12

Filtering and comparisons

Up to this point, you've been working with whatever output the shell gave you: all the processes, filesystem objects, and various Azure commands. But this type of output isn't always going to be what you want. Often you'll want to narrow down the results to a few items that specifically interest you, such as getting processes or files that match a pattern. That's what you'll learn to do in this chapter.

12.1 Making the shell give you just what you need

The shell offers two broad models for narrowing results, and they're both referred to as *filtering*. In the first model, you try to instruct the cmdlet that's retrieving information for you to retrieve only what you've specified. In the second model (discussed in section 12.5), which takes an iterative approach, you take everything the cmdlet gives you and use a second cmdlet to filter out the things you don't want.

Ideally, you'll use the first model, which we call *filter left*, as much as possible. It may be as simple as telling the cmdlet what you're after. For example, with `Get-Process`, you can tell it which process names you want:

```
Get-Process -Name p*,*s*
```

But if you want `Get-Process` to return only the processes with more than 1 GB of memory, regardless of their names, you can't tell the cmdlet to do that for you, because it doesn't offer any parameters to specify that information.

Similarly, if you're using the `Get-ChildItem`, it includes the `-Path` parameter, which supports wildcards. Although you could get all files and filter using `Where-Object`, we don't recommend it. Once again, this technique is ideal because the cmdlet has to retrieve only matching objects. We call this the *filter left*, sometimes referred to as *early filtering*, technique.

12.2 Filtering left

Filter left means putting your filtering criteria as far to the left, or toward the beginning, of the command line as possible. The earlier you can filter out unwanted objects, the less work the remaining cmdlets on the command line will have to do, and possibly less unnecessary information may have to be transmitted across the network to your computer.

The downside of the filter-left technique is that every single cmdlet can implement its own means of specifying filtering, and every cmdlet will have varying abilities to perform filtering. With `Get-Process`, for example, you can filter only on the `Name` or `Id` property of the processes.

When you're not able to get a cmdlet to do all the filtering you need, you can turn to a PowerShell Core cmdlet called `Where-Object` (which has the alias `where`). This uses a generic syntax, and you can use it to filter any kind of object after you've retrieved it and put it into the pipeline.

To use `Where-Object`, you need to learn how to tell the shell what you want to filter, and that involves using the shell's comparison operators.

12.3 Using comparison operators

In computers, a *comparison* always takes two objects or values and tests their relationship to one another. You might be testing whether they're equal, or whether one is greater than another, or whether one of them matches a text pattern of some kind. You indicate the kind of relationship you want to test by using a *comparison operator*. The result of the test in simple operations results in a Boolean value: `True` or `False`. Put another way, either the tested relationship is as you specified, or it isn't.

PowerShell uses the following comparison operators. Note that when comparing text strings, these aren't case sensitive; an uppercase letter is seen as equal to a lowercase letter:

- `-eq`—Equality, as in `5 -eq 5` (which is `True`) or `"hello" -eq "help"` (which is `False`)
- `-ne`—Not equal to, as in `10 -ne 5` (which is `True`) or `"help" -ne "help"` (which is `False`, because they are, in fact, equal, and we're testing to see if they're unequal)
- `-ge` *and* `-le`—Greater than or equal to, and less than or equal to, as in `10 -ge 5` (`True`) or `(Get-Date) -le '2020-12-02'` (which will depend on when you run this, and shows how dates can be compared)
- `-gt` *and* `-lt`—Greater than and less than, as in `10 -lt 10` (`False`) or `100 -gt 10` (`True`)

For string comparisons, you can also use a separate set of case-sensitive operators if needed: `-ceq`, `-cne`, `-cgt`, `-clt`, `-cge`, `-cle`.

If you want to compare more than one thing at once, you can use the logical operators `-and` and `-or`. Each takes a subexpression on either side, and we usually enclose them in parentheses to make the line easier to read:

- `(5 -gt 10) -and (10 -gt 100)` is `False`, because one or both subexpressions are `False`.
- `(5 -gt 10) -or (10 -lt 100)` is `True`, because at least one subexpression is `True`.

In addition, the logical `-not` operator reverses `True` and `False`. This can be useful when you're dealing with a variable or a property that already contains `True` or `False`, and you want to test for the opposite condition. For example, if you want to test whether a process isn't responding, you could do the following (you'll use `$_` as a placeholder for a process object):

```
$_.Responding -eq $False
```

PowerShell defines `$False` and `$True` to represent the `False` and `True` Boolean values. Another way to write that comparison is as follows:

```
-not $_.Responding
```

Because `Responding` normally contains `True` or `False`, the `-not` reverses `False` to `True`. If the process isn't responding (meaning `Responding` is `False`), your comparison will return `True`, indicating that the process is "not responding." We prefer the second technique because it reads, in English, more like what we're testing for: "I want to see if the process isn't responding." You'll sometimes see the `-not` operator abbreviated as an exclamation mark (`!`).

A couple of other comparison operators are useful when you need to compare strings of text:

- `-like`—Accepts `*`, `?`, and `[]` as wildcards, so you can compare to see if `"Hello" -like "*ll*"` (that would be `True`). The reverse is `-notlike`, and both are case insensitive; use `-clike` and `-cnotlike` for case-sensitive comparisons. You can find the other available wildcards in the `about_Wildcards` help file.
- `-match`—Makes a comparison between a string of text and a regular expression pattern. Its logical opposite is `-notmatch`, and as you might expect, `-cmatch` and `-cnotmatch` provide case-sensitive versions. Regular expressions are covered in a later chapter of this book.

The neat thing about the shell is that you can run almost all of these tests right at the command line (the exception is the one using the `$_` placeholder—it won't work by itself, but you'll see where it will work in the next section).

TRY IT NOW Go ahead and try any—or all—of these comparisons. Type them on a line—for example, `5 -eq 5`—press Enter, and see what you get.

You can find the other available comparison operators in the `about_Comparison_Operators` help file, and you'll learn about a few of the other ones in chapter 25.

12.4 Filtering objects out of the pipeline

Once you've written a comparison, where do you use it? Well, you can use it with the shell's generic filtering cmdlet, `Where-Object`.

For example, do you want to get rid of all processes but the ones using more than 100 MB of memory (`WorkingSet`)?

```
Get-Process | Where-Object -FilterScript {$_.WorkingSet -gt 100MB}
```

The `-FilterScript` parameter is positional, which means you'll often see this typed without it:

```
Get-Process | Where-Object {$_.WorkingSet -gt 100MB}
```

If you get used to reading that aloud, it sounds sensible: "where `WorkingSet` greater than 100 MB." Here's how it works: when you pipe objects to `Where-Object`, it examines each one of them using its filter. It places one object at a time into the `$_` placeholder and then runs the comparison to see whether it's `True` or `False`. If it's `False`, the object is dropped from the pipeline. If the comparison is `True`, the object is piped out of `Where-Object` to the next cmdlet in the pipeline. In this case, the next cmdlet is `Out-Default`, which is always at the end of the pipeline (as we discussed in chapter 11) and which kicks off the formatting process to display your output.

That `$_` placeholder is a special creature: you've seen it used before (in chapter 10), and you'll see it in one or two more contexts. You can use this placeholder only in the specific places where PowerShell looks for it, and this happens to be one of those places. As you learned in chapter 10, the period tells the shell that you're not comparing the entire object, but rather just one of its properties, `WorkingSet`.

We hope you're starting to see where `Get-Member` comes in handy. It gives you a quick and easy way to discover the properties of an object, which lets you turn around and use those properties in a comparison like this one. Always keep in mind that the column headers in PowerShell's final output don't always reflect the property names. For example, run `Get-Process`, and you'll see a column like `PM(MB)`. Run `Get-Process | Get-Member`, and you'll see that the actual property name is `PM`. That's an important distinction: always verify property names by using `Get-Member`; don't use a `Format-` cmdlet.

Above and beyond

PowerShell v3 introduced a new "simplified" syntax for `Where-Object`. You can use it only when you're doing a single comparison; if you need to compare multiple items, you still have to use the original syntax, which is what you've seen in this section.

Folks debate whether or not this simplified syntax is helpful. It looks something like this:

```
Get-Process | where WorkingSet -gt 100MB
```

Obviously, that's a bit easier to read: it dispenses with the curly brackets `{}` and doesn't require the use of the awkward-looking `$_` placeholder. But this new syntax doesn't mean you can forget about the old syntax, which you still need for more-complex comparisons:

```
Get-Process | Where-Object {$_.WorkingSet -gt 100MB -and $_.CPU -gt 100}
```

What's more, there are years' worth of examples out on the internet that all use the old syntax, which means you have to know it to use them. You also have to know the new syntax, because it will now start cropping up in developers' examples. Having to know two sets of syntax isn't exactly "simplified," but at least you know what's what.

12.5 Using the iterative command-line model

We want to go on a brief tangent with you now to talk about what we call the PowerShell iterative command-line model. The idea behind this model is that you don't need to construct these large, complex command lines all at once and entirely from scratch. Start small.

Let's say you want to measure the amount of virtual memory being used by the 10 most virtual-memory-hungry processes. But if PowerShell itself is one of those processes, you don't want it included in the calculation. Let's take a quick inventory of what you need to do:

1. Get processes.
2. Get rid of everything that's PowerShell.
3. Sort the processes by virtual memory.
4. Keep only the top 10 or bottom 10, depending on how you sort them.
5. Add up the virtual memory for whatever is left.

We believe you know how to do the first three steps. The fourth is accomplished using your old friend, `Select-Object`.

TRY IT NOW Take a moment and read the help for `Select-Object`. Can you find any parameters that would enable you to keep only the first or last object in a collection?

We hope you found the answer. Finally, you need to add up the virtual memory. This is where you need to find a new cmdlet, probably by doing a wildcard search with `Get-Command` or `Help`. You might try the `Add` keyword, or the `Sum` keyword, or even the `Measure` keyword.

TRY IT NOW See if you can find a command that would measure the total of a numeric property like virtual memory. Use `Help` or `Get-Command` with the `*` wildcard.

As you're trying these little tasks (and not reading ahead for the answer), you're making yourself into a PowerShell expert. Once you think you have the answer, you might start in on the iterative approach.

To start, you need to get processes. That's easy enough:

```
Get-Process
```

TRY IT NOW Follow along in the shell and run these commands. After each, examine the output, and see if you can predict what you need to change for the next iteration of the command.

Next, you need to filter out what you don't want. Remember, *filter left* means you want to get the filter as close to the beginning of the command line as possible. In this case, you'll use `Where-Object` to do the filtering, because you want it to be the next cmdlet in the pipeline. That's not as good as having the filtering occur on the first cmdlet, but it's better than filtering later on down the pipeline.

In the shell, press the up arrow on the keyboard to recall your last command, and then add the next command:

```
Get-Process | Where-Object { $_.Name -notlike 'pwsh*' }
```

You're not sure if it's `pwsh` or `pwsh.exe`, so you use a wildcard comparison to cover all your bases. Any process that isn't like those names will remain in the pipeline.

Run that to test it, and then press the up arrow again to add the next bit:

```
Get-Process | Where-Object { $_.Name -notlike 'pwsh*' } |
Sort-Object VM -Descending
```

Pressing Enter lets you check your work, and the up arrow lets you add the next piece of the puzzle:

```
Get-Process | Where-Object  { $_.Name -notlike 'pwsh*' } |
Sort-Object VM -Descending | Select -First 10
```

Had you sorted in the default ascending order, you would have wanted to keep the `-last 10` before adding this last bit:

```
Get-Process | Where-Object { $_.Name -notlike 'pwsh*' } |
Sort-Object VM -Descending | Select -First 10 |
Measure-Object -Property VM -Sum
```

We hope you were able to figure out at least the name of that last cmdlet, if not the exact syntax used here.

This model—running a command, examining the results, recalling it, and modifying it for another try—is what differentiates PowerShell from more traditional scripting languages. Because PowerShell is a command-line shell, you get those immediate results, as well as the ability to quickly and easily modify your command if the results aren't what you expect. You should also be seeing the power you have when you combine even the handful of cmdlets you've learned up to this point in the book.

12.6 Common points of confusion

Anytime we introduce `Where-Object` in a class, we usually come across two main sticking points. We tried to hit those concepts hard in the preceding discussion, but if you have any doubts, we'll clear them up now.

12.6.1 Filter left, please

You want your filtering criteria to go *as close to the beginning of the command line as possible*. If you can accomplish the filtering you need in the first cmdlet, do so; if not, try to filter in the second cmdlet so that the subsequent cmdlets have as little work to do as possible.

Also, try to accomplish filtering as close to the source of the data as possible. For example, if you're querying processes from a remote computer and need to use `Where-Object`—as we did in one of this chapter's examples—consider using PowerShell remoting to have the filtering occur on the remote computer, rather than bringing all of the objects to your computer and filtering them there. You'll tackle remoting in chapter 13, and we mention this idea of filtering at the source again there.

12.6.2 When $_ is allowed

The special `$_` placeholder is valid only in the places where PowerShell knows to look for it. When it's valid, it contains one object at a time from the ones that were piped into that cmdlet. Keep in mind that what's in the pipeline can and will change throughout the pipeline, as various cmdlets execute and produce output.

Also be careful of nested pipelines—the ones that occur inside a parenthetical command. For example, the following can be tricky to figure out:

```
Get-Process -Name (Get-Content c:\names.txt |
Where-Object -filter { $_ -notlike '*daemon' }) |
Where-Object -filter { $_.WorkingSet -gt 128KB }
```

Let's walk through that:

1. You start with `Get-Process`, but that isn't the first command that will execute. Because of the parentheses, `Get-Content` will execute first.
2. `Get-Content` is piping its output—which consists of simple `String` objects—to `Where-Object`. That `Where-Object` is inside the parentheses, and within its filter, `$_` represents the `String` objects piped in from `Get-Content`. Only those strings that don't end in *daemon* will be retained and output by `Where-Object`.
3. The output of `Where-Object` becomes the result of the parenthetical command, because `Where-Object` was the last cmdlet inside the parentheses. Therefore,

all of the names that don't end in *daemon* will be sent to the `-Name` parameter of `Get-Process`.

4. Now `Get-Process` executes, and the `Process` objects it produces will be piped to `Where-Object`. That instance of `Where-Object` will put one service at a time into its `$_` placeholder, and it will keep only those services whose `WorkingSet` property is greater than `128KB`.

Sometimes we feel like our eyes are crossing with all the braces and curly braces, periods, and parentheses, but that's how PowerShell works, and if you can train yourself to walk through the command carefully, you'll be able to figure out what it's doing.

12.7 Lab

Remember that `Where-Object` isn't the only way to filter, and it isn't even the one you should turn to first. We've kept this chapter brief to allow you more time to work on the hands-on examples. Keeping in mind the principle of *filter left*, try to accomplish the following:

1. Get the commands from the `PSReadLine` module.
2. Get the commands using the verb `Get` from the `PSReadLine` module.
3. Display all files under /usr/bin that are larger than 5 MB.
4. Find all modules on the PowerShell Gallery that start with `PS` and the author starts with Microsoft.
5. Get the files in the current directory where the `LastWriteTime` is in the last week. (Hint: `(Get-Date).AddDays(-7)` will give you the date from a week ago.)
6. Display a list of all processes running with either the name `pwsh` *or* the name `bash`.

12.8 Lab answers

```
1 Get-Command -Module PSReadLine
2 Get-Command Get-* -Module PSReadLine
3 Get-ChildItem /usr/bin/* | Where-Object {$_.length -gt 5MB}
4 Find-Module -Name PS* | Where-Object {$_.Author -like 'Microsoft*'}
5 Get-ChildItem | where-object LastWriteTime -ge (get-date).AddDays(-7)
6 Get-Process -Name pwsh,bash
```

12.9 Further exploration

Practice makes perfect, so try filtering some of the output from the cmdlets you've already learned about, such as `Get-ChildItem`, `Get-Process`, and even `Get-Command`. For example, you might try to filter the output of `Get-Command` to show only cmdlets. Or use `Test-Connection` to ping several computers or websites (such as google.com or facebook.com), and show the results only from computers that didn't respond. We're not suggesting that you need to use `Where-Object` in every case, but you should practice using it when it's appropriate.