Outline

In the previous exercise, we explored script formatting and general readability, but didn't make any changes in terms of functionality. In this lab, we'll add parameters to our script in order to give it some flexibility.

The 02_Split-Shakespeare.ps1 script functions exactly as intended, but it only works under very specific circumstances; one of the most obvious being that the Shakespeare.txt must exist in the location that we've hardcoded. We can make this location dynamic by adding a parameter block, which allows us to specify the location at runtime.

Exercise 1 - Create parameter block

At the top of the script, we've created a couple variables named `$FilePath` and `$OutputPath`. We'll be replacing these variables with parameters, so we can either comment the lines out by placing a hash mark (`#`) at the beginning of the line, or we can simply delete them.

Let's insert a Parameter Block after `line 1` of our script. The simplest way to create a parameter block is below.

```
param
(
    $Parameter1,
    $Parameter2
)
```

In our specific use case, we want to give our parameters the names `FilePath` and `OutputPath`, so our parameter block will look like this:

```
param
(
    $FilePath,
    $OutputPath
)
```

Insert the parameter block at the top of the script and remove the lines declaring `$Filepath` and `$OutputPath`, then save it. Now every time we run the script, we can specify both the filepath to import the text file from, and the directory we want to output to. Try copying Shakespeare.txt to a new location on your workstation, and supply the script with that location, along with a new output path. (Note: If you move the file, don't forget to move it back; you might break other exercises!)

We can do so by navigating to the folder where the script resides in a PowerShell session and invoke it with the following Syntax:

```
.\02_Split-Shakespeare.ps1 -FilePath C:\Path\To\Shakespeare.txt -OutputPath
C:\MyOutputPath
```

This approach allows us to supply any kind of information you want to keep dynamic inside a script at runtime, but we haven't added any kind of validation, and this can lead to some situations that will cause the script to fail.

Exercise 3 - Add parameter validation

Try running the script, but this time don't specify any values.

```
.\02_Split-Shakespeare.ps1
```

The script will still attempt to process until it hits a terminating error. In many cases, a script will run halfway through before terminating, often with disastrous results. There are a few approaches we can take to solve this problem, which we'll discuss in turn.

ErrorActionPreference

You can add a line into your script to treat any and all errors as terminating errors, which means the script will cease execution on any error. We can control this behavior by setting the `$ErrorActionPreference` variable.

```
$ErrorActionPreference = 'Stop'
```

This can work in very simple scripts, but in many cases could still lead to a large portion of the script running before an error is encountered, and it's generally not recommended to use this approach for variable validation.

This approach is generally more useful for creating predictable behavior in `try/catch` blocks, but that falls outside the scope of this exercise. You can read more about PowerShell preference variables here.

Parameter Attributes

Mandatory Attribute

One way to ensure we're always supplying each parameter with a value is to add the `Mandatory` attribute to each parameter. This will ensure that if no value has been specified, the script will prompt for input before proceeding. Both of the following examples are valid, and the syntax is a matter of personal preference.

```
```

```
param
(
    [Parameter(Mandatory)]
    $FilePath,

    [Parameter(Mandatory)]
    $OutputPath
)
```

```
param
(
    [Parameter(Mandatory = $true)]
    $FilePath,

    [Parameter(Mandatory = $true)]
    $OutputPath
)
```

You do not have to include the blank line between the parameters, or place the variable itself on a different line than the attributes; PowerShell is very forgiving in terms of formatting and the format of our example was chosen for ease of readability.

In the examples above, there is a 1:1 relationship between each attribute and it's corresponding variable, meaning you can control the behavior of each parameter independently.

Try replacing the parameter block in our script with one of the two above and rerun it. Now our script will prompt for input on every execution.

ValidateNotNullOrEmpty Attribute

Adding a mandatory attribute to our parameters is useful in making sure we always supply information to our script, but there's nothing preventing us from simply leaving the input blank. If we do so, the script will again attempt to process until it hits a terminating error. We can circumvent this by adding another attribute to our parameters called `ValidateNotNullOrEmpty`. This will ensure that we always supply a value to each parameter.

```
param
(
    [Parameter(Mandatory = $true)]
    [ValidateNotNullOrEmpty()]
    $FilePath,

    [Parameter(Mandatory = $true)]
    [ValidateNotNullOrEmpty()]
```

```
    $OutputPath
)
```

Try replacing the parameter block in our script with the one above and rerun it. This time if you leave the values blank, the script will terminate before attempting to run.

ValidateScript Attribute

By adding the ValidateNotNullOrEmpty attribute, we've protected against blank input, but someone might enter an invalid path. If we were to do so, the script would once again attempt to process until it hits an error. There's no built-in parameter attribute to check whether a filepath is valid, but PowerShell gives us the tools to let us create our own validation. We can do so using the `ValidateScript` attribute.

Let's again replace our parameter block with the example below.

```
param
(
    [Parameter(Mandatory = $true)]
    [ValidateNotNullOrEmpty()]
    [ValidateScript({Test-Path $PSItem})]
    $FilePath,

    [Parameter(Mandatory = $true)]
    [ValidateNotNullOrEmpty()]
    $OutputPath
)
```

In this instance, we've added a simple script as an argument inside the parenthesis of our `ValidateScript` attribute to test whether the value of `$Filepath` that we supply is valid. Because the arguments in our `ValidateScript` attribute are actually a code block, we're also required to encase them in a set of braces `{}`.

You can substitute any code you would like to use to validate your parameters based on your own needs. In our example, we could also use `Get-Item` instead of `Test-Path`; any condition that would cause the code block's execution to fail or to return `$false` would cause the script to exit, and both methods are equally valid.

Exercise 3 - Add parameter defaults

We've added flexibility to our scripts by adding parameters, but by making them mandatory we've forced them to be run interactively every single time they're invoked. This can be handy for sensitive operations, but what if we wanted to make our script run without needing input every time? Easy: we can add defaults by appending an equal sign and whatever value we want to assign to each parameter.

```
param
(
    [Parameter(Mandatory = $true)]
    [ValidateNotNullOrEmpty()]
    [ValidateScript({Test-Path $PSItem})]
    $FilePath = "$PSScriptRoot\..\Source\Shakespeare.txt",

    [Parameter(Mandatory = $true)]
    [ValidateNotNullOrEmpty()]
    $OutputPath = "$PSScriptRoot\Books"
)
```

This parameter block will *technically* assign a default value to each parameter, but if you inserted it into our script and ran it you'd still be prompted for input. This is because we've still marked each parameter as `Mandatory`, meaning the value must be supplied at runtime. Let's remove the `Mandatory` arguments from each parameter (or change each `$true` to `$false`) and run our script, which should now complete successfully.

```
param
(
    [Parameter()]
    [ValidateNotNullOrEmpty()]
    [ValidateScript({Test-Path $PSItem})]
    $FilePath = "$PSScriptRoot\Source\Shakespeare.txt",

    [Parameter()]
    [ValidateNotNullOrEmpty()]
    $OutputPath = "$PSScriptRoot\Books"
)
```

Our script is set to automatically validate that `Shakespeare.txt` exists inside inside the `Source` folder of this exercise, but we still have the ability to supply our own paths for both input and output. Rename the `Source` folder for this exercise to `Source2` and run the script. It won't prompt for input; it'll simply fail. This is because we've already assigned a value as a default; the path just doesn't exist anymore. Let's run our script, but this time we'll manually supply the values for the `FilePath` and `OutputPath` parameters.

```
# You may have to supply your own paths in the command below, depending on where the
current working directory for your PowerShell session.
. '.\02_Split-Shakespeare.ps1' -FilePath ".\Source2\Shakespeare.txt" -OutputPath
".\Books2"
```

Now we've supplied our script new input and output paths for this run, but our defaults remain in place.

The examples in this exercise are far from an exhaustive list of what's possible; you can read more about parameters and their attributes here and here.