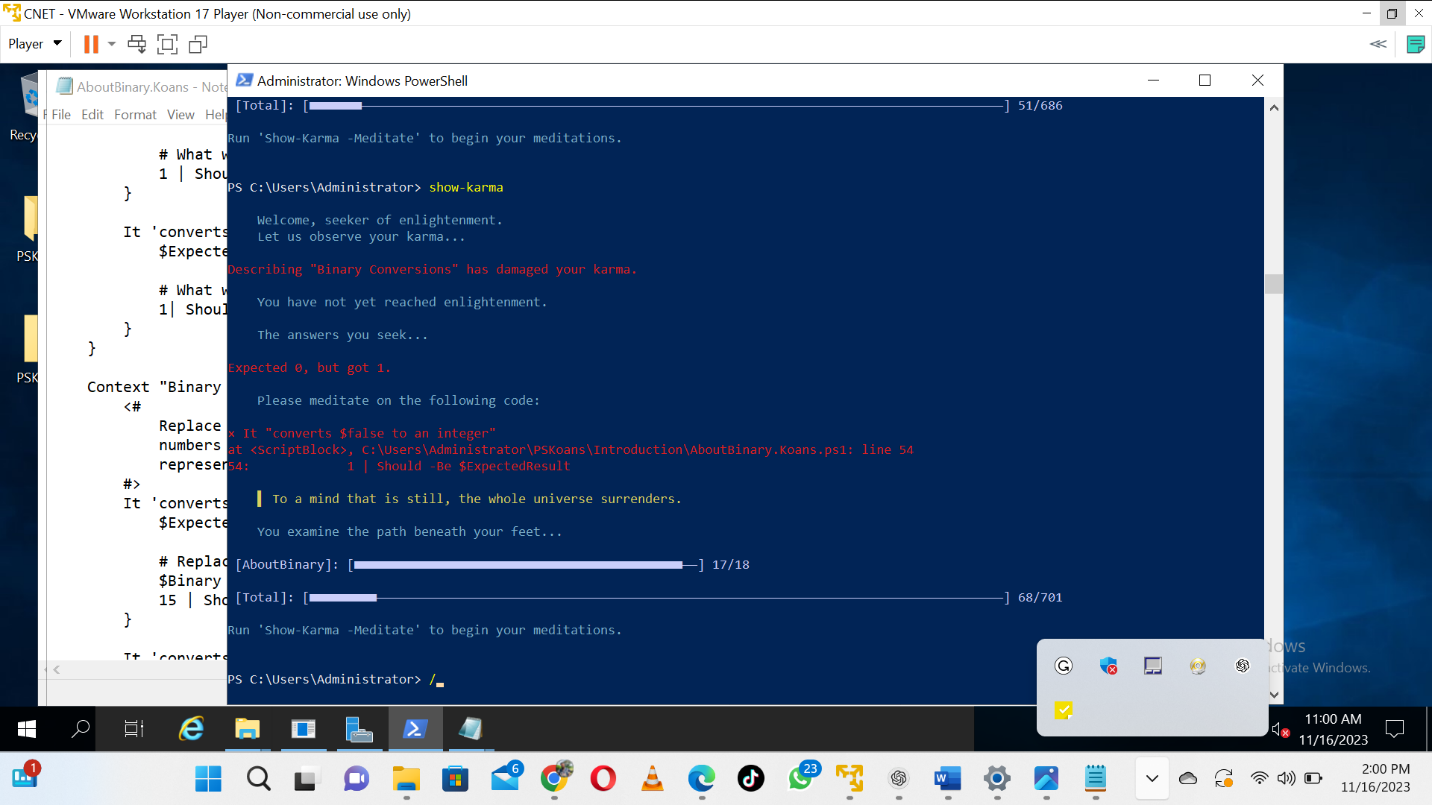
**LAB 6 ASSIGNMENT**



**ABOUT VERBS**

using module PSKoans

[Koan(Position = 105)]

param()

<#

Cmdlet Verbs

The fundamental building block of PowerShell are cmdlets and functions, both

of which are named according to a Verb-Noun syntax by convention.

You can see the list of PowerShell's recommended / approved verbs by calling

Get-Verb in your PowerShell console, or going to the following docs page:

https://docs.microsoft.com/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands

Each command should use a verb appropriate to the action it's taking, and a

noun that succinctly and clearly describes what it's acting upon.

In this topic, we'll cover a few of the most common verbs which you'll see

most frequently.

#>

Describe "Basic Verbs" {

BeforeAll {

# We'll be using this path later on.

$TestDrive = "C:\Users\Administrator\Documents\TestDrive"

$FilePath = join-path -path $TestDrive -childpath "path.txt"

if (Test-Path $FilePath) {

Remove-Item -Path $FilePath

}

}

Context "Get" {

<#

Cmdlets with the Get verb are used for retrieving data.

So for example "Get-Process" will retrieve all the processes running

on your current machine.

#>

It 'is for commands that retrieve data' {

<#

Using the Get-Command cmdlet, which retrieves all available

commands, find 5 commands with the Get verb.

Fill in each blank with the name of a different Get-\* command.

#>

$Answers = @(

"Get-Process"

"Get-Command"

"Get-Service"

"Get-Help"

"Get-Content"

)

$Answers | Should -BeIn (Get-Command -Verb Get).Name

$Answers | Get-Unique | Should -HaveCount 5 -Because "five unique cmdlets are required"

}

}

Context "New" {

<#

Cmdlets with the New verb are used to create data.

So for example 'New-Guid' will create a new GUID, or 'New-TimeSpan'

will create a new [timespan] object with the specified parameters.

#>

It 'is for commands that create data' {

<#

Using Get-Command, find 5 commands with the New verb.

Replace each \_\_\_\_ with the name of a New-\* command.

#>

$Answers = @(

"New-Item"

"New-Object"

"New-TimeSpan"

"New-Variable"

"New-Guid"

)

$Answers | Should -BeIn (Get-Command -Verb New).Name

$Answers | Get-Unique | Should -HaveCount 5 -Because "five unique cmdlets are required"

}

It 'can create a New-Item' {

<#

Let's try creating a file!

Use New-Item to create a new text file, in the location

specified with the $FilePath variable, which is defined above.

The necessary parameters are already filled in for you here. In

this case, the file we're creating is simply empty. You could

also specify a -Value parameter with some text to put in the

newly-created file.

Take some time to experiment with New-Item in your console if

you'd like to see what it can do! Start with:

Get-Command New-Item -Syntax

Get-Help New-Item -Examples

#>

$File = New-Item -Path $FilePath -ItemType File

# All "file" objects are of this type.

$File | Should -BeOfType [System.IO.FileInfo]

# An empty file has a "length" of zero.

$File.Length | Should -Be 0

}

}

Context "Add" {

<#

Cmdlets with the Add verb append data to an existing object or data

source.

Essentially, if the target doesn't exist then a cmdlet with the Add

verb will typically create it. If it does exist, the cmdlet will

add data to it, if data can be added without overwriting the

original data.

A common example one might see working with Office 365 is calendar

permissions. If you want to grant permissions to somebody who

doesn't have any, you use "Add-MailboxFolderPermission".

#>

It 'is for commands that append data' {

<#

Using Get-Command, find 5 commands with the Add verb.

Replace each \_\_\_\_ with the name of an Add-\* command.

#>

$Answers = "Add-Content", "Add-Type", "Add-Member", "Add-History", "Add-PSSnapin"

$Answers | Should -BeIn (Get-Command -Verb Add).Name

$Answers | Get-Unique | Should -HaveCount 5 -Because "five unique cmdlets are required"

}

It 'can Add-Content to a file' {

<#

Try adding this content to the file we created above using

Add-Content.

#>

"Mountains are merely mountains." | Add-Content -Path $FilePath

<#

Let's see what happens if we add a whole bunch of things! Fill

in these blanks with whatever you like.

#>

'Line 1' | Add-Content -Path $FilePath

'Line 2' | Add-Content -Path $FilePath

'Line 3' | Add-Content -Path $FilePath

'Line 4' | Add-Content -Path $FilePath

# Let's check the contents of the file.

$FileData = Get-Content -Path $FilePath

# How many lines did we end up with?

$FileData.Count | Should -Be $FileData.Count

<#

We can see that several lines of content were added to the file.

Add-\* cmdlets can only append data, they can't overwrite. With

this information, you should be able to determine what the

expected content of the file is at this point.

#>

$FileData = Get-Content -Path $FilePath

$ExpectedContent = @(

'Mountains are merely mountains.'

'Line 1'

'Line 2'

'Line 3'

'Line 4'

'The road onwards, the road back; which is the shorter?'

)

$FileData | Should -BeExactly $FileData

}

}

Context "Set" {

<#

Cmdlets with the Set verb will overwrite information that already

exists.

Some Set-\* cmdlets require the instance to already be present for

you to change it; you'll need to use a New-\* cmdlet first to create

an instance before you can overwrite information within it.

A common example one may see working with Office 365 is with

calendar permissions. If a user already has some permissions

configured, you can use "Set-MailboxFolderPermission" to change the

user's permissions.

#>

It 'is for commands that overwrite data' {

<#

Using Get-Command, find 5 commands with the Set verb.

Replace each \_\_\_\_ with the name of a Set-\* command.

#>

$Answers = "Set-Item", "Set-Location", "Set-Variable", "Set-Content", "Set-ExecutionPolicy"

$Answers | Should -BeIn (Get-Command -Verb Set).Name

$Answers | Get-Unique | Should -HaveCount 5 -Because "five unique cmdlets are required"

}

It 'can Set-Content for a file' {

<#

Let's try using Set-Content on our text file from before. But

first, let's check that it still has the contents we added. We

should still have 5 lines in it from before.

If you added extra lines to the file in the Add-Content koan

above, make sure to update the expected line count here!

#>

$LineCount = 5

Get-Content -Path $FilePath | Should -HaveCount $LineCount

# Now let's try setting the contents.

"Wherever you are, it's the place you need to be." | Set-Content -Path $FilePath

# So what should be in the file now?

$FileContent = Get-Content -Path $FilePath

"$FileContent" | Should -BeExactly $FileContent

# What happens if we set the contents again?

$FilePath | Set-Content -Path $FilePath

$FileContent | Should -BeExactly $FileContent

$NewContent = "Rest and be kind, you don't have to prove anything."

$NewContent | Set-Content -Path $FilePath

$UpdatedFileContent = Get-Content -Path $FilePath

$NewContent | Should -BeExactly $UpdatedFileContent

Get-Content -Path $FilePath | Should -BeExactly "Rest and be kind, you don't have to prove anything."

<#

You'll see that there's only one line of text in the file. This

is because the Set-Content command will completely overwrite

whatever is in the file already.

#>

}

}

Context "Remove" {

<#

Cmdlets with the Remove verb will delete data from an object or data

source.

Once again with calendar permissions in Office 365, you can use a

Remove cmdlet to completely remove a user's permissions to a

calendar.

#>

It "is for commands that delete data" {

<#

Using Get-Command, find 5 commands with the Remove verb.

Replace each \_\_\_\_ with the name of a Remove-\* command.

#>

$Answers = @(

'Remove-WindowsFeature', 'Remove-AppPackage', 'Remove-EtwTraceSession', 'Remove-AppProvisionedPackage', 'Remove-AutologgerConfig'

)

$Answers | Should -BeIn (Get-Command -Verb Remove).Name

$Answers | Get-Unique | Should -HaveCount 5 -Because "five unique cmdlets are required"

}

It 'can Remove-Item to delete a file' {

<#

We can use Remove-Item to delete the text file we've been

working with. Before we do, let's just double check the file

still exists. 'Leaf' here refers to a file; 'Container'

would be the corresponding type for a folder.

#>

Test-Path $FilePath -PathType Leaf | Should -BeTrue

# Pester has its own way of checking that files exist.

$FilePath | Should -Exist

# Use Remove-item to delete the file completely.

Remove-Item -Path $FilePath

# Let's check it was removed properly. Test-Path $FilePath -PathType Leaf | Should -BeTrue

Test-Path $FilePath | Should -BeFalse

<#

If we try to remove a file that doesn't exist, we should get

an error. What does that error look like?

#>

$Message = "cannot find path"

{ Remove-Item -Path $FilePath -ErrorAction Stop } | Should -Throw -ExpectedMessage $Message

}

}

}

**GET MEMBER**

using module PSKoans

[Koan(Position = 106)]

param()

<#

Get-Member

Following on from our last Koan about cmdlet verbs, lets cover the nouns.

As previously stated, the noun portion indicates the target of a given command.

If you use the "Get-Member" cmdlet on any object you'll see its type name, or data type.

For example:

PS C:\Users\Administrator\Documents\GitHub\Dev\PSKoans> Get-Process | Get-Member

TypeName: System.Diagnostics.Process

Name MemberType Definition

---- ---------- ----------

Handles AliasProperty Handles = Handlecount

Name AliasProperty Name = ProcessName

NPM AliasProperty NPM = NonpagedSystemMemorySize64

PM AliasProperty PM = PagedMemorySize64

SI AliasProperty SI = SessionId

VM AliasProperty VM = VirtualMemorySize64

WS AliasProperty WS = WorkingSet64

You'll also see properties and methods. Together with the TypeName, this tells us everything

we need to know in order to access the object's data and any actions it can perform.

Accessing a property or method is done via dot notation, which boils down to:

$Object.Property

$Object.Method()

We'll cover why we have to use parentheses with method calls later.

#>

Describe "Get Member" {

Context 'Exploring Object Properties' {

<#

Let's look at some object properties!

Properties define the state an object is in.

Using any three cmdlets you like (make sure you use three different

cmdlets!), use Get-Member in your console to peek at the properties

on the object.

Let's see an example; by sending the output from Get-Process into

Get-Member we can inspect the objects Get-Process outputs. You can

run the below command in your console:

Get-Process | Get-Member -MemberType Property

From the output of the above command, we will see that one of the

properties is named "Threads", so using that cmdlet name and that

property name will satisfy one of the tests below.

The others are up to you! Get-\* cmdlets will be most helpful here;

if you're not sure which to try, you can use the following command

to list Get-\* cmdlets you can try out:

Get-Command -Verb Get

#>

BeforeAll {

$Cmdlets = [System.Collections.Generic.HashSet[string]]::new()

$PropertyString = "property '{0}' should be present in output from {1}"

$UniqueString = 'unique cmdlets should be used for each test'

}

It 'lists one of the properties of the first unique command' {

$CmdletName = 'Get-Process'

$PropertyName = 'SessionId'

$Parameters = @{

<#

This is only needed if a cmdlet you provide has mandatory

parameters. Enter the parameter name and the value just

outside this comment block, but inside the @{ } like so:

ParameterName = "Value"

#>

}

$Reason = $PropertyString -f $PropertyName, $CmdletName

& (Get-Command -Name $CmdletName) @Parameters |

Get-Member -MemberType Property -Name $PropertyName |

Should -Not -BeNullOrEmpty -Because $Reason

$Cmdlets.Add($CmdletName) | Should -BeTrue -Because $UniqueString

}

It 'lists one of the properties of the second unique command' {

$CmdletName = 'Get-ADForest'

$PropertyName = 'RootDomain'

$Parameters = @{

<#

This is only needed if a cmdlet you provide has mandatory

parameters. Enter the parameter name and the value just

outside this comment block, but inside the @{ } like so:

ParameterName = "Value"

#>

}

$Reason = $PropertyString -f $PropertyName, $CmdletName

& (Get-Command -Name $CmdletName) @Parameters |

Get-Member -MemberType Property -Name $PropertyName |

Should -Not -BeNullOrEmpty -Because $Reason

$Cmdlets.Add($CmdletName) | Should -BeTrue -Because $UniqueString

}

It 'lists one of the properties of the third unique command' {

$CmdletName = 'Get-AppPackage'

$PropertyName = 'InstallLocation'

$Parameters = @{

<#

This is only needed if a cmdlet you provide has mandatory

parameters. Enter the parameter name and the value just

outside this comment block, but inside the @{ } like so:

ParameterName = "Value"

#>

}

$Reason = $PropertyString -f $PropertyName, $CmdletName

& (Get-Command -Name $CmdletName) @Parameters |

Get-Member -MemberType Property -Name $PropertyName |

Should -Not -BeNullOrEmpty -Because $Reason

$Cmdlets.Add($CmdletName) | Should -BeTrue -Because $UniqueString

}

}

Context 'Exploring Object Methods' {

<#

Now that we know how the state of an object is represented by the values of its properties,

we can now take a look at methods. In contrast to properties, methods need parentheses.

Inside the parentheses you can give additional parameters (oftentimes called arguments)

to the method, similary to parameters for a function. In fact, this is how most

programming languages treat functions and methods. If for example you want to

know if a string ends with a certain character, it is necessary that the method knows

which character you want to compare the string against.

Similar to above, you can inspect the methods available from an

object that a cmdlet outputs, by changing the -MemberType value

you provide to Get-Member:

Get-Process | Get-Member -MemberType Method

If you don't provide a -MemberType option and value, it will simply

list all the members, regardless of the kind of members they are.

You can reuse the same set of cmdlets from above here if you wish,

but you will need to check to see if there are methods available on

the objects they output!

#>

BeforeAll {

$Cmdlets = [System.Collections.Generic.HashSet[string]]::new()

$MethodString = "property '{0}' should be present in output from {1}"

$UniqueString = 'unique cmdlets should be used for each test'

}

It 'lists one of the methods of the first unique command' {

$CmdletName = 'Get-Process'

$MethodName = 'Dispose'

$Reason = $MethodString -f $MethodName, $CmdletName

& (Get-Command -Name $CmdletName) |

Get-Member -MemberType Method -Name $MethodName |

Should -Not -BeNullOrEmpty -Because $Reason

$Cmdlets.Add($CmdletName) | Should -BeTrue -Because $UniqueString

}

It 'lists one of the methods of the second unique command' {

$CmdletName = 'Get-AppxVolume'

$MethodName = 'GetHashCode'

$Reason = $MethodString -f $MethodName, $CmdletName

& (Get-Command -Name $CmdletName) |

Get-Member -MemberType Method -Name $MethodName |

Should -Not -BeNullOrEmpty -Because $Reason

$Cmdlets.Add($CmdletName) | Should -BeTrue -Because $UniqueString

}

It 'lists one of the methods of the third unique command' {

$CmdletName = 'Get-BpaModel'

$MethodName = 'ToString'

$Reason = $MethodString -f $MethodName, $CmdletName

& (Get-Command -Name $CmdletName) |

Get-Member -MemberType Method -Name $MethodName |

Should -Not -BeNullOrEmpty -Because $Reason

$Cmdlets.Add($CmdletName) | Should -BeTrue -Because $UniqueString

}

}

}

**BINARY**

using module PSKoans

[Koan(Position = 107)]

param()

<#

Binary

Binary is a base 2 number system, and only uses two numerals: 0 and 1.

Binary is the basis of all computing and computational storage. All data is

stored both on hard drives and in memory in a series of states that can be

interpreted as a series of 1s and 0s, which directly represents the

underlying on or off hardware states.

It's useful to know binary when working with PowerShell to understand how

numbers are commonly represented, which will be covered in more detail in a

future topic.

It's also extremely useful knowledge if you plan on delving deeper into

computing; the knowledge is applicable to networking, programming,

data science, databases, and essentially every other area of computing.

For example, the below binary represents a "byte" of data, which contains

eight "bits," each of which will be either 1 or 0, as mentioned earlier.

The table below demonstrates some of the structure of how this works. It's

very much comparable to the commonly-used base-10 number system, but since

binary is base 2 instead, each column is a power of 2, not of a power of 10.

| 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |

| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

+-----+-----+-----+-----+-----+-----+-----+-----+

0 0 0 0 0 0 0 1

Written in a simpler binary form: 00000001 (Actual value: 1)

That's simple enough. But since there are only two numerals in a binary

number system, you have to use extra columns to represent numbers larger

than 1.

As such, 00000011 has a value of 3.

As both 1 and 2 columns are

one, and 1 + 2 is equal to 3.

#>

Describe 'Binary Conversions' {

Context 'Boolean Conversions' {

It 'converts $false to an integer' {

$ExpectedResult = $false -as [int]

# What would $false be if converted to a number?

1 | Should -Be $ExpectedResult

}

It 'converts $true to an integer' {

$ExpectedResult = $true -as [int]

# What would $true be if converted to a number?

1| Should -Be $ExpectedResult

}

}

Context "Binary to Integer Conversion" {

<#

Replace the blanks below with the decimal value of the binary

numbers in each case. For example, the binary sequence "10" is

represented by the number 2 in the standard decimal system.

#>

It 'converts 1111 to an integer' {

$ExpectedValue = [Convert]::ToInt32(1111, 2)

# Replace the \_\_ with the decimal value of 1111

$Binary = "1111"

15 | Should -Be $ExpectedValue

}

It 'converts 1000 to an integer' {

$ExpectedValue = [Convert]::ToInt32(1000, 2)

# Replace \_\_ with the decimal value of 1000

$Binary = "1000"

8 | Should -Be $ExpectedValue

}

It 'converts 0010 to an integer' {

$ExpectedValue = [Convert]::ToInt32(0010, 2)

# Replace \_\_ with the decimal value of 0010

$Binary = "0010"

2 | Should -Be $ExpectedValue

}

It 'converts 1001 to an integer' {

$ExpectedValue = [Convert]::ToInt32(1001, 2)

# Replace \_\_ with the decimal value of 1001

$Binary = "1001"

9 | Should -Be $ExpectedValue

}

It 'converts 11111111 to an integer' {

$ExpectedValue = [Convert]::ToInt32(11111111, 2)

# Replace \_\_ with the decimal value of 11111111

$Binary = "11111111"

255 | Should -Be $ExpectedValue

}

It 'converts 10101010 to an integer' {

$ExpectedValue = [Convert]::ToInt32(10101010, 2)

# Replace \_\_ with the decimal value of 10101010

$Binary = "10101010"

170 | Should -Be $ExpectedValue

}

It 'converts 11001100 to an integer' {

$ExpectedValue = [Convert]::ToInt32(11001100, 2)

# Replace \_\_ with the decimal value of 11001100

$Binary = "11001100"

204 | Should -Be $ExpectedValue

}

It 'converts 11110001 to an integer' {

$ExpectedValue = [Convert]::ToInt32(11110001, 2)

# Replace \_\_ with the decimal value of 11110001

$Binary = "111g10001"

241 | Should -Be $ExpectedValue

}

}

Context "Integer to Binary Conversion" {

<#

Convert the following integers into their binary representation.

For example, 2 is represented in binary with the digits "10".

#>

It 'converts the integer 7 to binary' {

# Replace \_\_\_\_ with the binary value of 7

$Value = 7

$Binary = [Convert]::ToString(7, 2)

'111' | Should -Be $Binary

}

It 'converts the integer 12 to binary' {

# Replace \_\_ with the binary value of 12

$Value = 12

$Binary = [Convert]::ToString(12, 2)

'12' | Should -Be ([Convert]::ToInt32($Binary, 2))

}

It 'converts the integer 2 to binary' {

# Replace \_\_ with the binary value of 2

$Value = 2

$Binary = [Convert]::ToString(2, 2)

'2' | Should -Be ([Convert]::ToInt32($Binary, 2))

}

It 'converts the integer 14 to binary' {

# Replace \_\_ with the binary value of 14

$Value = 14

$Binary = [Convert]::ToString(14, 2)

'14' | Should -Be ([Convert]::ToInt32($Binary, 2))

}

It 'converts the integer 103 to binary' {

# Replace \_\_ with the binary value of 103

$Value = 103

$Binary = [Convert]::ToString(103, 2)

'103' | Should -Be ([Convert]::ToInt32($Binary, 2))

}

It 'converts the integer 250 to binary' {

# Replace \_\_ with the binary value of 250

$Value = 250

$Binary = [Convert]::ToString(250, 2)

'250' | Should -Be ([Convert]::ToInt32($Binary, 2))

}

It 'converts the integer 74 to binary' {

# Replace \_\_ with the binary value of 74

$Value = 74

$Binary = [Convert]::ToString(74, 2)

'74' | Should -Be ([Convert]::ToInt32($Binary, 2))

}

It 'converts the integer 32 to binary' {

# Replace \_\_ with the binary value of 32

$Value = 32

$Binary = [Convert]::ToString(32, 2)

'32' | Should -Be ([Convert]::ToInt32($Binary, 2))

}

}

}