Assignment #5

You will need to show the result of**the First 30 Answers. [beside the initial questioner]**

**You need to start from ABOUTBOOLEANS**

**NOTE: I want to actually see the result itself not just the game bar.**

**In the world document please paste all the 30 answers as text**

Describe "Booleans" {

    # Using only booleans, either $true or $false, fill in the blanks below.

    It '( 1 -gt 2 ) is either true or false' {  
        $\_\_\_\_ | Should -Be ( 1 -gt 2 ) -Because '1 is not greater than 2'  
    }

    It '( 1 -lt 2 ) is either true or false' {  
        $\_\_\_\_ | Should -Be ( 1 -lt 2 ) -Because '1 is less than 2'  
    }

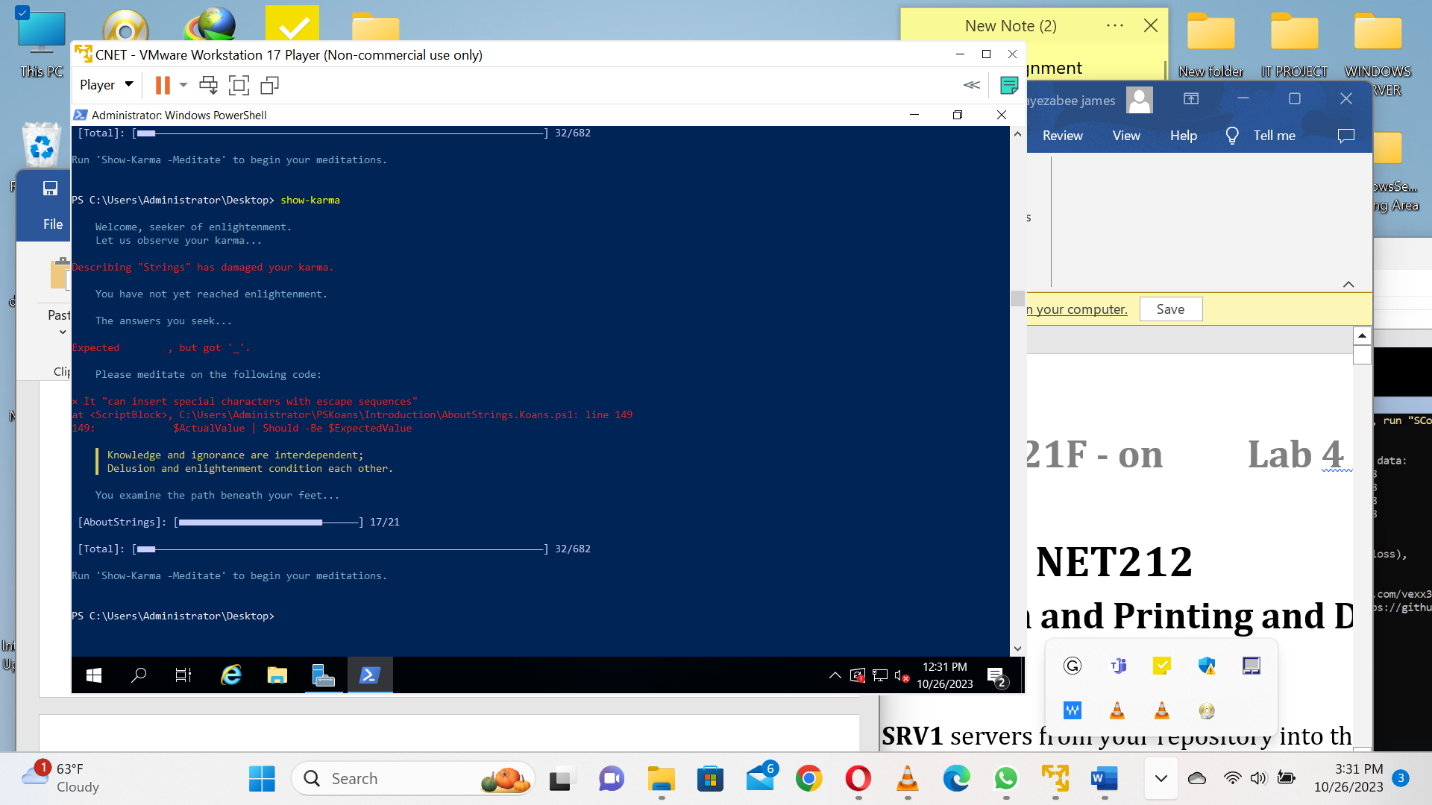
    It '( 10 -lt 20 ) is either true or false' {  
        $\_\_\_\_ | Should -Be ( 10 -lt 20 ) -Because '10 is less than 20'  
    }

    It '( 10 -gt 20 ) is either true or false' {  
        $\_\_\_\_ | Should -Be ( 10 -gt 20 ) -Because 'The lesser is not greater'  
    }

    It '( 3 -eq 3 ) is either true or false' {  
        $\_\_\_\_ | Should -Be ( 3 -eq 3 ) -Because 'A mirror reflects true'  
    }

    It '( 100 -lt 1 ) is either true or false' {  
        $\_\_\_\_ | Should -Be ( 100 -lt 1 ) -Because '100 is not less than 1'  
    }  
}

**ANSWER**



**Describe "Booleans"** {

    # Using only booleans, either $true or $false, fill in the blanks below.

    It '( 1 -gt 2 ) is either true or false' {

        $false | Should -Be ( 1 -gt 2 ) -Because '1 is not greater than 2'

    }

    It '( 1 -lt 2 ) is either true or false' {

        $true | Should -Be ( 1 -lt 2 ) -Because '1 is less than 2'

    }

    It '( 10 -lt 20 ) is either true or false' {

        $true | Should -Be ( 10 -lt 20 ) -Because '10 is less than 20'

    }

    It '( 10 -gt 20 ) is either true or false' {

        $false | Should -Be ( 10 -gt 20 ) -Because 'The lesser is not greater'

  It '( 10 -gt 20 ) is either true or false' {

        $false | Should -Be ( 10 -gt 20 ) -Because 'The lesser is not greater'

    }

    It '( 3 -eq 3 ) is either true or false' {

        $true | Should -Be ( 3 -eq 3 ) -Because 'A mirror reflects true'

    }

    It '( 100 -lt 1 ) is either true or false' {

        $false | Should -Be ( 100 -lt 1 ) -Because '100 is not less than 1'

    }

}

**Numbers**

In PowerShell, we can use several different data types to represent numbers.

Which type is used depends on what you need. The two most common types are:

- Integers

Integers represent whole numbers. This is the default numeric data type

that PowerShell uses for numbers that don't include a decimal portion.

- Doubles

Doubles are typically numbers which include decimal places or exponents.

This is the default numeric type that PowerShell will use for numbers

specified with decimal places or exponents, for example:

- 0.5

- 2e3

So 10 will be an integer and 10.0 will be a double, unless a specific type

is applied either with a type suffix or a cast, which will be covered in

more detail in a later topic. The examples below display some of the

differences inherent to the numeric types.

#>

**Describe 'Basic Number Types**' {

Context 'Double' {

It 'can result from an operation involving multiple types of numbers' {

<#

When doing arithmetic on different types of numbers, PowerShell

will automatically convert the less precise or more narrow type

to the other kind.

#>

$Int = 10

$Double = 10.0

'System.Int32' | Should -Be $Int.GetType().Fullname

'System.Double' | Should -Be $Double.GetType().Fullname

$Result = $Int \* $Double

# What type results when you multiply integers and doubles?

'System.Double' | Should -Be $Result.GetType().Fullname

}

}

Context 'Integers' {

It 'has to be a whole number' {

function Get-Number {

param(

[Parameter(Mandatory)]

[int]

$Number

)

$Number

}

$Pi = [Math]::PI

'System.Double' | Should -be $Pi.GetType().Fullname

# What number will return if you pass Pi into an int function?

[int]3 | Should -Be (Get-Number -Number $Pi)

}

It 'can be a larger number if needed' {

<#

Integers come in two flavours:

- int (Int32)

- long (Int64)

If an integer value exceeds the limits of the Int32 type, it is

automatically expanded to the larger Int64 type.

#>

# What exactly are the limitations of the [int] type?

$MaxValue = [int]::MaxValue

$MinValue = [int]::MinValue

$MaxValue | Should -Be $MaxValue

$MinValue | Should -Be $MinValue

# If you enter a number larger than that, the type should change.

$BigValue = [int]::MaxValue + 1

$BigValue | Should -BeOfType [double]

$BigValue | Should -BeGreaterThan $MaxValue

$SmallValue = [int]::MinValue - 1

$SmallValue | Should -BeOfType [double]

$SmallValue | Should -BeLessThan $MinValue

}

It 'allows you to request the larger type with a suffix' {

<#

By specifying the L suffix, a number is forced to use the long

type. here are more suffixes available, depending on your

PowerShell version. We'll cover those in a later topic.

#>

100L | Should -BeOfType [long]

$Value = [int]::Maxvalue + 1

$Value | Should -BeOfType [double]

[int]::MinValue -le $Value -and $Value -le [int]::MaxValue | Should -BeFalse

}

}

}

**Describe "Banker's Rounding**" {

<#

The default midpoint rounding method used in PowerShell is called

"Rounding to Even," or "Banker's Rounding". Numbers will be rounded to

the nearest even integer when rounding from a midpoint (###.5) value.

This behaviour stems from the underlying library method [math]::Round()

which is documented in more detail here:

https://docs.microsoft.com/en-us/dotnet/api/system.math.round#midpoint-values-and-rounding-conventions

Alternate overloads of the Round() method are also available if you need

to adjust the midpoint rounding method used for a particular use case.

#>

It 'rounds to the nearest even when cast to integer if it is a midpoint' {

# How will these numbers be rounded to integer?

2 | Should -Be ([int]2.5)

3 | Should -Be ([int]3.34)

13 | Should -Be ([int]12.7)

11 | Should -Be ([long]10.61)

6 | Should -Be ([long]5.5)

}

}

using module PSKoans

**Strings**

Strings in PowerShell come in two flavours: expandable strings, and string

literals. Expandable strings in PowerShell are created using double quotes,

while string literals are created with single quotes.

Expandable strings in PowerShell can evaluate expressions or variables

mid-string in order to dynamically insert values into a preset string, like

a sort of impromptu template string.

#>

Describe 'Strings' {

It 'is a simple string of text' {

'String' | Should -Be 'string'

}

Context 'Literal Strings' {

It 'assumes everything is literal' {

$var = 'Some things you must take literally'

$var | Should -Be $var

}

It 'can contain special characters' {

# 'Special' is just a title.

$complexVar = 'They interpret these characters literally: $ ` $()'

$complexVar | Should -Be $complexVar

}

It 'can contain quotation marks' {

$Quotes = 'This creates only one set of ''quotation marks'' in the string.'

# Single quotes are easier to work with in double-quoted strings.

"$Quotes" | Should -Be $Quotes

}

}

Context 'Evaluated Strings' {

It 'can expand variables' {

$var = 'apple'

"My favorite fruit is $Var" | Should -Be "My favorite fruit is $var"

}

It 'can do a simple expansion' {

"Your home directory is: $HOME" | Should -Be "Your home directory is: $HOME"

}

It 'handles other ways of doing the same thing' {

# Strings can handle entire subexpressions being inserted as well!

$String = "Your home folder is: $(Get-Item -Path $HOME)"

$String | Should -Be $String

}

It 'will expand variables that do not exist' {

<#

If a string contains a variable that has not been created,

PowerShell will still try to expand the value. A variable that

doesn't exist or has a null value will simply disappear when it

is expanded.

This could be the result of a typing mistake.

#>

$String = "PowerShell's home folder is: $SPHome"

$String | Should -Be $String

}

It 'can get confused about :' {

<#

In PowerShell, a colon (:) is used to define a scope or provider

path for a variable. For example, the Environment provider uses

the syntax $env:SomeVariableName, which refers to an environment

variable named 'SomeVariableName'. Environment variables are

initially set by the operating system and usually passed to

child processes, offering a limited way for processes to

communicate in a limited way.

When : is part of a string, PowerShell will try and expand a

variable in that scope or from that provider.

#>

$Number = 1

$String = "$Number:Get shopping"

$String | Should -Be $String

}

It 'can use curly braces to mark the variable name' {

<#

Variables followed by : or adjacent to other characters normally

included in a variable name can be referenced by enclosing the

variable name with curly braces like so: ${varName}

#>

$Number = 1

$String = "${Number}:Get shopping"

$String | Should -Be $String

}

It 'can escape special characters with backticks' {

$LetterA = 'Apple'

$String = "`$LetterA contains $LetterA."

$String | Should -Be $String

}

It 'can escape quotation marks' {

$String = "This is a `"string`" value."

$AlternateString = "This is a ""string"" value."

# A mirror image, a familiar pattern, reflected in the glass.

$Results = @(

$String

$AlternateString

)

$Results | Should -Be @($String, $AlternateString)

}

It 'can insert special characters with escape sequences' {

<#

All text strings in PowerShell are actually just a series of

character values. Each of these values has a specific number

assigned in the [char] type that represents that letter, number,

or other character.

.NET uses UTF16 encoding for [char] and [string] values.

However, with most common letters, numbers, and symbols, the

assigned [char] values are identical to the ASCII values.

ASCII is an older standard encoding for text, but you can still

use all those values as-is with [char] and get back what you'd

expect thanks to the design of the UTF16 encoding. An extended

ASCII code table is available at: https://www.ascii-code.com/

#>

$ExpectedValue = [char] 9

<#

If you're not sure what character you're after, consult the

ASCII code table above.

Get-Help about\_Special\_Characters will list the escape sequence

you can use to create the right character with PowerShell's

native string escape sequences.

#>

$ActualValue = "`\_"

$ActualValue | Should -Be $ExpectedValue

}

}

Context 'String Concatenation' {

It 'adds strings together' {

# Two become one.

$String1 = 'This String'

$String2 = 'is cool.'

$String1 + ' ' + $String2 | Should -Be 'This string is cool.'

}

It 'can be done more easily' {

# Water mixes seamlessly with itself.

$String1 = 'This string'

$String2 = 'is cool.'

"$String1 $String2" | Should -Be 'This string is cool.'

}

}

Context 'Substrings' {

It 'lets you select portions of a string' {

# Few things require the entirety of the library.

$String = 'At the very top!'

<#

The [string].Substring() method has a few variants, or

"overloads." The most common overloads are:

string Substring(int startIndex)

string Substring(int startIndex, int length)

In other words:

- Both variants return a string.

- One variant needs two index references, where to start and

stop in selecting the substring.

- The other only requires a starting index, and goes until the

end of the original string.

#>

'At the' | Should -Be $String.Substring(0, 6)

'the very top!' | Should -Be $String.Substring(7)

}

}

Context 'Here-Strings' {

<#

Here-strings are a fairly common programming concept, but have some

additional quirks in PowerShell that bear mention. They start with

the sequence @' or @" and end with the matching reverse "@ or '@

sequence.

The terminating sequence MUST be at the start of the line, or they

will be ignored, and the string will not terminate correctly.

#>

It 'can be a literal string' {

$LiteralString = @'

Hullo!

'@ # This terminating sequence must be at the start of the line.

# "Empty" space, too, is a thing of substance for some.

' Hello' | Should -Be $LiteralString

}

It 'can be an evaluated string' {

# The key is in the patterns.

$Number = 42

# Indentation sometimes gets a bit disrupted around here-strings.

$String = @"

I am number #$Number!

"@

'I am number #42' | Should -Be $String

}

It 'interprets all quotation marks literally' {

$AllYourQuotes = @"

All things that are not 'evaluated' are "recognised" as characters.

"@

'All things that are not ''evaluated'' are "recognised" as characters.' | Should -Be $AllYourQuotes

}

}

Context 'Arrays and Strings' {

It 'can be inserted into a string' {

<#

Arrays converted to string will display each member separated by

a space by default.

#>

$array = @(

'Hello'

'world'

)

'Hello world' | Should -Be "$array"

}

It 'can be joined with a different string by setting the OFS variable' {

<#

The $OFS variable, short for "Output Field Separator," defines

the separator used to join an array when it is converted to a

string.

By default, the OFS variable is unset, and a single space is

used as the separator.

#>

$OFS = '... '

$array = @(

'Hello'

'world'

)

'Hello... world' | Should -Be "$array"

# Removing the created OFS variable, the default will be restored.

Remove-Variable -Name OFS -Force

}

}

}