PowerShell For Loop, ForEach, and Do While/Until Explained

lazyadmin.nl/powershell/powershell-loops-for-foreach-do-while-until

April 6, 2021

One of the most fundamental functions in programming besides “If, Else” are loops. They allow you to process data or run a function until a certain condition is reached. In PowerShell, we can use different types of loops **For loop, ForEach, While, Do While**, and **Do Until**.

Knowing when to use what is important to write better Powershell scripts. Especially the difference between the three while loops is sometimes hard to understand.

In this article, we are going to take a look at the different loop functions that we can use in PowerShell, how to use them, and what the differences are between them.

Powershell For Loop

The For Loop in PowerShell is pretty much the most basic loop that you can use. It iterates a specified number of times through a part of the script until the condition is met.

To use the For loop in PowerShell you will need to specify a counter `$i`, the condition `$i -lt 5` (run as long as $i is less than 5) and increase the counter.

```
For ($i = 0; $i -lt 5; $i++) {
Write-host $i
}
```

Increasing the counter can also be done inside the script. This allows you to only increase the counter when a specific condition is met in your script.

```
For ($i = 0; $i -lt 5) {
Write-Host $i;
$i++
}
```

You can also iterate over an array with a simple For Loop in PowerShell. We can take the **array length** as part of the condition and iterate through the array until we have processed all the items in the array.

It’s easier to use a foreach loop to process an array than a for loop, more about that later.

```
$fruit = @('apple','pear','banana','lemon','lime','mango')
# -le means less or equal to
For ($i = 0; $i -le $fruit.length; $i++) {
Write-Host $fruit[$i];
}
```

PowerShell Foreach 1..10

PowerShell also has a shorthand that you can use for a loop. You can define a range, for example, `1..10` and then loop through each number in the range with a ForEach statement. This allows you to specify how many times you want to run the script and just pipe the script behind it.

```
$path = "C:\temp"
# Create a new file ForEach 1..10 number
1..10 | % {
$newFile = "$path\test_file_" + $_ + ".txt";
New-Item $newFile
}
```

PowerShell Foreach and ForEach-Object Loop

To process each item in an array or collection you can use the ForEach functions in PowerShell.

There are two types of foreach functions in PowerShell, we have **foreach** and **ForEach-Object**. The difference between the two is that `foreach` will load the entire collection in the memory before it starts processing the data, which `ForEach-Object` will process the data one at a time. This makes ForEach-Object perfect to pipe behind another function.

ForEach-Object has two shorthands that you can use, `ForEach` and `%`. The place where you use ForEach determines which version of the two is used.

```
$fruits = @('apple','pear','banana','lemon','lime','mango')
# Foreach - Loads all the fruits in the memory and process it
# Takes 21,4553 miliseconds
Foreach ($fruit in $fruits) {
Write-Host $fruit;
}
#Shorthand for foreach
# Takes 14,3926 miliseconds
$fruits.foreach( {
Write-Host $_;
})
# ForEach-Object
# Takes 7,8812 miliseconds
$fruits | ForEach-Object {Write-Host $_}
# Shorthand for ForEach-Object
# Takes 7.3507 miliseconds
$fruits | ForEach {Write-Host $_}
# Also a Shorthand for ForEach-Object
# Takes 7,2982 miliseconds
```

$fruits | % {Write-Host $_}
As you can see is foreach a bit slower in this example, but the reason for this is it takes a little bit to load the collection into the memory.

If we take a large data set, with 300.000 records for example, then we can see that foreach can be a lot faster than `ForEach-Object`. Only the memory consumption would be a bit higher of foreach.

```
$items = 0..300000
# Takes 180 ms to complete
(Measure-Command { foreach ($i in $items) { $i } }).TotalMilliseconds
# Takes 1300 ms to complete
(Measure-Command { $items | ForEach-Object { $_ } }).TotalMilliseconds
# Takes 850 ms to complete
(Measure-Command { $items.ForEach({ $i })}).TotalMilliseconds
```

When you use which function really depends on your dataset. If you need to process a lot of data, then `ForEach-Object` would be the best choice to avoid memory problems. Small data sets can perfectly be processed with `ForEach`.

Another advantage of `ForEach-Object` is that you can pipe another cmdlet behind it, like exporting it to CSV for example.

ForEach-Object Script Blocks

ForEach-Object allows you to use script blocks which can be really useful when you need to log the process or clean up when the script is completed.

```
$path = "C:\temp"
$totalSize = 0
Get-ChildItem $path | ForEach-Object -Begin {
Write-Host "Processing folder $path"
} -Process {
$totalSize += ($_.length / 1kb)
} -End {
Write-host $totalSize "Kb"
}
```

Powershell Do While Loop

The PowerShell **Do While loop** is used to run a script **as long as the condition is True or met**. You also have the **While loop** in PowerShell, without the Do part. They are basically the same, both run until a condition is met.

If you run the scripts below, you will see that they both great 10 test files in the temp folder. Both run as long as `$i` is less than 10.

```
$i = 0;
```

```
$path = "C:\temp"
# Do While loop
Do {
# Do Something
$newFile = "$path\dowhile_test_file_" + $i + ".txt";
New-Item $newFile
$i++;
}
While ($i -lt 10)
# While loop
$i = 0;
While ($i -lt 10) {
# Do Something
$newFile = "$path\while_test_file_" + $i + ".txt";
New-Item $newFile
$i++;
}
```

But there is **one big difference between the two**. Do While will always run one time atleast, no matter the condition that you are using. The while loop will only run when the condition is met.

Take the following example, we want to run the script, as long `$i` is less than 10. But we have set the counter to 15 to start with. As you will see when you run the script is that the Do While does run one time, and the while does not run at all.

```
$i = 15;
Do {
Write-host "Do While $i is less then 15";
$i++;
}
While ($i -lt 10)
$i = 15;
while ($i -lt 10)
{
Write-host "While $i is less then 15";
$i++;
}
```

You often use a do-while loop in PowerShell when you are waiting for a condition to be met, for example until a process is finished or as in the example below, until the internet connection is offline.

PowerShell won't continue to the Write-Host line until the while condition is met. So in this example, we are testing the internet connection with Test-Connection. It will return true when we have a connection, and otherwise, it will return false.

```
Do {
```

```
Write-Host "Online"
Start-Sleep 5
}
While (Test-Connection -ComputerName 8.8.8.8 -Quiet -Count 1)
Write-Host "Offline"
```

Creating a Sleep Loop in PowerShell

If you want to create a loop that checks if a service is back online, or for a file to be created then the best option is to use a sleep inside your loop.

Without the sleep part, the script will run as fast as possible which can be a couple of 1000 iterations per second or more.

With the **Start-Sleep** cmdlet, we can initiate a little break before the next iteration:

```
Do {
Write-Host "Online"
Start-Sleep 5
}
While (Test-Connection -ComputerName 8.8.8.8 -Quiet -Count 1)
Write-Host "Offline"
```

In the example above we wait 5 seconds before we test the connection to 8.8.8.8 again. You can also use milliseconds instead of seconds the start-sleep command.

```
Start-Sleep -Miliseconds 50
```

PowerShell Do Until Loop

Do Until is pretty much the same as Do While in PowerShell. The only difference is the condition in which they run.

- **Do While** keeps running **as long as the condition is true**. When the condition is false it will stop.
- **Do Until** keeps running **as long as the condition is false**. When the condition becomes true it will stop.

```
Do {
Write-Host "Computer offline"
Start-Sleep 5
}
Until (Test-Connection -ComputerName 'lab01-srv' -Quiet -Count 1)
Write-Host "Computer online"
```

When to use While or Until really depends on what you want to do. Is basis you can say that when the exit condition must be true, then use Until. If the exit condition is negative (false) then you should use While.

Stopping a Do While/Until loop

In PowerShell, we can use **Break** to stop a loop early. It's best to limit the use of breaks in your loops because they make your code harder to read. If you have structured your code well, then you really don't need it.

```
$i=0;
Do {
Write-Host "Computer offline"
Start-Sleep 1
$i++;
if ($i -eq 2) {
Break
}
}
Until (Test-Connection -ComputerName 'test' -Quiet -Count 1)
```

Using Continue in a PowerShell Loop

Another useful method that you can use inside a do-while/until loop in PowerShell is **Continue**. With Continue you can stop the script inside the Do body and continue to the next iteration of the loop.

This can be useful if you have a large script inside a Do block and want to skip to the next item when a certain condition is met.

```
$servers = @('srv-lab01','srv-lab02','srv-lab03')
Foreach ($server in $servers) {
if ((Test-Connection -ComputerName $server -Quiet -Count 1) -eq $false) {
Write-warning "server $server offline"
Continue
}
# Do stuff when server is online
}
```

PowerShell Loop Examples

Below you will find a couple of loop examples that may help to create your own loops.

PowerShell Infinite Loop

You can create an infinite loop in PowerShell with the While loop function. While will keep running as long as the condition is true. So to create an infinite loop we can simply do the following:

```
While ($true) {
# Do stuff forever
```

```
Write-Host 'looping'
# Use Break to exit the loop
If ($condition -eq $true) {
Break
}
}
```

To exit the infinite loop you can use either **Ctrl + C** or based on a condition with **Break**.

Powershell Loop Through Files

To loop through files we are going to use the foreach-object function. Files are objects that we can get with the get-childitem cmdlet, so we can pipe the foreach-object behind it. Another reason to use ForEach-Object instead of foreach is that we don't know upfront how many files we are going to process.

Foreach will load all the data in the memory, which we don't want with an unknown data set.

```
$path = "C:\temp"
$csvItems = 0
Get-ChildItem $path | ForEach-Object {
$_.Name;
If ($_.Extension -eq '.csv') {
$csvItems++;
}
}
Write-host "Total CSV Files are $csvItems";
```

If you want to include the subfolder as well you can use the -recurse option. Use -filter to get only the .csv files for example.

```
Get-ChildItem $path -recurse -filter *.csv
```

Loop through a text file

In PowerShell, you can loop through a text file line for line. This can be really useful if you want to automatically read out log files for example.

The example below works fine for small files, but the problem is that Get-Content will read the entire file into the memory.

```
$lineNr = 0
foreach($line in Get-Content c:\temp\import-log.txt) {
if($line -match 'warning'){
# Work here
Write-warning "Warning found on line $lineNr :"
Write-warning $line
}
```

```
$lineNr++;
}
```

To read and process larger files with PowerShell you can also use the .Net file reader:

```
foreach($line in [System.IO.File]::ReadLines("c:\temp\import-log.txt"))
{
$line
}
```

Wrapping Up

I hope the example helped with creating your PowerShell For, Foreach-Object, Do While, and Until loops. If you have any questions just drop a comment below.

Did you **Liked** this **Article**?
Get the latest articles like this **in your mailbox**
or share this article

I hate spam to, so you can unsubscribe at any time.