Timothy L. Warner

Sams Teach Yourself

Windows PowerShell®

in 24 Hours

SAMS

Timothy Warner

Sams **Teach Yourself**

Windows **PowerShell®**

in **24 Hours**

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

- **-le** (less than or equal to)
- **-like** (wildcard comparison)
- **-notlike** (wildcard comparison)
- **-in** (contained in an array)
- **-notin** (not contained in an array)
- **-contains** (contains the specified value)
- **-notcontains** (does not contain the specified value)

NOTE

The foregoing isn't a complete list; see the help file about_Comparison_Operators for the full information.

You might be familiar with the traditional mathematical comparison operators such as =, >, <, and so forth. The PowerShell team explicitly chose these textual representations to avoid conflict with the mathematical symbols in other contexts.

For example, in Linux and Cmd.exe you can redirect output by using the greater than symbol:

```
dir > cdrive.txt
```

We can actually do this in PowerShell as well; hence you see the need for alternatives.

The best way for you to get comfortable with using these comparison operators is to actually use them. To that point, work through the following Try It Yourself exercise on that very subject.

▼ TRY IT YOURSELF

Playing with PowerShell Comparison Operators

In this Try It Yourself exercise, you'll gain some experience in using PowerShell comparison operators "on their own" as well as in the context of the pipeline.

To complete this exercise, work from an elevated PowerShell console session.

1. Let's begin by running some simple expressions, all of which return either Boolean true or false. I won't give you the answers here; half the fun is thinking about the expected result and then comparing your idea with the actual output:

```
2 + 3
50 – 5
33 * 3
100 / 4
```

```
8 -ge 8
8 -lt 6
4 + 2 * 3 * (20 / 3)
```

The last expression was interesting because it reminds us of the order of mathematical operations. (Remember the old mnemonic PEMDAS?) Feel free to put parentheses around any expression or subexpression that you need PowerShell to evaluate first.

2. Now let's try some more interesting examples:

```
"Spam" -eq "spam"
"Spam -ceq "spam"
```

What did you see? You can prepend **c** to your comparison operator to make it case sensitive. By default, string comparisons are not case sensitive.

3. Let's do more with strings:

```
"abc" -ne "abc"
"Windows PowerShell" -like "*shell"
```

The **–like** operator is excellent when you need to use fuzziness in your expressions. For instance, combining **–like** with the asterisk in the last example says, "Does 'Windows PowerShell' match on any word ending in 's-h-e-l-l'?"

4. On to more generic examples:

```
"abc", "def" -contains "def"
"def" -in "abc", "def"
```

5. And now we'll consider a few practical situations where the use of comparison operators helps us with our PowerShell expressions.

For instance, here we can retrieve a list of PowerShell commands available on our system that are of the **CommandType** cmdlet:

```
Get-Command | Where-Object {$_.CommandType -eq "cmdlet"}
```

6. What if we need to retrieve any PowerShell command that includes the string **clear** in its name?

```
PS C:\> Get-Command | Where-Object {$_.name -like "*clear*"}

CommandType     Name                                               Source
-----------     ----                                               ------
Function        Clear-AssignedAccess
AssignedA...
Function        Clear-BCCache                                      BranchCache
Function        Clear-BitLockerAutoUnlock                          BitLocker
Function        Clear-Disk                                         Storage
```

```
Function        Clear-DnsClientCache                                DnsClient
Function        Clear-FileStorageTier                               Storage
Function        Clear-Host
Cmdlet          Clear-Content
Microsoft...
Cmdlet          Clear-EventLog
Microsoft...
Cmdlet          Clear-History
Microsoft...
Cmdlet          Clear-Item
Microsoft...
Cmdlet          Clear-ItemProperty
Microsoft...
Cmdlet          Clear-KdsCache                                      Kds
Cmdlet          Clear-Tpm
TrustedPl...
Cmdlet          Clear-Variable
Microsoft...
Cmdlet          Clear-WindowsCorruptMountPoint                      Dism
```

If you're anything like I am, you'll be using the asterisk wildcard operator all the time. After all, given how many thousands of PowerShell commands exist in the world today, I daresay nobody has all their names memorized, including Jeffrey Snover, the creator of Windows PowerShell.

NOTE

If you've ever played the awesome game Minecraft, you know that the cardinal rule in that game is never to dig straight up or straight down lest you find yourself in a lava pool.

With Windows PowerShell, the equivalent mantra is "filter left, format right." What this means is that the formatting commands that you'll learn about in the next couple of hours terminate the pipeline in most cases.

Therefore, you want to put your filtering commands as far to the left in the pipeline as possible, and then save the filtering, exporting, and converting commands for the end.

Measuring Objects

One of the features of Microsoft Excel that I especially enjoy is its ability to rapidly give me measurement data based on my selection. Take a look at Figure 7.1 to see what I mean.

We can do all of this and more by invoking the **Measure-Object** cmdlet. Here is the syntax from the command's first parameter set, which is the one that I use all the time:

```
Measure-Object [[-Property] <String[]>] [-Average ] [-InputObject <PSObject>]
[-Maximum ] [-Minimum ] [-Sum ] [<CommonParameters>]
```

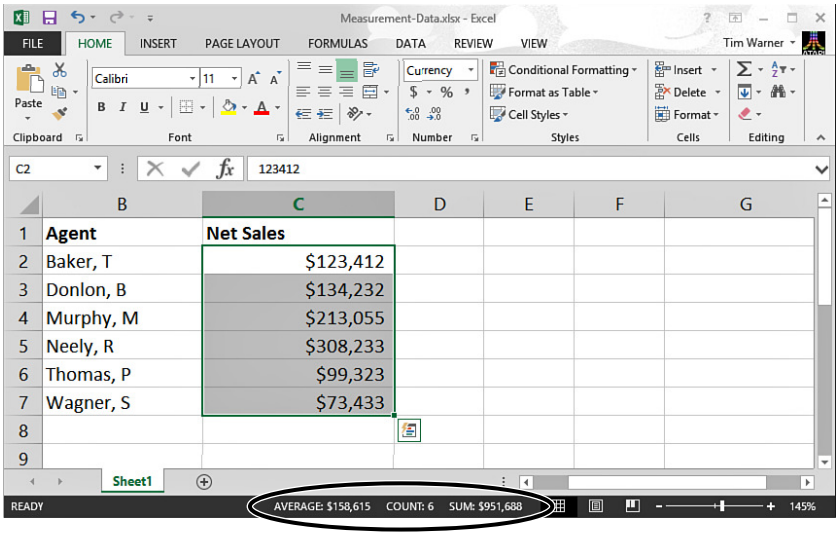

FIGURE 7.1
Microsoft Excel provides easy access to common measurement data.

The foregoing syntax doesn't tell us this directly, but **Measure-Object** (aliased to **measure**) performs the count operation by default. For instance:

```
PS C:\> Get-ChildItem -Path "C:\Windows\System32" | Measure-Object

Count     : 3397
Average   :
Sum       :
Maximum   :
Minimum   :
Property  :
```

Delving Deeper into Measure-Object

How about we try a more interesting example? Imagine we had a CSV file named student-grades.csv that had the following contents:

```
Lastname,grade
Abbott,81
Barclay,77
Miller,92
Night,66
Potter,80
Singleton,96
Williams,54
Zion,90
```

Now let's build a pipeline:

```
PS C:\Users\Tim\Desktop> Import-Csv -Path .\student-grades.csv |
➥Measure-Object -Property grade -Average -Maximum -Minimum

Count    : 8
Average  : 79.5
Sum      :
Maximum  : 96
Minimum  : 54
Property : grade
```

"Cool beans!" as we used to say in the 1980s. You may be wondering two things, like I did at first:

- How does the **–Property** parameter work?
- How did PowerShell know we wanted to work on the grade field in the CSV file?

To the first point, let's consult the **Measure-Object** help:

```
PS C:\Users\Tim\Desktop> Get-Help Measure-Object -Parameter Property

-Property <String[]>
    Specifies one or more numeric properties to measure. The default is the
    Count (Length) property of the object.

    Required?                    false
    Position?                    1
    Default value                Count
    Accept pipeline input?       false
    Accept wildcard characters?  false
```

Aha. We learn that the **–Property** parameter is positional in the first position, and that it defaults to performing a count. Mystery solved. We also see that the parameter expects a string value.

Honestly, I wonder whether the help file's statement that the **–Property** parameter doesn't accept pipeline input is an error. It's crucial to note that the PowerShell help system is not infallible by any means. In point of fact, one of the beauties of updateable help in PowerShell is that we see those fixes as they are made by the Windows PowerShell team.

At any rate, let's see the data that comes out of the **Import-Csv** part of the pipeline:

```
PS C:\Users\Tim\Desktop> Import-Csv –Path .\student-grades.csv | Get-Member

   TypeName: System.Management.Automation.PSCustomObject
```

```
Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
grade       NoteProperty System.String grade=81
Lastname    NoteProperty System.String Lastname=Abbott
```

Okay. So PowerShell dynamically creates a **NoteProperty** for each column header in the CSV file. The data type is string, which is completely compatible with the data that **Measure-Object** expects.

What Measure-Object Is and What It Isn't

We need to keep in mind that **Measure-Object** is excellent at giving us numeric statistics, but it won't tell us anything else about our data.

For instance, to continue our preceding example, how could we generate a report that lists the students with the top three grades? Like this:

```
$csv = Import-Csv –Path .\student-grades.csv
$csv | Sort-Object –Property score –desc | Select-Object –First 3

Lastname                                 grade
--------                                 -----
Singleton                                96
Potter                                   80
Zion                                     90
```

You can use the **Measure-Object** command to calculate text file statistics as well. Let's imagine that we have the following text file, boilerplate.txt:

> Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed et nulla mattis, luctus libero at, semper dolor. Aenean sed arcu a metus tincidunt mollis. Curabitur consectetur, arcu at rhoncus rhoncus, ante sem cursus mauris, sed lobortis erat dui sit amet odio. Duis nec sollicitudin nunc, sed lacinia ante. Sed egestas quis lorem tristique aliquam. Proin scelerisque vulputate leo, in ornare mi sodales sed. Sed quis quam finibus velit placerat auctor. Quisque aliquam justo dolor, ac rutrum elit tincidunt a. Quisque molestie cursus justo vitae dictum.
>
> Aliquam ornare nunc sit amet purus lobortis elementum. Quisque ornare rhoncus pellentesque. Quisque a quam vehicula urna ultricies ornare. Maecenas vel mi vitae ipsum tristique dapibus vitae in libero. Maecenas a turpis et magna facilisis tempus. Aliquam maximus placerat pulvinar. Maecenas vitae mauris a lorem vestibulum porttitor et eu eros. Donec vestibulum lectus in diam volutpat, a sollicitudin augue vehicula. Integer sed sapien