PowerShell scripts.*

### Introduction

While having a graphical user interface like Microsoft Windows is great for both ease-of-use and productivity, nothing beats commands for performing repetitive tasks. After all, a single command can replace clicking buttons, choosing menu items, and selecting choices in dialogs. Microsoft has provided a command-line interface ever since the first release of DOS. It also provided a way to put multiple commands into a text file with a “bat” extension (a “batch” file) and have one command execute all the commands in the file.

If you’ve ever used batch files to automate system tasks, you know they have numerous shortcomings. Error handling is rudimentary and the list of commands is short, so there’s really only a small set of things you can do with batch files, like copying or moving files.

Although the command shell (cmd.exe) and batch files are still available in Windows, they’ve been supplanted in functionality by Windows PowerShell. PowerShell has been around for more than a decade. Starting with Windows 7, it ships with the operating system.

There are two versions of PowerShell: the classic PowerShell, the latest version of which is 5.1 and is no longer developed, and PowerShell Core, the latest version of which at this writing (June 2023) is 7.3.4. The difference is that the former is built on the .NET Framework while the latter is based on .NET Core. PowerShell Core isn’t currently installed by default; to install it, open a PowerShell console window (see the “Starting with PowerShell” section) and type:

```
winget install -id Microsoft.PowerShell --source winget
```

PowerShell is Windows-only while PowerShell Core is cross-platform (there are versions for Windows, MacOS, and Linux). I’m only going to marginally discuss the differences between them and will just refer to them collectively as “PowerShell.” See <https://techgenix.com/powershell-core/> for some of the differences between the two versions.

The key things to know about PowerShell are:

- It’s built on .NET, so it has full access to .NET’s extensive class library.
- It also has full access to COM and WMI, so it can do just about any system administrative task.
- PowerShell statements are interpreted rather than compiled, so they can be executed interactively in PowerShell’s console window.
- PowerShell is dynamically rather than statically typed, so it works more like JavaScript than like .NET for data typing.
- Like .NET, everything in PowerShell is an object.

## What is PowerShell good for?

In a word, everything:

- Server administration: restart or shutdown servers, change passwords, restart services, terminate processes, or just about any other task you can think of.
- Web server administration: manage web sites, application pools, and virtual directories.
- Workstation administration: perform backups, create user accounts, set or determine configuration settings, kill processes, and so on.
- Application administration: many tasks you would perform using a GUI to manage tools like SQL Server and Exchange Server are available as PowerShell commands.
- Anything else you want to automate. Later in this document, we'll look at automating application installation and backups.

Here's a simple example: suppose you want to document the values of system environment variables. Here's how you do it using the Windows GUI:

- Click the Windows Start button, type "environment," and choose Edit the System Environment Variables to display the System Properties dialog.
- Click the *Environment Variables* button.
- For each one, click the Edit button, copy the name and the values, and paste them into the documentation.

Here's how to do it using PowerShell:

- Launch PowerShell; the next section discusses several ways to do that.
- Type the following:  
`dir env: | select key, value | export-csv result.csv`

Not only does this one statement save time, it can also be automated, which the GUI steps cannot.

## Starting with PowerShell

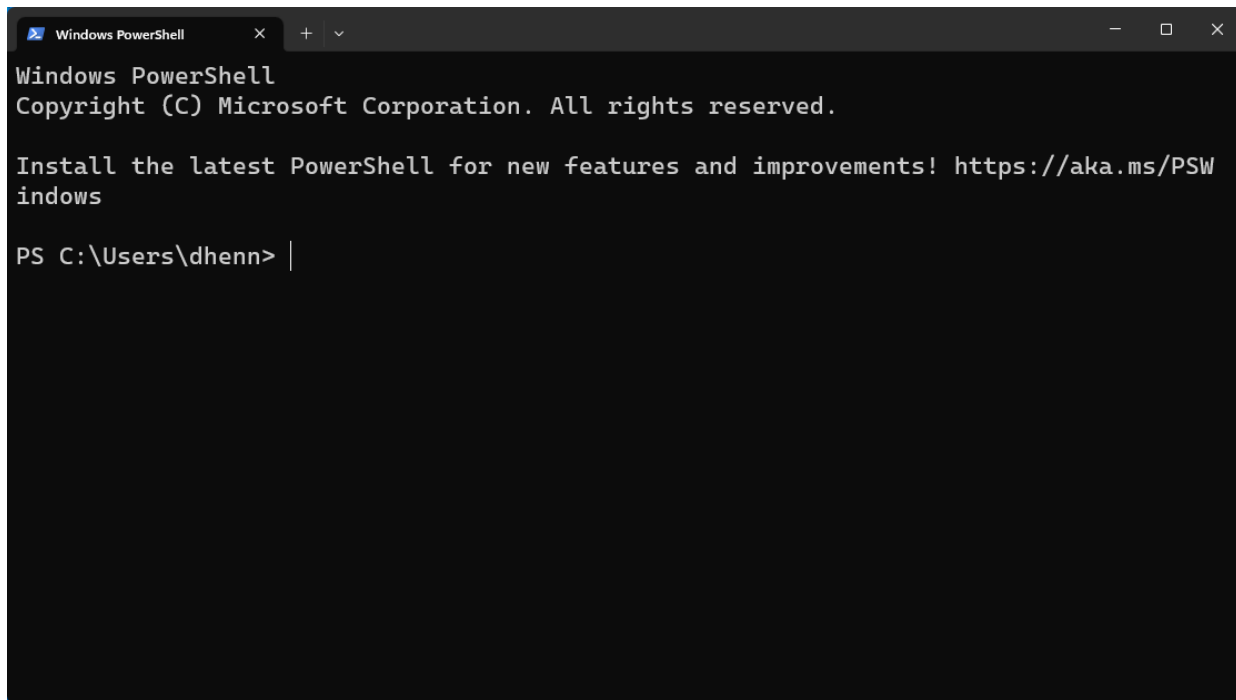
The easiest way to start playing with PowerShell is to open a console window. There are several ways you can do that:

- Open File Explorer, choose the File tab, and select either *Open Windows PowerShell* or *Open Windows PowerShell as administrator* to open a window in the current folder. (Note: this is no longer available in newer builds of Windows 11.)
- Type "PowerShell" ("pwsh" for PowerShell Core) in the address bar of File Explorer to open a window in the current folder.

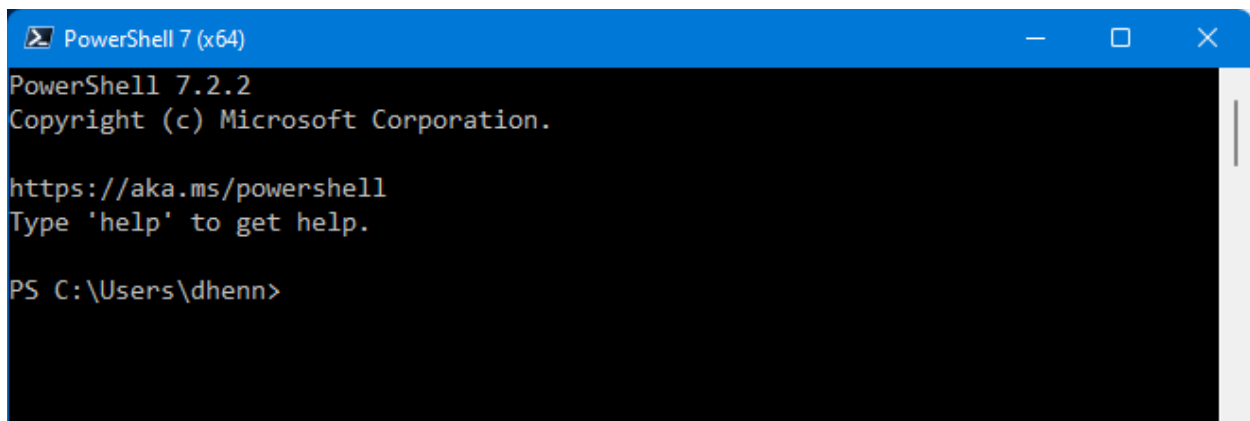
- Click the Windows Start button or press Windows-S to open Search, type “PowerShell,” and select *Windows PowerShell* from the list of matches (*PowerShell 7* for PowerShell Core). Right-click it and choose *Run as administrator* if necessary.

PowerShell is in `C:\Windows\System32\WindowsPowerShell\v1.0`, so you can also run `PowerShell.exe` in that folder. PowerShell Core is in `C:\Program Files\PowerShell\7`, so you can run `Pwsh.exe` in that folder.

The PowerShell console window is shown in **Figure 1** (this is actually Windows Terminal hosting a PowerShell window) and the PowerShell Core console window is shown in **Figure 2**.



**Figure 1.** The Windows PowerShell console looks very much like the `Cmd.exe` console.



**Figure 2.** The PowerShell Core console looks a little different than the PowerShell version.

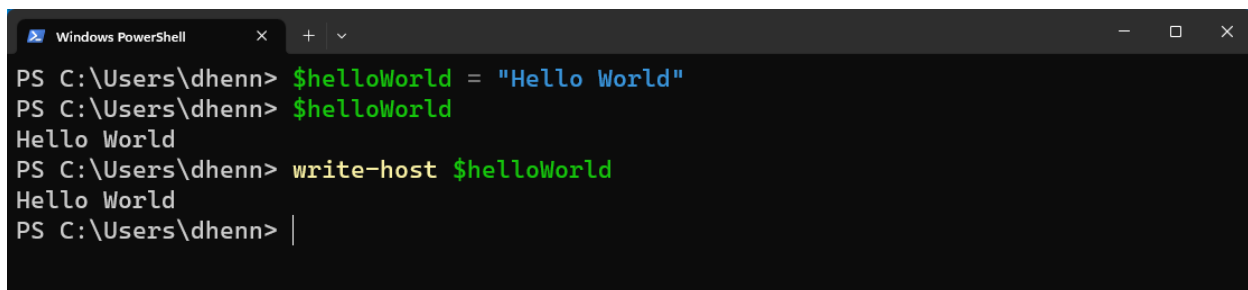
The console window supports a tabbed interface: the top of the window has tabs, a “+” button to create a new tab, and a down arrow button with additional options (type of console tab to create, settings dialog, and command palette). Any Cmd.exe command works. So, you can type `cls` (to clear the display), `dir`, `cd`, or the name of an executable or BAT file. Right-clicking a tab to access other options, such as color settings and tab control (rename, duplicate, close).

Let’s explore some of PowerShell using the console window.

### Variables

PowerShell variable names start with “\$,” are not case-sensitive, and can contain any character except “!,” “@,” “#,” “%,” “&,” comma, period, and space. You should avoid using reserved words (use `Get-Help about_reserved_words` for a list of them; you may have to run PowerShell as administrator and use `Update-Help` to download the latest help files first) and built-in variable names (`Get-Help about_automatic_variables` shows a list). The normal naming convention for variables is camel case, such as “myVariableName.”

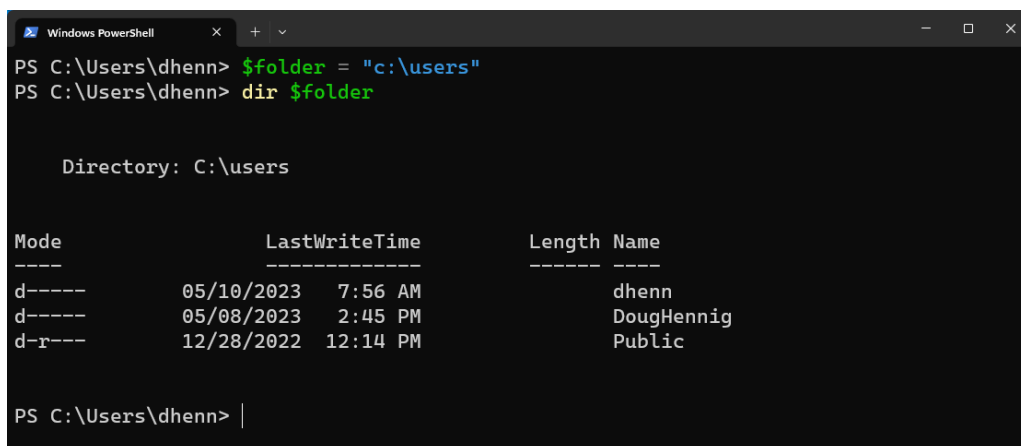
Variables are assigned a value with “=.” You can output the value of a variable using just its name or with the `Write-Host` command, as shown in **Figure 3**.



```
Windows PowerShell
PS C:\Users\dhenn> $helloWorld = "Hello World"
PS C:\Users\dhenn> $helloWorld
Hello World
PS C:\Users\dhenn> write-host $helloWorld
Hello World
PS C:\Users\dhenn> |
```

**Figure 3.** Assign a value to a variable using “=” and display it by typing its name or using `Write-Host`.

Variables can be passed to commands. In **Figure 4**, the variable `$folder` is passed to the `dir` command.



```
Windows PowerShell
PS C:\Users\dhenn> $folder = "c:\users"
PS C:\Users\dhenn> dir $folder

Directory: C:\users

Mode                LastWriteTime         Length Name
----                -
d-----          05/10/2023   7:56 AM             dhenn
d-----          05/08/2023   2:45 PM          DougHennig
d-r---          12/28/2022  12:14 PM             Public

PS C:\Users\dhenn> |
```

**Figure 4.** Variables can be passed to commands.

PowerShell supports the same data types as .NET; the more commonly used ones are shown in **Table 1**.

**Table 1.** Commonly used data types in PowerShell.

| Type        | Description                                   |
|-------------|---|
| [array]     | Array of values                               |
| [bool]      | Boolean True/False value                      |
| [char]      | Unicode 16-bit character                      |
| [datetime]  | Date and time                                 |
| [decimal]   | 128-bit decimal value                         |
| [double]    | Double-precision 64-bit floating point number |
| [hashtable] | Hashtable object                              |
| [int]       | 32-bit signed integer                         |
| [long]      | 64-bit signed integer                         |
| [single]    | Single-precision 32-bit floating point number |
| [string]    | Fixed-length string of Unicode characters     |
| [xml]       | Xml object                                    |

PowerShell automatically determines the data type when you assign a value:

```
$v1 = "1"  
$v2 = 1  
$v3 = 1.1
```

In this case, \$v1 is a string, \$v2 an int, and \$v3 a decimal. PowerShell does automatic type conversion when necessary:

```
"1" + 1  
1 + "1"
```

The first statement displays “11” while the second displays “2.”

You can also specify the data type in square brackets:

```
[int]$value = 10
```

String literals can either be surrounded with single or double quotes. The difference between the two is that double quotes cause “interpolation,” or expansion of variables, while single quotes don’t. See **Figure 5** for an example.

A screenshot of a Windows PowerShell terminal window. The window has a title bar with 'Windows PowerShell' and standard window controls. The terminal shows a series of commands and their outputs. The commands are: `$v1 = "quick"`, `$v2 = "The $v1 brown fox"`, `$v3 = 'The $v1 brown fox'`, `$v2`, `$v3`, and `$v1`. The outputs are: 'The quick brown fox', 'The \$v1 brown fox', and '\$v1' respectively. The prompt is `PS C:\Users\dhenn>`.

**Figure 5.** Double quotes cause interpolation; single quotes don't

Use “+” to concatenate strings. In this code, “Windows PowerShell” is displayed:

```
$v1 = "Windows"
$v2 = "PowerShell"
$v1 + ' ' + $v2
```

A period ends a variable in evaluation:

```
$here = get-item .
"Hello $here.Name"
```

This displays “Hello *path*.Name” rather than the expected “Hello *pathname*,” where *path* is the current folder. That happens because the period terminates the variable name rather than being used as the dot between the object and member names. In that case, use:

```
"Hello $($here.Name)"
```

Arrays are zero-based indexed lists of variables. You can specify an array using a comma-delimited list of values using one of the following ways:

```
$a = 1, 2, 3
$b = @(4, 5, 6)
```

Use “@()” for an empty array:

```
$c = @()
$c.Count
```

“+” adds an element to an array or can even combine arrays:

```
$a + 4
$a + $b
```

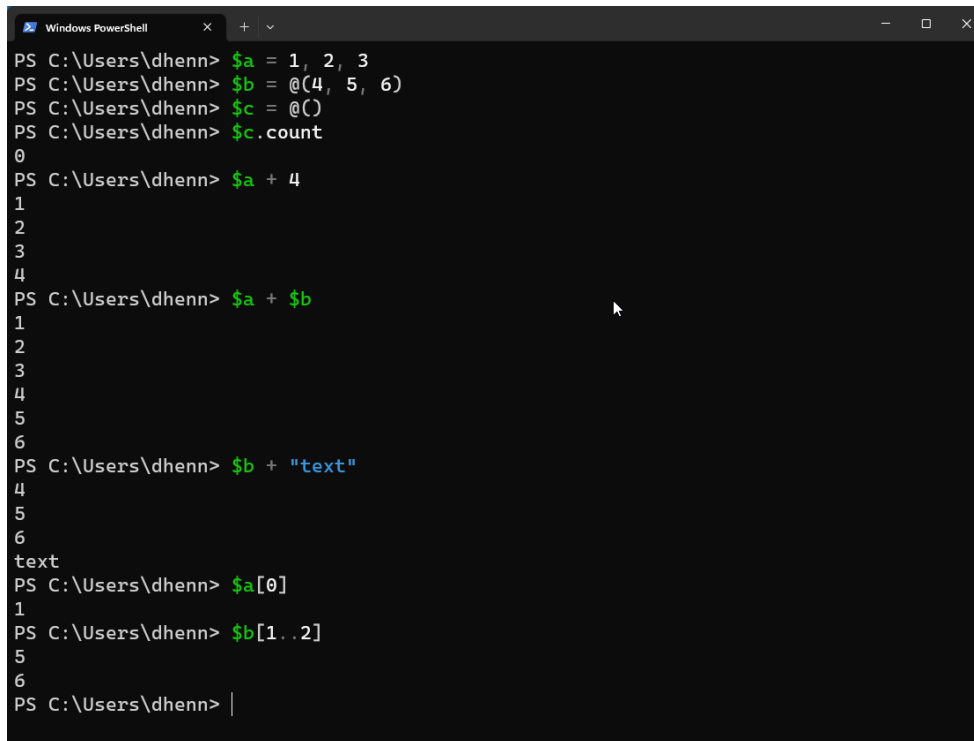
Arrays can be mixed data types:

```
$b + "text"
```

Use “array[index]” to access individual elements or “startIndex..EndIndex” for a range of values:

```
$a[0]  
$b[1..2]
```

See **Figure 6** for the results of these operations.



```
Windows PowerShell  
PS C:\Users\dhenn> $a = 1, 2, 3  
PS C:\Users\dhenn> $b = @(4, 5, 6)  
PS C:\Users\dhenn> $c = @()  
PS C:\Users\dhenn> $c.count  
0  
PS C:\Users\dhenn> $a + 4  
1  
2  
3  
4  
PS C:\Users\dhenn> $a + $b  
1  
2  
3  
4  
5  
6  
PS C:\Users\dhenn> $b + "text"  
4  
5  
6  
text  
PS C:\Users\dhenn> $a[0]  
1  
PS C:\Users\dhenn> $b[1..2]  
5  
6  
PS C:\Users\dhenn> |
```

**Figure 6.** Arrays are easy to work with in PowerShell.

A HashTable stores name-value pairs of any data type and length. To declare a HashTable, use similar syntax to arrays but with curly braces instead of parenthesis and semi-colons or carriage returns instead of commas.

```
$teams = @{"WPG" = "Jets"; "CGY" = "Flames"; "EDM" = "Oilers"}  
$teams  
$teams["CGY"]  
$teams["TOR"] = "Leafs"  
$teams  
$teams.Remove("TOR")  
$teams  
$teams.Clear()  
$teams
```

See **Figure 7** for the results.





```
PS C:\Users\dhenn> $teams = @{"WPG" = "Jets"; "CGY" = "Flames"; "EDM" = "Oilers"}
PS C:\Users\dhenn> $teams

Name          Value
----          -
EDM           Oilers
WPG           Jets
CGY           Flames

PS C:\Users\dhenn> $teams["CGY"]
Flames
PS C:\Users\dhenn> $teams["TOR"] = "Leafs"
PS C:\Users\dhenn> $teams

Name          Value
----          -
TOR           Leafs
EDM           Oilers
WPG           Jets
CGY           Flames

PS C:\Users\dhenn> $teams.Remove("TOR")
PS C:\Users\dhenn> $teams

Name          Value
----          -
EDM           Oilers
WPG           Jets
CGY           Flames

PS C:\Users\dhenn> $teams.Clear()
PS C:\Users\dhenn> $teams
PS C:\Users\dhenn> |
```

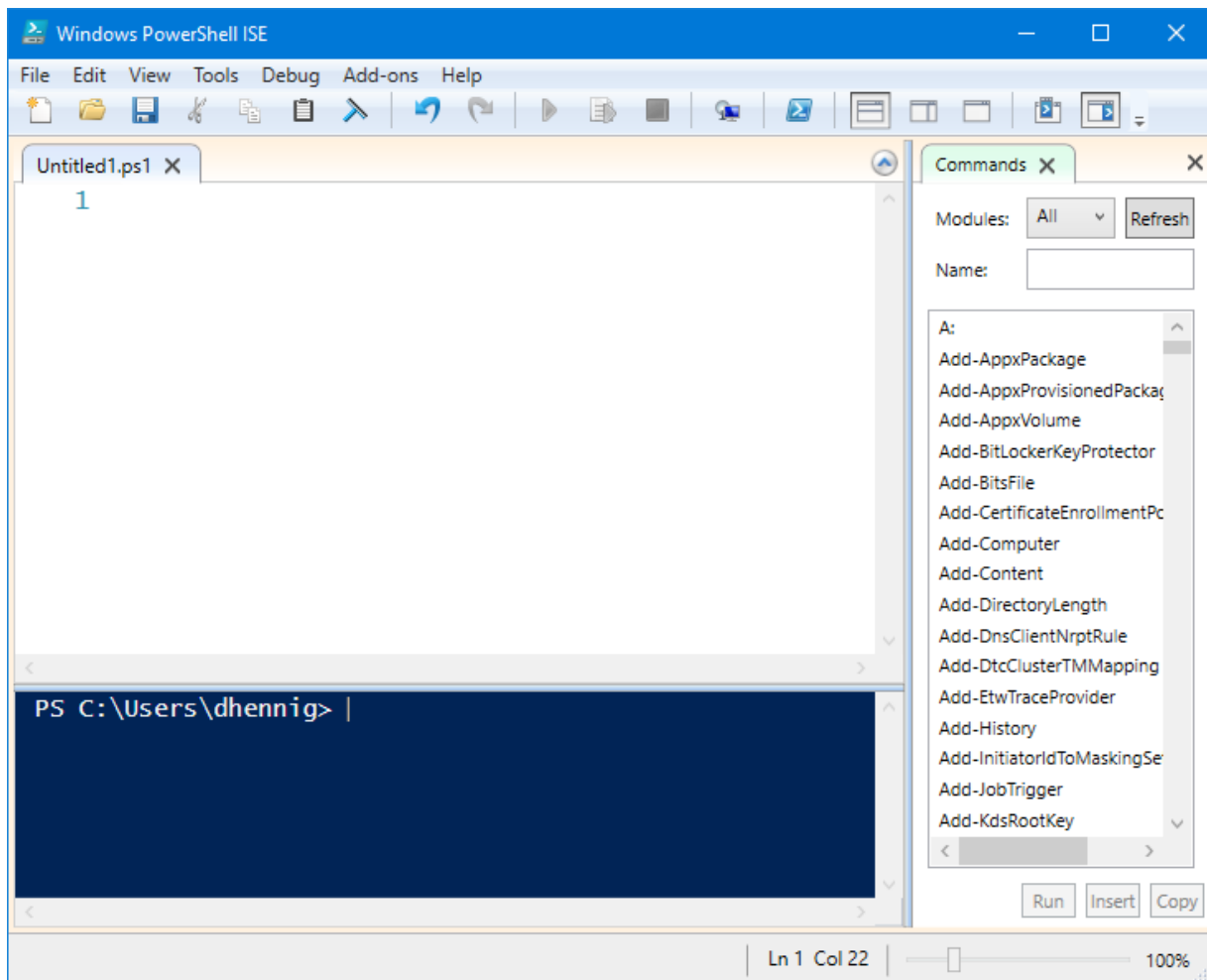
**Figure 7.** A HashTable is a list of name-value pairs that can be accessed by name or index.

“Splatting” allows you to execute a cmdlet (cmdlets are discussed later) with parameters from an array or HashTable rather than listing the parameters individually. To do that, pass the variable containing the array or HashTable to the cmdlet using an “@” prefix. Here’s an example of using splatting to zip some files:

```
$compress = @{
    Path = "C:\SomeFolder\*.*"
    CompressionLevel = "Fastest"
    DestinationPath = "C:\Backup\Backup.zip"
}
Compress-Archive -Force @compress
```

## PowerShell ISE

While the PowerShell console window is fine for playing around, eventually you'll want a better environment to work in. That environment is PowerShell ISE (interactive scripting environment). ISE has a built-in console window, a script editor, an integrated debugger, and IntelliSense for commands, so like the Visual Studio interactive development environment (IDE), it's a better environment to work in. See **Figure 8** to see what ISE looks like.



**Figure 8.** PowerShell ISE is an interactive development environment for PowerShell.

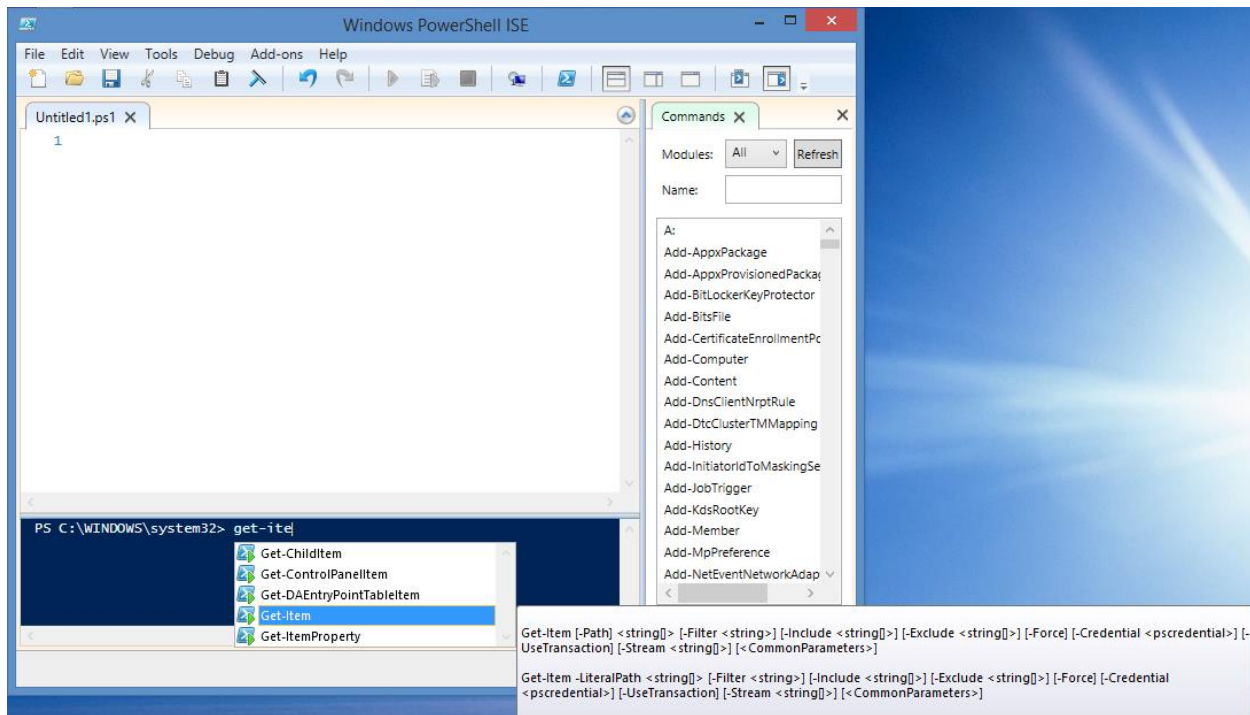
(Note: PowerShell Core doesn't have its own IDE. Instead, use Visual Studio Code with the PowerShell Extension.)

To run PowerShell ISE, search for "powershell ise" or run `PowerShell_ise.exe` in `C:\Windows\System32\WindowsPowerShell\v1.0`. You can also right-click a ps1 file (a PowerShell script file we'll discuss later) and choose Edit, or from a Windows Explorer window, type "powershell\_ise" in the address bar.

In addition to the usual menu, toolbar, and status bar of most IDEs, ISE has three main areas:

- The script pane is an editor where you work on a script file. It has IntelliSense and syntax coloring to assist with editing commands.
- The Commands pane shows a list of commands. You can search the list by name, filter by module, and see the syntax and parameters for the selected command.
- The console pane is like the console window but adds IntelliSense so it's a lot easier to work in.

In both the script and console panes, command completion makes it easy to enter a command. Type the first part of the command and, as you can see in **Figure 9**, IntelliSense lists matching commands. Hit Tab for command completion; that is, inserting the selected item into the text. This even works for non-command things like “CD .\;” a list of directories pops up and you can press Tab to enter the selected one.



**Figure 9.** The console pane in ISE includes IntelliSense.

## Commands

There are four types of commands in PowerShell:

- Native commands
- Cmdlets
- Scripts

- Functions

I discussed native commands earlier so let's look at the others.

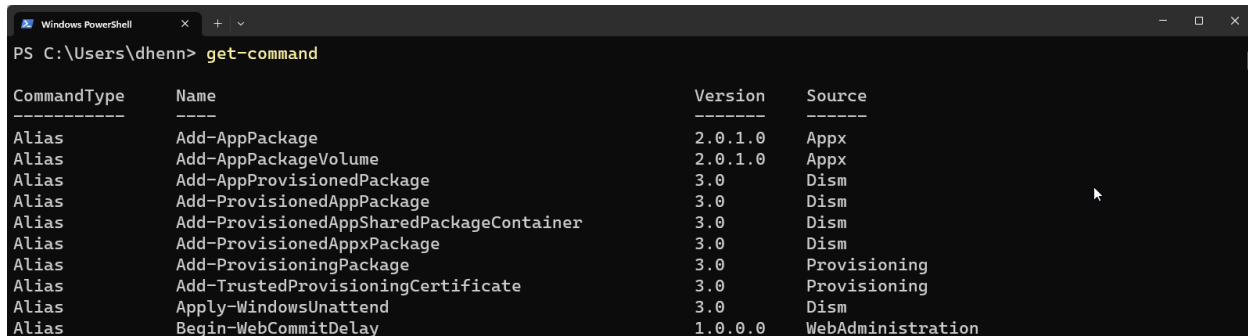
### Cmdlets

Cmdlets are commands built into PowerShell. Their names are standardized to use verb-noun syntax. For example, `Get-Item` displays information about the specified folder:

```
Get-item .  
$here = get-item .  
$here.Name  
$here.GetType()
```

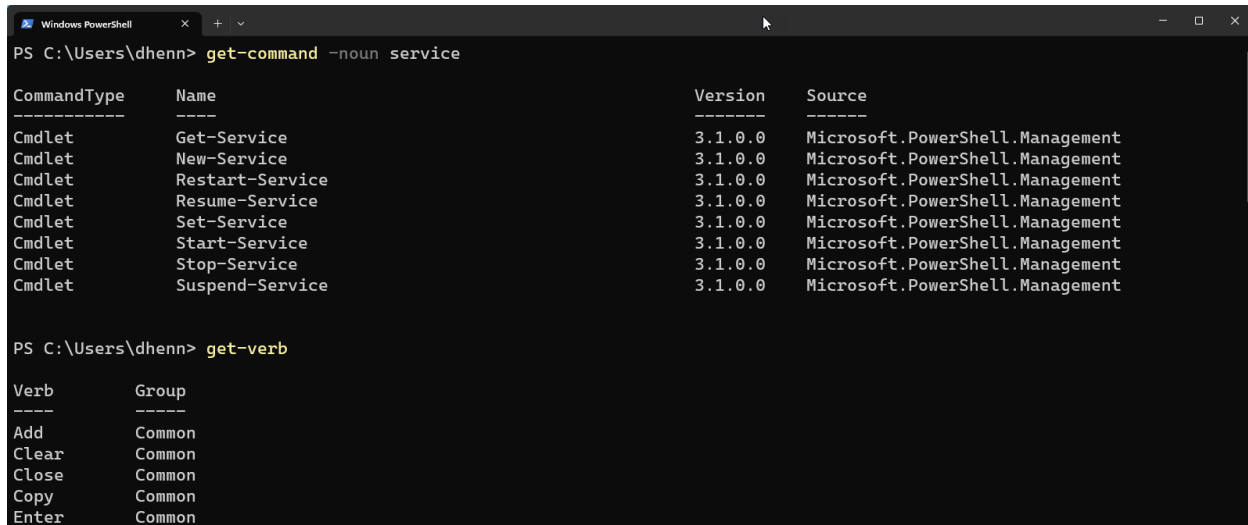
Notice that PowerShell cmdlets are also not case-sensitive; `Get-Item`, `Get-item`, and `get-item` all do the same thing.

To see a list of cmdlets, use `Get-Command`; see **Figure 10**.



**Figure 10.** Get-command lists all commands.

To see commands using a particular verb, use `Get-Command -verb verb`, such as `Get-Command -verb get`. Common verbs are add, clear, get, new, and set. For commands using a particular noun, use `Get-Command -noun noun`, such as `Get-Command -noun service`. For a list of verbs, use `Get-Verb`. See **Figure 11**.



```
PS C:\Users\dhenn> get-command -noun service

CommandType      Name
-----
Cmdlet            Get-Service
Cmdlet            New-Service
Cmdlet            Restart-Service
Cmdlet            Resume-Service
Cmdlet            Set-Service
Cmdlet            Start-Service
Cmdlet            Stop-Service
Cmdlet            Suspend-Service

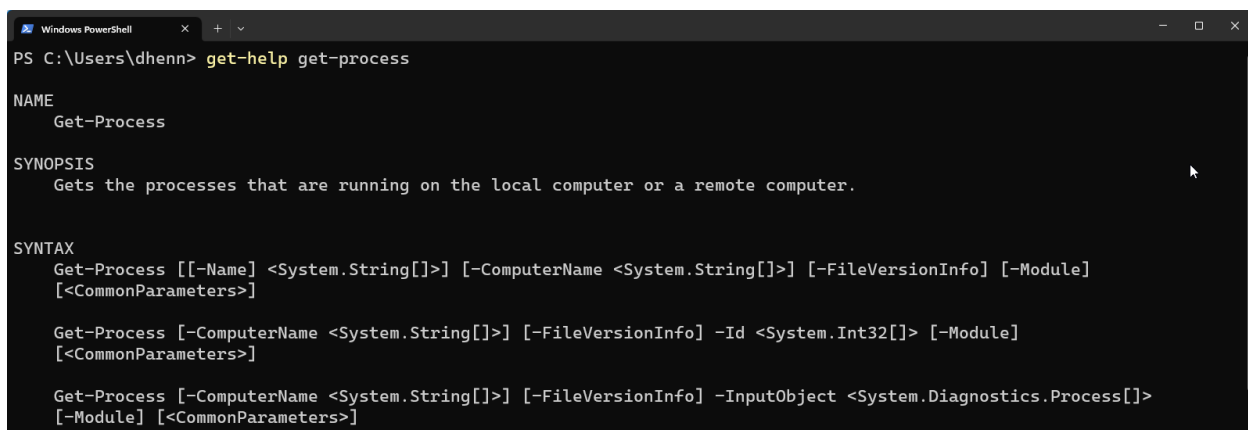
Version           Source
-----
3.1.0.0           Microsoft.PowerShell.Management
3.1.0.0           Microsoft.PowerShell.Management
3.1.0.0           Microsoft.PowerShell.Management
3.1.0.0           Microsoft.PowerShell.Management
3.1.0.0           Microsoft.PowerShell.Management
3.1.0.0           Microsoft.PowerShell.Management
3.1.0.0           Microsoft.PowerShell.Management
3.1.0.0           Microsoft.PowerShell.Management

PS C:\Users\dhenn> get-verb

Verb      Group
-----
Add       Common
Clear     Common
Close     Common
Copy      Common
Enter     Common
```

**Figure 11.** Get-Command -noun lists nouns and Get-Verb lists verbs.

Get-Help is a great command because it gives you help for PowerShell. Get-Help *cmdlet* displays help about the specified cmdlet (see **Figure 12**). Interestingly, the text displayed by Get-Help isn't built into PowerShell but is instead in the scripts that create the cmdlet; we'll see that later in the "Documenting your scripts" section. Get-Help *cmdlet* - examples shows examples of the command.



```
PS C:\Users\dhenn> get-help get-process

NAME
    Get-Process

SYNOPSIS
    Gets the processes that are running on the local computer or a remote computer.

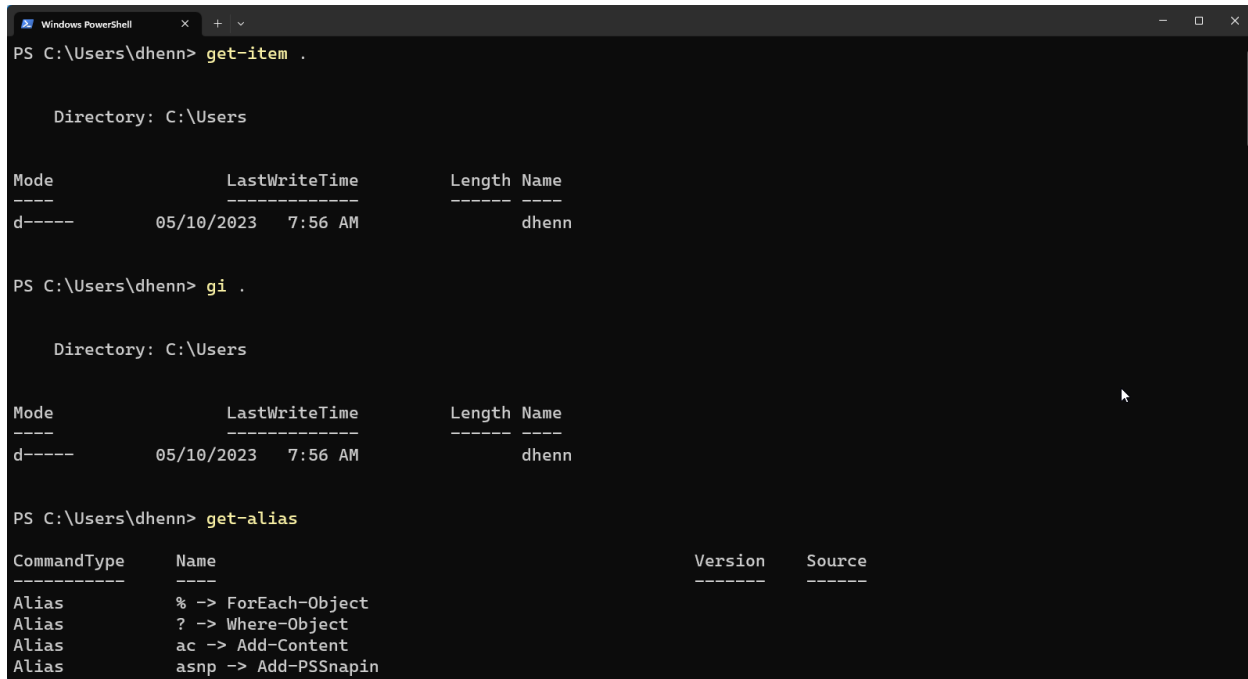
SYNTAX
    Get-Process [[-Name] <System.String[]>] [-ComputerName <System.String[]>] [-FileVersionInfo] [-Module]
    [-<CommonParameters>]

    Get-Process [-ComputerName <System.String[]>] [-FileVersionInfo] -Id <System.Int32[]> [-Module]
    [-<CommonParameters>]

    Get-Process [-ComputerName <System.String[]>] [-FileVersionInfo] -InputObject <System.Diagnostics.Process[]>
    [-Module] [-<CommonParameters>]
```

**Figure 12.** Get-help is a great way to see information about a specific item.

Many cmdlets have abbreviated names, known as aliases. For example, "gi" is the alias for Get-Item. Helpfully, some well-known DOS commands, like cd and dir, are just abbreviations for PowerShell commands (Set-Location and Get-ChildItem, in this case). Use Get-Alias or its alias gal to see a list of aliases. See **Figure 13**.



**Figure 13.** Aliases, such as “gi” for “get-item,” make typing in the command window easier.

You can create your own aliases using `Set-Alias alias command`. However, user-defined aliases are only in effect for that PowerShell session and are forgotten once you’ve closed it. If you want to use a certain set of aliases regularly, use the `Export-Alias` and `Import-Aliases` commands to write to and read from text files of aliases.

Abbreviations are fine for use at the command line but I don’t recommend using them in scripts for reasons of clarity.

**Table 2** lists some commonly used cmdlets.

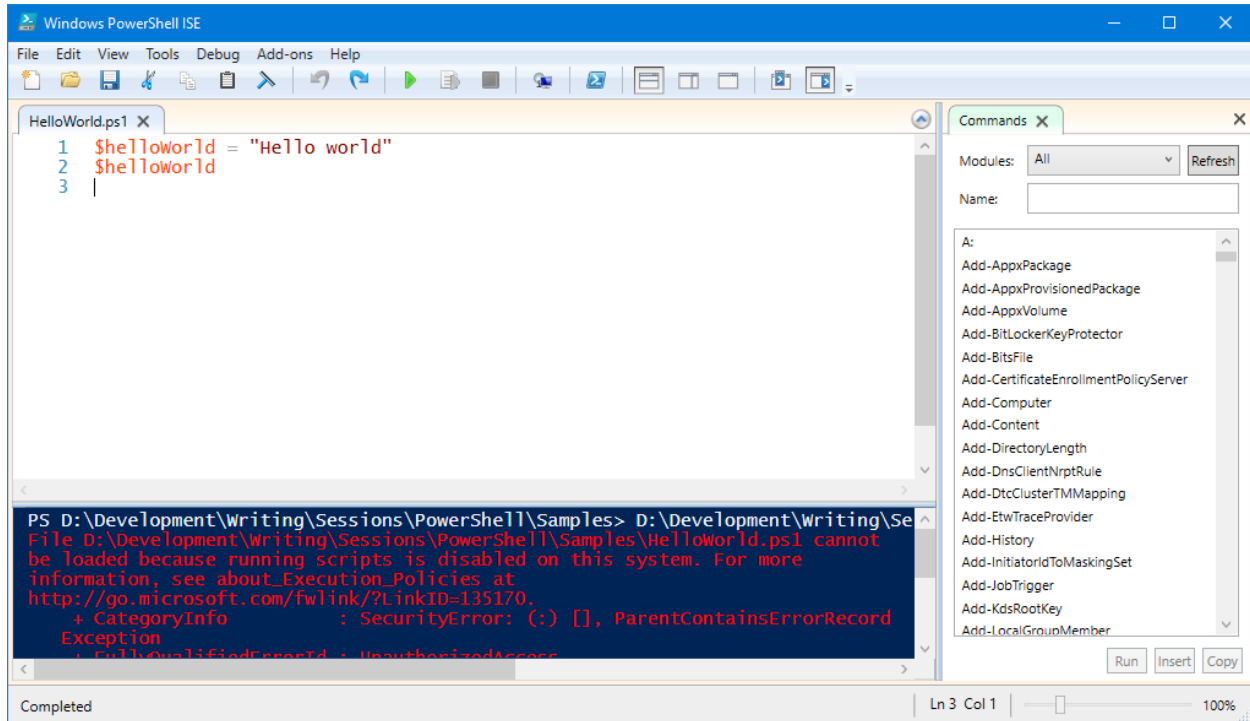
**Table 2.** Commonly used cmdlets.

| Name                    | Description   |
|-------------------------|---|
| <b>Add-Content</b>      | Writes to a file.   |
| <b>Get-ChildItem</b>    | Gets the items in the specified path. Not just for files; can work with other things such as environment variables and the Windows Registry, and has a lot of flexibility including recursively reading subdirectories. |
| <b>Get-Content</b>      | Reads the content of a file.  |
| <b>Get-Date</b>         | Gets the current date.  |
| <b>Get-ItemProperty</b> | Gets the properties of the specified item. Often used to get values from the Registry.  |
| <b>Get-Location</b>     | Gets the current location.  |
| <b>Get-Member</b>       | Displays a list of the members of the specified object (remember that everything in PowerShell is an object).   |
| <b>Get-Variable</b>     | Lists variables.  |
| <b>New-Object</b>       | Instantiates the specified class.   |
| <b>Remove-Item</b>      | Removes the specified item. Can be recursive and can handle non-empty folders.  |
| <b>Test-Path</b>        | Determines if the specified path exists. Works on any path, including Registry entries.   |
| <b>Write-Host</b>       | Outputs to the console.   |

## Scripts

A PowerShell script is a text file with a “ps1” extension that contains PowerShell statements. Although you can use any text editor to create a script, as we saw earlier, using PowerShell ISE is the preferred editing environment. You can type the statements in a new script or open and edit an existing script file.

To run a script, click the green arrow in the toolbar or press F5. One thing you’ll quickly find is that by default, PowerShell execution policy prevents running scripts (**Figure 14**).



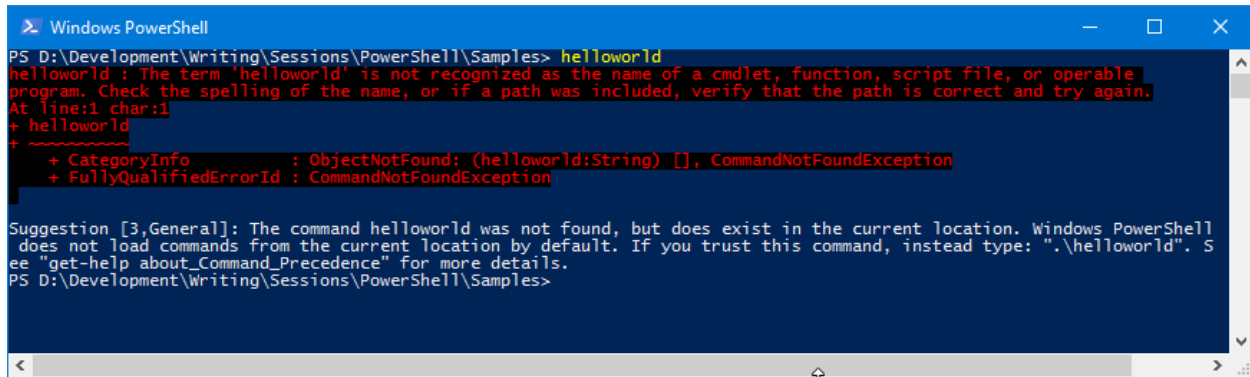
**Figure 14.** By default, running PowerShell scripts is disabled.

To allow script execution, use `Set-ExecutionPolicy level`, where *level* is one of the levels shown in **Table 3**. Note that you either need to run PowerShell as administrator or add `-scope currentuser` to the command.

**Table 3.** Use one of these settings with Set-ExecutionPolicy to control script execution.

| Type                | Description  |
|---------------------|--|
| <b>Restricted</b>   | Scripts cannot be executed.  |
| <b>AllSigned</b>    | All scripts must be signed by a trusted publisher.   |
| <b>RemoteSigned</b> | Scripts downloaded from the Internet must be signed by a trusted publisher.                                    |
| <b>Unrestricted</b> | Scripts can be executed. You are prompted for permissions before running scripts downloaded from the Internet. |

You can’t run a script by simply typing its name; see **Figure 15** for an example.



**Figure 15.** Running a script in the console window gives an error unless you prefix it with “.\.”

As the hint suggests, PowerShell doesn’t run scripts from the current directory by default. You have a couple of choices:

- Add the current directory to the PowerShell path by issuing `$env:Path += ";."`.
- Prefix the script name with `.\` to run it as an “encapsulated” script; that is, like a separate application. This is sometimes called “dot source running.” You can also run a script using `.\script` (that is, a period, a space, a period, a backslash, then the script name), which runs the script as part of the command session, which means variables can overwrite each other. This probably isn’t practical in a production environment but is handy for testing and debugging.

Speaking of debugging, debugging PowerShell scripts is very much like debugging in other IDEs. Set or clear a breakpoint on a statement by choosing Toggle Breakpoint from the Debug menu or pressing F9. Start running a script or continue to the next breakpoint using Run/Continue in the Debug menu or F5. Step through code using the Step Over, Step Into, and Step Out items in the Debug menu (F10, F11, and Shift-F11, respectively). Stop debugging using Stop Debugger in the Debug menu or Shift-F5. You can change the values of variables, execute other commands while execution is paused, and so on.

Some other things to know about scripts:

- “#” is the comment begin character:  
`# Get the current folder.`  
`$currFolder = Get-Location`
- “&” runs an executable program. For example, this code runs Notepad and opens the specified file:  
`$notepad = "notepad.exe"`  
`$file = "helloWorld.ps1"`  
`& $notepad $file`
- To force a script to be executed by PowerShell Core, add this at the start:  
`#requires -PSEdition Core`



Or use this to force it to PowerShell:

```
#requires -PSEdition Desktop
```

### PowerShell profile

A profile is a script that runs when PowerShell starts. The path of the profile is contained in the built-in variable `$profile`; by default, it contains

`"C:\Users\username\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1,"` but that script file isn't created automatically.

The following set of commands checks whether the file specified in `$profile` exists, creates the file, and opens it in Notepad:

```
Test-Path $Profile
New-Item -Path $Profile -ItemType file -Force
notepad $Profile
```

Some of the things you can do in the profile are load aliases, load frequently used modules (I'll discuss modules later), set the default path, and so on.

### Functions

The syntax for a function is:

```
function Name(parameters)
{
    code
}
```

Here's an example:

```
function Get-Greeting()
{
    "Hello, Doug"
}
```

Here are some function concepts:

- The function name doesn't have to follow verb-noun syntax. Here's the same function with a different name:

```
function Hello-World()
{
    "Hello, Doug"
}
```
- As you can see in these examples, parameters are not required in the function definition.

- Parameters can be specified as shown in the syntax above, using a `param` statement, which must be the first thing in the code block, or not at all but retrieved using the `$args` variable.

```
function Hello-World()
{
    param ($name)
    "Hello, $name"
}

function Test-Args()
{
    $passed = $args[0] + " " + $args[1]
    "The parameters are $passed"
}
```

- Parameters are referenced like variables—that is, they are prefixed with “\$”—and can either be data-typed or not. Multiple parameters are separated with commas.

```
function Hello-World([string]$name)
{
    "Hello, $name"
}
```

- Even if they are included in the definition, parameters don’t have to be passed unless they’re mandatory (see the next point); the variables specified as parameter names are null or false in that case. Use something like the following to determine if a parameter was passed a value:

```
if (!$name)
```

- You can specify that a parameter must be passed a value using the `mandatory` keyword. You’re prompted for missing values when the script is run in ISE but get an error otherwise.

```
function Hello-World([parameter(mandatory=$true)] $name)
{
    "Hello $name"
}
```

- You can specify a default value for parameters that aren’t passed:

```
function Hello-World()
{
    param ($name = "Unknown")
    "Hello, $name"
}

function Hello-World($name = "Unknown")
{
    "Hello, $name"
}
```

- Use the `return` statement to return a value from the function.

If a script file contains only a function, executing the script file won't do anything. You have to "load" the script file for PowerShell to recognize its functions. To do that, type period, space, and the path to the script file (which you need to prefix with ".\."). For example, to make the Get-Greeting function available, do the following (assuming it's in a script file named GetGreeting.ps1):

```
.\GetGreeting.ps1
```

Now you can type "Get-" and see "Get-Greeting" in the IntelliSense list.

Note that if you name the script file and the function the same, PowerShell gets a little confused and sometimes lists both. For that reason, I suggest using a different script file name.

To avoid having to do this every time you start ISE, you can load your functions at startup by adding the appropriate commands to your profile.

### Documenting your scripts

As I mentioned earlier, Get-Help doesn't display hard-coded text but instead reads the help information from the script file. This is called "comment-based help." To add documentation to your own script files, use syntax like the following:

```
function MyFunction($someParameter)
{
    <#
    .Synopsis
    This describes what the function does.

    .Description
    This is a longer description of the function.

    .Example
    Show the syntax here

    Describe what happens here. You can have multiple .Example sections.

    .Parameter someParameter
    Discuss the parameter.

    .Notes
    Anything you want, such as your name.

    .Link
    The URL for documentation, your web site, etc.
    #>

    # Code for the function
}
```

Other keywords are available; see <https://tinyurl.com/orlvxte> for more information on comment-based help.

To display help for the function, use `Get-Help function` or `Get-Help function -full`, where *function* is the function name.

## Modules

A module is a PowerShell script with a “psm1” extension. Modules usually consist of functions so they don’t execute when loaded but simply make those functions available to other scripts. The environment variable `psModulePath`, which you can see in PowerShell using `$env:psModulePath`, contains the paths for modules. To load a module, use `Import-Module moduleName`. The `Get-Module` command lists loaded modules.

You can create your own module by creating a script with a “psm1” extension rather than “ps1.” Putting your psm1 file into one of the folders in the modules path, usually `C:\Users\UserName\Documents\WindowsPowerShell\Modules`, makes it available to be imported into any script.

Lots of modules are available for PowerShell. Some useful ones are:

- `PSCX` (PowerShell Community Extensions; <https://github.com/Pscx/Pscx>) provides features like SQL queries against databases, reading from and writing to the clipboard, and so on.
- `PowerShellGet` (<https://tinyurl.com/53z3fvzk>) makes it easy to find and install other modules in the PowerShell Gallery, a central repository for PowerShell content.
- `Posh-Git` (available using `PowerShellGet`) provides PowerShell integration with the Git version control system.
- The popular application installer `Chocolatey` (<https://chocolatey.org>) is actually a PowerShell module that supports downloading and installing thousands of applications using a single command. We’ll see this in the “Automating application installation” section of this document.

## Operators

PowerShell uses the comparison operators shown in **Table 4**. Note that comparisons are case-insensitive; use a prefix of “c” (such as “-ceq”) for a case-sensitive comparison. If the item being compared is a scalar value, the result of the comparison is true or false. If the item is a collection or array, the result is any matching values.

PowerShell has three logical operators: “-not” (or its abbreviation “!”), “-and,” and “-or.”

PowerShell has several assignment operators. “++” is the increment operator; “\$i++” is the equivalent of “\$i = \$i + 1.” Similarly, “--” is the decrement operator. “+=” adds the specified value; “\$x += \$y” is the equivalent of “\$x = \$x + \$y.”

**Table 4.** PowerShell comparison operators.

| Operator            | Description                                    |
|---------------------|--|
| <b>-eq</b>          | Equals   |
| <b>-ne</b>          | Not equals                                     |
| <b>-gt</b>          | Greater than                                   |
| <b>-lt</b>          | Less than                                      |
| <b>-ge</b>          | Greater than or equals                         |
| <b>-le</b>          | Less than or equal                             |
| <b>-like</b>        | Like; you can use "*" as a wildcard character. |
| <b>-notlike</b>     | Not like                                       |
| <b>-contains</b>    | Contains                                       |
| <b>-notcontains</b> | Does not contain                               |
| <b>-in</b>          | In a list                                      |
| <b>-notin</b>       | Not in a list                                  |

## Branching code

PowerShell has an **if** statement that allows conditional branches. Here's the syntax:

```
if (some condition)
{
    statement(s)
}
```

The condition must be in parentheses and be an expression that returns a Boolean value, and the statements must be surrounded by curly braces. Typically, you'll compare one value to another using one of the comparison operators listed in PowerShell has three logical operators: **"-not"** (or its abbreviation **"!"**), **"-and,"** and **"-or."**

PowerShell has several assignment operators. **"++"** is the increment operator; **"\$i++"** is the equivalent of **"\$i = \$i + 1."** Similarly, **"--"** is the decrement operator. **"+="** adds the specified value; **"\$x += \$y"** is the equivalent of **"\$x = \$x + \$y."**

**Table 4.** You can combine expressions using logical operators discussed earlier.

The **else** and **elseif** statements provide alternate execution paths:

```
if (some condition)
{
    statement(s)
}
elseif
{
    statement(s)
}
elseif
{
    statement(s)
}
else
```

```
{  
    statement(s)  
}
```

The PowerShell `switch` statement is like that in JavaScript or C#: it tests the various possible values of an expression. Here's the syntax:

```
switch (expression)  
{  
    value1  
    {  
        statement(s)  
    }  
    value2  
    {  
        statement(s)  
    }  
    default  
    {  
        statement(s)  
    }  
}
```

Here are some comments about this:

- Like `if`, the expression must be in parentheses and the statements must be surrounded with curly braces.
- `default` is the section to execute if none of the values match. It's not required but is a good idea.

`Branching.ps1`, one of the files accompanying this document, shows examples of `if` and `switch` statements; see **Listing 1**.

**Listing 1.** `Branching.ps1` shows examples of branching commands.

```
$a = 3  
$b = ""  
  
# IF  
if ($a -eq 1)  
{  
    $b = "Red"  
}  
elseif ($a -eq 2)  
{  
    $b = "Blue"  
}  
if ($a -eq 3)  
{  
    $b = "Green"  
}  
Write-Host $b
```

```
# SWITCH
$a = 1
switch ($a)
{
    1 {$b = "Red"}
    2 {$b = "Blue"}
    3 {$b = "Green"}
    default {$b = "Unknown"}
}
Write-Host $b
```

## Loops

PowerShell has the traditional set of looping constructs: `do/while/until`, `for`, and `foreach`. In addition, it has two commands that control loop execution: `break` and `continue`.

### do while, while, and do until

`do while` and `while` execute a block of code as long as a condition is true. The difference is that `do while` tests the condition at the end so it executes at least once; `while` tests the condition at the start so it may not execute at all. This code outputs the numbers 1 through 5:

```
$i = 1
do
{
    Write-Host $i
    $i++
}
while ($i -le 5)
```

`do until` is the opposite of `do while`: it executes until the condition becomes true.

```
$i = 1
do
{
    Write-Host $i
    $i++
}
until ($i -gt 5)
```

### for

The `for` command use the same syntax as C, C#, JavaScript, and similar languages:

```
for (initialization; termination condition; repeat expression)
```

Initialization typically initializes a loop counter to a starting value, the termination condition specifies when the loop is done, and the repeat expression typically increments the loop counter. For example:

```
for ($i = 1; $i -lt 6; $i++)
```

```
{  
    Write-Host $i  
}
```

### foreach

`foreach` executes a block of code once for each item in an array or collection. It uses similar syntax to C# and other languages. For example, this code lists all the files in the current folder and its subdirectories:

```
$currFolder = Get-Location  
foreach($item in (Get-ChildItem -Path $currFolder -Recurse -File))  
{  
    $source = $item.DirectoryName + "\" + $item  
    Write-Host $source  
}
```

### break and continue

Use `break` if you need to break out of a loop early or `continue` to skip to the next iteration.

### Error handling

PowerShell has the familiar `try/catch/finally` construct for handling errors. You can even throw an exception. See the code in the “Backing up files” section of this document for an example.

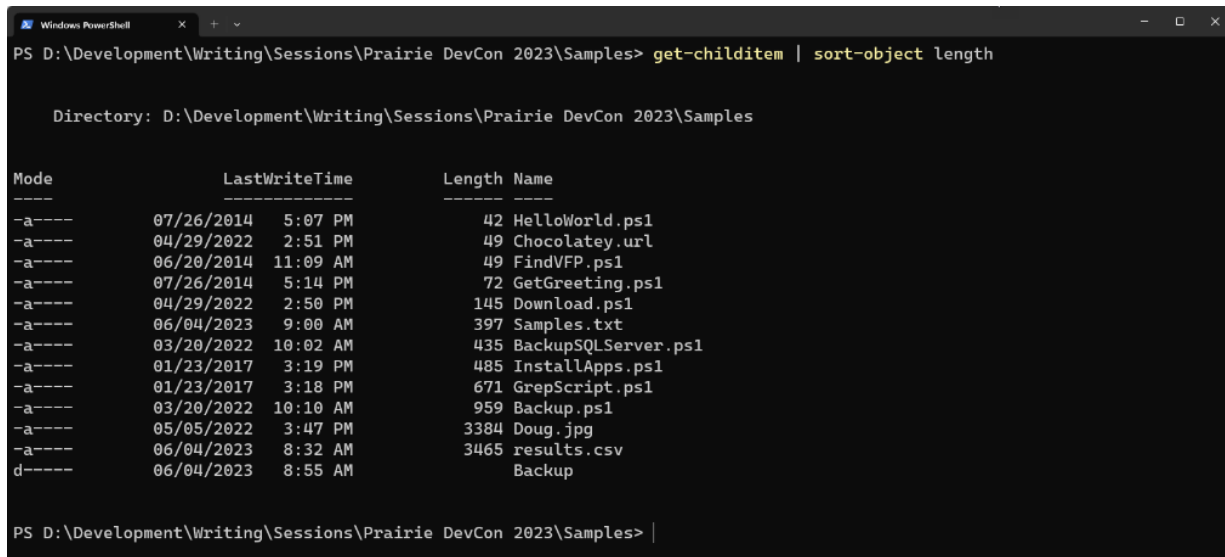
“\$?” is a built-in variable containing the status of the last statement. If it’s true, the statement succeeded. It can be used, for example, in branching code to do different things based on whether the code was successful or not.

If an error occurs, the built-in variable `$Error` can be used to retrieve the error information. The “Backing up files” section of this document also has an example.

### Pipelining

Pipelining means sending the output of one command to the input of another. For example, `Get-ChildItem` lists the files in the current folder while `Sort-Object` sorts a collection by some attribute. What happens if we send the output of `Get-ChildItem` to `Sort-Object`? See **Figure 16** for the results. The pipe (“|”) character causes the pipelining to occur.





```
PS D:\Development\Writing\Sessions\Prairie DevCon 2023\Samples> get-childitem | sort-object length

Directory: D:\Development\Writing\Sessions\Prairie DevCon 2023\Samples

Mode                LastWriteTime         Length Name
----                -
-a-----         07/26/2014    5:07 PM             42 HelloWorld.ps1
-a-----         04/29/2022    2:51 PM             49 Chocolatey.url
-a-----         06/20/2014   11:09 AM             49 FindVFP.ps1
-a-----         07/26/2014    5:14 PM             72 GetGreeting.ps1
-a-----         04/29/2022    2:50 PM            145 Download.ps1
-a-----         06/04/2023    9:00 AM           397 Samples.txt
-a-----         03/20/2022   10:02 AM          435 BackupSQLServer.ps1
-a-----         01/23/2017    3:19 PM          485 InstallApps.ps1
-a-----         01/23/2017    3:18 PM          671 GrepScript.ps1
-a-----         03/20/2022   10:10 AM          959 Backup.ps1
-a-----         05/05/2022    3:47 PM         3384 Doug.jpg
-a-----         06/04/2023    8:32 AM        3465 results.csv
d-----         06/04/2023    8:55 AM           Backup
```

**Figure 16.** The result of piping the output of Get-ChildItem to Sort-Object.

A special kind of piping cmdlet is a filter. A filter takes something as an input, removes unwanted items, and sends the rest as output. One built-in filter is Where-Object, which acts like the WHERE clause in a SQL statement. **Figure 17** shows the result of this command:

```
Get-ChildItem | Where-Object {$_.Length -gt 1000} | Sort-Object Length
```



```
PS D:\Development\Writing\Sessions\Prairie DevCon 2023\Samples> get-childitem | where-object {$_.length -gt 1000} | sort
-object length

Directory: D:\Development\Writing\Sessions\Prairie DevCon 2023\Samples

Mode                LastWriteTime         Length Name
----                -
-a-----         05/05/2022    3:47 PM         3384 Doug.jpg
```

**Figure 17.** Using Where-Object to filter a list.

The built-in variable `$_` means the current item being examined by the filter cmdlet. In this case, it only returns items whose Length property is greater than 1,000 bytes.

Your own scripts can support pipelining using the built-in variable `$input`, which contains the pipelined data. For example, this one-line script in a file named FindEdge.ps1 looks through the input piped to it for a file named “msedge.exe:”

```
$input | Where-Object {$_.Name -eq "msedge.exe"}
```

This command shows the location of msedge.exe somewhere under C:\Program Files (x86) (you could use C:\ instead but that would take longer):

```
Get-ChildItem 'C:\Program Files (x86)\Microsoft' -Recurse | .\FindEdge
```

You can also create your own filter. Although the FindEdge script acts like a filter, it doesn't start executing until \$input has been filled with data. A filter, on the other hand, starts processing immediately, using the \$\_ variable for the current item. The only difference between a filter and a function is the "filter" keyword.

For example, here's a filter which filters out anything not containing the specified keyword:

```
filter Grep($keyword)
{
    if (($_ | Out-String) -like "$keyword*")
    {
        $_
    }
}
```

The following command lists all PS1 files in the current folder and all subdirectories:

```
Get-ChildItem -Recurse | Grep .ps1
```

## Working with .NET

Since PowerShell is based on .NET, it's easy to use .NET classes in PowerShell statements. The new-object cmdlet instantiates the specified class; "System" is inferred so you don't have to specify that namespace. For example:

```
$web = New-Object Net.WebClient
$web.DownloadFile(
    "https://www.prairiedevcon.com/assets/images/speakers/doughennig.jpg",
    "$((Get-Location).Path)\Doug.jpg")
```

instantiates the System.Net.WebClient class and downloads a file. You can use Get-Member to see a list of the members of the instantiated object.

To call a static member of a class, use the syntax [Class]::Member. For example, this gets a new Guid:

```
$guid = [Guid]::NewGuid()
```

## Script examples

Let's look at some examples of scripts to learn more about how they work.

### Automating application installation

Chocolatey is a PowerShell utility that automates the installation of Windows applications. It saves you having to manually navigate to the publisher's web site, finding the correct download link, and downloading and installing the application. As of this writing, over 9,000 applications can be installed using Chocolatey, including Google Chrome, Firefox, Microsoft SQL Server Express, Microsoft Visual Studio, Notepad++, Filezilla, Git, Dropbox,

and so on. Obviously, you'll need valid license keys for some applications. You can read more about Chocolatey and see a list of the available applications at <https://community.chocolatey.org>. Scott Hanselman also had an informative blog post about it at <https://tinyurl.com/pmdhfv5>.

You can install Chocolatey with a single, albeit complex, command:

```
@powershell -NoProfile -ExecutionPolicy unrestricted -Command  
"iex ((new-object net.webclient).DownloadString('https://  
community.chocolatey.org/install.ps1'))" &&  
SET PATH=%PATH%;%systemdrive%\chocolatey\bin
```

This is three statements in one:

- @powershell invokes PowerShell from any command window, and the statement prior to “-Command” tells PowerShell to allow unrestricted script execution.
- The text between “-Command” and “&&” instantiates the .NET WebClient class, tells it to download Install.ps1 from community.chocolatey.org, and runs it.
- The SET PATH statement adds Chocolatey to your path so it can be executed from anywhere.

Using Chocolatey to install an application is as easy as typing `choco install appName` in a command window. <https://community.chocolatey.org/packages> lists the applications and their installation command lines.

Of course, you can even automate Chocolatey itself. Here's an example of a simple script that installs Chocolatey and then uses it to download and install Filezilla, Google Chrome, and Notepad++. Think about how quickly such a script could set up a new machine for you, whether a physical or virtual machine, whether local, remote, or in the cloud.

```
# Allow scripts to run; this needs to be executed in the console pane.  
#Set-ExecutionPolicy unrestricted -Scope CurrentUser  
  
# Download and install Chocolatey  
Invoke-Expression ((new-object net.webclient)  
    .DownloadString('https://community.chocolatey.org/install.ps1'))  
  
# Add Chocolatey to the path so we can run it from anywhere  
set PATH=%PATH%;%systemdrive%\chocolatey\bin  
# Start installing apps  
choco install Filezilla  
choco install GoogleChrome  
choco install NotepadPlusPlus
```

For even more automation, look at Boxstarter (<https://boxstarter.org>). It can perform completely unattended installs of Windows boxes, including installing the latest Windows updates.

### Backing up files

Backup.ps1 performs a backup by zipping files into a file named Backup\_*date*.zip in the Backup subdirectory of the specified folder, where *date* is the date the backup was made.

There are a variety of mechanisms for zipping files in PowerShell. Two common techniques are using the Compress-Archive cmdlet and the .NET System.IO.Compression.ZipFile class. Compress-Archive is much simpler but has limitations if you use recursion to zip files in subdirectories; see <https://tinyurl.com/44xu39n2> for details. Here's an example of Compress-Archive:

```
# Zip all the files in the folder.
$compress = @{
    Path = $sourceFolder + "\*.*"
    CompressionLevel = "Fastest"
    DestinationPath = $zipFileName
}
Compress-Archive -Force @compress
```

The script starts by ensuring a string parameter named \$sourceFolder was passed and that it's not null or empty. After assigning the name of the zip file to the \$zipFileName variable and ensuring the Backup folder exists, the script zips all the files in the folder. "\$?" is a built-in variable containing the status of the last statement, so it's used to write the appropriate message. If an error occurs, the built-in variable \$Error is used to retrieve the error information.

```
Param(
    [parameter(Mandatory = $true)]
    [ValidateNotNullOrEmpty()]
    [string]$sourceFolder
)

# Get the name of the zip file.
$zipFileName = $sourceFolder + "\Backup\Backup_$(Get-Date -f yyyy-MM-dd).zip"

try
{
    # Create the Backup folder if it doesn't exist.
    $exists = Test-Path ($sourceFolder + "\Backup")
    if (-not $exists)
    {
        md ($sourceFolder + "\Backup")
    }

    # Zip all the files in the folder.
    $compress = @{
        Path = $sourceFolder + "\*.*"
        CompressionLevel = "Fastest"
        DestinationPath = $zipFileName
    }
    Compress-Archive -Force @compress
```

```
# Check for errors
if ($?)
{
    Write-Host "Backup successful"
}
else
{
    Write-Host "Backup failed"
}
}

catch
{
    Write-Host "Error occurred at $(Get-Date): $($Error[0].Exception.Message)"
}

finally
{
    Write-Host "Backup finished at $(Get-Date)"
}
```

To see this script in action, pass it the name of a folder to backup, such as:

```
Backup (Get-Location)
```

### Backing up SQL Server databases

I have a scheduled task that backs up the SQL Server databases on my server to a ZIP file. The task executes this command:

```
powershell -File C:\Backup\BackupSQLServer.ps1
```

BackupSQLServer.ps1 has this content:

```
$SkipDBs = @('master','model','msdb','tempdb')
Get-SqlDatabase -ServerInstance MYSERVERNAME |
    Where { $SkipDBs -NotContains $_.Name } |
    foreach { Backup-SqlDatabase -DatabaseObject $_
        -BackupFile "C:\SQLBackup\$(($_.Name).bak)" }
$compress = @{
    Path = "C:\SQLBackup\*.*"
    CompressionLevel = "Fastest"
    DestinationPath = "C:\Backups\BackupSQL $(get-date -f yyyy-MM-dd).Zip"
}
Compress-Archive -Force @compress
```

This code backs up all the databases in the specified SQL Server instance except those listed in the \$SkipDBs variable to BAK files in C:\SQLBackup, then zips the files in that folder to a file named YYYY-MM-DD.zip (for example, 2023-05-20.zip).

### Resources

There are a ton of resources available for all things PowerShell; simply Google “powershell” and be prepared to be overwhelmed! A good place to start, though, is <https://learn.microsoft.com/en-us/powershell>, Microsoft’s PowerShell documentation center. It has links for webcasts, downloads, blogs, scripts, tutorials, and more.

### Summary

Windows PowerShell allows you to automate just about any task you can think of. It’s relatively easy to learn, is very powerful, and once you’ve created a script, can save you a lot of time doing tasks the manual way.

### Biography

Doug Hennig is CTO of Stonefield Software Inc. He has more than 40 years’ experience in the IT industry. He is the author of the several award-winning products, including Stonefield’s flagship product Stonefield Query. He has written several books and hundreds of articles in various software development magazines. Doug has spoken at software development conferences and user groups all over the world. He was a 15-time Microsoft Most Valuable Professional (MVP) and has done extensive consulting and development work for Microsoft.