**1. Get-Help**

As the name suggests, Get-Help is the cmdlet used to interact with Powershell’s integrated help system. It’s basically the self-help guide to PowerShell and the very first cmdlet you should familiarize yourself with. In fact, chapter 3 of “[Learn PowerShell in a Month of Lunches](https://www.manning.com/books/learn-powershell-in-a-month-of-lunches)” is dedicated to the Get-Help cmdlet, and it goes on for 17 pages!

Get-Help provides valuable information about cmdlets, parameters, aliases, etc. It also provides helpful examples to get you started.

Here’s the syntax to use Get-Help:

Get-Help <cmdlet\_name>

By default, Get-Help limits the information returned by the command. Add the -Full parameter to view the entire help documentation about a given cmdlet, including descriptions, examples, and additional notes.

Get-Help <cmdlet\_name> -Full

A screen shot of a computer program

Description automatically generated

**Important:**You need to update your local help files using the Update-Help cmdlet. It’s also a good idea to update your PowerShell help files every couple of months to ensure they are current.

**2. Get-Command**

Another important PowerShell cmdlet, especially for newcomers, is Get-Command. Get-Command, without additional parameters, returns all cmdlets, functions, and aliases installed on the computer.

Get-Command

A screen shot of a computer

Description automatically generated

Get-Command is a great way to see what commands you can run. However, by default, this command returns over a thousand results, which can be overwhelming. Thankfully, you can narrow down the results by using filters. For example, if you wanted to see what commands deal with certificates, you can filter for that specifically.

Get-Command -Name \*certificate\*

A screenshot of a computer program

Description automatically generated

Most cmdlets use common sense names, so filtering is an intuitive way to track down the cmdlet you’re looking for.

**3. Get-Member**

One quality that makes PowerShell so versatile is that almost everything in PowerShell is an object consisting of a name, methods, and properties. We can use the Get-Member cmdlet to inspect these objects.

<cmdlet\_name> | Get-Member

A computer screen shot of a program

Description automatically generated

In this example, I’ve piped the Get-Date cmdlet to the Get-Member cmdlet, which returns the object’s attributes. I’ve also added the -Name \*da\* filter to narrow down the results.

First, notice the usage of the pipeline operator.

A screen shot of a computer

Description automatically generated

The pipeline operator sends the results of the preceding command to the following command. In this example, the results from the Get-Date cmdlet are passed to the Get-Member cmdlet.

Next, pay attention to the TypeName.

A screen shot of a computer

Description automatically generated

The TypeName is the type of object that the command returns. In this case, the object type is System.DateTime.

Lastly, notice the MemberTypes returned for the object.

A computer screen shot of a program

Description automatically generated

MemberTypes is where you’ll identify the methods and properties of an object. Methods are actions you can perform against the object, and properties are the object’s attributes.

Get-Member is an excellent cmdlet, especially for PowerShell beginners, to quickly identify the methods and properties of an object.

**4. Out-File**

Need to save your PowerShell output to a raw text file? Out-File is the cmdlet you’re looking for.

To use the Out-File cmdlet, you can either use the piping operator to pass output to it or the -InputObject parameter to specify the objects to be written to the file.

Get-Process | Out-File C:\Temp\processes.txt

Or:

$proc = Get-Process Out-File C:\Temp\processes.txt -InputObject $proc

These commands provide the same results, creating a text file containing the output of the Get-Process cmdlet. It’s important to note that the information in the text file will be formatted as if sent to the terminal window.

A screenshot of a computer

Description automatically generated

**5. Export-Csv**

If you need to output your data in a more usable format, you’ll want to export it to a CSV file using the Export-Csv cmdlet.

Export-Csv does what its name implies and exports output to a CSV file. This cmdlet is extremely useful, especially when working with large data sets. However, it’s important to point out that Export-Csv exports ALL object property values by default, so you’ll want to filter the properties to return using the Select-Object cmdlet.

Get-Process | Select-Object -Property ID, ProcessName | Export-Csv C:\CSV\processes.csv

Here is the resulting CSV file:

A screenshot of a computer

Description automatically generated

**6. Get-ChildItem**

In PowerShell, you'll often need to work with items in containers such as files, certificates, or registries. Get-ChildItem returns all the items in one or more containers. If you only want to return specific items, you can use filtering to narrow down the results. You can also return items in subdirectories by using the -Recurse parameter. To limit the depth of the subdirectory search, use the -Depth parameter.

Get-ChildItem <folder\_name>

This example returns all items located in the specified directory.

A screenshot of a computer

Description automatically generated

This example is a bit more complex and returns all TXT files containing the word “taco” in the specified directory and subdirectories.

Get-ChildItem C:\Temp\\* -Recurse -Include \*taco\*.txt

A screenshot of a computer

Description automatically generated

The results show that this command returned two files, one located in the *C:\Temp\Extra* subdirectory and one in the *C:\Temp* directory.

**7. Out-GridView**

Most of the time, PowerShell output returns to the console window. However, if you need to interact with the output, you can use the Out-GridView cmdlet. Out-GridView sends output to a grid view GUI window.

Here’s the syntax to use Out-GridView:

<command> | Out-GridView

A screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

In this example, I’ve piped the Get-Service cmdlet to the Out-GridView cmdlet. The Out-GridView window automatically opened, containing the Get-Service command output, allowing me to interact with it.

If we want to take this example even further, we can use the -PassThru parameter to send multiple items down the pipeline, essentially letting you pick what output you want to pass on. Let’s update this example with the -PassThru parameter, and we’ll send our selected output to a TXT file.

Get-Service | Out-GridView -PassThru | Out-File C:\Files\services.txt

When we run this command, the Out-GridView window will open as usual. However, because we used the -PassThru parameter, we can select the output we want to pipe to the next command.

Hint: Hold the SHIFT key to highlight multiple items, and hold the CTRL key to select individual items. Click **OK** when you’ve finished selecting items.

A screenshot of a computer

Description automatically generated

Once we’ve made our selection and clicked **OK**, the output is sent to a text file. We can open the text file to ensure it contains the items we selected.

**8. Invoke-Item**

The Invoke-Item cmdlet launches items, including files, pictures, executables, etc. It can also launch more than one item at a time. Invoke-Item uses the default file association to determine the application to launch with the corresponding file type.

Here is the basic structure of an Invoke-Item command:

Invoke-Item <item\_path>

A computer screen with text

Description automatically generated

This example launches the indicated TXT file in Notepad.exe because of the default application assignment on my computer.

A screenshot of a computer

Description automatically generated

As mentioned above, Invoke-Item can also be used to open several files at once. This command will open all TXT files in the *C:\Temp* folder at once.

Invoke-Item “C:\Temp\\*.txt”

**9. Foreach**

Foreach loops iterate through a collection of items, performing actions against them. Foreach loops are a little more complex than the other commands we’ve covered so far. However, foreach loops become increasingly crucial as you dive into more advanced PowerShell tasks.

Here is the basic format of a foreach command in PowerShell:

foreach ($<item> in $<collection>){<statement list>}

Here’s a simple example of a foreach command that will identify files in a specified folder that are 0 KB in size.

$files = Get-ChildItem -Path C:\Files\ foreach ($file in $files){ if ($file.Length -eq 0kb){ Write-Host $file } }

A screenshot of a computer program

Description automatically generated

In this example, we assign the items located at *C:\Files\* to the $files variable. Then, we use the foreach command to loop through each item in the variable. Next, we’re using an if statement to determine if the file is equal to 0 KB. If the file is equal to 0 KB, the file is output to the console. The loop repeats until it has processed each item in the collection assigned to the variable $files.

We can check the folder for ourselves to ensure the data returned was accurate.

A screenshot of a computer

Description automatically generated

**10. If statements**

If statements are another extremely useful command in PowerShell. If statements execute logic based on conditions being met, similar to how people make decisions. For example, if I won the lottery, I would enjoy an early retirement. If not, then I would continue to upload amazing content to the internet. PowerShell can use the same kind of logic.

Here’s what the structure of an if statement looks like in PowerShell.

if ($<condition>){ <function to perform if the condition is true> }

When a condition is true, the function executes. If the condition returns false, it bypasses the function and proceeds to process the remainder of the script. For example:

$var = 42 If ($var -eq 42){ Write-Host “Congratulations. 42 is the meaning of life, the universe, and everything.” }

In this example, since the variable $var equals 42, and the condition is met, the Write-Host command will execute, and the message will display. If the variable equaled something different, like 24, the condition would return false, and the Write-Host command would not execute.