POWERSHELL FUNCTIONS

TLO KNOWLEDGE AND SKILLS

Conditions:

- Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of PowerShell functions.

Knowledge:

- Identify PowerShell functions.
- Understand how strings and expressions work in PowerShell scripts.

Skills:

- Working knowledge of how to manipulate strings in Powershell scripts.
- Understanding of how to use Powershell functions and regular expressions.

OBJECTIVES

- Define PowerShell Functions
- Identify String Manipulation
- Describe Regular Expressions
- Demonstrate Powershell Scripts

POWERSHELL FUNCTIONS

A function in PowerShell is a grouping of code that has an optional input and output.

It’s a way of collecting up a lot of code in order to perform one or more times by just pointing to it instead of duplicating that code repeatedly.

Think of it as a method in objects. Anything within your script can call it.

POWERSHELL FUNCTIONS

A function is a list of PowerShell statements that has a name that you assign. When you run a function, you type the function name. The statements in the list run as if you had typed them at the command prompt.

Functions can be as simple as:

```
Function Write-Something {

            Param($item)

            Write-Host “You passed the parameter $item into the function”

}
```

POWERSHELL FUNCTIONS

```
Write-Something "Hello!"
```

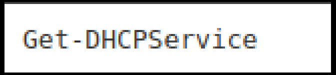
```
function Get-MultiplicativeResult {

Param ([int]$a,[int]$b)

$c = $a * $b

Write-Output $c

}
```

```
Get-MultiplicativeResult 5 10
```

```
function Get-DHCPService { Get-Service -Name DHCP }
```

POWERSHELL FUNCTIONS

This is a function that we can call like a cmdlet.



```
Get-DHCPService
```

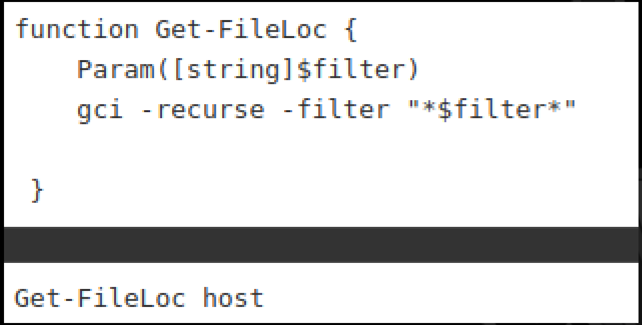
POWERSHELL FUNCTIONS

We’ve just created an easier way to get the DHCP service.

Functions are useful if you are reusing code alot.

You can put all of your reused code into a single function statement and call the code when you reference the name of the function.

The example here is the same idea but a little more complicated.



```
function Get-FileLoc {
    Param([string]$filter)
    gci -recurse -filter "*$filter*"

 }
```

```
Get-FileLoc host
```

APPLY YOUR LEARNING

Open up powershell and let’s build this function:

```
function Stop-PowerShell {
$procs=(Get-WmiObject Win32_Process | Where {$_.Name -eq 'powershell.exe'} `
        | ?{$_.ProcessId -ne $pid} | Select-Object -ExpandProperty ProcessId)

Do {
  ForEach ($i in $procs) {
        Get-Process powershell | ?{$_.Id -ne $pid} | Stop-Process | Out-Null
        $procs=(Get-WmiObject Win32_Process | Where {$_.Name -eq 'powershell.exe'} `
          | ?{$_.ProcessId -ne $pid} | Select-Object -ExpandProperty ProcessId)
  }
} While ($procs)
}
```

STRING MANIPULATION

It is important to understand that strings in PowerShell are always objects, whether you are dealing with literal strings or variables.

Consequently, the methods of string objects provide most of the functions you need. As usual, you can display them with the Get-Member cmdlet.

```
"this is a string" | get-member

$string = "a string"
$string | get-member
```

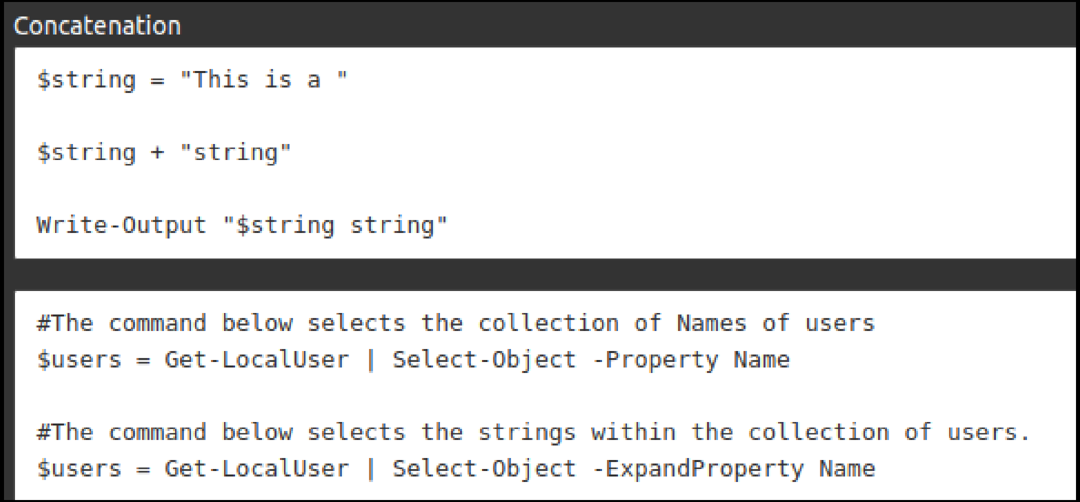
STRING MANIPULATION

Like we learned in objects, a variable is just a name that represents an object. You can see that when we can get a string literal and a string variable we get the same output because we are dealing with a string object in both scenarios.

We can use the built in string methods to manipulate and interact with string objects.

SELECT-OBJECCT

The difference in the command below is that Select-Object -Property creates a collection of the different user object names. We would then have to further convert the results of this command to get our output to a string.



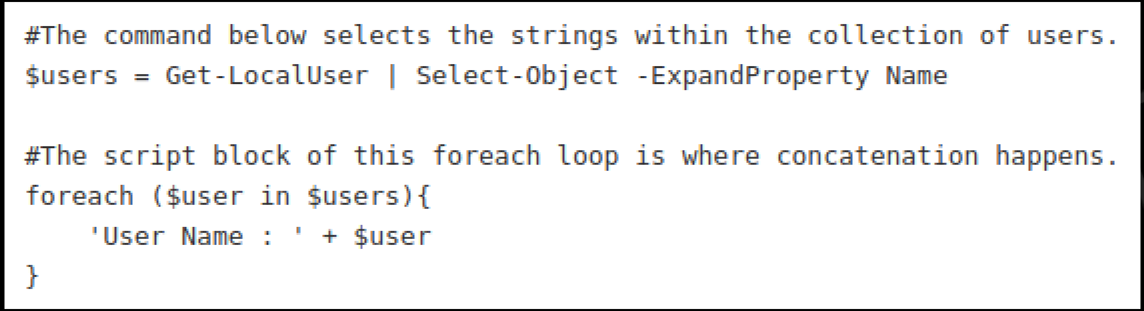
Concatenation

```
$string = "This is a "

$string + "string"

Write-Output "$string string"
```

```
#The command below selects the collection of Names of users
$users = Get-LocalUser | Select-Object -Property Name

#The command below selects the strings within the collection of users.
$users = Get-LocalUser | Select-Object -ExpandProperty Name
```

EXPAND PROPERTY

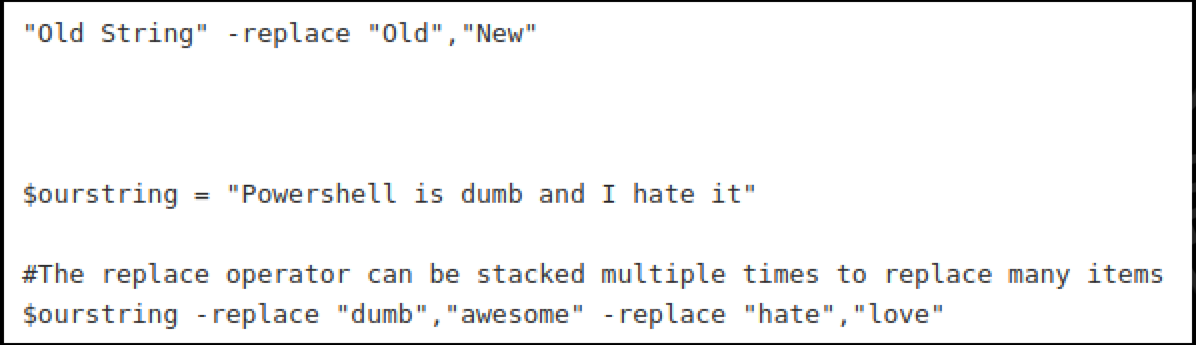
The Select-Object -ExpandProperty, takes the property argument passed, in this case Name and converts it to a string that is ready for string manipulation.



```
#The command below selects the strings within the collection of users.
$users = Get-LocalUser | Select-Object -ExpandProperty Name

#The script block of this foreach loop is where concatenation happens.
foreach ($user in $users){
    'User Name : ' + $user
}
```

REPLACE

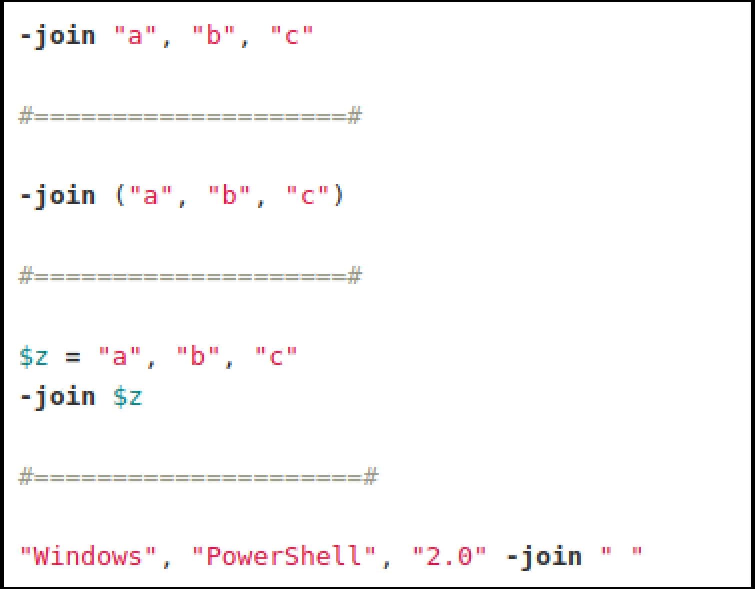
The replace operator replaces instances of string occurrences (old values) with the specified string (new value).



```
"Old String" -replace "Old","New"



$ourstring = "Powershell is dumb and I hate it"

#The replace operator can be stacked multiple times to replace many items
$ourstring -replace "dumb","awesome" -replace "hate","love"
```

JOIN

The join operator concatenates a set of strings into a single string. The strings are appended to the resulting string in the order that they appear in the command.



```
-join "a", "b", "c"

#====================#

-join ("a", "b", "c")

#====================#

$z = "a", "b", "c"
-join $z

#=====================#

"Windows", "PowerShell", "2.0" -join " "
```

SPLIT

The split() method is on every string object and is capable of splitting a string into an array based on a non-regex character.

```
"This.is.an.example.of.string.to.split.." -split "\."
```

REGULAR EXPRESSIONS

Regular Expressions (REGEX) are used synonymously across many programming language variants, and serve as a set of tools for filtering strings in large data outputs.

Their value cannot be overstated, as they have helped operators and analysts accomplished the most mundane of tasks to the most extreme and complex.

Think of regex as [CTRL]+F on adrenaline.

```
#Declare our string to apply our regex filter to.

$string = "HelloMyNameIS Nick123_55-4-11111-asdf_//123-45-6789-hELLOmYnAME Nick"
```

REGULAR EXPRESSIONS

Syntax	Example	Purpose
`+`	`[a-z]+`	Match previous character or character set at least once
`?`	`[c-e]?`	Match at most once; previous character or character set may or may not exist
`*`	`App* and` `Bana*`	Any character; match any number of times
`{min,max}`	`[0-9]{1,3}`	Match at least `min` times, and at most `max` times
`{n}`	`[0-9]{3}`	Match exactly `n` times

MATCH AND PATTERN

In Regex the \d character class is used to designate numbers, just as the [0-9] character class does.

The - is escaped as it’s usually used in regex to denote a character range.

The {2,3} repetition quantifier denotes that whatever is previously stated, in this case the character class \d, which again is any number 0-9, must contain at least 2 numbers, and may be repeated up to three instances in the string to be matched.

MATCH AND PATTERN

The repetition quantifier, {1,} is often used to denote one or more, when the quantity to match of a designated character class is not known.

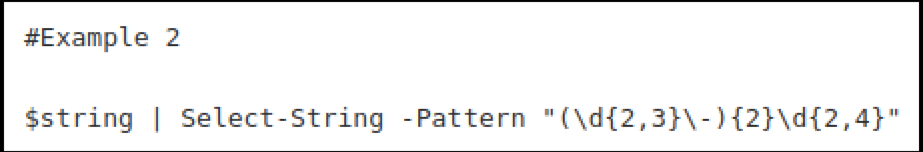
The -Match shell built-in parameter (Example 1) can be used to match a regex pattern of any provided string.

```
#Example 1

$string -Match "(\d{2,3}\-){2}\d{2,4}"
```

MATCH AND PATTERN

To match a string using the contents of a file(s), the CMDLET Select-String (Example 2) is used with the parameter -Pattern, which unlike the -Match parameter, does not just return a boolean indication of match.

Instead returns the entire line of the actual string that was matched.



```
#Example 2

$string | Select-String -Pattern "(\d{2,3}\-){2}\d{2,4}"
```

MATCH AND PATTERN

Passing the complete line of output with matching string to the ForEach loop, discussed in a previous lesson, used in conjunction with methods .Matches(), and .Value().

Example 3 extracts only the string that the -Pattern Select-String parameter matches.

```
#Example 3

#Foreach loops through all the matches and prints the collection to the screen.

$string | Select-String -Pattern "(\d{2,3}\-){2}\d{2,4}" | ForEach {$_.Matches.Value}
```

REGEX

This command goes and grabs a pentesting guide on the internet and stores the content of that page to a file called ptes.txt:

```
Invoke-WebRequest -Uri "http://www.pentest-standard.org/index.php/PTES_Technical_Guidelines" -UseBasicParsing | Select-Object -ExpandProperty Content | Out-File ptes.txt
```

We can see that this file is *HUGE*. It would take us hours to skim through it without regex. It's over a Gigabyte:

```
(Get-Content .\ptes.txt).Length
```

```
Get-Content .\ptes.txt | Select-String -Pattern "nmap" | % {$_.Matches.Value} | Select -First 2
```

```
Get-Content .\ptes.txt | Select-String -Pattern "nmap.+" | % {$_.Matches.Value} | Select -First 30 | Select -Last 10
```

```
Get-Content .\ptes.txt | Select-String -Pattern "nmap\s\-.+" | % {$_.Matches.Value} | Sort -Unique
```

We are able to grab contextual information by filtering on the string "nmap" using regex without having to search this file manually. Reinforce this concept to the students and remind them how powerful regex is.

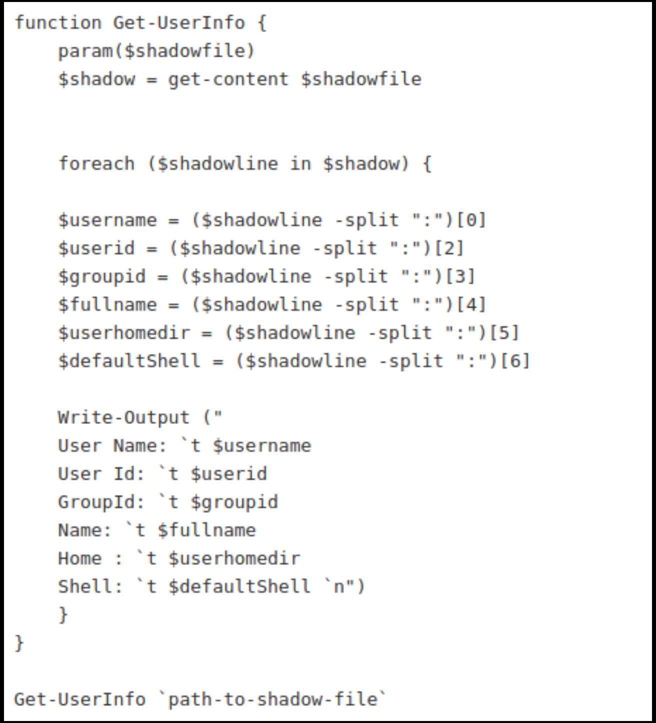
ANCHORS

Two regex characters we can use separately or together are the ‘^’ and ‘$’. Both are referred to as anchor characters as they search for patterns that are anchored to one end or the other.

^ - searches for pattern at the beginning of a line

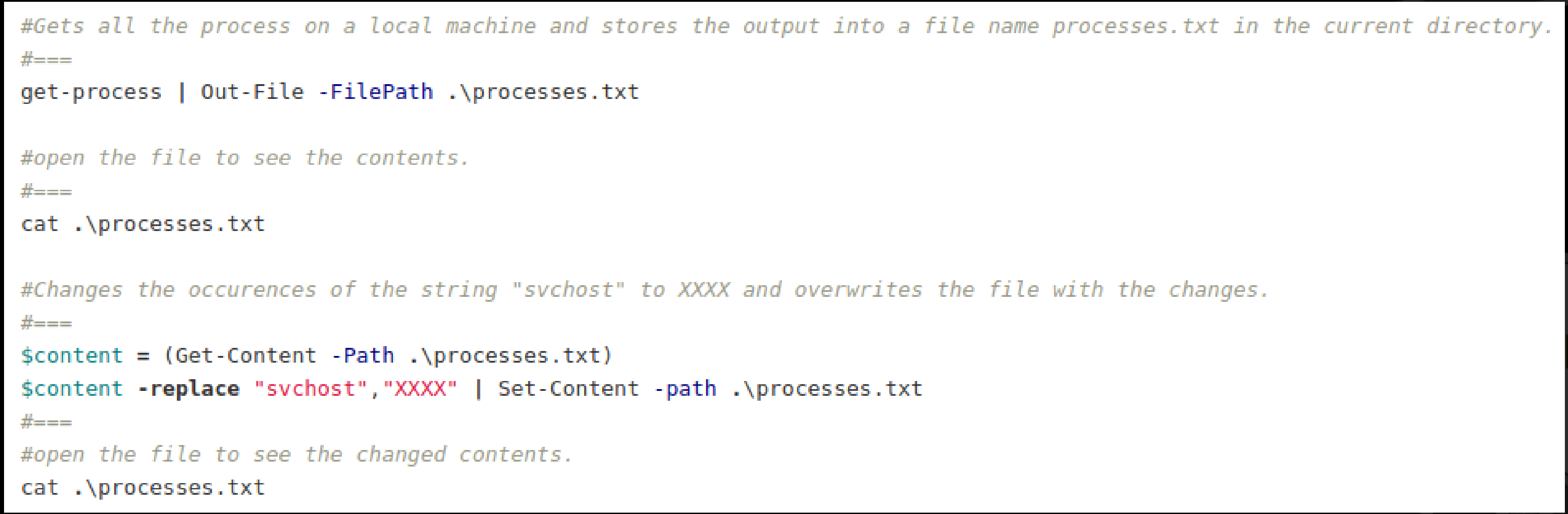
$ - searches for a pattern at the end of a line

POWERSHELL SCRIPTS



```
function Get-UserInfo {
    param($shadowfile)
    $shadow = get-content $shadowfile


    foreach ($shadowline in $shadow) {

    $username = ($shadowline -split ":")[0]
    $userid = ($shadowline -split ":")[2]
    $groupid = ($shadowline -split ":")[3]
    $fullname = ($shadowline -split ":")[4]
    $userhomedir = ($shadowline -split ":")[5]
    $defaultShell = ($shadowline -split ":")[6]

    Write-Output ("
    User Name: `t $username
    User Id: `t $userid
    GroupId: `t $groupid
    Name: `t $fullname
    Home : `t $userhomedir
    Shell: `t $defaultShell `n")
    }
}

Get-UserInfo `path-to-shadow-file`
```

SEARCH AND REPLACE STRINGS IN A FILE



```
#Gets all the process on a local machine and stores the output into a file name processes.txt in the current directory.
#===
get-process | Out-File -FilePath .\processes.txt

#open the file to see the contents.
#===
cat .\processes.txt

#Changes the occurences of the string "svchost" to XXXX and overwrites the file with the changes.
#===
$content = (Get-Content -Path .\processes.txt)
$content -replace "svchost","XXXX" | Set-Content -path .\processes.txt
#===
#open the file to see the changed contents.
cat .\processes.txt
```