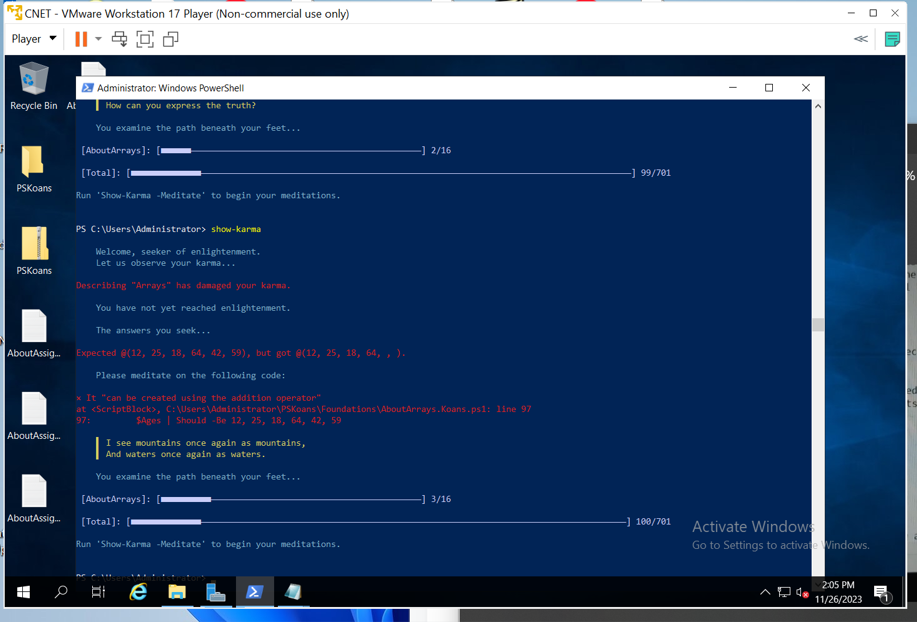
**Please complete from Koan all challenge until 100**.

****

**CODES**

**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))

}

}

}

**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?

0 | 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 11110001 to an integer' {

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

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

$Binary = "11110001"

241 | 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 = "11110001"

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))

}

}

}

**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?

0 | 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 11110001 to an integer' {

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

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

$Binary = "11110001"

241 | 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))

}

}

}

**using module PSKoans**

**[Koan(Position = 112)]**

**param()**

**<#**

**Assignment and Arithmetic Operators**

Just like many other programming languages, PowerShell has special operators designed to

work with data.

You can use the following command to get a full overview of how operators work:

Get-Help about\_Operators

Loosely, operators fall into a few categories: assignment (=), arithmetic, comparison,

redirection, and string operators.

In terms of order of operations, arithmetic operators typically execute before

comparison operators, followed by string operators, redirection operators, and finally

assignment operators execute last.

#>

Describe 'Assignment Operator' {

It 'is used to assign a value to variables' {

$ExpectedValue = 1 + 1

$ActualValue = $ExpectedValue

$ActualValue | Should -Be $ExpectedValue

}

It 'is also used to assign a value to properties or elements' {

# With arrays, we can assign values directly to indexes

$Array = 1, 2, 3

$Array[2] = 5

$Array | Should -Be @(1, 2, 5) # What would change?

}

It 'can assign values to many variables at once' {

$Var1 = $Var2 = $Var3 = $Var4 = 27

$Var1 | Should -Be $Var2

$Var2 | Should -Be $Var3

$Var3 | Should -Be $Var4

$Var4 | Should -Be 27

}

It 'can assign multiple values to multiple variables' {

$Var1, $Var2 = @("Correct", "Incorrect")

$Var1 | Should -Be "Correct"

$Var2 | Should -Be "Incorrect"

}

}

**Describe 'Arithmetic Operators'** {

<#

These can be used for standard arithmetic with numerical values, as well as some limited

usage with arrays and strings that can come in handy.

#>

Context 'Addition' {

It 'is used to add two items together' {

13 + 4 | Should -Be 17

7 + 6 | Should -Be 13

13.7 + 4 | Should -Be (13.7 + 4)

}

It 'can be used to concatenate strings' {

'My name is ' + 'Jim' | Should -Be 'My name is Jim'

'hello' + 'world' | Should -Be ('hello' + 'world')

}

It 'can be used to put arrays together' {

<#

As we will cover in AboutArrays, this is not so much 'adding' arrays together as it is

building a totally new array. It does, however, have its uses.

#>

$Array = 1, 2, 3, 4, 5

$ExpectedResult = $Array + 7

$Result = @(

1

2

3

4

5

7

)

$Result | Should -Be $ExpectedResult

}

It 'behaves according to the type of the left-hand item' {

'10.511' | Should -Be ('10.5' + 11)

(11 + '12.5') | Should -Be (11 + '12.5')

12.21 + 11 -eq 23.43 | Should -BeFalse

# Adding items into typed arrays will also cause the resulting value to be converted

[int[]] $Array = @(1, 2, 3, 4, 5)

$Array += '17'

$Array | Should -Be @(1, 2, 3, 4, 5, 17)

}

}

Context 'Subtraction' {

It 'works similarly to addition' {

12 - 7 | Should -Be 5

(11 - 3.5) | Should -Be (7.5)

}

It 'cannot be used with strings' {

{ 'hello' - 'h' } | Should -Throw

# However, this work if the string contains a useable number.

4.5 | Should -Be ('12' - '7.5')

# In other words, subtraction only operates on numerical values.

{ @(1, 2) - 1 } | Should -Throw -ExpectedMessage ' '

}

}

Context 'Multiplication' {

It 'can be used on both integer and non-integer numerals' {

48 | Should -Be (12 \* 4)

12.1 \* 2 | Should -Be 24.2

}

It 'can also be used on strings' {

'AAAA' | Should -Be ('A' \* 4)

('\_\_' \* 4) | Should -Be '\_\_\_\_\_\_\_\_'

}

}

Context 'Division' {

It 'is restricted to numeric use only' {

<#

As with subtraction, there's no useful meaning of using division on a string

so any attempts to do so will throw an error.

#>

{ 'hello!' / 3 } | Should -Throw -ExpectedMessage 'Cannot convert value'

<#

Unlike with other numerical operators, however, division often results

in a non-integer (double) value even when both operands are integers.

#>

3 / 4 | Should -Be 0.75

20 / 10 -eq 2 | Should -BeTrue

}

}

Context 'Modulus' {

# Modulus is a bit of an odd one, but common enough in programming. It performs a

# division, and then returns the integer value of the remainder.

It 'is usually used with integers' {

$Remainder = 15 % 7

1 | Should -Be $Remainder

}

It 'cannot be used on non-numeric values' {

$ModulusOnString = {

# Some things are better seen when you try them for yourself.

$String = 'hello!'

$String % 4

}

# Only a partially matching phrase from the error message is necessary.

$ModulusOnString | Should -Throw -ExpectedMessage ' '

$ModulusOnArray = {

# If you have trouble, try doing something similar in the console to see what happens.

$Array = 1, 10, 20

$Array % 4

}

$ModulusOnArray | Should -Throw -ExpectedMessage ' '

}

}

}

**Describe 'Assignment/Arithmetic Combination Operators'** {

It 'is a bit unwieldy to assign and increment without combination operators' {

$Value = 5

$Value = $Value + 5

10 | Should -Be $Value

}

It 'is possible to combine assignment and addition' {

$Value = 12

$Value += 7

19 | Should -Be $Value

}

It 'is also possible to combine subtraction with assignment' {

$Value = 19

$Value -= 3

16 | Should -Be $Value

}

It 'works the same way with division' {

$Value = 16

$Value /= 2

8 | Should -Be $Value

}

It 'works with multiplication as well' {

$Value = 8

$Value \*= 3

24 | Should -Be $Value

}

It 'even works with modulus' {

$Value = 12

$Value %= 4

0 | Should -Be $Value

}

It 'can get a bit confusing to follow' {

$Value = 0

$Value /= 3

$Value %= 5

$Value += 4

$Value \*= 7

$Value -= 7

$Value | Should -Be 21

}

}

**using module PSKoans**

**[Koan(Position = 113)]**

**param()**

**<#**

**Arrays and Iterable Collections**

Like many programming languages, PowerShell often uses arrays to keep

collections of objects together. Arrays tie in closely with PowerShell's

pipeline, which is one way to iterate over a collection with a good deal of

efficiency.

There are a few 'array-like' collection types available in PowerShell, all

of which are rooted in .NET classes and data types, and behave much the same

as they do in other .NET languages.

Arrays in particular are closely tied to the PowerShell pipeline, which are

covered in another topic.

#>

Describe 'Arrays' {

It 'is useful for grouping related objects and values' {

<#

The comma operator is used to create an array.

Spaces are typically ignored.

#>

$Ages = 12, 25, 18, 64

<#

Individual elements of an array can be accessed with square-bracket

index syntax. Arrays are zero-indexed; the first element is at index

[0], the second at [1], etc.

#>

$Ages[0] | Should -Be 12

$Ages[-1] | Should -Be $Ages[3]

}

It 'can be created with the @() operator' {

<#

The array subexpression operator @() is used to create an array

from multiple values or expressions. Within the parentheses, you can

use commas, semicolons, or even line breaks to divide each element.

#>

$Names = @(

'Steve'

'John'; 'Jaime' # This is a messy way to do things, but it does work

'Abigail', 'Serena', 'Kali'

<#

Having everything on its own line would be much cleaner and is a

more common usage of this syntax.

#>

)

# Where is index 4 in the above array?

'Serena' | Should -Be $Names[4]

<#

Although in many cases in PowerShell, an expression that only

returns one value will not become an array, this operator forces the

value or object to be wrapped in an array if the result is not

already an array; it guarantes the result will be an array.

#>

$Names[4] | Should -Be 'Serena'

$Array = @( 10 )

$Array.GetType().FullName | Should -Be 'System.Object[]'

}

It 'is a fixed size collection; elements cannot be added or removed' {

# Most arrays in PowerShell are fixed size. Elements cannot be directly added or removed.

$Ages = 12, 25, 18, 64

{ $Ages.Add(59) } | Should -Throw

{ $Ages.Remove(12) } | Should -Throw

}