Import-Module

Reference

Feedback

Module: Microsoft.PowerShell.Core

In this article

Adds modules to the current session.

Syntax

```
Import-Module
      [-Global]
      [-Prefix <String>]
      [-Name] <String[]>
      [-Function <String[]>]
      [-Cmdlet <String[]>]
      [-Variable <String[]>]
      [-Alias <String[]>]
      [-Force]
      [-PassThru]
      [-AsCustomObject]
      [-MinimumVersion <Version>]
      [-MaximumVersion <String>]
      [-RequiredVersion <Version>]
      [-ArgumentList <Object[]>]
      [-DisableNameChecking]
      [-NoClobber]
      [-Scope <String>]
      [<CommonParameters>]
```

```
Import-Module
      [-Global]
      [-Prefix <String>]
      [-Name] <String[]>
      [-Function <String[]>]
      [-Cmdlet <String[]>]
      [-Variable <String[]>]
      [-Alias <String[]>]
      [-Force]
      [-PassThru]
      [-AsCustomObject]
      [-MinimumVersion <Version>]
      [-MaximumVersion <String>]
      [-RequiredVersion <Version>]
      [-ArgumentList <Object[]>]
      [-DisableNameChecking]
```

```
      [-NoClobber]
      [-Scope <String>]
      -PSSession <PSSession>
      [<CommonParameters>]
```

```
Import-Module
      [-Global]
      [-Prefix <String>]
      [-Name] <String[]>
      [-Function <String[]>]
      [-Cmdlet <String[]>]
      [-Variable <String[]>]
      [-Alias <String[]>]
      [-Force]
      [-PassThru]
      [-AsCustomObject]
      [-MinimumVersion <Version>]
      [-MaximumVersion <String>]
      [-RequiredVersion <Version>]
      [-ArgumentList <Object[]>]
      [-DisableNameChecking]
      [-NoClobber]
      [-Scope <String>]
      -CimSession <CimSession>
      [-CimResourceUri <Uri>]
      [-CimNamespace <String>]
      [<CommonParameters>]
```

```
Import-Module
      [-Global]
      [-Prefix <String>]
      [-FullyQualifiedName] <ModuleSpecification[]>
      [-Function <String[]>]
      [-Cmdlet <String[]>]
      [-Variable <String[]>]
      [-Alias <String[]>]
      [-Force]
      [-PassThru]
      [-AsCustomObject]
      [-ArgumentList <Object[]>]
      [-DisableNameChecking]
```

```
      [-NoClobber]
      [-Scope <String>]
      [<CommonParameters>]
```

```
Import-Module
      [-Global]
      [-Prefix <String>]
      [-FullyQualifiedName] <ModuleSpecification[]>
      [-Function <String[]>]
      [-Cmdlet <String[]>]
      [-Variable <String[]>]
      [-Alias <String[]>]
      [-Force]
      [-PassThru]
      [-AsCustomObject]
      [-ArgumentList <Object[]>]
      [-DisableNameChecking]
      [-NoClobber]
      [-Scope <String>]
      -PSSession <PSSession>
      [<CommonParameters>]
```

```
Import-Module
      [-Global]
      [-Prefix <String>]
      [-Assembly] <Assembly[]>
      [-Function <String[]>]
      [-Cmdlet <String[]>]
      [-Variable <String[]>]
      [-Alias <String[]>]
      [-Force]
      [-PassThru]
      [-AsCustomObject]
      [-ArgumentList <Object[]>]
      [-DisableNameChecking]
      [-NoClobber]
      [-Scope <String>]
      [<CommonParameters>]
```

```
Import-Module
      [-Global]
      [-Prefix <String>]
      [-Function <String[]>]
      [-Cmdlet <String[]>]
      [-Variable <String[]>]
      [-Alias <String[]>]
      [-Force]
      [-PassThru]
      [-AsCustomObject]
      [-ModuleInfo] <PSModuleInfo[]>
      [-ArgumentList <Object[]>]
      [-DisableNameChecking]
      [-NoClobber]
      [-Scope <String>]
      [<CommonParameters>]
```

Description

The `Import-Module` cmdlet adds one or more modules to the current session. Starting in PowerShell 3.0, installed modules are automatically imported to the session when you use any commands or providers in the module. However, you can still use the `Import-Module` command to import a module. You can disable automatic module importing using the `$PSModuleAutoloadingPreference` preference variable. For more information about the `$PSModuleAutoloadingPreference` variable, see about_Preference_Variables.

A module is a package that contains members that can be used in PowerShell. Members include cmdlets, providers, scripts, functions, variables, and other tools and files. After a module is imported, you can use the module members in your session. For more information about modules, see about_Modules.

By default, `Import-Module` imports all members that the module exports, but you can use the **Alias**, **Function**, **Cmdlet**, and **Variable** parameters to restrict which members are imported. The **NoClobber**

parameter prevents `Import-Module` from importing members that have the same names as members in the current session.

`Import-Module` imports a module only into the current session. To import the module into every new session, add an `Import-Module` command to your PowerShell profile. For more information about profiles, see about_Profiles.

You can manage remote Windows computers that have PowerShell remoting enabled by creating a **PSSession** on the remote computer. Then use the **PSSession** parameter of `Import-Module` to import the modules that are installed on the remote computer. When you use the imported commands in the current session the commands implicitly run on the remote computer.

Starting in Windows PowerShell 3.0, you can use `Import-Module` to import Common Information Model (CIM) modules. CIM modules define cmdlets in Cmdlet Definition XML (CDXML) files. This feature lets you use cmdlets that are implemented in non-managed code assemblies, such as those written in C++.

For remote computers that don't have PowerShell remoting enabled, including computers that aren't running the Windows operating system, you can use the **CIMSession** parameter of `Import-Module` to import CIM modules from the remote computer. The imported commands run implicitly on the remote computer. A **CIMSession** is a connection to Windows Management Instrumentation (WMI) on the remote computer.

Examples

Example 1: Import the members of a module into the current session

This example imports the members of the **PSDiagnostics** module into the current session.

```
Import-Module -Name PSDiagnostics
```

Example 2: Import all modules specified by the module path

This example imports all available modules in the path specified by the `$env:PSModulePath` environment variable into the current session.

```
Get-Module -ListAvailable | Import-Module
```

Example 3: Import the members of several modules into the current session

This example imports the members of the **PSDiagnostics** and **Dism** modules into the current session.

```
$m = Get-Module -ListAvailable PSDiagnostics, Dism
Import-Module -ModuleInfo $m
```

The `Get-Module` cmdlet gets the **PSDiagnostics** and **Dism** modules and saves the objects in the `$m` variable. The **ListAvailable** parameter is required when you're getting modules that aren't yet imported into the session.

The **ModuleInfo** parameter of `Import-Module` is used to import the modules into the current session.

Example 4: Import all modules specified by a path

This example uses an explicit path to identify the module to import.

```
Import-Module -Name c:\ps-test\modules\test -Verbose

VERBOSE: Loading module from path 'C:\ps-test\modules\Test\T
VERBOSE: Exporting function 'my-parm'.
VERBOSE: Exporting function 'Get-Parameter'.
VERBOSE: Exporting function 'Get-Specification'.
VERBOSE: Exporting function 'Get-SpecDetails'.
```

Using the **Verbose** parameter causes `Import-Module` to report progress as it loads the module. Without the **Verbose**, **PassThru**, or **AsCustomObject** parameter, `Import-Module` doesn't generate any output when it imports a module.

Example 5: Restrict module members imported into a session

This example shows how to restrict which module members are imported into the session and the effect of this command on the session. The **Function** parameter limits the members that are imported from the module. You can also use the **Alias**, **Variable**, and **Cmdlet** parameters to restrict other members that a module imports.

The `Get-Module` cmdlet gets the object that represents the **PSDiagnostics** module. The **ExportedCmdlets** property lists all the cmdlets that the module exports, even though they were not all imported.

```
Import-Module PSDiagnostics -Function Disable-PSTrace, Enabl
(Get-Module PSDiagnostics).ExportedCommands
```

```
Key                           Value
---                           -----
Disable-PSTrace               Disable-PSTrace
Disable-PSWSManCombinedTrace Disable-PSWSManCombinedTrace
Disable-WSManTrace            Disable-WSManTrace
Enable-PSTrace                Enable-PSTrace
Enable-PSWSManCombinedTrace   Enable-PSWSManCombinedTrace
Enable-WSManTrace             Enable-WSManTrace
Get-LogProperties             Get-LogProperties
Set-LogProperties             Set-LogProperties
Start-Trace                   Start-Trace
Stop-Trace                    Stop-Trace

Get-Command -Module PSDiagnostics

CommandType     Name                Version    Source
-----------     ----                -------    ------
Function        Disable-PSTrace     6.1.0.0    PSDiagnostic
Function        Enable-PSTrace      6.1.0.0    PSDiagnostic
```

Using the **Module** parameter of the `Get-Command` cmdlet shows the commands that were imported from the **PSDiagnostics** module. The results confirm that only the `Disable-PSTrace` and `Enable-PSTrace` cmdlets were imported.

Example 6: Import the members of a module and add a prefix

This example imports the **PSDiagnostics** module into the current session, adds a prefix to the member names, and then displays the prefixed member names. The **Prefix** parameter of `Import-Module` adds the `x` prefix to all members that are imported from the module. The prefix applies only to the members in the current session. It doesn't change the module. The **PassThru** parameter returns a module object that represents the imported module.

```
Import-Module PSDiagnostics -Prefix x -PassThru

ModuleType Version    Name                ExportedCommands
---------- -------    ----                ----------------
Script     6.1.0.0    PSDiagnostics       {Disable-xPSTrace,

Get-Command -Module PSDiagnostics

CommandType     Name                                     Versi
-----------     ----                                     -----
Function        Disable-xPSTrace                         6.1.0
Function        Disable-xPSWSManCombinedTrace            6.1.0
Function        Disable-xWSManTrace                      6.1.0
Function        Enable-xPSTrace                          6.1.0
Function        Enable-xPSWSManCombinedTrace             6.1.0
Function        Enable-xWSManTrace                       6.1.0
Function        Get-xLogProperties                       6.1.0
Function        Set-xLogProperties                       6.1.0
Function        Start-xTrace                             6.1.0
Function        Stop-xTrace                              6.1.0
```

`Get-Command` gets the members that have been imported from the module. The output shows that the module members were correctly prefixed.

Example 7: Get and use a custom object

This example demonstrates how to get and use the custom object returned by `Import-Module`.

Custom objects include synthetic members that represent each of the imported module members. For example, the cmdlets and functions in a module are converted to script methods of the custom object.

Custom objects are useful in scripting. They're also useful when several imported objects have the same names. Using the script method of an object is equivalent to specifying the fully qualified name of an imported member, including its module name.

The **AsCustomObject** parameter is only usable when importing a script module. Use `Get-Module` to determine which of the available modules

is a script module.

```
Get-Module -List | Format-Table -Property Name, ModuleType -

Name          ModuleType
----          ----------
Show-Calendar     Script
BitsTransfer    Manifest
PSDiagnostics   Manifest
TestCmdlets       Script
...

$a = Import-Module -Name Show-Calendar -AsCustomObject -Pass
$a | Get-Member

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()
Show-Calendar ScriptMethod System.Object Show-Calendar();

$a."Show-Calendar"()
```

The `Show-Calendar` script module is imported using the **AsCustomObject** parameter to request a custom object and the **PassThru** parameter to return the object. The resulting custom object is saved in the `$a` variable.

The `$a` variable is piped to the `Get-Member` cmdlet to show the properties and methods of the saved object. The output shows a `Show-Calendar` script method.

To call the `Show-Calendar` script method, the method name must be enclosed in quotation marks because the name includes a hyphen.

Example 8: Reimport a module into the same session

This example shows how to use the **Force** parameter of `Import-Module` when you're reimporting a module into the same session. The **Force** parameter removes the loaded module and then imports it again.

```
Import-Module PSDiagnostics
Import-Module PSDiagnostics -Force -Prefix PS
```

The first command imports the **PSDiagnostics** module. The second command imports the module again, this time using the **Prefix** parameter.

Without the **Force** parameter, the session would include two copies of each **PSDiagnostics** cmdlet, one with the standard name and one with the prefixed name.

Example 9: Run commands that have been hidden by imported commands

This example shows how to run commands that have been hidden by imported commands. The **TestModule** module includes a function named `Get-Date` that returns the year and day of the year.

```
Get-Date

Thursday, August 15, 2019 2:26:12 PM

Import-Module TestModule
Get-Date

19227
```

```
Get-Command Get-Date -All | Format-Table -Property CommandTy

CommandType     Name         ModuleName
-----------     ----         ----------
Function        Get-Date     TestModule
Cmdlet          Get-Date     Microsoft.PowerShell.Utility

Microsoft.PowerShell.Utility\Get-Date

Thursday, August 15, 2019 2:28:31 PM
```

The first `Get-Date` cmdlet returns a **DateTime** object with the current date. After importing the **TestModule** module, `Get-Date` returns the year and day of the year.

Using the **All** parameter of `Get-Command` show all the `Get-Date` commands in the session. The results show that there are two `Get-Date` commands in the session, a function from the **TestModule** module and a cmdlet from the **Microsoft.PowerShell.Utility** module.

Because functions take precedence over cmdlets, the `Get-Date` function from the **TestModule** module runs, instead of the `Get-Date` cmdlet. To run the original version of `Get-Date`, you must qualify the command name with the module name.

For more information about command precedence in PowerShell, see about_Command_Precedence.

Example 10: Import a minimum version of a module

This example imports the **PowerShellGet** module. It uses the **MinimumVersion** parameter of `Import-Module` to import only version `2.0.0` or greater of the module.

```
Import-Module -Name PowerShellGet -MinimumVersion 2.0.0
```

You can also use the **RequiredVersion** parameter to import a particular version of a module, or use the **Module** and **Version** parameters of the `#Requires` keyword to require a particular version of a module in a script.

Example 11: Import using a fully qualified name

This example imports a specific version of a module using the **FullyQualifiedName**.

```
PS> Get-Module -ListAvailable PowerShellGet | Select-Object

Name          Version
----          -------
PowerShellGet 2.2.1
PowerShellGet 2.1.3
PowerShellGet 2.1.2
PowerShellGet 1.0.0.1

PS> Import-Module -FullyQualifiedName @{ModuleName = 'PowerS
```

Example 12: Import using a fully qualified path

This example imports a specific version of a module using the fully qualified path.

```
PS> Get-Module -ListAvailable PowerShellGet | Select-Object

Path
----
C:\Program Files\PowerShell\Modules\PowerShellGet\2.2.1\Powe
C:\program files\powershell\6\Modules\PowerShellGet\PowerShe
C:\Program Files\WindowsPowerShell\Modules\PowerShellGet\2.1
```

```
C:\Program Files\WindowsPowerShell\Modules\PowerShellGet\1.0

PS> Import-Module -Name 'C:\Program Files\PowerShell\Modules
```

Example 13: Import a module from a remote computer

This example shows how to use the `Import-Module` cmdlet to import a module from a remote computer. This command uses the Implicit Remoting feature of PowerShell.

When you import modules from another session, you can use the cmdlets in the current session. However, commands that use the cmdlets run in the remote session.

```
$s = New-PSSession -ComputerName Server01
Get-Module -PSSession $s -ListAvailable -Name NetSecurity

ModuleType Name             ExportedCommands
---------- ----             ----------------
Manifest   NetSecurity      {New-NetIPsecAuthProposal, New-N

Import-Module -PSSession $s -Name NetSecurity
Get-Command -Module NetSecurity -Name Get-*Firewall*

CommandType     Name
-----------     ----
Function        Get-NetFirewallAddressFilter
Function        Get-NetFirewallApplicationFilter
Function        Get-NetFirewallInterfaceFilter
Function        Get-NetFirewallInterfaceTypeFilter
Function        Get-NetFirewallPortFilter
Function        Get-NetFirewallProfile
Function        Get-NetFirewallRule
Function        Get-NetFirewallSecurityFilter
Function        Get-NetFirewallServiceFilter
Function        Get-NetFirewallSetting

Get-NetFirewallRule -DisplayName "Windows Remote Management*
  Format-Table -Property DisplayName, Name -AutoSize
```

```
DisplayName                                                     Nam
-----------                                                     ---
Windows Remote Management (HTTP-In)                             WIN
Windows Remote Management (HTTP-In)                             WIN
Windows Remote Management - Compatibility Mode (HTTP-In)        WIN
```

`New-PSSession` creates a remote session (**PSSession**) to the `Server01` computer. The **PSSession** is saved in the `$s` variable.

Running `Get-Module` with the **PSSession** parameter shows that the **NetSecurity** module is installed and available on the remote computer. This command is equivalent to using the `Invoke-Command` cmdlet to run `Get-Module` command in the remote session. For example:

`Invoke-Command $s {Get-Module -ListAvailable -Name NetSecurity`

Running `Import-Module` with the **PSSession** parameter imports the **NetSecurity** module from the remote computer into the current session. The `Get-Command` cmdlet retrieves commands that begin with `Get` and include `Firewall` from the **NetSecurity** module. The output confirms that the module and its cmdlets were imported into the current session.

Next, the `Get-NetFirewallRule` cmdlet gets Windows Remote Management firewall rules on the `Server01` computer. This is equivalent to using the `Invoke-Command` cmdlet to run `Get-NetFirewallRule` on the remote session.

Example 14: Manage storage on a remote computer without the Windows operating system

In this example, the administrator of the computer has installed the Module Discovery WMI provider, which allows you to use CIM

commands that are designed for the provider.

The `New-CimSession` cmdlet creates a session on the remote computer named RSDGF03. The session connects to the WMI service on the remote computer. The CIM session is saved in the `$cs` variable. `Import-Module` uses the **CimSession** in `$cs` to import the **Storage** CIM module from the `RSDGF03` computer.

The `Get-Command` cmdlet shows the `Get-Disk` command in the **Storage** module. When you import a CIM module into the local session, PowerShell converts the CDXML files for each command into PowerShell scripts, which appear as functions in the local session.

Although `Get-Disk` is typed in the local session, the cmdlet implicitly runs on the remote computer from which it was imported. The command returns objects from the remote computer to the local session.

```
$cs = New-CimSession -ComputerName RSDGF03
Import-Module -CimSession $cs -Name Storage
# Importing a CIM module, converts the CDXML files for each
# PowerShell scripts. These appear as functions in the local
Get-Command Get-Disk

CommandType     Name                  ModuleName
-----------     ----                  ----------
Function        Get-Disk              Storage

# Use implicit remoting to query disks on the remote compute
# module was imported.
Get-Disk

Number Friendly Name           OperationalStatus  Total Size
------ -------------           -----------------  ----------
0      Virtual HD ATA Device   Online                  40 GB
```

Parameters

-Alias

Specifies the aliases that this cmdlet imports from the module into the current session. Enter a comma-separated list of aliases. Wildcard characters are permitted.

Some modules automatically export selected aliases into your session when you import the module. This parameter lets you select from among the exported aliases.

Expand table

Type:	String[]
Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	True

-ArgumentList

Specifies an array of arguments, or parameter values, that are passed to a script module during the `Import-Module` command. This parameter is valid only when you're importing a script module.

You can also refer to the **ArgumentList** parameter by its alias, **args**. For more information about the behavior of **ArgumentList**, see about_Splatting.

Expand table

Type:	Object[]
Aliases:	Args

Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-AsCustomObject

Indicates that this cmdlet returns a custom object with members that represent the imported module members. This parameter is valid only for script modules.

When you use the **AsCustomObject** parameter, `Import-Module` imports the module members into the session and then returns a **PSCustomObject** object instead of a **PSModuleInfo** object. You can save the custom object in a variable and use member-access enumeration to invoke the members.

Expand table

Type:	SwitchParameter
Position:	Named
Default value:	False
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-Assembly

Specifies an array of assembly objects. This cmdlet imports the cmdlets and providers implemented in the specified assembly

objects. Enter a variable that contains assembly objects or a command that creates assembly objects. You can also pipe an assembly object to `Import-Module`.

When you use this parameter, only the cmdlets and providers implemented by the specified assemblies are imported. If the module contains other files, they aren't imported, and you might be missing important members of the module. Use this parameter for debugging and testing the module, or when you're instructed to use it by the module author.

Expand table

Type:	Assembly[]
Position:	0
Default value:	None
Required:	True
Accept pipeline input:	True
Accept wildcard characters:	False

-CimNamespace

Specifies the namespace of an alternate CIM provider that exposes CIM modules. The default value is the namespace of the Module Discovery WMI provider.

Use this parameter to import CIM modules from computers and devices that aren't running a Windows operating system.

This parameter was introduced in Windows PowerShell 3.0.

Expand table

Type:	String

Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-CimResourceUri

Specifies an alternate location for CIM modules. The default value is the resource URI of the Module Discovery WMI provider on the remote computer.

Use this parameter to import CIM modules from computers and devices that aren't running a Windows operating system.

This parameter was introduced in Windows PowerShell 3.0.

Expand table

Type:	Uri
Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-CimSession

Specifies a CIM session on the remote computer. Enter a variable that contains the CIM session or a command that gets the CIM session, such as a Get-CimSession command.

`Import-Module` uses the CIM session connection to import modules from the remote computer into the current session. When you use the commands from the imported module in the current session, the commands run on the remote computer.

You can use this parameter to import modules from computers and devices that aren't running the Windows operating system, and Windows computers that have PowerShell, but don't have PowerShell remoting enabled.

This parameter was introduced in Windows PowerShell 3.0.

Expand table

Type:	CimSession
Position:	Named
Default value:	None
Required:	True
Accept pipeline input:	False
Accept wildcard characters:	False

-Cmdlet

Specifies an array of cmdlets that this cmdlet imports from the module into the current session. Wildcard characters are permitted.

Some modules automatically export selected cmdlets into your session when you import the module. This parameter lets you select from among the exported cmdlets.

Expand table

Type:	String[]
Position:	Named

Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	True

-DisableNameChecking

Indicates that this cmdlet suppresses the message that warns you when you import a cmdlet or function whose name includes an unapproved verb or a prohibited character.

By default, when a module that you import exports cmdlets or functions that have unapproved verbs in their names, PowerShell displays the following warning message:

> WARNING: Some imported command names include unapproved verbs which might make them less discoverable. Use the Verbose parameter for more detail or type Get-Verb to see the list of approved verbs.

This message is only a warning. The complete module is still imported, including the non-conforming commands. Although the message is displayed to module users, the naming problem should be fixed by the module author.

Expand table

Type:	SwitchParameter
Position:	Named
Default value:	False
Required:	False
Accept pipeline input:	False

Accept wildcard characters:	False

-Force

This parameter causes a module to be loaded, or reloaded, over top of the current one. Some modules load external assemblies. The import fails if you are importing a module that loads a newer version of an assembly. The **Force** parameter can't override the error. You must start a new session to load the new version.

Expand table

Type:	SwitchParameter
Position:	Named
Default value:	False
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-FullyQualifiedName

The value can be a module name, a full module specification, or a path to a module file.

When the value is a path, the path can be fully qualified or relative. A relative path is resolved relative to the script that contains the using statement.

When the value is a name or module specification, PowerShell searches the **PSModulePath** for the specified module.

A module specification is a hashtable that has the following keys.

- `ModuleName` - **Required** Specifies the module name.

- `GUID` - **Optional** Specifies the GUID of the module.
- It's also **Required** to specify at least one of the three below keys.
  - `ModuleVersion` - Specifies a minimum acceptable version of the module.
  - `MaximumVersion` - Specifies the maximum acceptable version of the module.
  - `RequiredVersion` - Specifies an exact, required version of the module. This can't be used with the other Version keys.

Expand table

Type:	ModuleSpecification[]
Position:	0
Default value:	None
Required:	True
Accept pipeline input:	True
Accept wildcard characters:	False

-Function

Specifies an array of functions that this cmdlet imports from the module into the current session. Wildcard characters are permitted. Some modules automatically export selected functions into your session when you import the module. This parameter lets you select from among the exported functions.

Expand table

Type:	String[]
Position:	Named
Default value:	None

Required:	False
Accept pipeline input:	False
Accept wildcard characters:	True

-Global

Indicates that this cmdlet imports modules into the global session state so they're available to all commands in the session.

By default, when `Import-Module` cmdlet is called from the command prompt, script file, or scriptblock, all the commands are imported into the global session state.

When invoked from another module, `Import-Module` cmdlet imports the commands in a module, including commands from nested modules, into the calling module's session state.

> **Tip**
>
> You should avoid calling `Import-Module` from within a module. Instead, declare the target module as a nested module in the parent module's manifest. Declaring nested modules improves the discoverability of dependencies.

The **Global** parameter is equivalent to the **Scope** parameter with a value of **Global**.

To restrict the commands that a module exports, use an `Export-ModuleMember` command in the script module.

Expand table

Type:	SwitchParameter
Position:	Named

Default value:	False
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-MaximumVersion

Specifies a maximum version. This cmdlet imports only a version of the module that's less than or equal to the specified value. If no version qualifies, `Import-Module` returns an error.

Expand table

Type:	String
Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-MinimumVersion

Specifies a minimum version. This cmdlet imports only a version of the module that's greater than or equal to the specified value. Use the **MinimumVersion** parameter name or its alias, **Version**. If no version qualifies, `Import-Module` generates an error.

To specify an exact version, use the **RequiredVersion** parameter. You can also use the **Module** and **Version** parameters of the **#Requires** keyword to require a specific version of a module in a script.

This parameter was introduced in Windows PowerShell 3.0.

Expand table

Type:	Version
Aliases:	Version
Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-ModuleInfo

Specifies an array of module objects to import. Enter a variable that contains the module objects, or a command that gets the module objects, such as the following command: `Get-Module -ListAvailable`. You can also pipe module objects to `Import-Module`.

Expand table

Type:	PSModuleInfo[]
Position:	0
Default value:	None
Required:	True
Accept pipeline input:	True
Accept wildcard characters:	False

-Name

Specifies the names of the modules to import. Enter the name of the module or the name of a file in the module, such as a `.psd1`, `.psm1`, `.dll`, or `.ps1` file. File paths are optional. Wildcard characters aren't permitted. You can also pipe module names and filenames to `Import-Module`.

If you omit a path, `Import-Module` looks for the module in the paths saved in the `$env:PSModulePath` environment variable.

Specify only the module name whenever possible. When you specify a filename, only the members that are implemented in that file are imported. If the module contains other files, they aren't imported, and you might be missing important members of the module.

> **Note**
>
> While it's possible to import a script (`.ps1`) file as a module, script files are usually not structured like script modules file (`.psm1`) file. Importing a script file doesn't guarantee that it's usable as a module. For more information, see about_Modules.

Expand table

Type:	String[]
Position:	0
Default value:	None
Required:	True
Accept pipeline input:	True
Accept wildcard characters:	True

-NoClobber

Prevents importing commands that have the same names as existing commands in the current session. By default, `Import-Module` imports all exported module commands.

Commands that have the same names can hide or replace commands in the session. To avoid command name conflicts in a session, use the **Prefix** or **NoClobber** parameters. For more information about name conflicts and command precedence, see "Modules and Name Conflicts" in about_Modules and about_Command_Precedence.

This parameter was introduced in Windows PowerShell 3.0.

Expand table

Type:	SwitchParameter
Aliases:	NoOverwrite
Position:	Named
Default value:	False
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-PassThru

Returns an object representing the imported module. By default, this cmdlet doesn't generate any output.

Expand table

Type:	SwitchParameter
Position:	Named

Default value:	False
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-Prefix

Specifies a prefix that this cmdlet adds to the nouns in the names of imported module members.

Use this parameter to avoid name conflicts that might occur when different members in the session have the same name. This parameter doesn't change the module, and it doesn't affect files that the module imports for its own use. These are known as nested modules. This cmdlet affects only the names of members in the current session.

For example, if you specify the prefix UTC and then import a `Get-Date` cmdlet, the cmdlet is known in the session as `Get-UTCDate`, and it's not confused with the original `Get-Date` cmdlet.

The value of this parameter takes precedence over the **DefaultCommandPrefix** property of the module, which specifies the default prefix.

Expand table

Type:	String
Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False

Accept wildcard characters:	False

-PSSession

Specifies a PowerShell user-managed session (**PSSession**) from which this cmdlet imports modules into the current session. Enter a variable that contains a **PSSession** or a command that gets a **PSSession**, such as a `Get-PSSession` command.

When you import a module from a different session into the current session, you can use the cmdlets from the module in the current session, just as you would use cmdlets from a local module. Commands that use the remote cmdlets run in the remote session, but the remoting details are managed in the background by PowerShell.

This parameter uses the Implicit Remoting feature of PowerShell. It's equivalent to using the `Import-PSSession` cmdlet to import particular modules from a session.

`Import-Module` can't import core PowerShell modules from another session. The core PowerShell modules have names that begin with Microsoft.PowerShell.

This parameter was introduced in Windows PowerShell 3.0.

Expand table

Type:	PSSession
Position:	Named
Default value:	None
Required:	True
Accept pipeline input:	False
Accept wildcard characters:	False

-RequiredVersion

Specifies a version of the module that this cmdlet imports. If the version isn't installed, `Import-Module` generates an error.

By default, `Import-Module` imports the module without checking the version number.

To specify a minimum version, use the **MinimumVersion** parameter. You can also use the **Module** and **Version** parameters of the **#Requires** keyword to require a specific version of a module in a script.

This parameter was introduced in Windows PowerShell 3.0.

Scripts that use **RequiredVersion** to import modules that are included with existing releases of the Windows operating system don't automatically run in future releases of the Windows operating system. This is because PowerShell module version numbers in future releases of the Windows operating system are higher than module version numbers in existing releases of the Windows operating system.

Expand table

Type:	Version
Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

-Scope

Specifies a scope to import the module in.

The acceptable values for this parameter are:

- **Global**. Available to all commands in the session. Equivalent to the **Global** parameter.
- **Local**. Available only in the current scope.

By default, when `Import-Module` cmdlet is called from the command prompt, script file, or scriptblock, all the commands are imported into the global session state. You can use the `-Scope Local` parameter to import module content into the script or scriptblock scope.

When invoked from another module, `Import-Module` cmdlet imports the commands in a module, including commands from nested modules, into the caller's session state. Specifying `-Scope Global` or `-Global` indicates that this cmdlet imports modules into the global session state so they're available to all commands in the session.

The **Global** parameter is equivalent to the **Scope** parameter with a value of **Global**.

This parameter was introduced in Windows PowerShell 3.0.

Expand table

Type:	String
Accepted values:	Local, Global
Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	False

`-Variable`

Specifies an array of variables that this cmdlet imports from the module into the current session. Enter a list of variables. Wildcard characters are permitted.

Some modules automatically export selected variables into your session when you import the module. This parameter lets you select from among the exported variables.

Expand table

Type:	String[]
Position:	Named
Default value:	None
Required:	False
Accept pipeline input:	False
Accept wildcard characters:	True

Inputs

String

You can pipe a module name to this cmdlet.

PSModuleInfo

You can pipe a module object to this cmdlet.

Assembly

You can pipe an assembly object to this cmdlet.

Outputs

None

By default, this cmdlet returns no output.

PSModuleInfo

If you specify the **PassThru** parameter, the cmdlet generates a **System.Management.Automation.PSModuleInfo** object that represents the imported module.

PSCustomObject

If you specify the **AsCustomObject** and **PassThru** parameters together, the cmdlet generates a **PSCustomObject** object that represents the module.

Notes

Windows PowerShell includes the following aliases for `Import-Module`:

- `ipmo`
- Before you can import a module, the module must be accessible to your local computer and included in the `PSModulePath` environmental variable. For more information, see about_Modules.

  You can also use the **PSSession** and **CIMSession** parameters to import modules that are installed on remote computers. However, commands that use the cmdlets in these modules run in the remote session on the remote computer.
- If you import members with the same name and the same type into your session, PowerShell uses the member imported last by default. Variables and aliases are replaced, and the originals aren't accessible. Functions, cmdlets, and providers are merely shadowed by the new members. They can be accessed by

qualifying the command name with the name of its snap-in, module, or function path.

- To update the formatting data for commands imported from a module, use the `Update-FormatData` cmdlet. If the formatting file for a module changes, use the `Update-FormatData` cmdlet to update the formatting data for imported commands. You don't need to import the module again.

- Starting in Windows PowerShell 3.0, the core commands installed with PowerShell are packaged in modules. In Windows PowerShell 2.0, and in host programs that create older-style sessions in later versions of PowerShell, the core commands are packaged in snap-ins (**PSSnapins**). The exception is **Microsoft.PowerShell.Core**, which is always a snap-in. Also, remote sessions, such as those started by the `New-PSSession` cmdlet, are older-style sessions that include core snap-ins.

  For information about the **CreateDefault2** method that creates newer-style sessions with core modules, see the CreateDefault2 Method.

- In Windows PowerShell 2.0, some property values of the module object, such as the **ExportedCmdlets** and **NestedModules** property values, weren't populated until the module was imported.

- If you attempt to import a module that contains mixed-mode assemblies that aren't compatible with Windows PowerShell 3.0+, `Import-Module` returns an error message like the following one.

  > Import-Module : Mixed mode assembly is built against version 'v2.0.50727' of the runtime and cannot be loaded in the 4.0 runtime without additional configuration information.

  This error occurs when a module that's designed for Windows PowerShell 2.0 contains at least one mixed-module assembly. A

mixed-module assembly that includes both managed and non-managed code, such as C++ and C#.

To import a module that contains mixed-mode assemblies, start Windows PowerShell 2.0 by using the following command, and then try the `Import-Module` command again.

```
PowerShell.exe -Version 2.0
```

- To use the CIM session feature, the remote computer must have WS-Management remoting and Windows Management Instrumentation (WMI), which is the Microsoft implementation of the Common Information Model (CIM) standard. The computer must also have the Module Discovery WMI provider or an alternate CIM provider that has the same basic features.

  You can use the CIM session feature on computers that aren't running a Windows operating system and on Windows computers that have PowerShell, but don't have PowerShell remoting enabled.

  You can also use the CIM parameters to get CIM modules from computers that have PowerShell remoting enabled, including the local computer. When you create a CIM session on the local computer, PowerShell uses DCOM, instead of WMI, to create the session.

- By default, `Import-Module` imports modules in the global scope even when called from a descendant scope. The top-level scope and all descendant scopes have access to the module's exported elements.

  In a descendant scope, `-Scope Local` limits the import to that scope and all its descendant scopes. Parent scopes then don't see the imported members.

  > **Note**

> `Get-Module` shows all modules loaded in the current session. This includes modules loaded locally in a descendant scope. Use `Get-Command -Module modulename` to see which members are loaded in the current scope.

- `Import-Module` doesn't load class and enumeration definitions in the module. Use the `using module` statement at the beginning of your script. This imports the module, including the class and enumeration definitions. For more information, see about_Using.
- During development of a script module, it's common to make changes to the code then load the new version of the module using `Import-Module` with the **Force** parameter. This works for changes to functions in the root module only. `Import-Module` doesn't reload any nested modules. Also, there's no way to load any updated classes or enumerations.

  To get updated module members defined in nested modules, remove the module with `Remove-Module`, then import the module again.

  If the module was loaded with a `using` statement, you must start a new session to import updated definitions for the classes and enumerations. Classes and enumerations defined in PowerShell and imported with a `using` statement can't be unloaded.

Related Links

- about_Modules
- Export-ModuleMember
- Get-Module
- New-Module
- Remove-Module

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback