23 Adding logic and loops

Looping (or going through a list of objects one at a time) is a fundamental concept in any language, and PowerShell is no exception. There will come a time when you will need to execute a block of code numerous times. PowerShell is well equipped to handle this for you.

23.1 Foreach and Foreach-Object

This section may be a bit confusing, as there is a difference between `Foreach` and `Foreach-Object`. Take a look at figure 23.1 for a visual representation of how `Foreach` works.

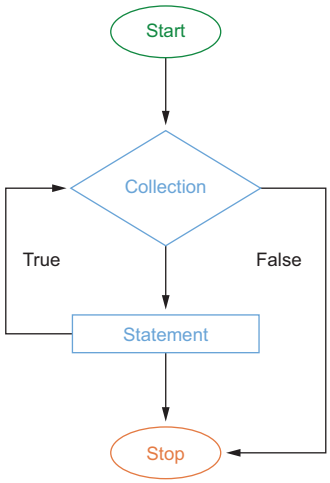

Figure 23.1 Diagram of how `Foreach` works

23.1.1 Foreach

Probably the most common form of looping is the `Foreach` command. `Foreach` allows you to iterate through a series of values in a collection of items, such as an array. The syntax of a `Foreach` command is

```
Foreach (p
temporary variable IN collection object)
{Do Something}
```

The process block (the part surrounded by `{}`) will execute as many times as the number of collection objects. Let's look at the following command and break it down:

```
PS C:\Scripts> $array = 1..10
PS C:\Scripts> foreach ($a in $array) {Write-output $a}
```

First, we made a variable called `$array` that will contain an array of numbers from 1 to 10. Next, we are making a temporary variable (`$a`) and assigning it to the current item in the collection that we are working with. The variable is available only inside the script block and will change as we iterate through the array.

Finally, the script block represented by the curly braces `{ }` will output `$a` to the screen (figure 23.2).

```
Administrator: PowerShell

PS C:\Scripts> $array = 1..10
PS C:\Scripts> foreach ($a in $array){Write-Host $a}
1
2
3
4
5
6
7
8
9
10
PS C:\Scripts>
```

Figure 23.2 Writing the output of an array using `foreach`

23.1.2 Foreach-Object

The `Foreach-Object` cmdlet performs an operation defined in a script block on each item in the input collection objects. Most frequently, the `Foreach-Object` is called via the pipeline.

> **TIP** Use `Foreach` if you are looping through multiple objects, and use `Foreach-Object` if you are using it in the pipeline.

Let's look at the command `Get-ChildItem | ForEach-Object {$_.name}`. First, we are running the command `Get-ChildItem` and sending the objects down the pipeline to the `Foreach-Object` cmdlet.

Next, we are saying for every item received from `Get-ChildItem`, run the command `$_.name` (figure 23.3). If you recall from earlier in the text, `$_` is simply the current object in the pipeline. By using `$_.Name`, we are taking the `name` property from the object and displaying it on the screen.

For both the `Foreach` and `Foreach-Object` cmdlets, the commands are executed sequentially, meaning it will take `item[0]`, run the commands you have specified, followed by the following `item[1]`, and so on until the input collection is empty. Usually this isn't a problem, but eventually, if you have a lot of commands in the process block or your input collection is enormous, you can see where executing these one at a time would impact your script's run time.

```
PS /mnt/c/Users/James> Get-ChildItem | Foreach-Object {$_.Name}
3D Objects
AppData
Application Data
Contacts
Cookies
Creative Cloud Files
Documents
Downloads
Favorites
```

Figure 23.3 This shows how to use `foreach-object` with the pipeline.

Hopefully, before you started diving into the chapter, you used the help feature to look at all the parameters available for `Foreach-Object`.

TRY IT NOW Run `get-help Foreach-Object` and review the results.

Above and beyond

The `%` is also an alias for the `ForEach-Object` command. The command from earlier could have been written

```
Get-ChildItem | %{$_.name}
```

which would have yielded the same results. But let's remember that it is always best to use full cmdlet names.

23.1.3 *Foreach-Object -Parallel*

As we mentioned before, the main drawback with the `Foreach-Object` command has been that it runs sequentially. There have been a few community-driven modules to help enable a parallel feature for the `Foreach-Object` command. With the introduction of PowerShell 7 (preview 3), a new `-Parallel` parameter was added to the `Foreach-Object` command. Instead of the command(s) being run sequentially, we can now run the same commands on most or all of our input objects at the same time. For example, suppose you are creating 1,000 new users in Active Directory. You could run the command

```
import-csv c:\scripts\newusers.csv |
ForEach-Object {New-aduser -Name $_.Name }
```

which would run the `New-Aduser` command 1,000 times sequentially. Or you can run the command with the `Parallel` parameter:

```
import-csv c:\scripts\newusers.csv |
ForEach-Object -Parallel {New-aduser -Name $_.Name }
```

The following command takes an array of numbers (1–5) and pipes it to a traditional `Foreach-Object` command, writes the output to the screen, and sleeps for 2 seconds (figure 23.4).

```
1..5 | ForEach-Object {Write-Output $_; start-sleep -Seconds 2}
```

```
PS C:\Scripts> 1..5 | ForEach-Object {Write-Output $_;Start-Sleep -Seconds 2}
1
2
3
4
5
PS C:\Scripts>
```

Figure 23.4 Takes an array, pipes to `Foreach-Object`, then runs a second command

We can see by using the `measure-command` cmdlet that this will take 10 seconds to complete.

```
PS C:\Scripts> measure-command {1..5 | ForEach-Object {Write-Output "$_";
➥ start-sleep -Seconds 2}}

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 10
Milliseconds      : 47
Ticks             : 100471368
TotalDays         : 0.000116286305555556
TotalHours        : 0.00279087133333333
TotalMinutes      : 0.16745228
TotalSeconds      : 10.0471368
TotalMilliseconds : 10047.1368
```

When we add the `-parallel` parameter, we will execute what is inside the command block on all the numbers in the array at once.

```
1..5 | ForEach-Object -parallel {Write-Output "$_"; start-sleep -Seconds 2}
```

By using the `parallel` parameter, we decreased our run time from 10 seconds to 2 seconds.

```
PS C:\Scripts> measure-command {1..5 | ForEach-Object -parallel {Write-Output
➥ "$_"; start-sleep -Seconds 2}}

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 2
Milliseconds      : 70
Ticks             : 20702383
```

```
TotalDays         : 2.39610914351852E-05
TotalHours        : 0.000575066194444444
TotalMinutes      : 0.0345039716666667
TotalSeconds      : 2.0702383
TotalMilliseconds : 2070.2383
```

Because each script block is running simultaneously, the order in which the results are returned to the screen cannot be guaranteed. There is also a throttle limit or the maximum number of script blocks that can be run in parallel at once that we need to make sure you know about—the default is 5. In our example, we had only 5 items in our input collection, so all 5 script blocks were running simultaneously. However, if we change our example from 5 items to 10 items, we will notice the run time changes from 2 seconds to 4 seconds. We can, however, change the throttle limit to a higher one by using the `-throttlelimit` parameter.

```
1..10 | ForEach-Object -parallel {Write-Output "$_"; start-sleep -Seconds 2}
➥ -ThrottleLimit 10
```

TRY IT NOW Change the array to 10 items; then use the `measure-command` cmdlet to see how long it takes to execute.

There is, however, a limitation with the `parallel` feature. In order to run each script block simultaneously, a new runspace is created. This can lead to significant performance degradation if the script blocks you are running are resource intensive.

23.2 While

If you have done any kind of scripting or programming before, then a `while` loop should not be new concept to you. A `while` loop is an iterative loop, or it will run until the terminating condition is satisfied. Like the `Foreach` loop we just talked about, the `while` loop has a script block, where you can put your commands to be executed (figure 23.5). The basic syntax is as follows: `While (condition) {commands}`.

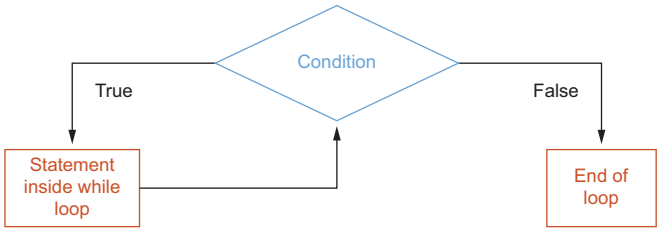

Figure 23.5 Diagram showing how a `while` loop works

- *Condition*—A Boolean (`$True` or `$False`) statement. The loop will execute while the condition is `True` and will terminate when the condition is `False`. Example: `While ($n -ne 10)`.
- *Commands*—Simple or complex commands that you want to execute while the condition is `True`.

Here is a quick example:

```
$n=1
While ($n -le 10){Write-Output $n; $n++}
```

We can also start adding logic operators such as `-and` and `-or` into our condition statement:

```
While ($date.day -ne 25 -and $date.month -ne 12)
{Write-Host "Its not Christmas Yet"}
```

TIP If you were to run the above command, it would run indefinitely unless you happened to run it on `25-December`. Use Ctrl-C to break the execution.

23.3 Do While

As we mentioned before, the `while` loop will execute only while the condition is `true`. But what if you wanted to execute the loop at least once regardless of whether the condition was `true` or not? That is where the `Do While` loop comes into play.

With `Do {commands} While (condition)`, notice that the script block and condition block are reversed. This will allow us to execute the script block at least one time, then evaluate our condition to see if we need to repeat the loop:

```
$date = get-date

do {
    Write-Output "Checking if the month is December"
    $date = $date.AddMonths(1)
} while ($date.Month -ne 12 )
```

23.4 Lab

1. Find a directory that has a lot of items in it. Use a `Foreach` loop and count the number of characters in each filename.
   - Do the same, but this time use the `-parallel` parameter.
2. Start the notepad process (or text editor of your choice); then write a `do while` loop that will display the following text until the process is closed: `$process is open`.

23.5 Lab answers

1.
```
$items = Get-ChildItem SOMEWHERE YOU |CHOSE
foreach ($i in $items){Write-Output "The character length of $i is
➥ "($i).Length}
```
2.
```
start-process notepad
$Process = "notepad"
do {
    Write-Host "$process is open"
} while ((get-process).name -contains "notepad")
```