CHAPTER 6

Extending PowerShell

To load a single function for use in a script, we use the dot-sourcing method discussed previously. However, what if we have many related functions that we want to make available to our script? Do we need to dot source each function individually? Fortunately, the answer is no. PowerShell provides an elegant way to group a set of functions together into a module so they can be loaded in just one statement. We've already used this feature in previous examples, and now we're going to discuss how we can create our own modules. We will also cover how to merge multiple script files into a module by using a manifest. Modules can be written in PowerShell script or in .NET languages like C#. Modules are one of the great extensibility features of PowerShell, and you can find many free modules available for download. Some, such as the SQL Server Module, SQLPS, are written by Microsoft, and others by vendors or the Windows community. We'll discuss some of the popular modules available and how to use them.

The Four Types of Modules

There are four types of PowerShell modules: binary, script, dynamic, and manifest. Binary modules are modules written in a compiled .NET language like C#. Because binary modules can only be written in C# or VB .NET, we are not going to cover creating them as it is beyond the scope of this book. In PowerShell 1.0, all modules had to be written in a .NET language.

Dynamic modules are modules created within a script that are not saved. Rather, they just persist for the duration of the PowerShell session.

A manifest module is a module that includes a special file called a manifest that documents the module and supports the ability to include other modules and related files.

Script modules are the most important for our purposes because they are written in the PowerShell language. We will cover creating and using them in great detail. For more-detailed technical information about modules, see the following link: `http://msdn.microsoft.com/en-us/library/dd878324%28v=vs.85%29.aspx`.

Script Modules

To create a script module we need to save the source code of the functions in one script file, using the extension `.psm1`, and then store it in a folder where PowerShell looks for modules. Where would that be? We can find that out by looking at the environment variable `$env:psmodulepath`. Enter it on the console:

```
$env:psmodulepath.
```

On my machine, I get the following:

```
C:\Users\BryanCafferky\Documents\WindowsPowerShell\Modules;C:\Program Files\
WindowsPowerShell\Modules;C:\Windows\system32\WindowsPo
werShell\v1.0\Modules\;c:\Program Files (x86)\Microsoft SQL Server\110\Tools\PowerShell\
Modules\;C:\Users\BryanCafferky\Documents\P
owerShell\Modules\
```

The first folder, in bold, is the best place to put our custom modules. It is there for that purpose. By default PowerShell will look for modules under the current userListings Documents folder in `WindowsPowerShell\Modules\`. We don't want to mess around with anything under the Windows folder, as Microsoft maintains those. Let's assume we want to store the following script as a module:

```
# Simple module example

function Invoke-UdfAddNumber([int]$p_int1, [int]$p_int2)
{
     Return ($p_int1 + $p_int2)
}

function Invoke-UdfSubtractNumber([int]$p_int1, [int]$p_int2)
{
     Return ($p_int1 - $p_int2)
}
```

The module will be named `umd_Simple`. First, we need to create a folder named `umd_Simple` in our `\Documents\WindowsPowerShell` folder. If the folder does not have the same name as the module, PowerShell cannot find it. Then, we need to save the script in the `umd_simple` folder with the name `umd_simple.psm1`.

Let's review the requirements of creating a module:

- It consists of a script file that contains one or more functions.
- It must be stored in a folder that is listed in `$env:psmodulepath`.
- The script must be stored in its own folder with the same name as the script file, less the extension.
- The script must be saved as the module name we want using the extension `.psm1`.

All these conditions must be met in order for PowerShell to be able to locate and use the module.

Now that we have the script `umd_Simple.psm1` stored in our `Documents` folder under `WindowsPowerShell\umd_simple`, we can import the module with this statement:

```
Import-Module umd_Simple
```

With the module imported, let's try the functions as shown here:

```
Invoke-UdfAddNumber 1 4
Invoke-UdfSubtractNumber 4 1
```

We should get the following output:

```
5
3
```

It's that simple. This module is not very useful, but it demonstrates the process of creating and using modules.

■ **Note** As a general rule, avoid putting spaces in any object names which includes folders and script names. It just makes things more complicated to deal with. Letters, numbers, and underscores are fine to use.

Microsoft wanted to make it as easy as possible for users to create script modules. In doing so, they automatically default some behaviors. For example, all functions are automatically visible to the script that imports the module, but variables of the module are not visible. Generally, that works pretty well. However, suppose we want to be able to access a module variable. An object-orientated programming best practice is to never let outside code directly access an object's variables. Instead, a Set function is provided to modify the variable's value, and a Get function is provided to retrieve the variable's value. This insulates the function from the outside world; i.e., external code cannot just assign any value, since the Set function can validate the data. Also, the internal implementation of the property may need to be changed over time, say from an int to a string, but the external interface, i.e., the Get and Set functions, can be maintained so that these changes do not affect external use of the properties.

Before we get into variations of modules, I want to introduce a more robust example. This module will load a sales tax rate table and a foreign currency conversion table. It exposes functions to set the path to the sales tax file; get the path to the sales tax file; get the tax rate for a given state; and get the currency conversion rate for a given foreign currency. The function to get the exchange rate uses parameter sets, so it can return either the number of units of the foreign currency to equal one US dollar or the number of US dollars to equal one foreign currency unit.

Let's review the code below.

```
<#  Module Name:  umd_state.psm1

    Author:       Bryan Cafferky

    Purpose:      A module demostration.

#>
<#
    .SYNOPSIS
      Demonstrate the creation and use of a custom module written in PowerShell.

    .Description
          This module was created to demonstrate the use of custom PowerShell modules.

     .Notes
          Author:  Bryan Cafferky for Pro PowerShell Development.
          Version: 1.0

#>

[string]$script:salestaxfilepath = ($env:HomeDrive + $env:HOMEPATH + `
 "\Documents\StateSalesTaxRates.csv")

$inexchangerate = Import-CSV ($env:HomeDrive + $env:HOMEPATH + `
 "\Documents\currencyexchangerate.csv")
```

```
[hashtable]$script:salestax = @{}

function Set-UdfSalesTaxFilepath {
        [CmdletBinding()]
        param (
            [Parameter(Mandatory = $true)]
            [ValidateScript({ Test-Path $_ })]
            [string]$p_taxfilepath
            )
            }
{
     $script:salestaxfilepath = $p_taxfilepath
}

function Get-UdfSalesTaxFilepath
{
     Write-Host $script:salestaxfilepath
}

function Invoke-UdfStateRateLoad
{

    $insalestax = Import-CSV $salestaxfilepath

    foreach ($item in $insalestax) {$script:salestax[$item.StateCode] = $item.SalesTaxRate}

}

function Get-UdfStateTaxRate
{
 [CmdletBinding()]
        param (
          [string]    $p_statecode
          )

   if ($script:salestax.Count -eq 0)
   {
      "Here"
      Invoke-UdfStateRateLoad
   }

   $script:salestax["$p_statecode"]

}

#  Note:  Site: http://www.xe.com/currency/usd-us-dollar?r provided Currency Conversion
values.

$exchangerate = Import-CSV ($env:HomeDrive + $env:HOMEPATH + "\Documents\
currencyexchangerate.csv")
```

```
function Get-UdfExchangeRate
{
 [CmdletBinding()]
        param (
          [Parameter(Mandatory = $true, Position = 0)]
          [string]    $p_currencycd,
          [Parameter(Mandatory = $true, Position = 1)]
          [string]    $p_asofdate,
          [Parameter(Mandatory = $true, Position = 2, ParameterSetName = "unitsperusd")]
          [switch]    $UnitsPerUSD,
          [Parameter(Mandatory = $true, Position = 2, ParameterSetName = "usdperunits")]
          [switch]    $USDPerUnits,
          [Parameter(Mandatory = $true, Position = 2, ParameterSetName = "currencyname")]
          [switch]    $CurrencyName
          )

   foreach ($item in $exchangerate)
   {
      if ($item.CurrencyCD -eq $p_currencycd -and $item.AsOfDate -eq $p_asofdate)
      {
          if ($CurrencyName) { Return $item.CurrencyName  }
          if ($UnitsPerUSD)  { Return $item.UnitsPerUSD   }
          if ($USDPerUnits)  { Return $item.USDPerUnit    }
      }

   }
 }
```

There's a lot happening in this script, so we're going to walk through it slowly in order to understand everything it is doing. Note: The actual sales tax rates and conversion rates may not be accurate, so do not rely on them. The goal here is to provide a realistic example of how a script module might be used in database development.

In this code, I included both custom comments and PowerShell-supported comment tags to remind us that we should include these whenever possible. Remember, the comment tags are automatically provided to users when they request help. The first non-comment line declares a string variable, as shown here:

```
[string]$script:salestaxfilepath = $env:HomeDrive + $env:HOMEPATH + `
"\Documents\StateSalesTaxRates.csv"
```

This variable stores the location of a file be loaded. Remember, by default this variable is not visible to the script that imports the module. To avoid hard-coded file paths, we use the environment variables `$env:HomeDrive` and `$env:HOMEPATH` to get the path to the documents folder for the current user. This is approach is repeated elsewhere.

Then, we load an external CSV file into the variable `$inexchangerate` with the statement that follows:

```
$inexchangerate = Import-CSV $env:HomeDrive + $env:HOMEPATH + `
 "\Documents\currencyexchangerate.csv"
```

We're going to load a state sales tax hashtable, so we declare it with the following line:

```
[hashtable]$script:salestax = @{}
```

This hashtable is empty but will be loaded by a function.

We want to allow the calling script to be able to change the sales tax file that gets loaded, but, as mentioned, it is not a good practice to give an external program access to a module variable. Instead, we create a special function to assign a value to the variable and another function to get the value of the variable. Let's look at the function that sets the variable's value:

```
function Set-UdfSalesTaxFilepath {
        [CmdletBinding()]
        param (
            [Parameter(Mandatory = $true)]
            [ValidateScript({ Test-Path $_ })]
            [string]$p_taxfilepath
            )
            }
{
     $script:salestaxfilepath = $p_taxfilepath
}
```

In this code, notice that we are using the `CmdLetBinding ValidateScript` feature to test the existence of the path being passed to the `Set-UdfSalesTaxFilepath` function. This is included to demonstrate how using a `Set` function can enhance the extensibility of the code. If the path exists, it is assigned to the module variable `$script:salestaxfilepath`.

To retrieve the value of the variable, the following function is provided:

```
function Get-UdfSalesTaxFilepath
{
     Return $script:salestaxfilepath
}
```

This code just returns the value of `$script:salestaxfilepath`. Although, in this case, it seems unnecessary to have a `Get` function, it is a good practice as requirements may change and the need to add additional logic is always a possibility.

Let's test these functions with the following code:

```
Import-Module umd_state

Set-UdfSalesTaxFilepath "c:\test\salestax.csvt"
Get-UdfSalesTaxFilepath
```

We need to import the module before we can use its function. The call to `Set-UdfSalesTaxFilepath` will fail, assuming the folder does not exist. The second line should display the default value assigned to `$script:salestaxfilepath`

To create this module, I thought a currency conversion function would be useful. I found the site: `http://www.xe.com/currency/usd-us-dollar?r=`, which provided some currency conversion values for a given date. I extracted that to a CSV file, which is loaded with the following statement:

```
$exchangerate = Import-CSV ($env:HomeDrive + $env:HOMEPATH + "\Documents\
currencyexchangerate.csv")
```

Unlike the sales tax function, we are not loading this into a hashtable. That's because there are several properties we need to get access to and hash tables only support a key/value pair. The following function uses the $exchangerate variable to return a selected property. Let's look at the following code:

```
function Get-UdfExchangeRate
{
 [CmdletBinding()]
        param (
          [Parameter(Mandatory = $true, Position = 0)]
          [string]    $p_currencycd,
          [Parameter(Mandatory = $true, Position = 1)]
          [string]    $p_asofdate,
          [Parameter(Mandatory = $true, Position = 2, ParameterSetName = "unitsperusd")]
          [switch]    $UnitsPerUSD,
          [Parameter(Mandatory = $true, Position = 2, ParameterSetName = "usdperunits")]
          [switch]    $USDPerUnits,
          [Parameter(Mandatory = $true, Position = 2, ParameterSetName = "currencyname")]
          [switch]    $CurrencyName
          )

   foreach ($item in $exchangerate)
   {
      if ($item.CurrencyCD -eq $p_currencycd -and $item.AsOfDate -eq $p_asofdate)
      {
          if ($CurrencyName) { Return $item.CurrencyName  }
          if ($UnitsPerUSD)  { Return $item.UnitsPerUSD   }
          if ($USDPerUnits)  { Return $item.USDPerUnit    }
      }

   }
 }
```

In this code, we use CmdLetBinding with parameter sets to provide the required functionality. All calls to Get-UdfExchangeRate must pass the country currency code and an as-of date—i.e., the date for the exchange rate. The third parameter must be one of three possible switches; UnitsPerUSD, USDPerUnits, CurrencyName. The switch that is passed determines what the function passes back. UnitsPerUSD is the number of units of the foreign currency required to equal the value of one US dollar. USDPerUnits is the number of US dollars required to equal one foreign currency unit. CurrencyName is the name of the currency for the currency code passed. By putting the switch parameter in the same position, making it mandatory, and assigning a separate parameter set name for each, we are requiring one switch to be passed, and only one switch. The foreach loop just cycles through each row in $exchangerates collection, and if a row has a CurrencyCD value and AsOfDate that match the parameters, a value will be returned depending on the switch passed.

Let's try the function out with the following code:

```
Get-UdfExchangeRate "GGP" '2014-12-22' -UnitsPerUSD
Get-UdfExchangeRate "GGP" '2014-12-22' -USDPerUnits
Get-UdfExchangeRate "GGP" '2014-12-22' –CurrencyName
```

A Warning about Variable Scopes

In the previous module, the functions to set and get a property value—i.e., Set-UdfSalesTaxFilepath and Get-UdfSalesTaxFilepath—respectively prefix the variable with the script scope, i.e., $script:salestaxfilepath. This is required due to a nuance about default variables scopes. A variable created in the module outside of a function gets a script-level scope, but a variable created in a function gets a local-function-level scope. If a function just reads a variable, it will use the script-level variable if there is no local-function-scoped variable of the same name. However, if the function assigns a value to a variable, it will create a local-function variable of that name, even if there is already one with a script-level scope (or a global one for that matter). This is called dynamic scoping and may be different than what you have seen in other programming languages. To illustrate, let's look at the following code:

```
$name = 'somevalue'                 # Creates the variable and assigns it to script scope

function Set-Name([string]$p_name)
{
        $name = $p_name            # Creates a new variable and assigns it to local-function scope
}

function Get-Name
{
         Write-Host $name    # Since no local variable $name exists, goes script-level
         variable.
}
```

This code is a very simple example of using Set and Get functions to assign a variable value. Let's test the code with the following statements:

```
Set-Name "test"
Get-Name
```

The call Get-Name returned "somevalue" instead of the value we set, i.e., "test." A small change to the code can correct this problem. Let's look at the revised code:

```
$name = 'somevalue'

function Set-Name([string]$p_name)
{
         $script:name = $p_name # Forces function to use the script level variable.
}

function Get-Name
{
         Write-Host $name
}
```

In this code, by simply prefixing the $name variable with the script scope, the Set function correctly assigns the value to the script-level variable. Test it with the following statements:

```
Set-Name "test"
Get-Name
```

Although you can just assign the scope when required, it is a better practice when using the Set and Get functions to assign and read module variables to always use the scope prefix. This makes the code easier to read and avoids bugs creeping into the module if changes are made.

Using a Module Like an Object

A nice feature of modules is that you can assign the loaded module to an object variable and then access the methods and properties using standard object notation. Let's look at the following code to see what I mean:

```
$mathobject = Import-Module umd_Simple -AsCustomObject

$mathobject.'Invoke-UdfAddNumber'(2, 3 )
```

The first line of code imports the module and uses the parameter -AsCustomObject, which causes PowerShell to return the module as an object that is assigned to $mathobject. Then the $mathobject's Invoke-UdfAddNumber method is called to add two numbers. PowerShell requires the function be enclosed in quotes because of the dash in the function name. Notice that when calling the method from an object variable we enclose the parameters in parentheses and separate them with a comma. It may not be obvious why you would want to use a module as an object. One reason is the readability of your code. In a long script, if you are importing many modules, it may not be clear where each function is coming from. The object notation makes it easy to see where the function resides. Another possible use is so that you can pass the module object to a function. An issue with PowerShell is that it does not support namespaces. A namespace is a tag used to group objects and methods much like a schema does for database objects. The same object or method name can exist in different namespaces. Prefixing the object or method with the namespace uniquely identifies the one you want. In PowerShell, if a function of the same name is defined in two different modules, there is no way to tell which one will get called. Calling the functions as methods of an object, as shown here, avoids these namespace collisions, i.e. the object variable acts like a namespace identifier.

Exporting Members

So far we have seen the default behavior that all functions are exposed to the script that imports it and all variables are not exposed. However, we can take direct control and specify which functions and variables to expose using the Export-ModuleMember cmdlet. Let's look at a simple example of this with the umd_Simple module we saw earlier:

```
# Simple module example

$somevar = 'Default Value'

function Invoke-UdfAddNumber([int]$p_int1, [int]$p_int2)
{
     Return ($p_int1 + $p_int2)
}

function Invoke-UdfSubtractNumber([int]$p_int1, [int]$p_int2)
{
     Return ($p_int1 - $p_int2)
}

Export-ModuleMember –function Invoke-UdfAddNumber
```

In this code, just by adding the line `Export-ModuleMember Invoke-UdfAddNumber`, we change the behavior to only allow external code to call `the Invoke-UdfAddNumber function`. There is a module variable added to help us see how this is affected. In this case, the module variable is still not available to external code. However, if we change the last line to the following code, the variable *will* be available to the client:

```
Export-ModuleMember -function Invoke-UdfAddNumber -variable somevar
```

Let's save this version of the script to `umd_Simple.psm1` in the Windows PowerShell folder with the same name. With this change, we can run the following two statements:

```
$mathobject = Import-Module umd_Simple -AsCustomObject -Force  # Force will reload if
necessary.

$mathobject.'Invoke-UdfAddNumber'(2, 3)
$mathobject.somevar = 'test'
```

However, we get an error if we try the following statement:

```
$mathobject.'Invoke-UdfSubtractNumber'(5, 3)
```

As useful as modules are, there is one limitation with them that bothers me. We need to include the code to all the functions in one script. I think this is bad from a code-maintenance standpoint. First, let's assume we need to change a single function. Ideally, we would check out the source code to the single function, make our changes, test them, and check it back in for it to be deployed to production. If the function is in a module, we need to check out the script, which may contain hundreds of functions, then test, check in, and deploy the entire module. In the process, we may have inadvertently changed a function without realizing it. It's a good practice to have things modular so we can develop, test, and deploy units of work. Also, since there are many functions in a single module, there is a better chance another developer will need to make changes to the same module script at the same time. This leads to change-control issues. Even just locating the function code is more difficult. You have to load the entire module and search for the function you need.

The ideal solution to this would be to have the module dot source the function script files it needs. Then we get the discoverability of modules with the convenience of dot sourcing. The problem with dot sourcing within the module is that when the module loads, the current folder may not be the module's folder. We could hard code a path to where the function scripts are, but that would cause maintenance issues later. It turns out there is a PowerShell module variable `$PSScripRoot` that gets assigned a value when a module starts to load. By leveraging this variable, we can have the module script load individual function script files that are stored in the module folder. Let's look at the following module script:

```
<#
.Author
   Bryan Cafferky

.SYNOPSIS
    A simple module to demonstrate using dot sourcing to load the functions.

.DESCRIPTION
    When this module is imported, the functions are loaded using dot sourcing.

#>

. ($PSScriptRoot + "\Invoke-UdfAddNumber.ps1")

. ($PSScriptRoot + "\Invoke-UdfSubtractNumber.ps1")
```

In this script, we simply concatenate the variable `$PSScriptRoot` with the script file name that contains the function we want loaded. These functions are made available to the script that imported the module. Sample code to load and test this module can be seen here:

```
Import-Module umd_ModuleDotSource

Invoke-UdfAddNumber 1 2

Invoke-UdfSubtractNumber 5 1
```

To prove that the module is discoverable to PowerShell enter this command:

```
Get-Module -ListAvailable
```

You should see a list of modules, including `umd_ModuleDotSource`. Thus, we get the benefit of discoverability while being able to maintain our functions in separate script files.

Flexibility in Modules

When you read most of the books and documentation about modules, you get the sense that modules must be static and hold a bunch of functions. However, modules are scripts, which means they can execute code that customizes what and how things are loaded. To get an idea of what I mean, let's look at the code for the free open-source SQLPSCX module listed here:

```
#-----------------------------------------------------------------#
# SQLPSX.PSM1
# Author: Bernd, 05/18/2010
#
# Comment: Replaces Max version of the SQLPSX.psm1
#-----------------------------------------------------------------#

$PSXloadModules = @()
$PSXloadModules = "SQLmaint","SQLServer","Agent","Repl","SSIS","Showmbrs"
$PSXloadModules += "SQLParser","adolib"
if ($psIse) {
   $PSXloadModules += "SQLIse"
}

$oraAssembly = [System.Reflection.Assembly]::LoadWithPartialName("Oracle.DataAccess")
if ($oraAssembly) {
   $PSXloadModules += "OracleClient"
   if ($psIse) {
       $PSXloadModules += "OracleIse"
   }
}
else { Write-Host -BackgroundColor Black -ForegroundColor Yellow "No Oracle found" }

$PSXremoveModules = $PSXloadModules[($PSXloadModules.count)..0]
```

```
$mInfo = $MyInvocation.MyCommand.ScriptBlock.Module
$mInfo.OnRemove = {
   foreach($PSXmodule in $PSXremoveModules){
       if (gmo $PSXmodule)
       {
         Write-Host -BackgroundColor Black -ForegroundColor Yellow "Removing SQLPSX
         Module - $PSXModule"
         Remove-Module $PSXmodule
       }
   }

   Write-Host -BackgroundColor Black -ForegroundColor Yellow "$($MyInvocation.MyCommand.
   ScriptBlock.Module.name) removed on $(Get-Date)"
}

foreach($PSXmodule in $PSXloadModules){
 Write-Host -BackgroundColor Black -ForegroundColor Yellow "Loading SQLPSX Module -
 $PSXModule"
 Import-Module $PSXmodule -global
}
Write-Host -BackgroundColor Black -ForegroundColor Yellow "Loading SQLPSX Modules is Done!"
```

Looking at this code, we can see that the script loads multiple other modules. It also applies conditional logic to decide what to install. Consider the following lines from the module.

```
if ($psIse) {
   $PSXloadModules += "SQLIse"
}
```

In the code above, we can see that the module script loads a number of other modules. The switch `$psIse` is checked and if true, the module `SQLIse` is added to the list of modules to be loaded. The point is to remember that you have flexibility in how you code modules. The following lines check for the existence of the `Oracle.DataAccess` assembly, and if loads the `OracleClient` module was loaded and if the `$psIse` switch was passed, it loads the `OracleIse` module as well:

```
$oraAssembly = [System.Reflection.Assembly]::LoadWithPartialName("Oracle.DataAccess")
if ($oraAssembly) {
   $PSXloadModules += "OracleClient"
   if ($psIse) {
       $PSXloadModules += "OracleIse"
   }
}
else { Write-Host -BackgroundColor Black -ForegroundColor Yellow "No Oracle found" }
```

Since this module loads other modules, it has code to remove those modules when the `SQLPSX` module is unloaded. The code here does this:

```
$PSXremoveModules = $PSXloadModules[($PSXloadModules.count)..0]

$mInfo = $MyInvocation.MyCommand.ScriptBlock.Module
$mInfo.OnRemove = {
    foreach($PSXmodule in $PSXremoveModules){
        if (gmo $PSXmodule)
        {
          Write-Host -BackgroundColor Black -ForegroundColor Yellow "Removing SQLPSX Module -
          $PSXModule"
          Remove-Module $PSXmodule
        }
    }

    Write-Host -BackgroundColor Black -ForegroundColor Yellow "$($MyInvocation.MyCommand.
    ScriptBlock.Module.name) removed on $(Get-Date)"
}
```

In the first line of the code, the array variable `$PSXremoveModules` is being loaded from the list of loaded modules. The line after this uses `$MyInvocation`, a PowerShell variable that holds information about the currently executing script. The line `$mInfo = $MyInvocation.MyCommand.ScriptBlock.Module` is just getting the name of the module. The interesting thing is that this is used to attach code to the `OnRemove` vent of the module; i.e., if the user executes the `Remove-Module` cmdlet for this module, the code in the braces will execute. Let's look at the loop that executes:

```
foreach($PSXmodule in $PSXremoveModules){
    if (gmo $PSXmodule)
    {
      Write-Host -BackgroundColor Black -ForegroundColor Yellow "Removing SQLPSX Module -
      $PSXModule"
      Remove-Module $PSXmodule
    }
}
```

We can see that the code loops through the array of module names. The statement "`if (gmo $PSXmodule)`" is executing the `Get-Module` cmdlet for each module so that the code in the braces that removes the module will only execute for modules that are loaded. If you are wondering what "`gmo`" means, you are not alone. It is an alias for `Get-Module`. Avoid using aliases in your scripts. It is fine for interactive use in the CLI, but should be avoided in scripts that are to be used by multiple users. The point of this code review is to show you how flexible module scripts can be.

Making Our Module More Flexible

A concern that may arise with a module is that it takes a long time to load and consumes a lot of memory, i.e. PowerShell is loading many exported functions and variables. Suppose some of the functions are used very often, but others are rarely used. Why load the rarely used functions unless we need them? Let's revise the module we created earlier, `umd_ModuleParms`, that dot sourced its members, to include a switch parameter that tells the module to conditionally load the rarely used functions. Note: The module file for the book already has this change. Please review the code below.

```
param ( [switch]$IncludeExtended )

. ($PSScriptRoot + "\Invoke-UdfAddNumber.ps1")

. ($PSScriptRoot + "\Invoke-UdfSubtractNumber.ps1")
```

```
if ($IncludeExtended)
{
  Write-Host "Adding extended function: $PSScriptRoot\Invoke-UdfMultiplyNumber.ps1"
  . ($PSScriptRoot + "\Invoke-UdfMultiplyNumber.ps1")
}
```

The first line of code defines a switch parameter `$IncludeExtended`. Since it is a switch, it automatically gets assigned `$false` if it is not passed. Then the code dot sources the two functions `Invoke-UdfAddNumber.ps1` and `Invoke-UdfSubtractNumber.ps1`. However, the function `Invoke-UdfMultiplyNumber.ps1` is only loaded if the caller passed `$true` to the module. In other words, by default, `Invoke-UdfMultiplyNumber.ps1` will not be loaded. Let's look at code that loads the module in Listing 6-1.

Listing 6-1. Loading a subset of module functions

```
Import-Module umd_ModuleParms -Force

Invoke-UdfAddNumber 1 2                 # Returns 3

Invoke-UdfSubtractNumber 5 1            # Returns 4

Invoke-UdfMultiplyNumber 5 6           # Generates an error because the function is not loaded.

Get-Module umd_ModuleParms              # Confirms that the new function is not loaded.
```

When we execute the statements in Listing 6-1, we see that `Invoke-UdfAddNumber` and `Invoke-UdfSubtractNumber` work fine. However, the call to `Invoke-UdfMultiplyNumber` fails. `Get-Module` shows us the function `Invoke-UdfMultiplyNumber` was not imported. The `Force` parameter on `Import-Module` tells PowerShell to reload the module if it is already loaded. Note: To list all the functions in memory, we can enter `Get-ChildItem function:` which lists the function name and the module it was loaded from.

To import the module and pass a value to `$IncludeExtended`, we need to use the `Import-Module ArgumentList` parameter as shown in Listing 6-2.

Listing 6-2. Loading the optional module function

```
Import-Module umd_ModuleParms -Force -ArgumentList $true # Displays message new function loaded.

Invoke-UdfAddNumber 1 2                 # Returns 3

Invoke-UdfSubtractNumber 5 1            # Returns 4

Invoke-UdfMultiplyNumber 5 6            # Returns 30

Get-Module umd_ModuleParms              # Confirms that the new function is loaded.
```

Listing 6-2 passes `$true` to the moduleListings `ArgumentList` parameter and `Invoke-UdfMultiplyNumber` is loaded. Now the call to `Invoke-UdfMultiplyNumber` succeeds. `Get-Module` confirms that `Invoke-UdfMultiplyNumber` is loaded. An interesting point is that internally the module named the parameter, i.e. `$IncludeExtended`, but when the module is loaded, we cannot pass the parameter by name. This makes sense since PowerShell does not know what parameters are defined for the module until it has been loaded. However, using meaningful parameter names in the module itself can make the code easier to understand. For a large module, customizing what gets loaded can save on resources. Alternatively, we could break functions up into separate modules.

Module Manifest

A module manifest is an optional file with a .psd1 extension that documents the module and how it is to be loaded—i.e., what files should be included. A manifest can be used for both binary and script modules. In its simplest usage, a manifest can provide basic information about the module, such as the author, copyright, description, and so forth. However, a manifest can also be used to nest modules and load custom .NET types and formats. When PowerShell executes the `Import-Module` cmdlet, it will look for a manifest to process before it looks for the module script. If there is none, it just loads the module. To make creating a manifest easier for developers, PowerShell provides the `New-ModuleManifest` cmdlet. Let's give it a try by entering the following command:

```
New-ModuleManifest
```

Because no parameters were specified, we are prompted for them. Most of the parameters will be written to the manifest file, but a few control the operation of the cmdlet. The parameters and their meanings are provided in Table 6-1.

Table 6-1. *Module Manifest Parameters*

Parameter	Description	Example
`Path`	Path and file name of manifest file	`C:\user1\modules\MyModule.psd1`
`AliasesToExport`	List of aliases to be exported	`Gtb,xyz`
`Author`	Module author's name	`Bryan Cafferky`
`ClrVersion`	Required CLR version	`4.0.30319`
`CmdletsToExport`	List of cmdlets to be exported	`Get-Something, Set-Something`
`CompanyName`	Name of the company	`BPC Global Solutions LLC`
`Copyright`	Ownership copyright information	`c BPC Global Solutions LLC All rights reserved.`
`DefaultCommandPrefix`	To avoid potential name collisions, adds this prefix to exported functions and cmdlets	`bryan`
`Description`	Tells what the module does	`Provides useful ETL functions`
`DotNetFrameworkVersion`	.Net framework version required by this module.	
`FileList`	List of files to include	`File1, file2`
`FormatsToProcess`	Format files to process	`@()`
`FunctionsToExport`	Functions that are exported.	`Get-Something, Set-Something`
`GUID`	Unique module identifier. If not entered, PowerShell will generate one.	`50cdb55f-5ab7-489f-9e94-4ec21ff51e59`

(*continued*)

Table 6-1. (*continued*)

Parameter	Description	Example
`HelpInfoURI`	A link to where help information is located	`http://www.somehelp.com`
`ModuleList`	List of modules to be loaded	`@()`
`ModuleVersion`	Documents the module's version	`1.0.0.0`
`NestedModules`	Modules within modules	`@()`
`PassThru`	Use to pass the data to the pipe in addition to writing the output file	`PassThru`
`PowerShellHostName`	The PowerShell host name. If needed, this can be found in the variable `$host.name`.	`Windows PowerShell ISE Host`
`PowerShellHostVersion`	The minimum PowerShell host version required.	`2.0`
`PowerShellVersion`	The minimum PowerShell version required	`2.0`
`PrivateData`	This is any private data that needs to be passed to the root module.	`System.IO.Compression.FileSystem`
`ProcessorArchitecture`		`IA64`
`RequiredAssemblies`	List any required assemblies	`@()`
`RequiredModules`	Non-root modules to be imported	`MyOtherModule.psm1`
`RootModule`	The core module and possibly the only module	`MyModule.psm1`
`ScriptsToProcess`	A script to be executed in the caller's session state	`Myprocess.ps1`
`TypesToProcess`	Type files to process on module import	`@()`
`VariablesToExport`	Variables to export	`*`
`Confirm`	Tell PowerShell to confirm before creating the manifest	`Confirm`
`WhatIf`	Asks PowerShell to tell you what is would do without actually doing it, i.e., creating the manifest file	`WhatIf`

For more details on the module manifest, consult the following link:

`http://msdn.microsoft.com/en-us/library/dd878337(v=vs.85).aspx`

If there is a manifest, we can use `Get-Module` to retrieve this information. For example, for the open-source module `WPK`, we can run the following script to get information from the manifest:

```
Import-module WPK

$i = Get-Module WPK
```

```
$i.Author
$i.CompanyName
$i.Copyright
$i.Description
$i.guid
```

This will give the following output:

```
James Brundage
Microsoft
2009

The WPF Powershell Kit is a PowerShell module for making quick user interfaces using Windows
Presentation Foundation and Windows PowerShell

Guid
----
00000000-0000-0000-0000-000000000000
f23582a5-01e1-4519-856b-01b8d6997bc5
```

The WPK module is available as part of the IsePackV2 module download available at the link below. However, WPK may not remain compatible with new versions of PowerShell.

http://social.technet.microsoft.com/wiki/contents/articles/4308.popular-powershell-modules.aspx

Dynamic Modules

Temporary modules can be created by a script, used, and then discarded automatically when the script exits. These are called dynamic modules and are basically an extension of the script-block concept. As an example, Listing 6-3 has two functions: one that extracts only letters from a string and another that extracts only digits from a string.

Listing 6-3. An example of a dynamic module

```
# Example of a Dynamic Module...

$scrblock = {
function Get_UdfLetters([string]$p_instring)
  {
     Return ($p_instring -replace '[^A-Z ]','')
  }

function Get_UdfNumbers([string]$p_instring)
  {
     Return ($p_instring -replace '[^0-9]','')
  }

}

$mod = new-module -scriptblock $scrblock -AsCustomObject

$mod | Get-Member
```

Let's look at the output to the console:

```
        TypeName: System.Management.Automation.PSCustomObject
Name                    MemberType      Definition
----                    ----------      ----------
Equals                  Method          bool Equals(System.Object obj)
GetHashCode             Method          int GetHashCode()
GetType                 Method          type GetType()
ToString                Method          string ToString()
Get_UdfLetters          ScriptMethod    System.Object Get_UdfLetters();
Get_UdfNumbers          ScriptMethod    System.Object Get_UdfNumbers();
```

In Listing 6-3, we start by assigning two function definitions to a script-block variable `$scrblock`. Then we use the `New-Module` cmdlet to create a module in memory using the script block we defined. Notice the `-AsCustomObject` parameter. This tells PowerShell to pass back the module as a PowerShell object that is assigned to the variable `$mod`. Just to prove `$mod` is a valid module, we pipe `$mod` into `Get-Member`, which should show us the two functions we defined, as shown in the output above.

The nice thing about creating our dynamic module as a PowerShell custom object is that we can use object method-calling syntax to use its functions. Let's look at the following code:

```
$mod.Get_UdfLetters("mix of numbers 12499 and letters")
```

Displays the output below to the console.

```
mix of numbers  and letters

$mod.Get_UdfNumbers("mix of numbers 12499 and letters")
```

Displays the output below to the console.

```
12499
```

Notice that we are violating the cmdlet naming convention by separating the verb and noun with an underscore instead of a dash. This is because if we use a dash, PowerShell will require us to enclose the function name in quotes when we call it. Since this is just a temporary module, I think breaking the recommended naming convention is justified. In most cases, we would probably just want to create standard script modules. However, dynamic modules do offer some flexibility that may come in handy. Since the script block is just a variable, we could modify it on the fly, changing what the module does and how it works. Let's look at a revised version of the script in Listing 6-4 to see how this works:

Listing 6-4. Changing the script block definition dynamically

```
$option = Read-Host "Enter 'A' to Allow underscore or 'D' to disallow the underscore "

if ($option -ne 'A') {
$scrblock = {
function Get_UdfLetters([string]$p_instring)
  {
     Return ($p_instring -replace '[^A-Z ]','')
  }
function Get_UdfNumbers([string]$p_instring)
  {
     Return ($p_instring -replace '[^0-9]','')
  }

 }
}
else
{
$scrblock =
{
function Get_UdfLetters([string]$p_instring)
  {
     Return ($p_instring -replace '[^A-Z _]','')
  }

function Get_UdfNumbers([string]$p_instring)
  {
     Return ($p_instring -replace '[^0-9_]','')
  }
 }
}

$mod = new-module -scriptblock $scrblock -AsCustomObject

$mod.Get_UdfLetters("mix of numbers-_ 12499 and_ letters")

$mod.Get_UdfNumbers("mix of numbers-_ 12499 and_ letters")
```

When we run this script, we are prompted about whether we want to allow the underscore character to remain in the string parameter or not. If we enter "A", the script block is defined that allows the underscore to remain, but if we enter anything else, the script-block variable is assigned the code that removes underscores. This is a simple example, but there may be occasions when you need to dynamically change the module's code. It's just another tool in the toolbox.

Popular Free PowerShell Modules

Modules provide a way for PowerShell to be extended to include functionality not available in the base language. There are many free modules available, and it is worth familiarizing yourself with some of the more popular ones to see if they can help you. Some of these are documented at the following link:

http://social.technet.microsoft.com/wiki/contents/articles/4308.popular-powershell-modules.aspx

As useful as PowerShell community–developed modules are, there are drawbacks to using them. The biggest drawback is that they are developed for specific versions of PowerShell and the related .NET frameworks version. Thus, when you upgrade to a new version of PowerShell, you may break existing code that uses these modules. This can also cause an issue with deploying PowerShell code. For example, your code may work fine on your machine under Windows 7 using PowerShell 2.0 but fail on the target environment running PowerShell 4.0. Unfortunately, these version requirements are not always obvious on the module download sites. Besides the potential version problems, deployments become more complex, as you will need to install the required modules on the target environment. Exacerbating this problem is that some modules depend on other modules. For example, the module SQLIse depends on the modules SQLPSX and PowerShellPack to function. This can further complicate deployments.

OData PowerShell Explorer

OData is a new open data-sharing standard being advanced by Microsoft and OASIS, an international open standards consortium. The idea is to establish one flexible standard that can be applied to share data from anywhere. The OData PowerShell Explorer available at http://psodata.codeplex.com/ provides interesting insights into the capabilities of using PowerShell to access OData sources. PowerShell can also serve up data as an OData source. At this point I think the future of OData as a standard is uncertain, but I encourage you to play with the OData Explorer to get an idea of the potential

PowerShell Community Extensions (PSCX)

The PowerShell Community Extensions (PSCX) module adds a lot of useful cmdlets that extend PowerShell in many areas. When you follow the link to the download page, be sure to select the version appropriate to your version of PowerShell. Fortunately, this module has a release that supports PowerShell 4.0.
You can find the version for PowerShell 3.0 and 4.0 and the help documentation about this module at http://pscx.codeplex.com/releases. The help file is named about_pscx.help.txt. Note: The link on the page Popular PowerShell Module's brings you to is the latest release. For the prior version, use https://pscx.codeplex.com/releases/view/98267.

A number of the PSCX cmdlets compress and decompress files into and from various formats. It also has some cmdlets to work with *Microsoft Message Queing*, MSMQ. Let's look at the following code to give a sense of some of the functionality this module provides.

```
"This is data in the clipboard." | Set-Clipboard

$myclipdata = Get-Clipboard

$myclipdata

Enable-OpenPowerShellHere        # Add Windows PowerShell open button to Windows Explorer.

"This is data." | Set-Clipboard

"some data" > mytestfile.txt

Set-ReadOnly mytestfile.txt      # Sets the ReadOnly attribute on so the file can only be read.

Set-Writable mytestfile.txt      # Turns off the Readonly attribute so the file can be modified.

Out-Speech "This is something"   # Uses Windows speech API.

# Zip all files with .txt extension into myzip.zip in current folder.
Write-Zip -Path *.txt -OutputPath 'myzip.zip'
```

An interesting set of cmdlets for database developers are the ones related to ADO.NET. These provide easy access to databases using .NET drivers. For SQL Server, we've previously seen code that directly accesses the .NET libraries to read data using ADO.NET. Let's look at the code that does this using the PSCX cmdlets:

```
#  Reading from SQL Server...
Get-AdoConnection -Server "(local)"  -Database "AdventureWorks" `
                  -ProviderName "System.Data.SQLClient"
$Provider="System.Data.SQLClient"
$ConnectionString="Data Source=.;Initial Catalog=AdventureWorks;Integrated Security=SSPI"
$Connection = Get-AdoConnection $Provider $ConnectionString
$Query = "SELECT top 10 * FROM person.person; select top 5 * from sales.customer;"

$mydata = Invoke-AdoCommand -ProviderName $Provider -Connection $Connection `
          -CommandText $Query -AsDataSet
$mydata.tables[0]  | Out-GridView
$mydata.tables[1]  | Out-GridView

$Connection.Close()
```

This code connects to the AdventureWorks database on the local machine and selects the top ten rows from `person.person`. It displays the result in a grid using `Out-GridView`. Notice that we first get a connection to the database with the statement `$Connection = Get-AdoConnection $Provider $ConnectionString`. `$Connection` holds the connection object that is used to execute the query in the statement `$mydata = Invoke-AdoCommand -ProviderName $Provider -Connection $Connection -CommandText $Query -AsDataSet`. The parameter `-AsDataSet` is critical for getting the data in a consumable format. We have two queries being executed. The result sets are stored in a collection named `tables`. Since `$mydata` is loaded with the query results, we can get the first resultset as `$mydata.tables[0]`, which we pipe to `Out-GridView`, and the second result set as `$mydata.tables[1]`, which is also piped to `Out-GridView`.

Accessing SQL Server is easy in the Windows world, as Microsoft provides many options. However, it can be more challenging to access other database products. PostgreSQL is a popular open source database, so let's look at how we would code statements that read from a PostgreSQL source. Note: If you need a good PostgreSQL ADO.NET driver for Windows, you can get one at `http://www.devart.com/dotconnect/postgresql/download.html`. They have a free Express edition, which is what I used for the example. You can also purchase a version with more-advanced functionality, such as support for Entity Frameworks. Now let's take a look at the code below.

```
# Reading from PostgreSQL...
$Provider="Devart.Data.PostgreSQL"
$ConnectionString="Data Source=localhost;Database=postgres;Initial
Schema=public;Port=5432;User ID=postgres;password=mypassword;Connection Lifetime=1000;"
$Connection = Get-AdoConnection $Provider $ConnectionString
$Query = "SELECT * FROM tmp_hold_log limit 10"

$mypgdata = Invoke-AdoCommand -ProviderName $Provider -Connection $Connection `
            -CommandText $Query -AsDataSet

$mypgdata.tables[0]  | Out-GridView

$Connection.Close()
```

The PostgreSQL code is not very different from the SQL Server version. The biggest difference is the connection string. Different .NET drivers support different options and names, so you need to find out what your driver supports. Integrated security is handy but is not typically supported by other database vendors, especially ones that do not run on Windows, so we need to specify a username and password. Note: We will talk about encrypting data in Chapter 9, which you may want to apply here. The SQL syntax for different databases varies, so we need to be careful not to use SQL Server syntax.

ShowUI

The ShowUI module offers an interesting option for adding a GUI to your PowerShell scripts. It provides a nice set of high-level PowerShell functions that make it easy to add visual elements like windows, buttons, text boxes, and so forth to your scripts. You can install ShowUI individually, or it can be installed as part of the module IsePackV2. Once installed, you can find some helpful examples in the folder named Examples in the ShowUI module folder. Unlike the other method of creating a GUI we've seen that uses a screen painter to generate the windows object code, ShowUI must be coded manually with location details specified via function parameters. This may be a good option for a simple GUI interface. Unfortunately, I found that ShowUI does not work with PowerShell 4.0, but does seem to work with 2.0. I hope the developers will bring this module up to date, but until they do, the usefulness of this module is limited.

SQL Server PowerShell Extensions

The SQL Server PowerShell Extensions, SQLPSX, is a free third-party set of modules that add SQL Server functionality as well as some support for Oracle. The cmdlets available are mostly administrative and probably are not ones you would include in the PowerShell code you deploy. However, they offer some nice features in working with SQL Server. Let's take a look at some examples:

```
Get-AgentJob "(local)" | Out-GridView
```

The statement above will get a list of SQL Agent jobs on the (local) server and display them in the grid view. Another example of using the SQLPSX modules can be seen below.

```
Get-SqlDatabase "(local)" AdventureWorks | Get-SqlTable -name 'person' -Schema person |
Get-SqlColumn | Select-Object -Property Name, DataType, Nullable | Out-GridView
```

This statement will get a database reference object, pipe it into the `Get-Table` cmdlet, which then gets a table reference object that is piped into the `Get-SqlColumn` cmdlet to get a list of columns in the table. This is piped into the `Select-Object` cmdlet, which pulls out the properties we want and then pipes that into the `Out-Gridview` for a nice readable display.

For documentation on the cmdlets available in this module, see the following URL:

```
http://files.powershellstation.com/SQLPSX/index.htm
```

In reading the help pages for SQLPSX, it is clear that the modules are intended mainly for database administration tasks like copying databases, moving SSIS packages, setting up replication, and things like that. The cmdlets I find most interesting are the `Get` cmdlets like the ones I demonstrated. However, some of these tasks can easily be done using the SQLPS module we've used throughout this book. For example, the code that follows will do the same thing as the first SQLPSX example:

```
Import-Module "sqlps" -DisableNameChecking

set-location SQLSERVER:\SQL\BryanCafferkyPC\DEFAULT\JobServer\Jobs

Get-ChildItem
```

One nice thing about this approach is that it sets the location context using the provider interface. This means that we only have to set the location once and subsequent commands will work in that context—i.e., SQL Agent jobs.

We can replace the example that gets column information with the following code that uses SQPPS:

```
Import-Module "sqlps" -DisableNameChecking

set-location SQLSERVER:\SQL\BryanCafferkyPC\DEFAULT\Databases\Adventureworks\Tables

$mytable = Get-ChildItem | Where-Object -Property Name -EQ 'Person'
$mytable.Columns | Select-Object -Property Name, DataType, Nullable | Out-GridView
```

The good thing about the SQLPS examples is that the SQLPS module is already installed on a machine that has SQL Server installed. I've also found that the SQLPS module tends to work pretty well across PowerShell versions. My recommendation is to use SQLPSX for daily administrative functions but to avoid deploying applications that depend on it. SQLPS is a better choice for that, or you can just use the SQL Client .NET library directly.

One thing to be aware of is that sometimes one module can cause problems with another module. For example, I found that if I imported the SQLPS module after I imported the SQLPSX module, the SQLPSX module did not work correctly. In fact, the SQLPS module can interfere with other modules as well. This is due to the provider interface. Using the `Remove-Module` cmdlet should correct the issue. However, you must set the location to a non–SQL Server context before PowerShell will let you unload the module. The statements below should do the trick.

```
set-location C:\Users       # Sets context to the File System provider.
Remove-Module SQLPS -Force  # Unloads the module from memory.
```

SQLISE

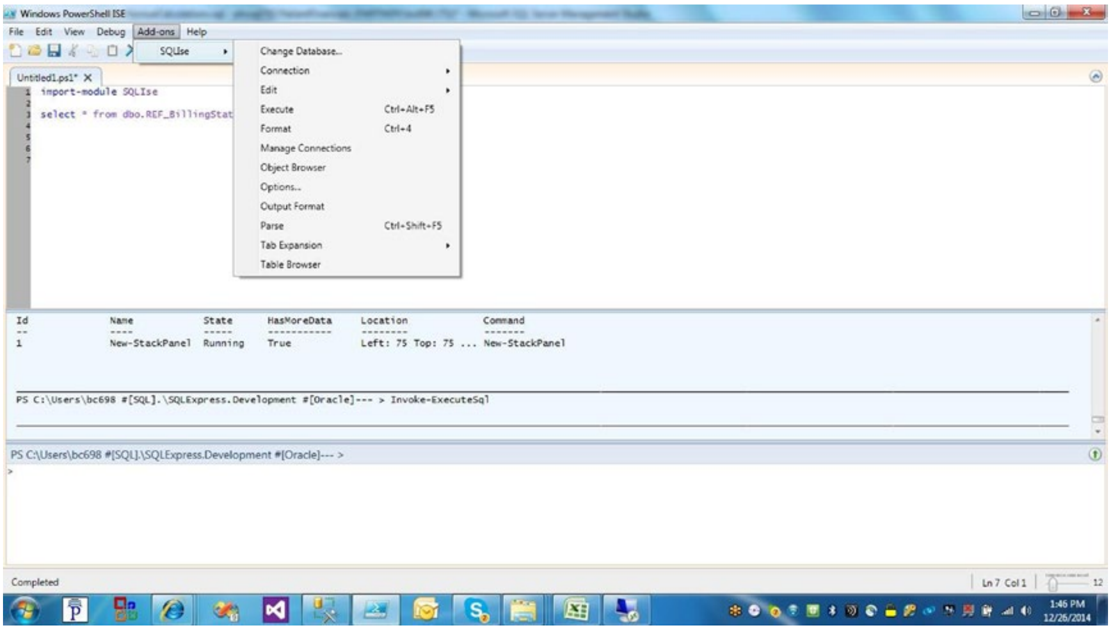
SQLIse is a module that provides SQL Query Analyzer–like functionality within the PowerShell ISE. To use this module, you need to install ISEPackV2 and SQLPSX. As interesting as this is, I found it would not install under PowerShell 4.0, but I did get it to work under PowerShell 2.0—although the install did generate a lot of warning messages about running code from the Internet. This link provides some background on this module:

http://sev17.com/2010/03/09/sqlise-a-powershell-based-sql-server-query-tool/

SQLIse provides SQL Server Management Studio's Query Analyzer–like functionality right within the PowerShell ISE. It exposes itself as an add-on in the PowerShell ISE. Assuming you have everything installed correctly, you can start to use SQLIse by importing it with the command here:

```
Import-Module SQLIse
```

There's a lot of functionality here that can best be conveyed by looking at the SQLIse menu under Add-ons. Figure 6-1 shows that menu.



```
import-module SQLIse

select * from dbo.REF_BillingStat
```

```
Id            Name            State      HasMoreData    Location            Command
--            ----            -----      -----------    --------            -------
1             New-StackPanel  Running    True           Left: 75 Top: 75 ... New-StackPanel

PS C:\Users\bc698 #[SQL].\SQLExpress.Development #[Oracle]--- > Invoke-ExecuteSql
```

```
PS C:\Users\bc698 #[SQL].\SQLExpress.Development #[Oracle]--- >
>
```

Figure 6-1. *Using the SQLIse Add-On*

Windows Automation Snap-In (WASP)

The Windows Automation Snap-In (WASP) is a useful module when you want to manipulate Windows applications as if you were a user interacting with the program. We've seen this used previously to automate navigating Internet Explorer to a website, going to a specific page, and downloading a file. It is a powerful functionality. However, such code tends to be a bit unstable because the application interface you are relying on can change. Let's look at a simple example of using WASP below.

```
Import-Module WASP

notepad.exe

start-sleep 3

Select-Window -ProcessName "notepad" | Send-Keys "This is an example of using WASP."
```

WASP was originally a PowerShell snap-in so for PowerShell versions earlier than 3.0, use the `Add-PSSnapIn` cmdlet to load it. The code above starts Notepad.exe, waits three seconds, sets the focus to the window running Notepad, and enters the text "`This is an example of using WASP`." WASP has a lot of features that are critical when you need to manipulate applications. However, many things can go wrong in this kind of application automation. In the code, for example, if you already had Notepad running when you ran the script, it would create a new instance of Notepad and would perhaps not write to the correct window. Nonetheless, this is a module that is good to be familiar with.

WASP is particularly useful for automating web access via *Internet Explorer*, IE. Let's look at the Listing 6-5 that starts Internet Explorer, navigates to Bing, and does a search for PowerShell WASP examples.

Listing 6-5. Using the WASP module

```
Import-Module WASP    # Use Import-Module starting with PowerShell 3.0

$url = "http://www.bing.com"

stop-process -processname iexplore*

$ie = New-Object -comobject InternetExplorer.Application
$ie.visible = $true
$ie.silent = $true
$ie.Navigate( $url )
start-sleep 2

Select-Window "iexplore" | Set-WindowActive

$txtArea=$ie.Document.getElementById("sb_form_q")

$txtArea.InnerText="PowerShell WASP Examples"
start-sleep 2
Select-Window "iexplore" | Set-WindowActive| Send-Keys "{ENTER}"
```

Listing 6-5 starts by importing the WASP module and assigning variable `$url` the link to Bing. The `stop-process` cmdlet is to end any Internet Explorer processes, so if you have any IE windows open, you may want to close them before running this code. Then the code below opens IE and navigates to the web page:

```
$ie = New-Object -comobject InternetExplorer.Application
$ie.visible = $true
$ie.silent = $true
$ie.Navigate( $url )
start-sleep 2
```

In the first line above, we are storing a reference to the Internet Explorer application in variable `$ie`. This makes it easy to assign properties like `visible` and `silent` and use its methods, such as `navigate`, which the script does to bring up the Bing page. Then the statement "`start-sleep 2`" pauses the script for 2 seconds to wait for the prior actions to complete. If the prior actions do not complete before we continue, any manipulation to the page such as keystrokes are lost. No errors are raised. The line below makes the IE window active and simulates pressing the enter key.

```
Select-Window "iexplore" | Set-WindowActive| Send-Keys "{ENTER}"
```

The next bit of code is interesting and important if we want to automate webpage access. This is where we enter something into the search textbox on the page. You might think that when the Bing page comes up, it will just automatically place the cursor in the search box. However, it does not, so to get the cursor there so we can enter some search terms, we use the statement seen here:

```
$txtArea=$ie.Document.getElementById("sb_form_q")
```

The method, `GetElementByID` will get us to the search text box, where we can use the following code to enter the search terms:

```
$txtArea.InnerText="PowerShell WASP Examples"
start-sleep 2
Select-Window "iexplore" | Set-WindowActive| Send-Keys "{ENTER}"
```

In the code above, we assign a value to `InnerText` which will enter text into the search text box. We need to give it time to complete this operation, so we use the `start-sleep` cmdlet. Then we need to simulate pressing the Enter key. But first, we need to make sure the focus is on the IE window, which we can do with the statements `Select-Window "iexplore" | Set-WindowActive|`. By piping this into the subsequent `Send-Keys` cmdlet, the cmdlet knows which windows to send the keystrokes to. Notice that to send the Enter key, we use "`{ENTER}`". This is a special name with which to tell send keys to send a named key rather than text. There are a number of these, which you can find in the documentation. For example, `Send-Keys {TAB}` will send a tab key.

You may be wondering how we can determine the control name for the search text box. A method we can use that often works is to use Internet Explorer to help us. First, we bring up the page that has the control we need to manipulate. Then we click on the control, in this case the search text box. Then we right mouse-click to bring up a list of possible actions, as shown in Figure 6-2.

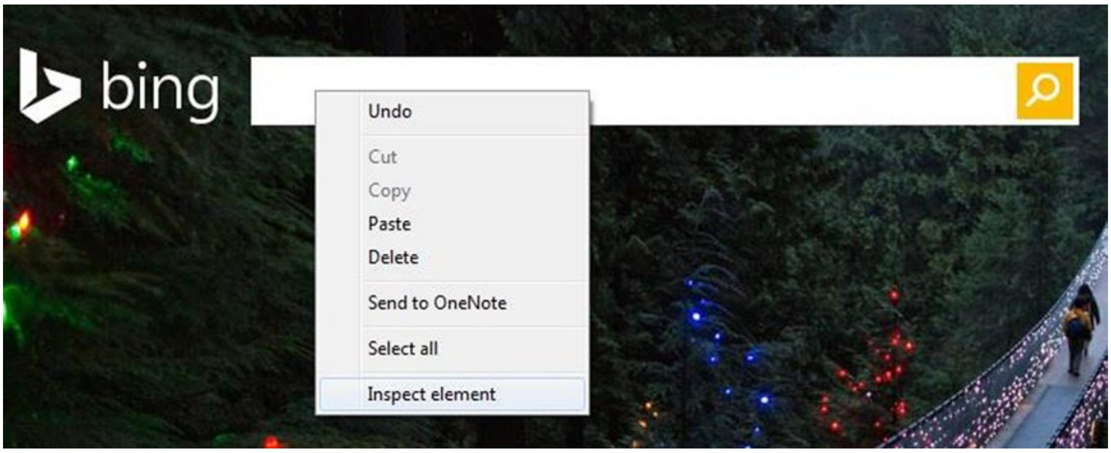
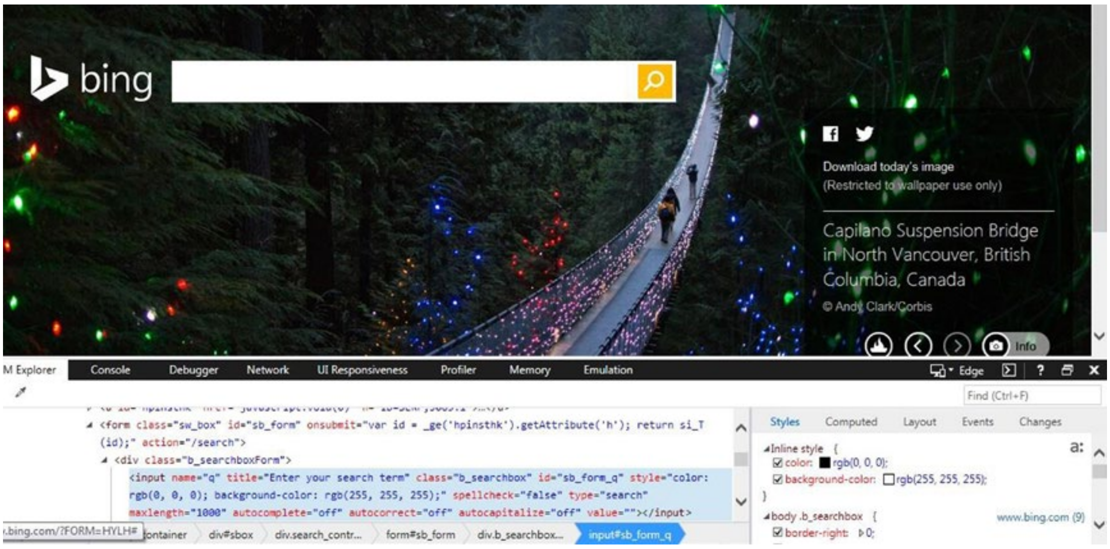


Figure 6-2. *Browser context menu showing the Inspect element option*

From the action menu, shown here, select Inspect Element. This will cause information about the selected element to display, as shown in Figure 6-3.



```
◢ <form class="sw_box" id="sb_form" onsubmit="var id = _ge('hpinsthk').getAttribute('h'); return si_T
  (id);" action="/search">
  ◢ <div class="b_searchboxForm">
      <input name="q" title="Enter your search term" class="b_searchbox" id="sb_form_q" style="color:
      rgb(0, 0, 0); background-color: rgb(255, 255, 255);" spellcheck="false" type="search"
      maxlength="1000" autocomplete="off" autocorrect="off" autocapitalize="off" value=""></input>
```

```
◢Inline style {
  ☑ color: ■rgb(0, 0, 0);
  ☑ background-color: □rgb(255, 255, 255);
}
◢body .b_searchbox {                    www.bing.com (9)
  ☑ border-right: ▷0;
```

Figure 6-3. *Properties of a selected element*

The selected block is the element we want. We can see that the ID is “`sb_form_q`”. Therefore, we can use the `GetElementByID("sb_form_q")` to position the script to this element. Admittedly, this can be a trial-and-error process, and as mentioned before, the element name could be changed without us knowing. These cautions aside, WASP provides some very powerful functionality.

Summary

This chapter covered extending PowerShell with modules. We discussed the four types of modules, which are script, manifest, binary, and dynamic. The most important of these is the script module because it is easy to implement, can be written in the PowerShell language, and provides an effective way to create reusable code. We discussed how to develop and deploy script modules. Then, we considered a weakness in script modules—i.e., all the functions must be stored in a single script file. We considered two approaches to solving this issue. We discussed what a module manifest is and how to create one. PowerShell supports the creation of temporary modules, called dynamic modules, within a script that can be created, used, and discarded. It is not always necessary to develop your own modules. We discussed a number of free open source modules that you can download and use in your applications.