Outline

In this lab, we'll explore what a function is, how to use them, why you should (or shouldn't!), and how to import them.

Exercise 1 - Create a Function

What Is a Function?

A function at it's core is essentially a command, but it differs from a PowerShell cmdlet in many ways, the largest being that we're using PowerShell to write PowerShell, instead of using a pre-compiled binary language like .NET or C#. For the sake of simplicity, we won't discuss the finer differences in this exercise, but you can read the official documentation here and here.

Let's start by writing a `Simple Function`. All we need to do is use the `Function` keyword, provide a name, and then place any code we want to be part of the function inside braces `{}`. The function below returns "Hello World!" any time you run it. Try pasting the code into a PowerShell window and then type `Write-Greeting` to confirm.

```
Function Write-Greeting
{
    Return "Hello World!"
}
```

You can name a function almost anything you want, but Microsoft does have a few recommendations. These are not requirements, but they are considered industry best practices, and are highly encouraged:

- Functions should use a Verb-Noun syntax.
    - Verbs should be singular and imperative; they are commands, after all.
    - Microsoft has a list of officially approved verbs; you can find these by using the `Get-Verb` cmdlet or by visiting Microsoft's website.
- Nouns don't have to be (and often aren't) singular, or even a single word.
    - There's no official naming standard for nouns; you're limited only by your imagination.
- Use Pascal Case. This means that the first letter of each word should be capitalized, even if there are no spaces in between.
    - Example: `invoke-mysuperawesomefunction` would be `Invoke-MySuperAwesomeFunction` in Pascal Case.

Why Use Functions?

Functions should be common, repeatable segments of code that perform a specific function (hence the name), ideally in a wide variety of scenarios. They can exist as standalone objects, as part of larger scripts, or even inside of other functions! The idea is that we want to write as little PowerShell as possible, and functions allow us to perform a potentially large number of operations with a single line.

An added benefit to using this approach is that it can make it easier to read and troubleshoot our scripts when things don't go smoothly. Functions don't have to be entirely about reducing line count; they can also be a matter of readability. Generally a function won't contain a single line of PowerShell, but you might choose to create one because the syntax of a particular line is difficult to read, and is used multiple times in a script. This is a perfectly valid approach.

For example, this could be difficult to read for some:

```
# Find book title
$BookTitle = $Book.TrimStart("`n").Split("`n")[0]
```

We can make this into a function, and call it `Find-BookTitle`.

```
Function Find-BookTitle($Book)
{
    Return $Book.TrimStart("`n").Split("`n")[0]
}
```

Now we can simplify our difficult-to-read line by calling our function. This is extremely easy to read, and you probably wouldn't need to leave any comments explaining what it does.

```
$BookTitle = Find-BookTitle $Book
```

You probably noticed that we didn't use a parameter block in the function above, but instead just specified the parameter name we wanted inside parenthesis; this is a short-hand way of declaring parameters. While it's generally not recommended to use this type of syntax with standalone functions that are packaged as part of a module, it's perfectly fine when creating simple functions inside a script where we have a predictable type of input. These types of functions are commonly called `Helper Functions`.

Exercise 2 - Create Helper Functions

Open 03_Split-Shakespeare.ps1, paste our `Find-BookTitle` function on line 14, and replace our difficult-to-read line on `Lines 43-44` with our simple line. You can also delete the comment immediately above.

Let's create a slightly longer helper function:

```
Function Split-Books($FilePath)
{
    # Read contents of complete file
    $Books = Get-Content -Raw $FilePath

    # Split into individual books in memory
    $Books = $Books -Split "\n\d{4}\n"

    # Discard copyright page
    $Books = $Books | Select-Object -Skip 1

    Return $Books
}
```

Paste this helper function immediately after our `Find-BookTitle` function, and then replace `Lines 40-47` with the following:

```
$Books = Split-Books $FilePath
```

Another helper function, but this time instead of doing a 1:1 replacement of code, we'll start seeing some reduction in our line count:

```
Function Initialize-OutputPath($OutputPath)
{
    IF (!(Test-Path $OutputPath))
    {
        New-Item -ItemType Directory -Path $OutputPath
    }
}
```

Insert it after the `Split-Books` function, and replace `Lines 41-46` with the following:

```
Initialize-OutputPath $OutputPath
```

We can also replace `Lines 52-57` with the following:

```
Initialize-OutputPath $OutputPath\$BookTitle
```

Maximizing Returns

So far you're probably thinking that we've increased the overall line length by 5, and the script is only marginally easier to read at best; you're right. Where helper functions shine is when they allow us to eliminate large segments of code, not one or two lines. If you look inside the `ForEach` loop inside the script, you'll see that we're separating our workflow into an `If`, `ElseIf`, and `Else` logic based on whether the books found are a play, sonnet, or unknown. Inside each of those blocks we have almost identical commands, which would make excellent candidates for some additional helper functions.

Open 03_Split-Shakespeare-HelperFunctions.ps1. In this version of the script, we've created some new helper functions called `Find-BookType` and `Out-Book`, and replaced almost the entirety of our ForEach loop, which reduced the line count from 167 to 124. Even in a script as small as this, we were able to reduce our footprint by 25%. This is because we're no longer repeating the same code three separate times in our loop. We were also able to remove a large amount of the comments because our script logic is now easy to follow.