

# Cyber Aces

## Module 3 – System Administration

### PowerShell – Flow Control & Output

By Tim Medin and Tom Hessman

Presented by Tim Medin

v15Q1

This tutorial is licensed for personal use exclusively for students and teachers to prepare for the Cyber Aces competition. You may not use any part of it in any printed or electronic form for other purposes, nor are you allowed to redistribute it without prior written consent from the SANS Institute.

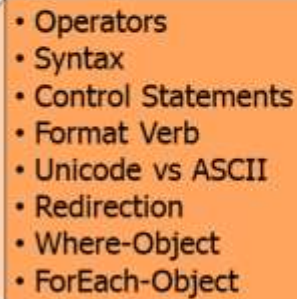
Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

1

Welcome to Cyber Aces, Module 3! This module provides an introduction to the latest shell for Windows, PowerShell. In this session we will discuss flow control and output in PowerShell.

# Course Roadmap

- Introduction
- Cmdlets
- Scripting & Syntax
- **Flow Control & Output**
- Practical Uses
- Conclusions

- 
- Operators
  - Syntax
  - Control Statements
  - Format Verb
  - Unicode vs ASCII
  - Redirection
  - Where-Object
  - ForEach-Object

Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

2

## Course Roadmap

In this section, you will be introduced to PowerShell's flow control and output cmdlets.

# Operators

Get-Help about\_operators

- Comparison
  - eq equals
  - ne not equals
  - gt greater than
  - lt less than
  - le less or equal
  - ge greater or equal
  - match regex match
  - notmatch negative regex match
  - like wildcard match
  - notlike neg. wildcard match
- Arithmetic (Math)
  - + add
  - subtract
  - \* multiply
  - / divide
  - % modulus (remainder)
- Logical
  - and logical and
  - or logical or
  - xor exclusive or
  - not logical not

Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

3

Operator	Description	Example Usage
-eq	Equal to	2 + 2 -eq 4
-lt	Less than	1 -lt 2
-gt	Greater than	2 -gt 1
-ge	Greater than or Equal to	4 -ge 4 4 -ge 3
-le	Less than or equal to	1 -le 1 13 -le 37
-ne	Not equal to	13 -ne 37
-not	Logical Not	-not (2 -eq 1)
!	Logical Not	!(2 -eq 1)
-and	Logical And	(2+2 -eq 4) -and (1+1 -eq 2)
-or	Logical Or	(2+2 -eq 0) -or (1+1 -eq 2)
-like	Match using the wildcard character	"PowerShell" -like "*shell"
-notlike	Opposite of -Like	"PowerShell" -notlike "*bash"
-match	Matches using a Regular Expression and populates the \$matches variable	"Sunday" -match "[A-Z]+"
-notmatch	Does not match on Regular Expression, populates \$matches	"Sunday" -notmatch "[0-9]+"
-contains	Containment operator	\$days -contains "sun"
-notcontains	Opposite of contains	\$days -notcontains "blah"
-replace	Replaces (does not return a Boolean)	"Monday" -replace "Fri" Output: Friday

# Review

- 1) What is the proper syntax to check if "\$a" is greater than 4?

```
$a >> 4  
$a -gt 4  
$a -ge 4  
$a gt 4  
$a > 4
```

- 2) Which of these commands will check if "\$a" ends with the string "find me"?

```
$a -contains "*find me"  
$a -like "find me"  
$a -like "*find me"  
$a -find "find me"  
$a -endswith "find me"
```

## Review

- 1) What is the proper syntax to check if "\$a" is greater than 4?

```
$a >> 4  
$a -gt 4  
$a -ge 4  
$a gt 4  
$a > 4
```

- 2) Which of these commands will check if "\$a" ends with string "find me"?

```
$a -contains "*find me"  
$a -like "find me"  
$a -like "*find me"  
$a -find "find me"  
$a -endswith "find me"
```

# Answers

- 1) What is the proper syntax to check if "\$a" is greater than 4?
  - `$a -gt 4`
  - The `>` operator is used for redirection (see Get-Help about\_redirection)
- 2) Which of these commands will check if "\$a" ends with string "find me"?
  - `$a -like "*find me"`
  - The asterisk at the beginning means it will match anything at the beginning, since there is no asterisk at the end it must exactly match "find me".

Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

5

## Answers

- 1) What is the proper syntax to check if "\$a" is greater than 4?

**`$a -gt 4`**

The `>` operator is used for redirection (see Get-Help about\_redirection)

- 2) Which of these commands will check if "\$a" ends with string "find me"?

**`$a -like "*find me"`**

The asterisk at the beginning means it will match anything at the beginning, since there is no asterisk at the end it must exactly match "find me".

# If...Then...Else

- Use to conditionally execute code

```
if ($a -ne 0) { $b = 4 / $a }
```
- Can use an Else

```
if ($a -eq 0)
{ "Can't div 0" }
else
{ $b = 4 / $a }
```
- Can stack with "elseif"

```
if ($a -eq 0)
{ "zero" }
elseif ($a -gt 0)
{ "positive" }
else
{ "negative" }
```

Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

6

## If...Then...Else

The "If..Then..Else" statement is one of the most basic methods of controlling the flow of a script. The basic syntax of the "If" statement in PowerShell is:

```
if (condition) {do stuff}
elseif (condition) {do other stuff}
elseif (condition) {do other stuff}
...
else {do something else}
```

That is a bit of pseudocode, so let's use a real example to see how it works. Let's say we have a variable "\$a" and we want to know if it is zero.

```
PS C:\> if ($a -eq 0) {"zero"}
```

No output, which must mean the variable is not zero. Let's modify our "If" statement to be a bit more verbose.

```
PS C:\> if ($a -eq 1) {"zero"} else {"non-zero"}
non-zero
```

Now the "If" statement tells us if the variable is non-zero or not, but what if the variable is positive.

```
PS C:\> if ($a -eq 0) {"zero"} elseif ($a -gt 0) {"positive"}
```

No output, we forgot to output something if "\$a" is negative.

```
PS C:\> if ($a -eq 0) {"zero"} elseif ($a -gt 0) {"positive"} else
{"negative"}
negative
```

We've used our "If" statement to let us know if our variable is zero, positive, or negative. There can be multiple "ElseIf" sections. Also, the script block, denoted with curly braces ({}), can contain all sorts of PowerShell magic, including other "If" statements.

# Where-Object Filtering

- If the Where-Object's script block returns true (or has output) then the object is passed further down the pipeline
- The current pipeline object (\$\_) represents each object as it is passed down the pipeline
- Find all files with a size (length) greater than 20MB

```
ls -Recurse | ? { $_.Length -ge 20000000 }
```

  - Gets each file in the file system, and pipe it into Where-Object (alias ?)
  - The script-block is executed, if the results are true then the object is passed down the pipeline. As there is no cmdlet at the end, the file properties are output
- What if we wanted to delete these files? It's really simple!

```
ls -Recurse | ? { $_.Length -ge 20000000 } | del
```

  - The files that make it through the filter are piped into del (alias for Remove-Item) and deleted
  - We could just as easily move the file to another location using Move-Item (alias mi, move, and mv)

```
...<earlier command> | Move-Item -Destination Z:\BigFiles
```
- Find large zip files

```
ls -r | ? { $_.Length -ge 20000000 -and $_.Extension -eq ".zip" }
```

Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

7

## Where-Object Filtering

The "Where-Object" cmdlet (alias "?") was lightly addressed in the variables section, but it deserves more attention. Let's use "Where-Object" to find all files bigger than 20MB.

```
PS C:\> ls -Recurse | ? { $_.Length -ge 20000000 }
```

The "ls" is an alias for "Get-ChildItem". As we saw in the "Common Cmdlets" section, "Get-ChildItem" does a directory listing. The "-Recurse" option recursively searches each directory.

Each object is passed down the pipeline, one at a time (represented by "\$\_"), and each object's length (size) is checked to see if it is greater than 20MB. If it is, then the object is passed further down the pipeline. If not, it is discarded. In this case, "further down the pipeline" is just output. Let's get a little more hi-tech with this example and search for files that are greater than 20MB and have the ".zip" extension.

```
PS C:\> ls -Recurse | ? { $_.Length -ge 20000000 -and $_.Extension -eq ".zip" }
```

Directory: C:\

Mode	LastWriteTime	Length	Name
-a---	10/10/2009 10:10 PM	31415926	mybig.zip

Note: PowerShell version 3 does not require the curly braces or the current pipeline object so the following works in PowerShell version 3.

```
PS C:\> ls -Recurse | ? Length -ge 20000000 -and Extension -eq ".zip"
```

# ForEach-Object

- Used to operate on each object in the pipeline
- Rename all files to Uppercase

```
PS C:\BigFiles> ls | Format-List Name
Name: aaa.zip
Name: bbb.zip
PS C:\BigFiles> ls | % { Move-Item -Path $_
                        -Destination $_.FullName.ToUpper() }
PS C:\BigFiles> ls | Format-List Name
Name: AAA.ZIP
Name: BBB.ZIP
```
- Each file is fed into the ForEach-Object cmdlet (alias %)
- The file is "moved" from its old name to its new upper case name
- The FullName property is a string, which has a "ToUpper" method that returns the string (text) in upper case

Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

8

## ForEach-Object

The ForEach-Object cmdlet is incredibly powerful. Linux has a command similar to this, xargs, but it is not nearly as powerful. If multiple commands are nested it is quite difficult to write (and read) the xargs command. For example, if we wanted to read the contents of a csv file which contained a list of large files, then move those files if they are still too big:

```
PS C:\> Import-Csv largefiles.csv | % { Get-Item $_.FileName } | ? {
    $_.Length -gt 20000000 } | Move-Item -Destination C:\StillTooBig
```

1. Import the csv file
2. Use the ForEach-Object cmdlet (alias %) to get the file (get-item) using the FileName column in the csv
3. Use Where-Object to only pass files that are greater than 20MB
4. Move the file to destination, this command receives the original path from the object passed down the pipeline

The "ForEach-Object" cmdlet is very powerful, and extremely useful. If you use PowerShell at all, you will need to know this command.



# Select-Object

- Used to filter out properties so only certain properties are passed down the pipeline
- Alias select
- Less data down the pipeline can mean increased speed

```
PS C:\> ls | Select-Object Name,Length
```

Name	length
Program Files	
Users	
Windows	
autoexec.bat	24
config.sys	10

Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

9

## Select-Object

The Select-Object cmdlet is used for removing properties from objects as they pass down the pipeline. It can also be used to select the first X lines, the last X lines, and/or skip the first X lines.

One of the most powerful, but advanced options, is to add properties to objects as they move down the pipeline. If you imported a csv file and it included two columns, firstname and lastname, you could create another column named Fullname using a command similar to the one below.

```
PS C:\> Import-Csv users.csv | Select *, @{Name="FullName";  
Expression={$_.Firstname + " " + $_.Lastname}}
```

# Output and the Format Verb

- Default format is Table (Format-Table, alias ft)
- Can be used to show only certain columns  

```
PS C:\> ps | ft Name, Id, Handles
```

Name	Id	Handles
explorer	1524	654
powershell	2052	657
- List, the other common format (Format-List, alias fl)  

```
PS C:\> ps | fl
```

\_\_NounName : Process  
Name : explorer  
Handles : 710  
VM : 213209088  
WS : 32026624  
...
- Only default fields are displayed, use the \* to display all  

```
PS C:\> ps | fl *
```

Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

10

## Output and the Format Verb

By default, most commands just display the basic properties of an object. That is a good thing, because otherwise the screen would be mostly cluttered with useless information. But what if we do want more or all of the information? Let's see how to view the objects in different methods. By default, the "Get-Process" cmdlet's output is similar to this:

```
PS C:\> Get-Process
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
711	24	20024	31196	203	13.23	1524	explorer
...							

The output above is the "Table" format, and can be explicitly chosen by using the following command:

```
PS C:\> Get-Process | Format-Table
```

We can use the "Format" cmdlets to show specific properties. Also, here is the alias for Format-Table (ft).

```
PS C:\> ps | ft Name, Id
```

Name	Id
explorer	1524

The other most notable output format is the "List" format. To output the results in a list format, use "Format-List" or the alias "fl".

```
PS C:\> ps | fl
```

Id : 1524  
Handles : 712  
CPU : 13.6396128  
Name : explorer  
...

Uh oh, in this case the list format actually displays less information. This isn't typically the case, but it does this for "Process" objects. To display everything use the \* for the list of properties.

```
PS C:\> ps | fl *
```

# File Output

- Out-File
    - Send output to a file
    - Writes Unicode, a character set that works for all languages and special, language specific, characters
    - Unicode may not be handled properly by some editors or parsers
    - Use the -Encoding parameter to specify ASCII
    - The redirection operator (>) outputs Unicode as well
  - Export-CSV will save the objects, as objects, in a csv format
    - Objects can be imported, as objects, with Import-CSV
    - A good way to export data for use with Excel
    - Can be used to "pause" the current work and return to it later
    - Both \$a and \$b will have the same contents and can be used in the same way
- ```
$a = Get-ChildItem C:\  
$a | Export-CSV my.csv  
$b = Import-CSV my.csv
```

Cyber Aces Module 3 - ©2015 The SANS Institute. Redistribution Prohibited.

11

## File Output

The Out-File cmdlet is used to write output from the pipeline to a file. However, it writes the content using Unicode. Unicode represents each character using more than one byte and it may not work with some editors or parsers. Unicode does support hundreds of different character sets and the special characters associated with those languages.

Reference: <https://en.wikipedia.org/wiki/Unicode>

The Export-CSV cmdlet is very powerful. It is very handy for parsing and manipulating data from other sources. It can also be used to save variables or arrays for later use. Each row will be an object, and each column is a property of that object. To export objects, simply pipe it into Export-CSV. To retrieve the data, use the Import-CSV cmdlet to read the csv file.

# Review

- 1) Which of these commands will find all files that end with the extension ".txt"?

```
ls -Recurse -Extension -eq ".txt"
ls -r | ? { $_.Extension -eq ".txt" }
ls -r | ? { $_.Extension ".txt" }
ls -r | ? { $_.Extension == ".txt" }
ls -r | ? { $_.Extension = ".txt" }
```

- 2) Which command would find all files larger than 20MB where the file name begins with "archive"?

```
ls -r | ? { $_.Length -gt 20000000 && $_.Name -like "archive" }
ls -r | ? { $_.Length -gt 20000000 and $_.Name -like "archive*" }
ls -r | ? { $_.Length -gt 20000000 -and $_.Name -like "archive*" }
ls -r | ? { $_.Length -gt 20000000 -and $_.Name -like "*archive*" }
ls -r | ? { $_.Length -gt 20000000 -and $_.Name -like "archive" }
```

## Review

- 1) Which of these commands will find all files that end with the extension ".txt"?

```
ls -Recurse -Eq ".txt"
ls -r | ? { $_.Extension -eq ".txt" }
ls -r | ? { $_.Extension ".txt" }
ls -r | ? { $_.Extension == ".txt" }
ls -r | ? { $_.Extension = ".txt" }
```

- 2) Which command would find all files larger than 20MB where the file name begins with "archive"?

```
ls -r | ? { $_.Length -gt 20000000 && $_.Name -like "archive" }
ls -r | ? { $_.Length -gt 20000000 and $_.Name -like "archive*" }
ls -r | ? { $_.Length -gt 20000000 -and $_.Name -like "archive*" }
ls -r | ? { $_.Length -gt 20000000 -and $_.Name -like "*archive*" }
ls -r | ? { $_.Length -gt 20000000 -and $_.Name -like "archive" }
```

# Answers

- 1) Which of these commands will find all files that end with the extension ".txt"?
  - `ls -r | ? { $_.Extension -eq ".txt" }`
  - The proper comparison operator is "-eq"
- 2) Which command would find all files larger than 20MB where the file name begins with "archive"?
  - `ls -r | ? { $_.Length -gt 20000000 -and $_.Name -like "archive*" }`
  - The proper Logical AND operator is -and
  - Since "archive" must be at the beginning we don't use a wildcard (\*) in front; however, we do need it at the end of the search string. Without the wildcard the statement will only match a file that is exactly named "archive" and won't match the file "archive1.zip"

## Answers

- 1) Which of these commands will find all files that end with the extension ".txt"?

```
ls -r | ? { $_.Extension -eq ".txt" }
```

The proper comparison operator is "-eq"

- 2) Which command would find all files larger than 20MB where the file name begins with "archive"?

```
ls -r | ? { $_.Length -gt 20000000 -and $_.Name -like "archive*" }
```

The proper Logical AND operator is -and

Since "archive" must be at the beginning we don't use a wildcard (\*) in front; however, we do need it at the end of the search string. Without the wildcard the statement will only match a file that is exactly named "archive" and won't match the file "archive1.zip"

## Exercise Complete!

- Congratulations! You have completed the session on flow control & output in PowerShell.

Exercise Complete