Outline

In this lab, we'll discuss functions inside scripts, modules, and scopes.

Exercise 1 - Convert a Script to a Function

Converting our entire script into a function is easy: all we have to do is paste the code into a function definition. As we learned in a previous exercise, the syntax is as follows:

```
Function Convert-Shakespeare
{
  <# YOUR CODE GOES HERE #>
}
```

Convert the code in 04_Split-Shakespeare.ps1 into a function named `Convert-Shakespeare`, then try running the script.

Nothing happened. Why?

Declaring a function only gives the definition of what the function should do; it doesn't actually execute the code in the same way that a script does. In order to execute the code contained within, we'd have to call the function, along with any required parameters. Try calling `Split-Shakespeare` from the same PowerShell window where you ran the script.

If you ran it from a vanilla PowerShell session, you probably got an error stating that the function wasn't recognized. This is because of a principle called `Scope`: the function only exists inside the scope of the script. Once the script terminates, the memory of anything inside it is disposed of. We can work around this easily with a technique called `Dot Sourcing`. By prefacing the name of our script with a period `.` character, we can import the function from our script into our PowerShell session. If you ran the script from inside VSCode, the program automatically dot-sourced the script for you.

Try running the script with dot sourcing.

```
# You may have to modify the path below to reflect your local directory
. .\04_Split-Shakespeare.ps1
```

Now we can call `Split-Shakespeare` and it should operate as intended. While this technique can be useful in certain scenarios (like troubleshooting), it comes with it's own set of challenges: the entire contents of the script are imported from the `Script Scope` into your `Local Scope`. This means any variables or code outside of a function block will execute.

Let's add a couple lines to the top of the script to see how that works.

```
Write-Output $Fruit
$Fruit = "Oranges"
```

Now, before we run the script, let's create a variable called `Fruit`, and give it a value of `Apples`, then invoke the variable to see it's value.

```
$Fruit = "Apples"
$Fruit
```

Let's dot source our script again.

```
# You may have to modify the path below to reflect your local directory
. .\04_Split-Shakespeare.ps1
```

It'll output a value of `Apples`, but if we again call `$Fruit` from our PowerShell session, we can see the variable has been overwritten with a value of `Oranges`.

When we run the script normally, we'd have a case of Schrodinger's fruit: it exists both inside and outside of the script. Before we ran it, the value outside the script is `Apples`, while we're running the script the value inside the script is `Oranges`, but as soon as the script terminates, we retain our original value of `Apples`. With dot-sourcing, we rewrite our outside value with the value from inside the script.

Because of this behavior, dot-sourcing isn't a recommended approach for importing functions. Instead, it's recommended we use `PowerShell Modules`.

`Local` and `Script` are far from the only scopes that exist in PowerShell. You read more about scopes and dot sourcing here and here.

Exercise 2 - Convert a Script to a Module

Converting a script file into a module is trivial: all we have to do is change the extension from `.ps1` to `.psm1`. Any code inside the file will still execute, but we no longer have to worry about accidentally overwriting variables or dot sourcing. Rename 04_Split-Shakespeare.ps1 to `04_Split-Shakespeare.psm1`.

Now that we've created a module file, let's reassign our `$Fruit` variable value to `Apples`.

```
$Fruit = "Apples"
```

Now we can use the `Import-Module` cmdlet to bring the function into our session.

```
# You may have to modify the path below to reflect your local directory
Import-Module .\04_Split-Shakespeare.psm1
```

Even though our `$Fruit` variable has a value of `$Oranges` inside the execution of the module file, we'll see from our PowerShell session it's retained it's original value of `Apples`. You probably also noticed that this time we didn't get any feedback to the console; this is because the default output stream is suppressed during module import. So let's change the first few lines to get some output.

Remove the first two lines from the script and replace them with the code below.

```
Write-Warning $Fruit
$Fruit = "Oranges"
Write-Warning $Fruit
```

Save the module and try importing it again. We've still received no feedback, and it doesn't look like any of our changes have taken effect; this is because we've already loaded the module. By default, PowerShell won't try to reload any module it's already imported. We can force it to by using the `-Force` parameter on our `Import-Module` call.

```
# You may have to modify the path below to reflect your local directory
Import-Module .\04_Split-Shakespeare.psm1 -Force
```

Now we should see two warnings: one from before we assigned our `$Fruit` variable in the module, and one from after we assigned it, each with separate values. If we call our `$Fruit` variable after the import, we can see again that we still have our original value.

It's important to remember that any time you make a change in a module, you'll have to reimport it with the `-Force` flag to reflect the changes, unless you wish to restart your PowerShell session.

You can read more about modules here.