```
 ____    ____________________________________________________________________________________________________
//¯¯\\__//¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯\\___
\\__//¯¯¯ New-MockZip                                                                                ___//¯¯\\
 ¯¯¯\\__________________________________________________________________________________________________//¯¯\\__//
     ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯    ¯¯¯¯
```

```
\________________________________________________________________________________________________________________________/
  Start /¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯\
/¯¯¯¯¯¯¯
```

Greetings PowerShell Community,

Today, I've decided to share a custom function named “New-MockZip” that has embedded custom .Net classes that I've developed with the intention of performing various experiments with random number generation, string data in Base64 format, and the native PowerShell archive classes.

If you would like to follow along with the script, it is located at this hyperlink...

```
________________________________________________________________________________________________
| New-MockZip.ps1 | https://github.com/mcc85s/FightingEntropy/blob/main/Scripts/New-MockZip.ps1 |
¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯
```

```
                                                                                                                 _______/
\_______________________________________________________________________________________________________________/ Start
  Overview /¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯\
/¯¯¯¯¯¯¯¯¯¯
```

Before I get started, what PRACTICAL USE would something like this be good for...?
Well, consider the following:

SUPPOSE, I want to TEST various elements on a system.
If I want to test a piece of data, or benchmark a particular piece of hardware...?
Then this might be a great way to do that.

This function CAN throw quite a load at the processing unit, hard drive, or whatever else.
I developed this for a more practical reason than to just generate random information, but- I think that people might be able to learn something from this particular function, especially if they want to get their hands dirty with generating random entries for a database...? Obfuscation/deobfuscation...? Or really, whatever you can think of where you want to use a specific character set and get a bunch of data back.

Really, with some minor tweaking, this function could be custom tailored for a VERY LARGE RANGE of practical uses... but I'm not going to cover all of those ideas.

However, I WILL construct a way to logically divide a “standard workload”, and then use the System.IO.Compression assembly to use it's standard base classes, AND, to export the randomly generated data to an archive.

Note: If anyone wants to know the REAL reason why I made this, I'm trying to experiment with memory stream objects in PS, and building a (*.zip) file WITHOUT writing the individual ZipFileEntry files to disk FIRST, but I'm not quite certain my objective is possible without installing something like DotNetZip.

```
                                                                                                              __________/
\____________________________________________________________________________________________________________/ Overview
  Function New-MockZip /¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯\
/¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯
```

So, in order to build the complete function, I had to develop several classes that are embedded within the function. It IS possible to do all of this work WITHOUT creating classes...? However, point of this is to illustrate WHY the class approach is a sensible option. In C#, programs are written with a format of:

```
using xxx;
using xxx.yyy;

namespace CoolNamespace
{
    class CoolClass5000
    {
        int Index;
        string Name;
    }
}
```

But in PowerShell, you can achieve the same result by writing functions.
If you use: `[CmdLetBinding()]Param([Parameter(Mandatory,Position=0)][String]$Etc`...?
You can COMBINE really cool features that the PowerShell engine offers, while keeping the code DYNAMIC.

I'll start with the function wrapper.

```
    # // ________________________
    # // | Add Compression Types |
    # // ------------------------

    Add-Type -AssemblyName System.IO.Compression, System.IO.Compression.Filesystem

    # // ____________________________________________________________________________________________
    # // | Establish function and classes that create a (*.zip) file with random Base64 information |
    # // --------------------------------------------------------------------------------------------

    Function New-MockZip
    {
        [CmdLetBinding()]Param(
        [Parameter(Mandatory,Position=0)][String]$Path,
        [Parameter(Position=1)][UInt32]$Factor=1000,
        [Parameter(Position=2)][UInt32]$Count=100,
        [Parameter(Position=3)][UInt32]$Length=100,
        [Parameter(Position=4)][UInt32]$Width=120)

        # // __________________________________
        # // | Where the classes are gonna go |
        # // ----------------------------------
    }
```

Right now, this function will do absolutely nothing, because there's nothing going on within its' scope.
However, to explain what that code indicates so far...?

```
    [Parameter(Mandatory,Position=0)][String]$Path,

    # // __________________________________________________________
    # // | The base path where the workload will create new items |
    # // ----------------------------------------------------------

    [Parameter(Position=1)][UInt32]$Factor=1000,

    # // ____________________________________________________________________________
    # // | How many times per page we want to replicate the randomly generated data |
    # // ----------------------------------------------------------------------------

    [Parameter(Position=2)][UInt32]$Count=100,

    # // ___________________________________________________________
    # // | How many pages for the 'book' we're randomly generating |
    # // -----------------------------------------------------------

    [Parameter(Position=3)][UInt32]$Length=100,

    # // _________________________________________________________________
    # // | How many lines per page of the book, to be randomly generated |
    # // -----------------------------------------------------------------

    [Parameter(Position=4)][UInt32]$Width=120)

    # // ________________________________
    # // | How many characters per line |
    # // --------------------------------
```

The name of the parameters probably sounds like we're working with a 3D object, doesn't it...? That's because
the data that is being randomly generated, is creating something that has similar properties to a 3D object.
The reason I used these property names, is because people have a pretty easy time understanding 3D objects.

Now what I'll do is cover each individual class, and explain what they all do.

```
                                                                                                      ____________________/
\_____________________________________________________________________________________________________/ Function New-MockZip
  MockBox /¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯\
/¯¯¯¯¯¯¯¯¯
```

This particular class is meant to control the dimensions of the output, so every single page, line, and character is controlled by this particular class.

```
# // _________________________________________________
# // | Maintains the dimensions of the output object |
# // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

Class MockBox
{
    [UInt32] $Factor
    [UInt32] $Count
    [UInt32] $Depth
    [UInt32] $Length
    [UInt32] $Width
    MockBox([UInt32]$Factor,[UInt32]$Count,[UInt32]$Length,[Uint32]$Width)
    {
        $This.Factor = $Factor
        $This.Count  = $Count
        $This.Depth  = ([String]$This.Count).Length
        $This.Length = $Length
        $This.Width  = $Width
    }
    [String] ToString()
    {
        Return ($This.PSObject.Properties | % { "{0}: [{1}]" -f $_.Name, $_.Value }) -join ', '
    }
}
```

```
                                                                                                                  _________/
\_________________________________________________________________________________________________________________/ MockBox
  MockPage /¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯\
/¯¯¯¯¯¯¯¯¯¯
```

This is essentially a single page of random Base64 characters. There is plenty more that could be done to this particular class, however, this will keep the output pretty streamlined and allow access back into the properties for each individual page later on upon data insertion.

```
# // _____________________________________________
# // | A single page of random Base64 characters |
# // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

Class MockPage
{
    [UInt32] $Index
    [String] $Path
    [Object] $Content
    MockPage([UInt32]$Index,[Object]$Content)
    {
        $This.Index    = [UInt32]$Index
        $This.Content  = $Content
    }
    [String] ToString()
    {
        Return "<[MockPage]>"
    }
}
```

```
                                                                                                                 __________/
\________________________________________________________________________________________________________________/ MockPage
```

```
  MockBook /¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯\
/¯¯¯¯¯¯¯¯¯¯
```

Now, this is basically the binding for all of the random pages of data. It is doing a lot more than just holding it all together. I'll explain in depth what's going on in this class, because it does have a few layers and operations it is performing. I'll paste it, and then afterward, I will dissect it.

```
# // ____________________________________________________________
# // | A collection of single pages of random Base64 characters |
# // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

Class MockBook
{
    [Object] $Base64
    [UInt32] $Count
    [Object] $Box
    [String] $Path
    [Object] $Output
    [String] Status([UInt32]$Rank)
    {
        Return "({0:d$($This.Box.Depth)}/{1})" -f $Rank, $This.Box.Count
    }
    [UInt32] Percent([UInt32]$Rank)
    {
        If ($Rank -eq 0)
        {
            $Rank = $Rank + 1
        }
        Return (($Rank/$This.Box.Count)*100)
    }
    [Hashtable] Progress([String]$Activity,[UInt32]$Rank)
    {
        Return [Hashtable]@{

            Activity = $Activity
            Status   = $This.Status($Rank)
            Percent  = $This.Percent($Rank)
        }
    }
    MockBook([String]$Path,[Object]$Box)
    {
        # // __________________________________________
        # // | Cast all Base64 characters to an array |
        # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

        $This.Base64 = 065..090+097..122+048..057+043,047 | % { [Char]$_ }
        $This.Path   = $Path
        $This.Box    = $Box

        # // _______________________________________________________
        # // | Divides the workload cleanly for process indication |
        # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

        $Slot = Switch ($This.Box.Count)
        {
            {$_ -lt 100}
            {
                $Div   = [Math]::Round(100/$This.Box.Count,[MidpointRounding]::AwayFromZero)
                $Step  = [Math]::Round(100/$Div)
                0..($Div-1) | % { $_ * $Step }
            }
            {$_ -eq 100}
            {
                $Div   = 100
                $Step  = 1
                0..99 | % { $_ * $Step }

            }
            {$_ -gt 100}
            {
                $Div   = $This.Box.Count/100
```

```
                    0..99 | % { [Math]::Round($_ * $Div) }
                }
            }

            $Slot[-1]      = $This.Box.Count
            $Book          = @{ }

            # // __________________________________________________________________
            # // | Create the book by casting individual pages to hashtable $Book |
            # // ------------------------------------------------------------------

            $Splat = $This.Progress("Generating [~]",0)
            Write-Progress @Splat

            ForEach ($X in 0..($This.Box.Count-1))
            {
                $Page    = @{ }

                # // __________________________________________________________________
                # // | Create the page by casting individual lines to hashtable $Page |
                # // ------------------------------------------------------------------

                ForEach ($I in 0..($This.Box.Length-1))
                {
                    # // ___________________________________________________
                    # // | Create the line by randomly generating numbers |
                    # // | and using the variable $This.Base64             |
                    # // ---------------------------------------------------

                    $Random = $This.Random()
                    $Line   = $This.Base64[$Random] -join ''

                    $Page.Add($Page.Count,$Line)
                }

                $Book.Add($Book.Count,[MockPage]::New($Book.Count,$Page[0..($Page.Count-1)]))

                If ($X -in $Slot)
                {
                    $Splat = $This.Progress("Generating [~]",$X)
                    Write-Progress @Splat
                }
            }
            $Splat = $This.Progress("Generating [~]",100)
            Write-Progress @Splat -Complete

            # // __________________________________________________________
            # // | Save the hashtable content to the array, $This.Output |
            # // ----------------------------------------------------------

            $This.Output = $Book[0..($Book.Count-1)]
        }
        [UInt32[]] Random()
        {
            $Hash = @{ }

            0..($This.Box.Width-1) | % { $Hash.Add($Hash.Count,(Get-Random -Maximum 63)) }

            Return $Hash[0..($Hash.Count-1)]
        }
        [String] ToString()
        {
            Return "<[MockBook]>"
        }
    }
```

Ok, so I'm going to list all of the properties and methods of this particular class right here...

```
    [Object] $Base64
    [UInt32] $Count
    [Object] $Box
    [String] $Path
    [Object] $Output
    [String] Status([UInt32]$Rank)
    [UInt32] Percent([UInt32]$Rank)
    [Hashtable] Progress([String]$Activity,[UInt32]$Rank)
    [UInt32[]] Random()
    [String] ToString()
    MockBook([String]$Path,[Object]$Box)
```

```
In the class, I do not have the methods and properties in THAT particular order, but that's ok.
The properties will retain their order. The methods don't really need to retain their order.

The main thing to remember with these classes, is that the method which bears the SAME EXACT NAME as the class,
is the SAME EXACT THING, as the Main() method in C#. Or like, Main(Args[]).

The properties are prefixed with the dollar sign symbols, while the methods are not.
SOME of the methods have a parameter, but not all of them.
________________________________________________________________________________
| MockBook([String]$Path,[Object]$Box) method is the main method, and it requires: |
| [String] $Path : to an existing folder                                          |
| [Object] $Box :  dimensions of the output                                       |
--------------------------------------------------------------------------------
Now, the function way up above requests a bunch of parameters, and yet this function right here only
needs (2). That's because it's sorta cobbling together a bunch of parameters and feeding it into this class.

Imagine it like this... You have a bunch of clothes. You put them all on, and then you don't really put a whole
lot of thought into what each individual article of clothing is doing, right...? Well, that's the idea here.
```

```
    PS Prompt:\> $Zip.Box

    Factor : 1000
    Count  : 100
    Depth  : 3
    Length : 100
    Width  : 120

    PS Prompt:\>
```

```
By having an individual sub class, then internal classes and types can all rely on the same information while
making the sharing of those details pretty easy for the computer as well as the person writing the script.

It's an object. It's like “I am a box, I'm 1000 pounds, I'm 100 layers thick, (depth is for string formatting)
I'm 100 inches across and 120 inches wide.” Anyway...
_______________________________________________________________________________________________________
| [Hashtable] Progress([String]$Activity,[UInt32]$Rank): returns a [Hashtable] to splat to Write-Progress |
| and this particular method is calling (2) methods internally while ALSO returning the [Hashtable].      |
| [String] Status([UInt32]$Rank), and [UInt32] Percent([UInt32]$Rank) are both being called upon while    |
| returning a [Hashtable] object.                                                                         |
-------------------------------------------------------------------------------------------------------
The property [Object] $Base64, is basically doing some real cool stuff with hexadecimal, and then the method
[UInt32[]] Random() is collecting an array of random numbers so that those numbers can be used as an array
to index the particular $Base64 characters. I'll talk about the main method now.
```

```
    # // __________________________________________
    # // | Cast all Base64 characters to an array |
    # // ------------------------------------------

    $This.Base64 = 065..090+097..122+048..057+043,047 | % { [Char]$_ }
    $This.Path   = $Path
    $This.Box    = $Box
```

As you can see, `$This.Base64` is casting a lot of numbers in an array `065..090+097..122+048..057+043,047`
That particular arrangement is using a combination of techniques. `065..090` is the array selector, which
for PowerShell veterans... you'll obviously know what that means. For someone that's new to PowerShell,
the number `065` followed by `..` and a HIGHER number than `065` will procedurally iterate through all numbers
in that range. So, to illustrate what happens here...

```
PS Prompt:\> 065..090
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
PS Prompt:\>
```

The trailing 0 before 65/90, is mainly just to pad the number out.
Padding the number with leading zeros is mainly for outputting to a string.
It makes virtually no difference right here, because the engine is chopping the `0` off of each number.
Which means I could easily use `65..90`, and it would perform the same exact operation as above.

However, since SOME of the numbers are (3) digits, as opposed to only (2), I guess it's just a matter of
preference on the notation.

Using this specific array selector `65..90` followed by a `+` indicates that the array is allowing additional
items to be added to the chain. If you were to use a `,` here, it'll cause problems. You can only use a `+` after
an array selector (like `65..90`).

However, if we closely examine what's going on in the entire chain, `065..090+097..122+048..057+043,047` , clearly
the number `043` is a single number, and then it is followed by a `,` - that is fine. That will work, because it is
indicating that a single item was added, but there isn't an additional array selector, it's only adding the `047`
at the tail end. If you were to try this... `065..090+097..122+048..057+043,047..057`, that would fail. This
particular technique is a little challenging to learn without some helpful hints...

The point of this is because I want to select ALL capital letters (A-Z), ALL lowercase letters (a-z), ALL numbers
(0-9), and (2) symbols "+" and "/". That's effectively the Base64 character set. Each of these numbers when
combined with the `[Char]` object converts the number into the character. Another way I could have done this is
to use hexadecimal notation...

```
-------------------------------------------------------------------------------------------------------------
| 065..090+097..122+048..057+043,047 | % { "0x{0:x2}" -f $_ }, ← Returns hex strings which need invocation |
| 0x41..0x2f ← Returns the actual numbers in that range, and that will not return the desired range as is   |
-------------------------------------------------------------------------------------------------------------
```

Suffice to say, I am getting really deep into the semantics of the "unwritten rules" of PowerShell.
The point is, by having this particular (chain/array) of numbers, I can reproduce all of the desired characters.

By casting `065..090+097..122+048..057+043,047 | % { [Char]$_ }` to `$This.Base64`, then I can generate random
numbers in a chain of numbers and then pull the particular characters I want from that character array.
Another method of doing this is like this: `[Char[]]@(065..090+097..122+048..057+043,047)`.

So, in the end, the first line here is throwing a fair number of techniques into a single line.

But the end result is this...

```
PS Prompt:\> [Char[]]@(065..090+097..122+048..057+043,047)
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
0
1
2
3
4
5
6
7
8
9
+
/
PS Prompt:\>
```

I realize that I just dedicated an entire page to reproducing all of that, however, the single line that I wrote produces all of that. That is basically the “alphabet” or “base” of the randomly generated content I want to generate many, many pages of.

The next part is the count selector, which sorta involves a little bit of arithmetic.
If the count is less than 100, then I have to divide 100 by the number of times that the count goes into 100, so that the progress indicator knows what percentage to show to the user.

If the count is like, idk, 40...? Then what the function has to do is basically say 40 goes into 100 2 times FULLY, but then 1 more time PARTIALLY. That means there are (3) sections, and that means that the percentage is going to be 33, 66, 100. I'll explain in a moment.

If the count is exactly 100, then it'll be a 1-to-1 ratio with percentage, no adjustment required in that case.

If the count EXCEEDS 100, like for instance, maybe it's 140. Well, for every 1.4 items it processes, it has to increase the percentage by 1. That doesn't really work when you're dealing with integers... so the number has to be aggregated and then rounded.

```
# // _______________________________________________________
# // | Divides the workload cleanly for process indication |
# // -------------------------------------------------------

$Slot = Switch ($This.Box.Count)
{
    {$_ -lt 100}
    {
        $Div   = [Math]::Round(100/$This.Box.Count,[MidpointRounding]::AwayFromZero)
        $Step  = [Math]::Round(100/$Div)
        0..($Div-1) | % { $_ * $Step }
    }
    {$_ -eq 100}
    {
        $Div   = 100
        $Step  = 1
        0..99 | % { $_ * $Step }

    }
    {$_ -gt 100}
    {
        $Div   = $This.Box.Count/100
        0..99 | % { [Math]::Round($_ * $Div) }
    }
}

$Slot[-1]      = $This.Box.Count
```

Even though I've described what this particular section has to do...? It is probably pretty daunting to look at for some people. I'm using a LOT of techniques here, but ultimately, when I write these blocks of code, I have to consider WHAT the LEAST COMMON DENOMINATOR actually is, in each instance. If I can get away with performing the same operation in each scope, then I'll reserve those COMMON operations for AFTER the switch block.

So right here, we have a switch block that is assigning it's output to the variable $Slot.
The switch block is utilizing a combination of techniques that assign the output to the variable.

The reason for this entire operation is for PROCESSING EFFICIENCY. If I'm working with a count of like 65535, I will lose SO MUCH PERFORMANCE by having the function Write-Progress show itself for EACH and EVERY INDIVIDUAL NUMBER, and it will actually take far more time for the function to complete.

It doesn't seem intuitive until you start using various approaches, and the Write-Progress function is something you don't want to spam.

By using mathematics to (aggregate/calculate) the integers I DO want the Write-Progress function to show, we can filter a few of the integers from the RANGE of possible numbers, with the end result cutting most of that performance degradation out of the equation altogether.

In reality, CONCEPTUALIZING how this works... is a real pain in the neck. But- when you figure it out, then looking at it causes you to think “Ok, so that's how ya do it...?” Yep. ^ That's how you do it.

I'll break it down because that's quite a lot for somebody to understand...

```
Switch ($Object.Count)
{
    {$_ -lt 100} {<#..#>}  {$_ -eq 100} {<#..#>} {$_ -gt 100} {<#..#>}
}
```

Right here, the `Switch ($Object.Count) { {$_ -lt 100} {<#..#>} }`, is saying that the token `$_` is being compared to the number `100`, the token being the element `$Object.Count`. If the token doesn't match ANY of those conditions, you would want to use the `Default {<#..#>}`.

In this case, the number is ALWAYS going to fit (1) of those (3) conditions, so having the condition `Default {<#..#>}` is unnecessary.

If the condition is TRUE, then it will perform the operation in the following set of curly braces.
I didn't think something like this would work until I worked on some of DeploymentBunny's PSD scripts.
Once I saw that you could get away with doing this, I will implement it in certain instances.
It sure makes more sense than using a chain of these:

```
If     (<#condition#>) {<#action#>}
ElseIf (<#condition#>) {<#action#>}
Else                   {<#action#>}
```

I'm not saying that people should NEVER use ^ that, I'm just saying the switch approach is a more efficient use case here.

```
<# Count less than 100 #>

$Div   = [Math]::Round(100/$This.Box.Count,[MidpointRounding]::AwayFromZero)
$Step  = [Math]::Round(100/$Div)
0..($Div-1) | % { $_ * $Step }
```

So, if the count is less than 100, `$Div` has to basically figure out where between 1 and 99 it happens to be.
Then, it's gotta round itself so that the variable `$Step` can evenly space out the percentage.

I haven't tested every single condition, but the point of using the math in this manner, is so that I don't have to literally sit at the keyboard, and MANUALLY try every single combination, and then spend days testing that.
I think some people actually get paid a lot of money to use approaches like this...

In each of these operations, you'll see the last thing being `0..$Number | % { $_ * $Step }`
That's because it's capturing the full integers that represent each percentage point.

```
<# Count equal to 100 #>

$Div   = 100
$Step  = 1
0..99 | % { $_ * $Step }
```

This is actually more complicated looking than it really is, the `$Div = 100` is actually unnecessary...
However- I could actually be using that variable in the range, and had I found a way to displace the whole `0..$Number | % { $_ * $Step }` somehow to outside of the script block, then that would've allowed me to reduce the content of the script block by a line in each iteration.

The problem is that I couldn't do that with the last block.
Regardless, using variables helps me track down LEAST COMMON DENOMINATORS.

```
<# Count greater than 100 #>

$Div   = $This.Box.Count/100
0..99 | % { [Math]::Round($_ * $Div) }
```

Here, $Div is calculating what integer step is equivalent to a single percentage point. The last operation before the next section, is Slot[-1] = $This.Box.Count. The point of THAT, is because all of the rounding causes the last number to be somewhat short of 100 percent, and so the [-1] selector indicates the last item in the array should be set to the full number rather than a percentage.

```
    Slot[-1]       = $This.Box.Count
    $Book          = @{ }

    # // ___________________________________________________________________
    # // | Create the book by casting individual pages to hashtable $Book |
    # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

    $Splat = $This.Progress("Generating [~]",0)
    Write-Progress @Splat

    ForEach ($X in 0..($This.Box.Count-1))
    {
        $Page     = @{ }

        # // ___________________________________________________________________
        # // | Create the page by casting individual lines to hashtable $Page |
        # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

        ForEach ($I in 0..($This.Box.Length-1))
        {
            # // ___________________________________________________
            # // | Create the line by randomly generating numbers |
            # // | and using the variable $This.Base64            |
            # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

            $Random = $This.Random()
            $Line   = $This.Base64[$Random] -join ''

            $Page.Add($Page.Count,$Line)
        }

        $Book.Add($Book.Count,[MockPage]::New($Book.Count,$Page[0..($Page.Count-1)]))

        If ($X -in $Slot)
        {
            $Splat = $This.Progress("Generating [~]",$X)
            Write-Progress @Splat
        }
    }
    $Splat = $This.Progress("Generating [~]",100)
    Write-Progress @Splat -Complete
```

The # // Comments are providing a pretty good idea of what's happening.

The point of using the [Hashtable] is mainly because they're a lot faster than arrays, and they actually perform the same exact role in this circumstance. If you use this technique, your scripts will actually run a lot faster because the main issue with arrays is this simple... every time you ADD an element to an ARRAY, it has to DESTROY the array and RECREATE the entire thing.

With a [Hashtable], this tradeoff is nonexistent. However, keeping the items in a [Hashtable], is sort of a pain in the neck. To explain WHY I'm passing the content of the [Hashtable] BACK to an array at the end, I will do that in nearly every circumstance where I use a [Hashtable] as an array, because the number is acting as a KEY, not an INDEX.

Now to move onto the main class.

```
                                                                                                 __________/
\_______________________________________________________________________________________________/ MockBook
  MockZip /¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯\
/¯¯¯¯¯¯¯¯¯
```

This paricular class acts as the controller class for the entire function, as it is doing more work than anything else, and is actually acting as multiple items- including a class factory. A class factory controls a number of pre-defined classes that have already been specified, in nearly an identical way to a namespace in C#.

I will put the class in entirety right below, and then break portions of it down.

```
# // ___________________________________________________________
# // | Create a scratch zip file of randomly generated content |
# // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

Class MockZip
{
    [String] $Name
    [String] $Path
    [String] $Fullname
    [UInt32] $Exists
    [Object] $Box
    [Object] $Archive
    [Object] $Book
    MockZip([String]$Path,[UInt32]$Factor,[UInt32]$Count,[UInt32]$Length,[UInt32]$Width)
    {
        # // _____________________________
        # // | Test for the initial path |
        # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

        If (![System.IO.Directory]::Exists($Path))
        {
            Throw "Invalid Path"
        }

        If ($Factor -le 0)
        {
            # // ____________________________________________________
            # // | Factor determines how many times each individual |
            # // | page content is repeated in the output file      |
            # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

            Throw "Exception [!] Factor must be higher than 0, suggested: 1000"
        }

        If ($Count -le 1)
        {
            # // ______________________________________________________________
            # // | Count determines how many individual pages will be written |
            # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

            Throw "Exception [!] Count must be higher than 1, suggested: 100"
        }

        If ($Length -le 1)
        {
            # // _____________________________________________________________
            # // | Length determines how many lines will be written per page |
            # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

            Throw "Exception [!] Length must be higher than 1, suggested: 100"
        }

        If ($Width -le 1)
        {
            # // __________________________________________________________
            # // | Width determines how many characters each line will be |
            # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

            Throw "Exception [!] Width must be higher than 1, suggested: 120"
        }

        # // ____________________________________________________
        # // | Establish Date/Time -> Name -> Path -> Fullname |
        # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

        $This.Name       = [DateTime]::Now.ToString("yyyy_MMdd_HHmmss")
        $This.Path       = "{0}\{1}" -f $Path, $This.Name
        $This.Fullname   = "{0}\{1}.zip" -f $Path, $This.Name
        $This.Box        = $This.GenerateBox($Factor,$Count,$Length,$Width)

        # // ______________________
        # // | Establish new base |
        # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

        $This.NewBase()

        # // _________________________________________________
        # // | Establish random data to insert into zip file |
        # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

        $This.Book       = $This.GenerateBook()

        # // _________________________________________________
        # // | Create the zip file, and then inject the book |
        # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

        $This.Create()
        $This.Open()
        $This.InjectBook()

        # // _____________________
        # // | Save the zip file |
        # // ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

        $This.Close()
        $This.RemoveBase()
    }
    [Object] GenerateBox([UInt32]$Factor,[UInt32]$Count,[UInt32]$Length,[UInt32]$Width)
    {
        Return [MockBox]::New($Factor,$Count,$Length,$Width)
    }
    [Object] GenerateBook()
    {
        Return [MockBook]::New($This.Path,$This.Box)
    }
    InjectBook()
    {
        If (!$This.Book)
        {
            Throw "Must create a book first..."
        }

        $Depth = ([String]$This.Book.Output.Count).Length
        ForEach ($X in 0..($This.Book.Output.Count-1))
        {
            $Item      = $This.Book.Output[$X]
            $Item.Path = "{0}\{1:d$Depth}.txt" -f $This.Path, $X
            $Hash      = @{ }

            Switch ($This.Box.Factor)
            {
                {$_ -gt 1}
                {
                    ForEach ($I in 0..($This.Box.Factor-1))
                    {
                        $Hash.Add($I,$This.Book.Output[$X].Content)
                    }
                }
                {$_ -eq 1}
                {
                    $Hash.Add(0,$This.Book.Output[$X].Content)
                }
            }

            $Content   = $Hash[0..($Hash.Count-1)]
            [System.IO.File]::WriteAllLines($Item.Path,$Content)

            $This.AddFile($Item.Path)
        }
    }
```

```
        CheckBase()
        {
            $This.Exists     = [System.IO.Directory]::Exists($This.Path)
        }
        NewBase()
        {
            $This.CheckBase()
            If (!$This.Exists)
            {
                [System.IO.Directory]::CreateDirectory($This.Path)
                Write-Host "Created [+] Directory: [$($This.Path)]"
                $This.Exists = 1
            }
        }
        RemoveBase()
        {
            $This.CheckBase()
            If ($This.Exists)
            {
                [System.IO.Directory]::Delete($This.Path)
                Write-Host "Removed [+] Directory: [$($This.Path)]"
                $This.Exists = 0
            }
        }
        Create()
        {
            If ([System.IO.File]::Exists($This.FullName))
            {
                Throw "Exception [!] File: [$($This.Fullname)] already exists"
            }

            $Item = [System.IO.Compression.ZipFile]::Open($This.Fullname,"Create")

            If (![System.IO.File]::Exists($This.Fullname))
            {
                Throw "Exception [!] File: [$($This.Fullname)] was NOT created"
            }

            Write-Host "Created [+] File: [$($This.Fullname)]"
            $Item.Dispose()
        }
        Open()
        {
            If (![System.IO.File]::Exists($This.Fullname))
            {
                Throw "Exception [!] File: [$($This.Fullname)] does not exist"
            }

            $This.Archive = [System.IO.Compression.ZipFile]::Open($This.Fullname,"Update")

            If (!$This.Archive)
            {
                Throw "Exception [!] File: [$($This.Fullname)] could not be opened"
            }

            Write-Host "Opened [+] File: [$($This.Fullname)]"
        }
        Close()
        {
            If (!$This.Archive)
            {
                Throw "Exception [!] File: [$($This.Fullname)] is not yet opened"
            }

            Write-Host "Closing [~] File: [$($This.Fullname)]"
            $This.Archive.Dispose()

            $This.Archive = $Null

            Write-Host "Closed [+] File: [$($This.Fullname)]"
        }
        AddFile([String]$Path)
        {
            $ID = $Path | Split-Path -Leaf
            [System.IO.Compression.ZipFileExtensions]::CreateEntryFromFile(
                $This.Archive,
                $Path,
                $ID,
                [System.IO.Compression.CompressionLevel]::Fastest
            ) | Out-Null
            [System.IO.File]::Delete($Path)
        }
    }
```

Most of the `# // Comments` are doing a great job explaining what is happening.
But also, the name of the `Methods()` are also doing a pretty swell job explaining what's happening.

Allow me to rattle off the properties and the methods.

```
    [String] $Name
    [String] $Path
    [String] $Fullname
    [UInt32] $Exists
    [Object] $Box
    [Object] $Archive
    [Object] $Book
    MockZip([String]$Path,[UInt32]$Factor,[UInt32]$Count,[UInt32]$Length,[UInt32]$Width)
    [Object] GenerateBox([UInt32]$Factor,[UInt32]$Count,[UInt32]$Length,[UInt32]$Width)
    [Object] GenerateBook()
    InjectBook()
    CheckBase()
    NewBase()
    RemoveBase()
    Create()
    Open()
    Close()
    AddFile([String]$Path)
```

There's a fair amount goin' on here, but the bulk of this particular class boils down to `InjectBook()`, as it is effectively the entire point of the function. Some prerequisites need to be instantiated and created. First off, the methods `GenerateBox()` and `GenerateBook()` are basically acting as factory methods, which means that those particular functions are calling in the prior listed classes.

`GenerateBox()` is literally taking the dimensions, and creating an object that can be reproduced by the book generation. The way that I am ordering the operations in the main `MockZip()` method is allowing me to perform parameter validation, as well as to wait until ALL of the testing fields have been successful, before it even starts to combine elements and create a single (folder/file).

Once all of those testing fields have been validated, it'll run this chunk below.

```
    # // ___________________________________________________
    # // | Establish Date/Time -> Name -> Path -> Fullname |
    # // ---------------------------------------------------

    $This.Name       = [DateTime]::Now.ToString("yyyy_MMdd_HHmmss")
    $This.Path       = "{0}\{1}" -f $Path, $This.Name
    $This.Fullname   = "{0}\{1}.zip" -f $Path, $This.Name
    $This.Box        = $This.GenerateBox($Factor,$Count,$Length,$Width)
```

```
-----------------------------------------------------------------------------------
| [DateTime]::Now.ToString("yyyy_MMdd_HHmmss") is virtually the same thing as... |
| Get-Date -UFormat %Y_%m%d_%H%M%S                                                |
-----------------------------------------------------------------------------------
```

Even though I know both of these methods and how to get the output to be in the format I'm looking for, that doesn't mean I have them memorized. I constantly forget the formatting.

/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯__/¯¯\

```
Me: Meh.
    Was that a CAPITAL Y, or a lowercase y...?
    What if I just spam everything all at once, and see what THAT does...?
    Will people throw some tomatos at me if I continue to have this conversation in the lesson plan...?
    What are the chances of finding a real gray M&M...?
    Did Mariah Carey actually beat Marshal Mathers with a spatula...?
```

\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/¯¯\__/

Look, you can use either one of those methods up above.
The reason why it's included in the class is so that when the parameters are passed into the class, it will then start cobbling together the information to start producing some content.

The truth is, in order to generate the content that will be saved to the file system, the base directory has to exist first, so by having this class control the name of the path + (date/time), it'll allow all of the validation to cause a unique file path to be created (you could also use like a GUID for this sort of effect too).

The method `NewBase()` accesses the method `CheckBase()`, and if that path exists, it will create a directory with the path pointing to the validated information + path.

The method `[Object] GenerateBook()` is featured below.

```
    $Depth = ([String]$This.Book.Output.Count).Length
    ForEach ($X in 0..($This.Book.Output.Count-1))
    {
        $Item      = $This.Book.Output[$X]
        $Item.Path = "{0}\{1:d$Depth}.txt" -f $This.Path, $X
        $Hash      = @{ }

        Switch ($This.Box.Factor)
        {
            {$_ -gt 1}
            {
                ForEach ($I in 0..($This.Box.Factor-1))
                {
                    $Hash.Add($I,$This.Book.Output[$X].Content)
                }
            }
            {$_ -eq 1}
            {
                $Hash.Add(0,$This.Book.Output[$X].Content)
            }
        }

        $Content   = $Hash[0..($Hash.Count-1)]
        [System.IO.File]::WriteAllLines($Item.Path,$Content)

        $This.AddFile($Item.Path)
    }
```

`$Depth` is meant primarily for padding the file names with leading `0`'s because the way that the filesystem works, if you name files `0..100` then what'll happen is that they won't remain in order if you use something like:

```
-----------------------
| Get-ChildItem $Path |
-----------------------
```

Cause they'll show up like this...

```
    0.txt
    1.txt
    10.txt
    11.txt
    12.txt
    2.txt
    3.txt
    4.txt
    5.txt
    6.txt
    7.txt
    8.txt
    9.txt
```

Now, I can tell you exactly why this happens. The double digit numbers have a higher precedence.
This will continue up the stack depending on the number of characters. So if you have 65535 entries, that's 5 characters right there, so you want to use a strategy like this...

```
    $Max   = 65535
    $Depth = ([String]$Max).Length
    0..2+65532..65535 | % { "{0:d$Depth}.txt" -f $_ }
```

```
    PS Prompt:\> $Max   = 65535
    PS Prompt:\> $Depth = ([String]$Max).Length
    PS Prompt:\> 0..2+65532..65535 | % { "{0:d$Depth}.txt" -f $_ }
    00000.txt
    00001.txt
    00002.txt
    65532.txt
    65533.txt
    65534.txt
    65535.txt
    PS Prompt:\>
```

Now, putting the $Depth outside of the loop causes the loop to run slightly faster because it's not recalculating the variable or anything like that. A lot of scripts out there in the wild will spam the same variable over and over again, and the script user or author won't particularly notice it if it's just one little variable here or there... However, these things can add up and severely impact the functionality of a script or function.

The rest of the lines in the script up above will generate a [Hashtable] and add all of the content for each page, the factor is literally just (duplicating/multiplying) the information. Once each page is complete, it will access the [System.IO.File] class, and write all of that content to a file.

```
    AddFile([String]$Path)
    {
        $ID = $Path | Split-Path -Leaf
        [System.IO.Compression.ZipFileExtensions]::CreateEntryFromFile(
        $This.Archive,
        $Path,
        $ID,
        [System.IO.Compression.CompressionLevel]::Fastest
        ) | Out-Null
        [System.IO.File]::Delete($Path)
    }
```

The method AddFile() will insert the newly created file into the zip file, which was created in the main method.

I did not fully explain those methods, because the name of those methods that deal with the (*.zip) file are named Create(), Open(), and Close(). Pretty easy to understand what those methods are doing by the NAME of the method, right...? Same goes for AddFile(), as it is merely adding the file to the zip file, and then the file is deleted.

```
                                                                                                          _________/
\________________________________________________________________________________________________________/ MockZip
  Conclusion /¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯\
/¯¯¯¯¯¯¯¯¯¯¯¯
```

Now, that's effectively the end of this lesson plan. Questions or comments...? You know where to find me.

```
                                                                                                       ___________/
\_____________________________________________________________________________________________________/ Conclusion
```

```
 ________________________________________________________
|¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯|
|                                   Michael C. Cook Sr. |
|                                     Security Engineer |
|                                  Secure Digits Plus LLC |
|_______________________________________________________|
 ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯
```