**How to Write a PowerShell Script Module**

* 11/21/2019
* 7 minutes to read
  + [](https://github.com/joeyaiello)
  + [](https://github.com/sdwheeler)
  + [](https://github.com/davidsmatlak)

A script module is any valid PowerShell script saved in a .psm1 extension. This extension allows the PowerShell engine to use rules and module cmdlets on your file. Most of these capabilities are there to help you install your code on other systems, as well as manage scoping. You can also use a module manifest file, which describes more complex installations and solutions.

**Writing a PowerShell script module**

To create a script module, save a valid PowerShell script to a .psm1 file. The script and the directory where it's stored must use the same name. For example, a script named MyPsScript.psm1 is stored in a directory named MyPsScript.

The module's directory needs to be in a path specified in $env:PSModulePath. The module's directory can contain any resources that are needed to run the script, and a module manifest file that describes to PowerShell how your module works.

**Create a basic PowerShell module**

The following steps describe how to create a PowerShell module.

1. Save a PowerShell script with a .psm1 extension. Use the same name for the script and the directory where the script is saved.

Saving a script with the .psm1 extension means that you can use the module cmdlets, such as [Import-Module](https://docs.microsoft.com/en-us/powershell/module/Microsoft.PowerShell.Core/Import-Module). The module cmdlets exist primarily so that you can import and export your code onto other user's systems. The alternate solution would be to load your code on other systems and then dot-source it into active memory, which isn't a scalable solution. For more information, see [Understanding a Windows PowerShell Module](https://docs.microsoft.com/en-us/powershell/scripting/developer/module/understanding-a-windows-powershell-module?view=powershell-7.1" \l "module-cmdlets-and-variables). By default, when users import your .psm1 file, all functions in your script are accessible, but variables aren't.

An example PowerShell script, entitled Show-Calendar, is available at the end of this article.

PowerShellCopy

function Show-Calendar {

param(

[DateTime] $start = [DateTime]::Today,

[DateTime] $end = $start,

$firstDayOfWeek,

[int[]] $highlightDay,

[string[]] $highlightDate = [DateTime]::Today.ToString()

)

#actual code for the function goes here see the end of the topic for the complete code sample

}

1. To control user access to certain functions or variables, call [Export-ModuleMember](https://docs.microsoft.com/en-us/powershell/module/Microsoft.PowerShell.Core/Export-ModuleMember) at the end of your script.

The example code at the bottom of the article has only one function, which by default would be exposed. However, it's recommended you explicitly call out which functions you wish to expose, as described in the following code:

PowerShellCopy

function Show-Calendar {

}

Export-ModuleMember -Function Show-Calendar

You can restrict what's imported using a module manifest. For more information, see [Importing a PowerShell Module](https://docs.microsoft.com/en-us/powershell/scripting/developer/module/importing-a-powershell-module?view=powershell-7.1) and [How to Write a PowerShell Module Manifest](https://docs.microsoft.com/en-us/powershell/scripting/developer/module/how-to-write-a-powershell-module-manifest?view=powershell-7.1).

1. If you have modules that your own module needs to load, you can use Import-Module, at the top of your module.

The Import-Module cmdlet imports a targeted module onto a system, and can be used at a later point in the procedure to install your own module. The sample code at the bottom of this article doesn't use any import modules. But if it did, they would be listed at the top of the file, as shown in the following code:

PowerShellCopy

Import-Module GenericModule

1. To describe your module to the PowerShell Help system, you can either use standard help comments inside the file, or create an additional Help file.

The code sample at the bottom of this article includes the help information in the comments. You could also write expanded XML files that contain additional help content. For more information, see [Writing Help for Windows PowerShell Modules](https://docs.microsoft.com/en-us/powershell/scripting/developer/module/writing-help-for-windows-powershell-modules?view=powershell-7.1).

1. If you have additional modules, XML files, or other content you want to package with your module, you can use a module manifest.

A module manifest is a file that contains the names of other modules, directory layouts, versioning numbers, author data, and other pieces of information. PowerShell uses the module manifest file to organize and deploy your solution. For more information, see [How to write a PowerShell module manifest](https://docs.microsoft.com/en-us/powershell/scripting/developer/module/how-to-write-a-powershell-module-manifest?view=powershell-7.1).

1. To install and run your module, save the module to one of the appropriate PowerShell paths, and use Import-Module.

The paths where you can install your module are located in the $env:PSModulePath global variable. For example, a common path to save a module on a system would be %SystemRoot%/users/<user>/Documents/PowerShell/Modules/<moduleName>. Be sure to create a directory for your module that uses the same name as the script module, even if it's only a single .psm1 file. If you didn't save your module to one of these paths, you would have to specify the module's location in the Import-Module command. Otherwise, PowerShell wouldn't be able to find the module.

Starting with PowerShell 3.0, if you've placed your module in one of the PowerShell module paths, you don't need to explicitly import it. Your module is automatically loaded when a user calls your function. For more information about the module path, see [Importing a PowerShell Module](https://docs.microsoft.com/en-us/powershell/scripting/developer/module/importing-a-powershell-module?view=powershell-7.1) and [Modifying the PSModulePath Installation Path](https://docs.microsoft.com/en-us/powershell/scripting/developer/module/modifying-the-psmodulepath-installation-path?view=powershell-7.1).

1. To remove a module from active service in the current PowerShell session, use [Remove-Module](https://docs.microsoft.com/en-us/powershell/module/Microsoft.PowerShell.Core/Remove-Module).

**Note**

Remove-Module removes a module from the current PowerShell session, but doesn't uninstall the module or delete the module's files.

**Show-Calendar code example**

The following example is a script module that contains a single function named Show-Calendar. This function displays a visual representation of a calendar. The sample contains the PowerShell Help strings for the synopsis, description, parameter values, and code. When the module is imported, the Export-ModuleMember command ensures that the Show-Calendar function is exported as a module member.

PowerShellCopy

<#

.Synopsis

Displays a visual representation of a calendar.

.Description

Displays a visual representation of a calendar. This function supports multiple months

and lets you highlight specific date ranges or days.

.Parameter Start

The first month to display.

.Parameter End

The last month to display.

.Parameter FirstDayOfWeek

The day of the month on which the week begins.

.Parameter HighlightDay

Specific days (numbered) to highlight. Used for date ranges like (25..31).

Date ranges are specified by the Windows PowerShell range syntax. These dates are

enclosed in square brackets.

.Parameter HighlightDate

Specific days (named) to highlight. These dates are surrounded by asterisks.

.Example

# Show a default display of this month.

Show-Calendar

.Example

# Display a date range.

Show-Calendar -Start "March, 2010" -End "May, 2010"

.Example

# Highlight a range of days.

Show-Calendar -HighlightDay (1..10 + 22) -HighlightDate "December 25, 2008"

#>

function Show-Calendar {

param(

[DateTime] $start = [DateTime]::Today,

[DateTime] $end = $start,

$firstDayOfWeek,

[int[]] $highlightDay,

[string[]] $highlightDate = [DateTime]::Today.ToString()

)

## Determine the first day of the start and end months.

$start = New-Object DateTime $start.Year,$start.Month,1

$end = New-Object DateTime $end.Year,$end.Month,1

## Convert the highlighted dates into real dates.

[DateTime[]] $highlightDate = [DateTime[]] $highlightDate

## Retrieve the DateTimeFormat information so that the

## calendar can be manipulated.

$dateTimeFormat = (Get-Culture).DateTimeFormat

if($firstDayOfWeek)

{

$dateTimeFormat.FirstDayOfWeek = $firstDayOfWeek

}

$currentDay = $start

## Process the requested months.

while($start -le $end)

{

## Return to an earlier point in the function if the first day of the month

## is in the middle of the week.

while($currentDay.DayOfWeek -ne $dateTimeFormat.FirstDayOfWeek)

{

$currentDay = $currentDay.AddDays(-1)

}

## Prepare to store information about this date range.

$currentWeek = New-Object PsObject

$dayNames = @()

$weeks = @()

## Continue processing dates until the function reaches the end of the month.

## The function continues until the week is completed with

## days from the next month.

while(($currentDay -lt $start.AddMonths(1)) -or

($currentDay.DayOfWeek -ne $dateTimeFormat.FirstDayOfWeek))

{

## Determine the day names to use to label the columns.

$dayName = "{0:ddd}" -f $currentDay

if($dayNames -notcontains $dayName)

{

$dayNames += $dayName

}

## Pad the day number for display, highlighting if necessary.

$displayDay = " {0,2} " -f $currentDay.Day

## Determine whether to highlight a specific date.

if($highlightDate)

{

$compareDate = New-Object DateTime $currentDay.Year,

$currentDay.Month,$currentDay.Day

if($highlightDate -contains $compareDate)

{

$displayDay = "\*" + ("{0,2}" -f $currentDay.Day) + "\*"

}

}

## Otherwise, highlight as part of a date range.

if($highlightDay -and ($highlightDay[0] -eq $currentDay.Day))

{

$displayDay = "[" + ("{0,2}" -f $currentDay.Day) + "]"

$null,$highlightDay = $highlightDay

}

## Add the day of the week and the day of the month as note properties.

$currentWeek | Add-Member NoteProperty $dayName $displayDay

## Move to the next day of the month.

$currentDay = $currentDay.AddDays(1)

## If the function reaches the next week, store the current week

## in the week list and continue.

if($currentDay.DayOfWeek -eq $dateTimeFormat.FirstDayOfWeek)

{

$weeks += $currentWeek

$currentWeek = New-Object PsObject

}

}

## Format the weeks as a table.

$calendar = $weeks | Format-Table $dayNames -AutoSize | Out-String

## Add a centered header.

$width = ($calendar.Split("`n") | Measure-Object -Maximum Length).Maximum

$header = "{0:MMMM yyyy}" -f $start

$padding = " " \* (($width - $header.Length) / 2)

$displayCalendar = " `n" + $padding + $header + "`n " + $calendar

$displayCalendar.TrimEnd()

## Move to the next month.

$start = $start.AddMonths(1)

}

}

Export-ModuleMember -Function Show-Calendar