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.
This training material was originally developed to help students, teachers, and mentors prepare for the Cyber Aces Online Competition. This module focuses on the basics of what an operating systems is as well as the two predominant OS's, Windows and Linux. In this session we will provide a walkthrough of the installation a Windows VM using VMware Fusion (MacOS) and VMware Player (Windows & Linux). These sessions include hands-on labs, but before we begin those labs we need to install the operating systems used in those labs. We will be using VMware to virtualize these operating systems. You can use other virtualization technologies if you like, but instruction for their setup and use are not included in this training.

The three modules of Cyber Aces Online are Operating Systems, Networking, and System Administration.

For more information about the Cyber Aces program, please visit the Cyber Aces website at https://CyberAces.org/.
Module 3 - System Administration
PowerShell
• Introduction
• Cmdlets
• Scripting, Variables & Syntax
• Flow Control & Output
• Practical Uses

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

### Example Usage

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>-eq</td>
<td>Equal to</td>
<td>2 + 2 -eq 4</td>
</tr>
<tr>
<td>-lt</td>
<td>Less than</td>
<td>1 -lt 2</td>
</tr>
<tr>
<td>-gt</td>
<td>Greater than</td>
<td>2 -gt 1</td>
</tr>
<tr>
<td>-ge</td>
<td>Greater than or Equal to</td>
<td>4 -ge 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 -ge 3</td>
</tr>
<tr>
<td>-le</td>
<td>Less than or equal to</td>
<td>1 -le 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 -le 37</td>
</tr>
<tr>
<td>-ne</td>
<td>Not equal to</td>
<td>13 -ne 37</td>
</tr>
<tr>
<td>-not</td>
<td>Logical Not</td>
<td>-not (2 -eq 1)</td>
</tr>
<tr>
<td>!</td>
<td>Logical Not</td>
<td>!(2 -eq 1)</td>
</tr>
<tr>
<td>-and</td>
<td>Logical And</td>
<td>(2+2 -eq 4) -and (1+1 -eq 2)</td>
</tr>
<tr>
<td>-or</td>
<td>Logical Or</td>
<td>(2+2 -eq 0) -or (1+1 -eq 2)</td>
</tr>
<tr>
<td>-like</td>
<td>Match using the wildcard character</td>
<td>&quot;PowerShell&quot; -like &quot;*shell&quot;</td>
</tr>
<tr>
<td>-notlike</td>
<td>Opposite of -Like</td>
<td>&quot;PowerShell&quot; -notlike &quot;*bash&quot;</td>
</tr>
<tr>
<td>-match</td>
<td>Matches using a Regular Expression and populates the $matches variable</td>
<td>&quot;Sunday&quot; -match &quot;[A-Z]*&quot;</td>
</tr>
<tr>
<td>-notmatch</td>
<td>Does not match on Regular Expression, populates $matches</td>
<td>&quot;Sunday&quot; -notmatch &quot;[0-9]*&quot;</td>
</tr>
<tr>
<td>-contains</td>
<td>Containment operator</td>
<td>$days -contains &quot;sun&quot;</td>
</tr>
<tr>
<td>-notcontains</td>
<td>Opposite of contains</td>
<td>$days -notcontains &quot;blah&quot;</td>
</tr>
<tr>
<td>-replace</td>
<td>Replaces (does not return a Boolean)</td>
<td>&quot;Monday&quot; -replace &quot;Fri&quot; Output: Friday</td>
</tr>
</tbody>
</table>
1) What is the proper syntax to check if "$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"
   - $a -endswith "find me"
1) What is the proper syntax to check if "$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".
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:

```powershell
if (condition) {do 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.

```powershell
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.

```powershell
($a -eq 1) {"zero"} else {"non-zero"}
```

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

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

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

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

```powershell
Get-ChildItem -Recurse | ? { $_.Length -ge 20000000 } | del
```

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.

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

Directory: C:\

<table>
<thead>
<tr>
<th>Mode</th>
<th>LastWriteTime</th>
<th>Length</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a---</td>
<td>10/10/2009 10:10 PM</td>
<td>31415926</td>
<td>mybig.zip</td>
</tr>
</tbody>
</table>

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

```powershell
ls -Recurse | ? Length -ge 20000000 -and Extension -eq ".zip"
```
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.
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.

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


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.
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" }`
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
Is this section, you will be introduced to PowerShell's flow control and output cmdlets.