Lecture 9: Functions and Active Directory

Functions are the core of PowerShell. All of the cmdlets, we have been using are basically built-in functions. In writing scripts, we have tried to divide the script into logical steps using comments. This process is the foundation for writing functions which are simply blocks of Windows PowerShell statements that are assigned a unique name. Many of the scripts we have written can be easily converted to functions. Functions help to break a script into smaller logical units and are the key to modular scripting. The latter helps to develop a script library for the reuse of code. Functions have to be loaded into the current session's memory before they can be run. Typing a functions name will run the statements as if you had typed them at the command prompt. PowerShell functions are very efficient, but to handle large collections in the pipeline PowerShell uses a special function called a filter. A filter uses the streaming variable $_ to efficiently use memory when working with large collections. There is also a special function parameter called "switch" which allows for Boolean testing.

In writing functions there are two rules that should be followed:

- Keep the verb-noun naming convention
- Keep functions to a single purpose.

Functions can be named whatever you like, but best practice is to use the same verb-noun syntax as used for cmdlets. Consistency of names is one of PowerShell's main benefits. Thus, keeping the same naming conventions will help others to quickly determine the logic and intent of the function. For example, if you were to create a function that would retrieve the contents of a file, you could call the function "Read-File". However, PowerShell does not have a verb called read which displays file contents.. The best choice then would be to call your function "Get-File" which is what PowerShell uses to retrieve the contents of a file. A listing of all verbs can be found using the Get-Verb cmdlets.

To make the reusability of code easier, functions should be kept to a single purpose. Keeping functions to a single purpose helps the reusability of code because each function is a single building block which can be combined to provide the functionality you desire. It also keeps the parameters to three which makes the code easier to read. As a general rule of thumb, if you have more than three parameters, you may not have broken the function down to a single purpose. The main reason for using single purpose functions is to organize scripts and build "modular" code which will help to solve problems faster and manage complex scripts.

In this way, you can combine individual scripts and functions to solve specific problems. For example, when you press the TV remote, you are not concerned with how the infra-red signal travels to the TV and initiates the ON circuitry of the TV box. You only want the TV to turn on. Functions are like that. They have inputs and outputs and work in a simple way to solve problems with the user knowing the details of how the function works.

A Function Syntax

Functions do not need to be complex in order to be useful. In fact, the most useful scripts are simple functions. The latter are functions that do not have parameters. A menu can be created which can combine several simple functions into a helpful and powerful program. The function syntax is very simple.

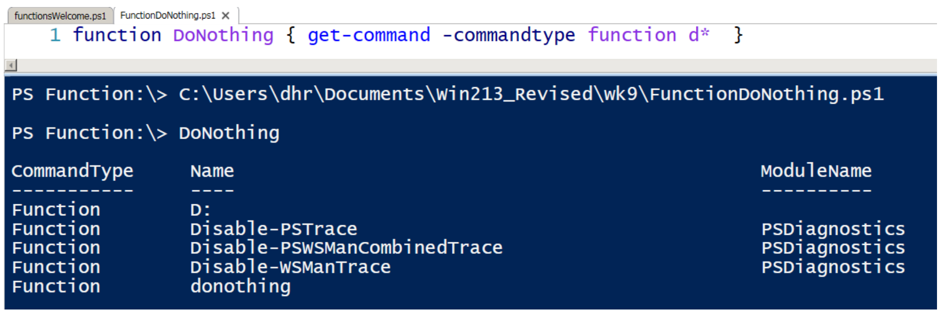
Function Syntax:

Function *FunctionName* **(parameters-optional)**
{
……….
Code block to be executed
…………
} #end of FunctionName

The function must be defined with the keyword **Function,** followed by any parameters to be passed in parenthesis. Parameters are like variables which we have used many times. The commands to be executed are contained in a script block. It is common practice to place a comment at the end of the function code block with the function name to improve readability and code copying. Not all functions will have parameters and if there are no parameters the parenthesis's can be omitted and the function is called a simple function.

Type: **Function DoNothing { get-command –commandtype function d* }**

To execute the function in the ISE, click on the Run icon, notice that unlike a script the commands did not run. This is because, unlike a script, a function must be loaded into memory first, before it can be executed. To execute the function we type its name, which then runs the command(s) inside the script block. Functions are stored on the function PSDrive. We can see that typing the function's name retrieves all functions beginning with the letter "d"). Our function is not very helpful, but it met the minimum syntax, and so was created on the function drive.



```
function DoNothing { get-command -commandtype function d*   }
```

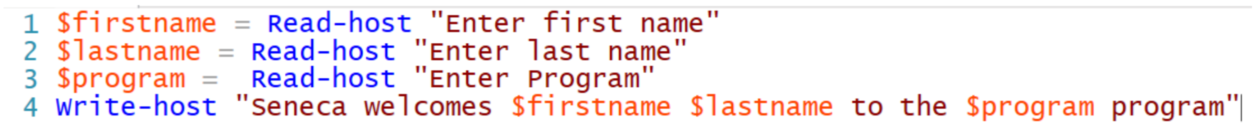
```
PS Function:\> C:\Users\dhr\Documents\Win213_Revised\wk9\FunctionDoNothing.ps1

PS Function:\> DoNothing

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        D:
Function        Disable-PSTrace                                    PSDiagnostics
Function        Disable-PSWSManCombinedTrace                       PSDiagnostics
Function        Disable-WSManTrace                                 PSDiagnostics
Function        donothing
```

Functions are nothing more than lines of code, like scripts we have written before, which have been

Figure 1: Simple Function to Display Basic Syntax

given a unique name. The lines of code will be run by calling the name of the function. One line of code, will run multiple lines of code. Functions are also the ideal way to provide code to other users. When you write a function, you design it so the user doesn't need to understand how it works. You define it with inputs and outputs which get the results the user expected. Functions are the BEST way to package your code.

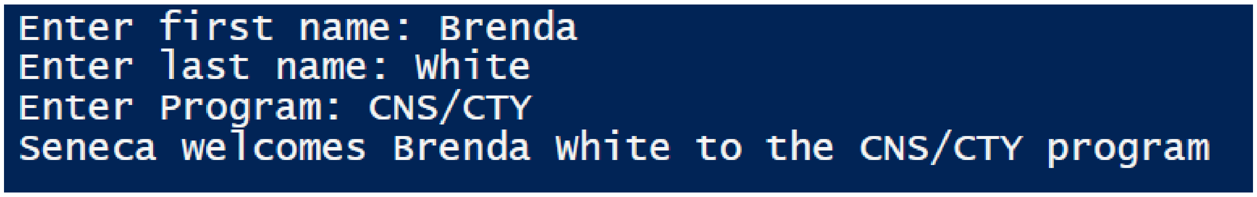
For example, the first script we wrote in this class is the following 4 lines of code, which got user input about your name, and program and displayed a welcome message. Let's review the script and convert it to a function to see the differences.



```
$firstname = Read-host "Enter first name"
$lastname = Read-host "Enter last name"
$program =  Read-host "Enter Program"
Write-host "Seneca welcomes $firstname $lastname to the $program program"
```

Figure 2: Code for Welcome Script

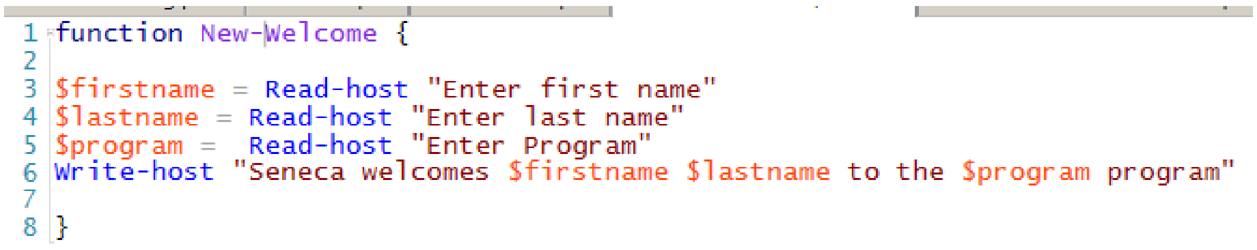
Run the code and make sure it is working correctly. You should have output like Figure 3 below:



```
Enter first name: Brenda
Enter last name: White
Enter Program: CNS/CTY
Seneca welcomes Brenda White to the CNS/CTY program
```

Figure 3: Output from Welcome Script

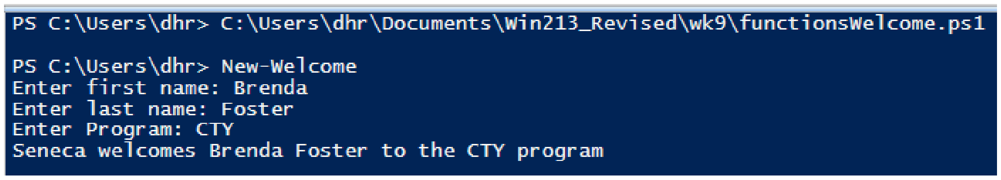
Now we will convert the script to a function. Add the function keyword and name the function New-Welcome. Place the entire script inside the script block.



```
function New-Welcome {

$firstname = Read-host "Enter first name"
$lastname = Read-host "Enter last name"
$program =  Read-host "Enter Program"
Write-host "Seneca welcomes $firstname $lastname to the $program program"

}
```

Figure 4: Converting Welcome Script to a Function New-Welcome

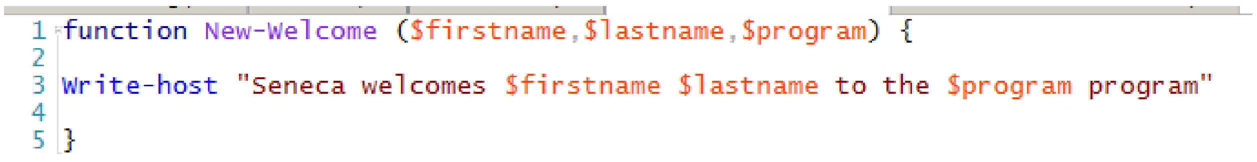
Clicking on the run icon loads the function into memory. Typing the name of the function then executes the program.



```
PS C:\Users\dhr> C:\Users\dhr\Documents\Win213_Revised\wk9\functionsWelcome.ps1

PS C:\Users\dhr> New-Welcome
Enter first name: Brenda
Enter last name: Foster
Enter Program: CTY
Seneca welcomes Brenda Foster to the CTY program
```

Figure 5: Output of the New-Welcome Function

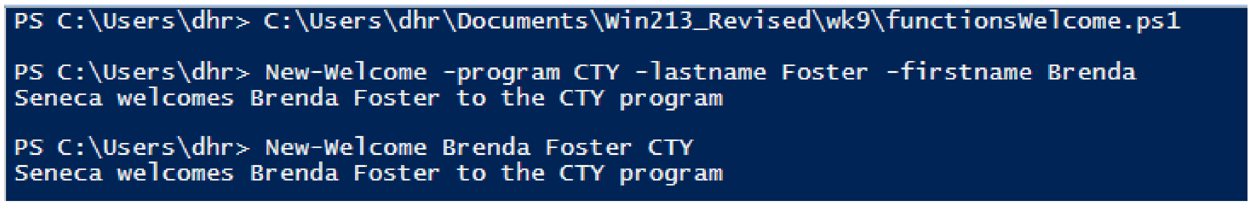
This function is called a simple function because it has no parameters. Parameters are like variables that are used inside the function. Since we need 3 pieces of information from every user, let's add parameters to the function. We place $firstname,$lastname and $program which are enclosed in parenthesis, outside of the script block. Now we can eliminate the Read-host lines because the user will provide the information on the command line. You function should look like below:



```
function New-Welcome ($firstname,$lastname,$program) {

Write-host "Seneca welcomes $firstname $lastname to the $program program"

}
```

Figure 6: Passing parameters to the function

Notice on the first running of the script, the named parameters can be given in any order. But if you don't use the named parameters, it is important to give the information in the correct order, otherwise the program will not display the message correctly.



```
PS C:\Users\dhr> C:\Users\dhr\Documents\Win213_Revised\wk9\functionsWelcome.ps1

PS C:\Users\dhr> New-Welcome -program CTY -lastname Foster -firstname Brenda
Seneca welcomes Brenda Foster to the CTY program

PS C:\Users\dhr> New-Welcome Brenda Foster CTY
Seneca welcomes Brenda Foster to the CTY program
```

Figure 7: Passing Parameters to the New-Welcome function

This function is OK if only you were going to use it. If another user is going to use it, we need to add a help message and provide some error checking. We want to ensure that the user provided all parameters, and if not, prompt to get the correct information.

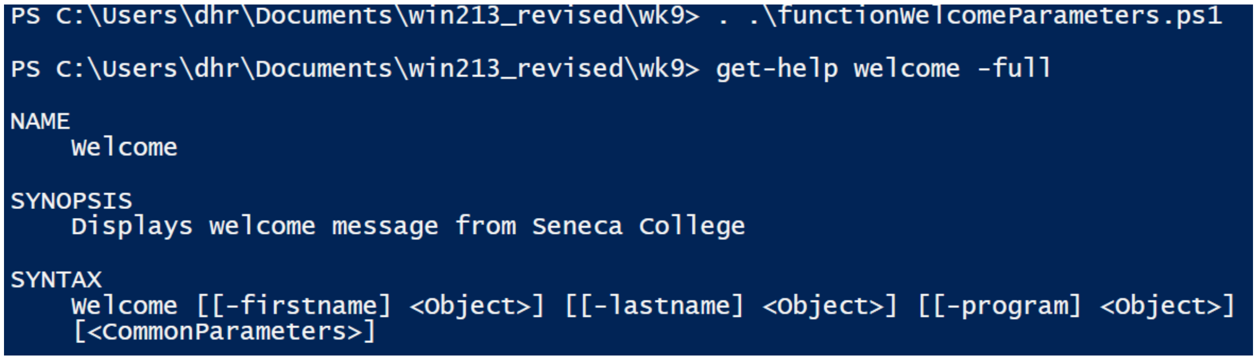
1 Adding a Help File: New-Welcome

PowerShell provides comment based help which is written into the function or script. PowerShell also accepts XML help files which can be updated with update-help, but we will not be using them in this course. The best place for comment based help is at the beginning of the ps1 file. The help file is enclosed inside a comment block and each keyword begins with a period.

```
<#
.SYNOPSIS
   Displays welcome message
.DESCRIPTION
   The welcome function takes 3 parameters firstname,lastname and program
   and displays a welcome message. If the user does not provide a parameter,
   the function will prompt for the missing parameter. The parameters are passed
   in the following order, position 1, firstname, position 2, last name, position
   3, program.
.EXAMPLE
   PS > welcome FirstName LastName CNS/CTY
.EXAMPLE
   PS > welcome Firstname
   Enter last name  <Lastname>
   Enter program  <program>
.NOTES
    Author - YourNameHere
    DateLastModified- Today's date
#>
```

Figure 8: Adding Comment Based Help

To view the help file we need to load the ps1 function into memory and then use the get-help command with all of the parameters, full, detailed, and example. To load the function into memory on the command line, we "dot-source" the function (notice the dot at the beginning of the command, followed by a space and then the path to the file). Dot-sourcing tells PowerShell to read all of the function code into memory and copy any variables to the Global scope so they will be available to any other function during the session.



```
PS C:\Users\dhr\Documents\win213_revised\wk9> . .\functionWelcomeParameters.ps1

PS C:\Users\dhr\Documents\win213_revised\wk9> get-help welcome -full

NAME
    Welcome

SYNOPSIS
    Displays welcome message from Seneca College

SYNTAX
    Welcome [[-firstname] <Object>] [[-lastname] <Object>] [[-program] <Object>]
    [<CommonParameters>]
```

Figure 9: Dot sourcing the function and partial listing of the help file.

Notice that PowerShell help also adds information about the function such as Syntax which was not in the help file we created. With PowerShell, help and user defined help files, the user has all the information to run the function correctly: how the function works, the 3 parameters required and the order in which they must be entered.

2 Error Checking: New-Welcome

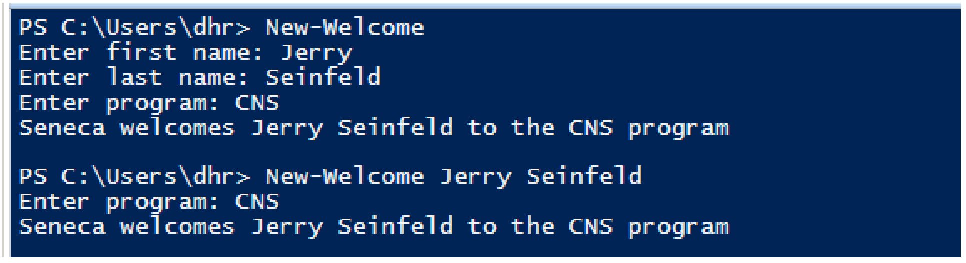
What if the user forgets and only gives 2 parameters instead of 3? Or, what if the user gives no parameters at all on the command line? We need error checking to ensure that the correct number of parameters are entered, and if not, prompt the user for the correct information. We do this using an if statement combined with the static method IsNullOrEmpty. (Next week we will learn another way to ensure the parameters are entered by making parameters mandatory)

```
#checking if a parameter is missing and prompting the user for the missing info.
if ([string]::IsNullOrEmpty($firstname)){$firstname = Read-host "Enter first name"}
if ([string]::IsNullOrEmpty($lastname)){$lastname = Read-host "Enter last name"}
if ([string]::IsNullOrEmpty($program)){$program = Read-host "Enter program"}
```

Figure 10: Checking if the correct number of Parameters are Passed

Using the .NET method IsNullOrEmpty inside an IF statement does the job. If the parameter is passed and the IF condition is FALSE, then the IF script block is ignored. On the other hand, if a parameter is missing, and the IF statement is TRUE, then the IF script block is run and the user is prompted to enter the missing information. You need to test for each specific parameter.

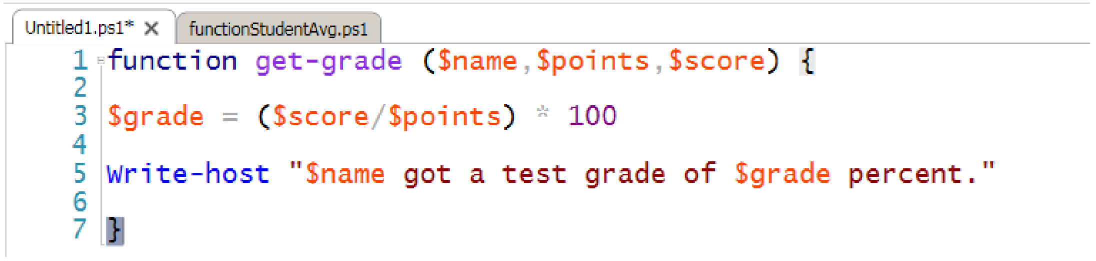
Run your code and try different scenarios.



```
PS C:\Users\dhr> New-Welcome
Enter first name: Jerry
Enter last name: Seinfeld
Enter program: CNS
Seneca welcomes Jerry Seinfeld to the CNS program

PS C:\Users\dhr> New-Welcome Jerry Seinfeld
Enter program: CNS
Seneca welcomes Jerry Seinfeld to the CNS program
```

Figure 11: Error Testing Welcome program

Now we will write a second function to calculate student grades. Open a new tab in the ISE and create a new function called get-grade. This function will pass 3 parameters, student name, the second is the total points available for the test, and the third parameter is the student score on the test.



Untitled1.ps1* functionStudentAvg.ps1

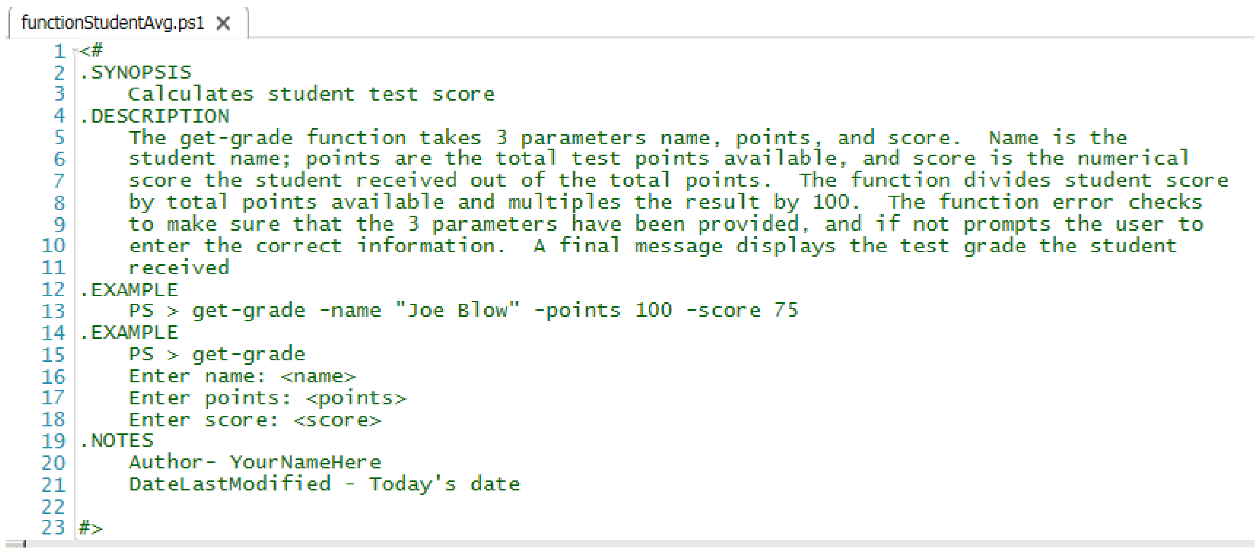
```
function get-grade ($name,$points,$score) {

$grade = ($score/$points) * 100

Write-host "$name got a test grade of $grade percent."

}
```

Figure 12: Source Code to find student grade

In the script block, grade is equal to the student score divided by the total points available and the result multiplied by 100. The function displays a message giving the test grade.

3 Adding a Help File: Get-Grade

Now that the basic code is written, we need to add the help file and error checking.



functionStudentAvg.ps1

```
<#
.SYNOPSIS
    Calculates student test score
.DESCRIPTION
    The get-grade function takes 3 parameters name, points, and score.  Name is the
    student name; points are the total test points available, and score is the numerical
    score the student received out of the total points.  The function divides student score
    by total points available and multiples the result by 100.  The function error checks
    to make sure that the 3 parameters have been provided, and if not prompts the user to
    enter the correct information.  A final message displays the test grade the student
    received
.EXAMPLE
    PS > get-grade -name "Joe Blow" -points 100 -score 75
.EXAMPLE
    PS > get-grade
    Enter name: <name>
    Enter points: <points>
    Enter score: <score>
.NOTES
    Author- YourNameHere
    DateLastModified - Today's date

#>
```

Figure 13: Help File FunctionStudentAvg.ps1

4 Error Checking: Get-Grade

Error Checking is required to test if the user entered a value on the command line. If the user didn't enter the name, we want to prompt to get the information. A better technique to use than in the first script, is to use the while condition statement.

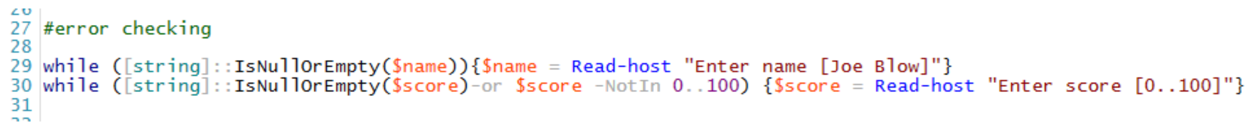
Beginning in PowerShell 3.0 a range operator "-In" and "–NotIn" were added to test in a number is in a specific range. For example,

Type: $a= 7
type: $a –In 6..10
Type: $a –NotIn 1..5

Notice the value is true; 7 is in the range of 6 to 10, and 7 is not in the range 1 to 5. These range operators will check if score is a value between 1 and 100.

A better approach than using the if statement to test a value, is to use the while statement. Since the while statement is a loop, the read-host statement will continually prompt until the correct information is given and the while condition becomes TRUE.

Also, parameters can have default values; since all tests are out of 100, we can set the value of $points to 100. This also is a form of error checking, if the user does not provide a points value, the default value will be used and the script will run correctly. However, if the user does provide a value, then the value on the command line will be used instead.



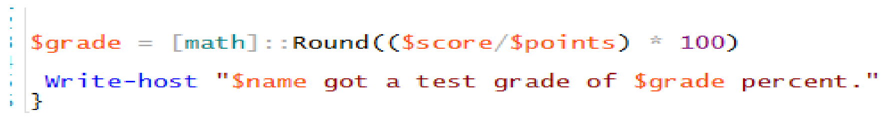
```
#error checking

while ([string]::IsNullOrEmpty($name)){$name = Read-host "Enter name [Joe Blow]"}
while ([string]::IsNullOrEmpty($score)-or $score -NotIn 0..100) {$score = Read-host "Enter score [0..100]"}

```

Figure 14: Error Checking in FunctionStudentAvg.ps1

One last change, to improve program output, is to round the variable grade so that it will always be an integer. We do this using the built in .NET Round method which is part of the math class.

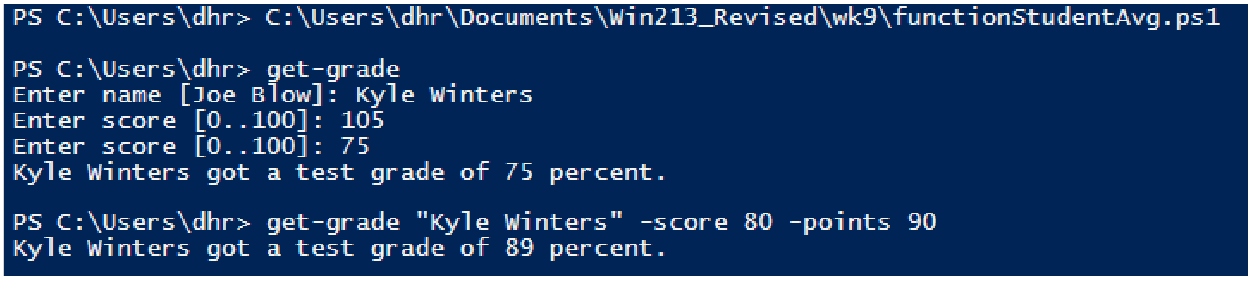
Type: [math]::Round(8.76)
Type: [math]::Round(8.1)

The round method assesses the decimal component and if over ".5" rounds to the next higher value. Make the following change to your code.



```
$grade = [math]::Round(($score/$points) * 100)
 Write-host "$name got a test grade of $grade percent."
}
```

Figure 15: Using the Round static method



```
PS C:\Users\dhr> C:\Users\dhr\Documents\Win213_Revised\wk9\functionStudentAvg.ps1

PS C:\Users\dhr> get-grade
Enter name [Joe Blow]: Kyle Winters
Enter score [0..100]: 105
Enter score [0..100]: 75
Kyle Winters got a test grade of 75 percent.

PS C:\Users\dhr> get-grade "Kyle Winters" -score 80 -points 90
Kyle Winters got a test grade of 89 percent.
```

Figure 16: Output of Get-Grade function

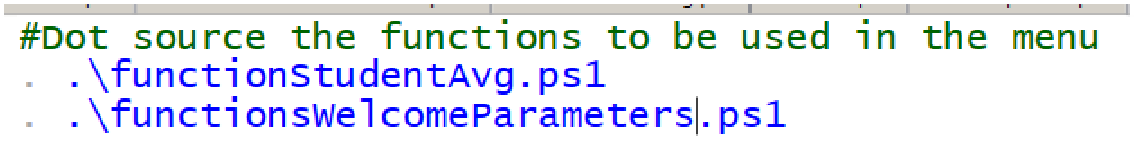
When you run the function, notice if the user does not pass any parameters to the function, the user is prompted for the correct input. An important point is when entering the student name with Read-Host, quotes are not needed, but when entering the name on the command line, quotes are needed. Otherwise "Winters" would be interpreted as parameter two, which would generate an error.

If the user does not provide the correct value for the student score, notice he/she is prompted for the correct information. In the first running of the script, the default value of 100 points was used, but in the second running of the script, the value of 90 points was provided. This value replaced the default value to create a grade of 89%. The two scripts are working perfectly; let's see how we can built a modular program with functions.

B Building a Menu

Now we will tie the two functions together with a menu. When using a menu with functions, you will need make 3 changes from how we used a menu with scripts

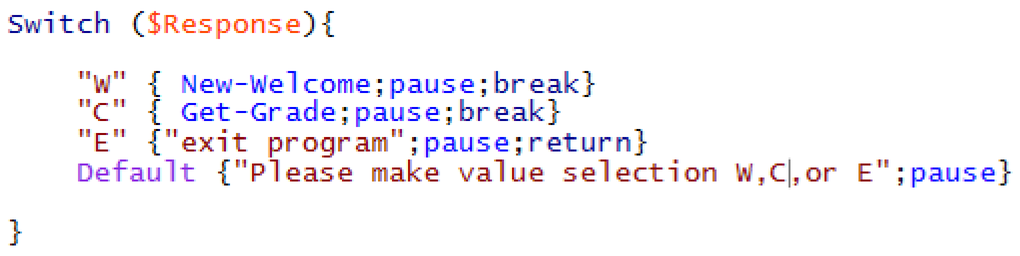
1 Dot source the functions at the top of the main program containing the menu.



```
#Dot source the functions to be used in the menu
. .\functionStudentAvg.ps1
. .\functionsWelcomeParameters.ps1
```

Figure 17: Dot Source the Functions at top of Main Program

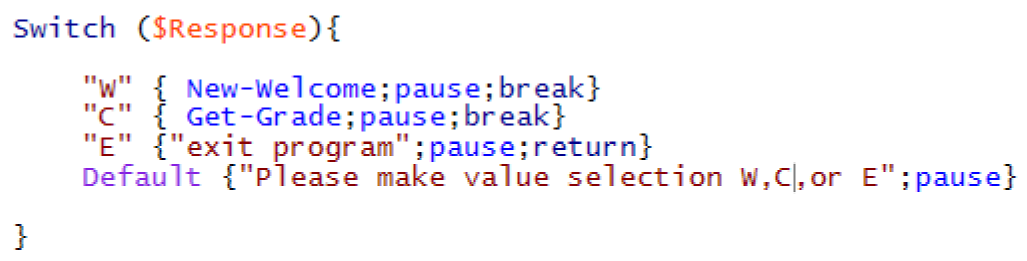
2 Call the Function name in the Switch option script block and pass any parameters



```
Switch ($Response){

    "W" { New-Welcome;pause;break}
    "C" { Get-Grade;pause;break}
    "E" {"exit program";pause;return}
    Default {"Please make value selection W,C,or E";pause}

}
```

Figure 18: Calling Function and Passing Parameters

3 Change the exit on the last option, to end the menu to return. (exit closes the PowerShell session and PowerShell console)



```
Switch ($Response){

     "W" { New-Welcome;pause;break}
     "C" { Get-Grade;pause;break}
     "E" {"exit program";pause;return}
     Default {"Please make value selection W,C,or E";pause}

}
```

Figure 39: Changing Exit to Return

C Installing Active Directory

Active Directory is a directory service that centrally stores all network resources, computers, users, shares and printers. AD is managed by one or more domain controllers that work in a "multimaster configuration", which means any changes on one domain controller will be replicated to all domain controllers. To install Active Directory, we used dcpromo.exe, however, dcpromo is now deprecated, except for unattended installations.

Installing Active Directory Domain Services (ADDS) is one of the core services that must be implemented when creating a centralized infrastructure. There are three deployment scenarios related to the Active Directory deployment:

- Creating a new Active Directory forest (Win213)
- Creating a new Active Directory Domain in an existing forest
- Creating a new Active Directory Domain Controller in an existing domain.

These deployment options are available as part of Active Directory Domain Services Configuration Wizard. Deploying Active Directory Domain Services is not a simple matter. There are prerequisites that must be met and multiple items that need to be configured before installing ADDS.

1 Infrastructure and Role Prerequisites

In order to install Active Directory on a machine, there are 4 basic prerequisites that need to be made.

1. ExecutionPolicy must be set to all scripts to run.
2. IP address must be set to a static configuration

3. Computer name must be set correctly for Domain.

4. The Administrator account must have a complex password

5. DNS must be installed and properly configured before Domain is installed

In addition to infrastructure prerequisites, there are role-based prerequisites that need to be deployed. These role-based prerequisites are shown here.

1. Active Directory module for Windows PowerShell
2. Active Directory Administrative Center tools
3. AD DS snap-ins and command-line tools

Once you have your computer renamed with a static IP address and the Active Directory Domain Services RSAT tools are installed, it is time to add the Active Directory Domain Services role, the DNS Server role, and the Group Policy management feature

After installing these features, it is time to test the system to ensure that the prerequisites are completed. This is done with a new built-in diagnostic cmdlet called Test-ADDSForestInstallation. Any errors need to be corrected before running the main cmdlet to install domain services, Install-ADDSForest. The latter is part of a module which must be first imported called ADDSDeployment.

We've Learned

1. Functions are an important structure in PowerShell and are stored on the Function PSDrive. Functions are the best tool to build a script library and begin modular scripting. Functions are simple to create using the Function keyboard. Simple functions do not have parameters. Complex functions use parameters to pass information to the function. Named parameters can be used in any order. Unnamed parameters must be used in the order the function uses them.

2. Functions which will be used by others should always have help files and error checking. The help files explain how to use the parameters and the order of the parameters in the function. And, error checking ensures that the correct information is passed to the function prior to is necessary to ensure that the correct information is passed to the function prior to execution.

3. Unlike scripts which run when the ps1 file name is entered on the command line, a function is a named block of code which must be loaded into memory first. Execution of the function is done by typing the functions name. Function names should follow the verb-noun format and the function should be limited to a single task.

4. Creating a menu with functions is very similar to using a menu with scripts. The only changes are dot sourcing the functions at the top of the main program, calling the function name in the option script block of the switch statement and changing exit to return in the last menu option.

5. Active Directory requires infrastructure and role based prerequisites in order for it to install correctly.