

<http://www.sqlservercentral.com/articles/Stairway+Series/97805/>

Printed 2014/05/06 09:55PM

Stairway to SQL PowerShell Level 5: SQL Server PowerShell Building Blocks

By [Ben Miller](#), 2013/03/19

You should be on your way with PowerShell with the previous levels and now the topic shifts to using Functions and Assemblies built in to .NET for use with automation. This topic will be used in your PowerShell adventures throughout your career and this Stairway.

First let's address how you use assemblies, as this is kind of a review from the previous levels on the loading of the SMO assemblies. The importance of loading assemblies into PowerShell is the fact that there are many functions available through .NET assemblies. Think of an ArrayList or an SntpClient object. These objects are part of the .NET Framework and are used by .NET coding, but the beauty of PowerShell is that it is built on top of Windows and .NET Framework so it can use them as well. In Table 5.1 you will see a list of objects and namespaces with the assemblies that are commonly used or that could be used. The assemblies listed in Table 5.1 are automatically loaded as part of the .NET Framework.

Object Name	Namespace	Assembly
SntpClient	System.Net.Mail	System.dll
ArrayList	System.Collections	Mscorlib.dll
DataTable	System.Data	System.Data.dll

Table 5.1 – Objects, Namespaces, Assemblies

By using the statements in Listing 5.1 as a template, you could substitute in a different assembly and that assembly would load into your PowerShell environment. Notice the different ways you can load assemblies in PowerShell. Listing 5.1 shows examples of loading assemblies outside of the .NET Framework core.

```
[System.Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo")
[System.Reflection.Assembly]::Load("Microsoft.SqlServer.Smo")
[System.Reflection.Assembly]::Load("Microsoft.SqlServer.Smo, Version=10.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91")
[System.Reflection.Assembly]::LoadFrom("c:\Sample.Assembly.dll")
[System.Reflection.Assembly]::LoadFile("c:\Sample.Assembly.dll")

Add-Type -AssemblyName "Microsoft.SqlServer.Smo"
Add-Type -AssemblyName "Microsoft.SqlServer.Smo, Version=10.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"
```

Listing 5.1 – Variations of Loading Assemblies

The main purpose of loading assemblies is to bring them into the memory space called an AppDomain for .NET and have access to the objects in that assembly and the methods and properties associated with the objects. When the assembly is loaded, you can now use New-Object to create an object of a type that is included in the assembly. SMO assemblies contain objects used in connecting and manipulating SQL Server objects. To gain access to these objects, you need to load them with one of the command variations above in Listing 5.1. If the assembly loads successfully, as you have seen in previous levels, it will indicate success by an output line showing the assembly that was loaded. If the assembly was not loaded, you will not get any output back on execution of the command. This is particularly important in the case that you use Add-Type to load the assembly without specifying the version and PublicKeyToken, because by default PowerShell will load the earliest version of the assembly that it knows about. Yes it is hard coded in the PowerShell environment. So to load a specific version in PowerShell using Add-Type, you must specify the long form as noted in Listing 5.1.

There is one other use of objects inside assemblies. There are assemblies that have objects with enumerations, options or functions that can be used without loading the assembly, these are called "static members". Consider Listing 5.2 and see that I did not load an assembly but the use of the Format function of a String object is used for formatting output with placeholders.

```
[String]::Format("{0} is a good guy.", "Ben")

Output: Ben is a good guy
```

Listing 5.2 – Use of a static function in String object

Using an assembly object in the same way as String is used in Listing 5.2 would look like this: [System.Reflection.Assembly]::Load() to load an assembly. The Assembly object inside System.Reflection namespace has a method called Load and it can be used without creating a new object and using it from there. It can be called directly. For the purpose of this Level, these are the concepts that you need to know about assemblies. You will see more about assemblies in later levels.

Functions in SQL PowerShell

Now let's take a look at functions and see how they work, how you create them and why you would want to use them. Functions are building blocks for you to create reusable pieces of PowerShell. Functions are self-contained inside either PS1 files or in a PowerShell Module. Listing 5.3 illustrates the format of a function and covers the optional blocks within a function depending on the desired use of the function.

```
Function Get-ProcessExamples {
    Param (
        $sqlserver
    )

    Begin { Write-Host "This is a pre-processor $sqlserver" }
    Process { Write-Host "This is the main body $sqlserver" }
    End { Write-Host "This is the post-processor $sqlserver" }
}
```

Listing 5.3 – Basic building blocks of a Function

Let's go over what is in the function in Listing 5.3. First you will notice that there is a name of the function, and you need to know that it follows a naming convention. This was illustrated in the Introduction level as to say that the name is Verb-Noun. In the case above you see Get-ProcessExamples and that follows this convention: Get is the

Verb and denotes that we are going to receive something back, and in this case we are seeing output from this function. Inside the function you see a Param block that allows you to parameterize your function and allow it to be more dynamic instead of static. Following the Param block, there is a Begin block. This block allows you to have a pre-processor set of statements, if you will, that run when the function first runs and then does not run again until the function is called again. After the Begin block, there is a Process block. This is the main body of the function and can run against multiple rows or objects and is the main piece of code if you do not specify the other blocks by name. The End block is simply a post-processor or one time piece of code that runs before the function ends.

When creating functions you can put multiple functions in one file and they will load in together if you use the dot sourcing method taught in an earlier level. But the main thing you should remember is that functions are reusable and in automation, reusable is golden. As a DBA or any database person, you will have things that you either interrogate or look at regularly, and if you had a function you could call to get that information, it would be much more efficient than typing in the code interactively every time you wanted to use that code.

The function in Listing 5.4 illustrates that I can pass a SQL Server name into the function and I will get back a value of the Max Memory for that server. This is only the tip of the iceberg in what you CAN do.

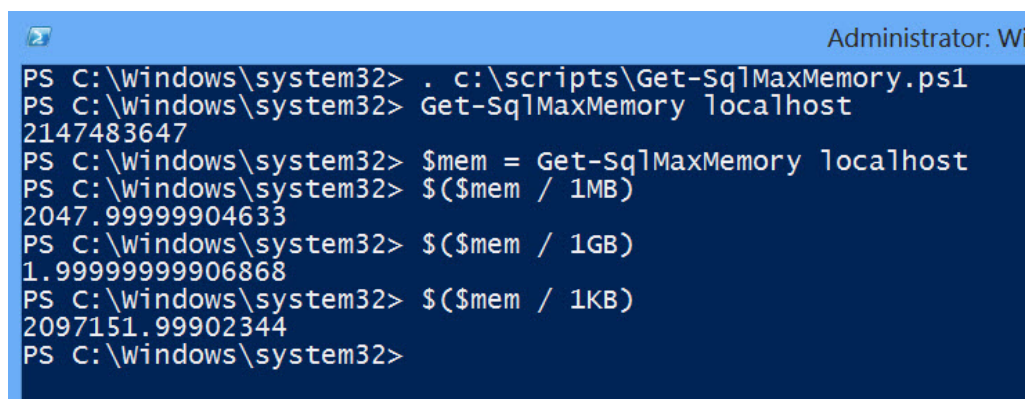
In Listing 5.3 you see the skeleton of a function, and in Listing 5.4 you see the actual function defined for getting the Max Memory setting. Figure 5.1 shows the results. Notice in Listing 5.4, I omitted the Begin, Process and End blocks. When you do that, it implies the Process block.

```
Function Get-SqlMaxMemory {
    Param ( [string]$sqlserver )

    $sqlconn = New-Object -TypeName Microsoft.SqlServer.Management.Smo.Server -ArgumentList $sqlserver

    $maxmem = $sqlconn.Configuration.MaxServerMemory.RunValue
    Write-Output $($maxmem/1MB)
}
```

Listing 5.4 – Create function Get-SqlMaxMemory



The screenshot shows a PowerShell console window with the title bar 'Administrator: Wi'. The command prompt is 'PS C:\windows\system32>'. The user enters the command '. c:\scripts\Get-SqlMaxMemory.ps1', which is followed by 'Get-SqlMaxMemory localhost'. The output is '2147483647'. The user then enters '\$mem = Get-SqlMaxMemory localhost', followed by '\$(\$mem / 1MB)', which outputs '2047.99999904633'. The user then enters '\$(\$mem / 1GB)', which outputs '1.99999999906868'. Finally, the user enters '\$(\$mem / 1KB)', which outputs '2097151.99902344'. The prompt returns to 'PS C:\windows\system32>'.

Figure 5.1 – Output of Get-SqlMaxMemory

Let's go over Listing 5.4 and Figure 5.1. In the listing, you see the function definition. Alone it really does not do anything, but the definition allows it to be loaded into memory and used. The first line in Figure 5.1 shows how it is loaded for use. This is using the "dot source" method that was discussed in previous levels. The "dot source" uses a period in the front and then the location and filename. This method loads whatever is in the file into your PowerShell session. There will not be any output unless there is an error or if inside the file, there is a Write statement. The fact that you have a Function definition inside that is valid, will add the function into the memory space of the PowerShell session, so that from then on, you can reference the function as shown in Figure 5.1. Ensure that your SQL Server is running, and then execute the function. The results of executing the function shows that the Max Memory setting has not been changed from the default setting. I could go and change this setting using sp_configure or the GUI in SSMS and execute the function again. But now I have a function that I can call to get this information. I don't have to manually run sp_configure or open SQL Server Management Studio to find out what the SQL Server Max Memory setting is, but I can reuse this function and even "dot source" it inside my profile. In the next level you will see this function inside a module and then you will put it in a profile and it will be available each time you open PowerShell.

This was a pretty short level, but the important stuff has been discussed. Expanding on this will be a task that you will want to experiment with. Try your hand at it and create a function that gets the Min Server Memory setting from SQL Server.