about_Preference_Variables - PowerShell

By sdwheeler

Archived: 2026-04-06 00:17:45 UTC

Variables that customize the behavior of PowerShell.

PowerShell includes a set of variables that enable you to customize its behavior. These preference variables work like the options in GUI-based systems.

The preference variables affect the PowerShell operating environment and all commands run in the environment. Some cmdlets have parameters that allow you to override the preference behavior for a specific command.

The following table lists the preference variables and their default values.

Variable	Default Value
`$ConfirmPreference`	`High`
`$DebugPreference`	`SilentlyContinue`
`$ErrorActionPreference`	`Continue`
`$ErrorView`	`ConciseView`
`$FormatEnumerationLimit`	`4`
`$InformationPreference`	`SilentlyContinue`
`$LogCommandHealthEvent`	`$false` (not logged)
`$LogCommandLifecycleEvent`	`$false` (not logged)
`$LogEngineHealthEvent`	`$true` (logged)
`$LogEngineLifecycleEvent`	`$true` (logged)
`$LogProviderHealthEvent`	`$true` (logged)
`$LogProviderLifecycleEvent`	`$true` (logged)
`$MaximumHistoryCount`	`4096`
`$OFS`	Space character (`" "`)
`$OutputEncoding`	`UTF8Encoding` object
`$ProgressPreference`	`Continue`
`$PSDefaultParameterValues`	`@{}` (empty hash table)
`$PSEmailServer`	`$null` (none)

Variable	Default Value
$PSModuleAutoLoadingPreference	All
$PSNativeCommandArgumentPassing	Windows on Windows, Standard on Non-Windows
$PSNativeCommandUseErrorActionPreference	$false
$PSSessionApplicationName	'wsman'
$PSSessionConfigurationName	'http://schemas.microsoft.com/powershell/Microsoft.PowerShell'
$PSSessionOption	PSSessionOption object
$PSStyle	PSStyle object
$Transcript	$null (none)
$VerbosePreference	SilentlyContinue
$WarningPreference	Continue
$WhatIfPreference	$false

PowerShell includes the following environment variables that store user preferences. For more information about these environment variables, see about_Environment_Variables.

- $Env:PSExecutionPolicyPreference
- $Env:PSModulePath

Note

Changes to preference variables apply only in the scope they are made and any child scopes thereof. For example, you can limit the effects of changing a preference variable to a single function or script. For more information, see about_Scopes.

This document describes each of the preference variables.

To display the current value of a specific preference variable, type the variable's name. For example, the following command displays the $ConfirmPreference variable's value.

```
$ConfirmPreference
```

```
High
```

To change a variable's value, use an assignment statement. For example, the following statement changes the $ConfirmPreference parameter's value to **Medium**.

```
$ConfirmPreference = "Medium"
```

The values that you set are specific to the current PowerShell session. To make variables effective in all PowerShell sessions, add them to your PowerShell profile. For more information, see about_Profiles.

When you run commands on a remote computer, the remote commands are only subject to the preferences set in the remote computer's PowerShell client. For example, when you run a remote command, the value of the remote computer's `$DebugPreference` variable determines how PowerShell responds to debugging messages.

For more information about remote commands, see about_Remote.

Determines whether PowerShell automatically prompts you for confirmation before running a cmdlet or function.

The `$ConfirmPreference` variable takes one of the `ConfirmImpact` enumeration values: **High**, **Medium**, **Low**, or **None**.

Cmdlets and functions are assigned a risk of **High**, **Medium**, or **Low**. When the value of the `$ConfirmPreference` variable is less than or equal to the risk assigned to a cmdlet or function, PowerShell automatically prompts you for confirmation before running the cmdlet or function. For more information about assigning a risk to cmdlets or functions, see about_Functions_CmdletBindingAttribute.

If the value of the `$ConfirmPreference` variable is **None**, PowerShell never automatically prompts you before running a cmdlet or function.

To change the confirming behavior for all cmdlets and functions in the session, change `$ConfirmPreference` variable's value.

To override the `$ConfirmPreference` for a single command, use a cmdlet's or function's **Confirm** parameter. To request confirmation, use `-Confirm`. To suppress confirmation, use `-Confirm:$false`.

Valid values of `$ConfirmPreference`:

- **None**: PowerShell doesn't prompt automatically. To request confirmation of a particular command, use the **Confirm** parameter of the cmdlet or function.
- **Low**: PowerShell prompts for confirmation before running cmdlets or functions with a low, medium, or high risk.
- **Medium**: PowerShell prompts for confirmation before running cmdlets or functions with a medium, or high risk.
- **High**: PowerShell prompts for confirmation before running cmdlets or functions with a high risk.

PowerShell can automatically prompt you for confirmation before doing an action. For example, when cmdlet or function significantly affects the system to delete data or use a significant amount of system resources.

```
Remove-Item -Path C:\file.txt
```

```
Confirm
Are you sure you want to perform this action?
Performing operation "Remove File" on Target "C:\file.txt".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [?] Help (default is "Y"):
```

The estimate of the risk is an attribute of the cmdlet or function known as its **ConfirmImpact**. Users can't change it.

Cmdlets and functions that might pose a risk to the system have a **Confirm** parameter that you can use to request or suppress confirmation for a single command.

Most cmdlets and functions keep the default value of **Medium** for **ConfirmImpact**. `$ConfirmPreference` is set to **High** by default. Therefore, it's rare that commands automatically prompt for confirmation when users don't specify the **Confirm** parameter. To extend automatic confirmation prompting to more cmdlets and functions, set the value of `$ConfirmPreference` to **Medium** or **Low**.

This example shows the effect of the `$ConfirmPreference` variable's default value, **High**. The **High** value only confirms high-risk cmdlets and functions. Since most cmdlets and functions are medium risk, they aren't automatically confirmed and `Remove-Item` deletes the file. Adding `-Confirm` to the command prompts the user for confirmation.

```
$ConfirmPreference
```

```
High
```

```
Remove-Item -Path C:\temp1.txt
```

Use `-Confirm` to request confirmation.

```
Remove-Item -Path C:\temp2.txt -Confirm
```

```
Confirm
Are you sure you want to perform this action?
Performing operation "Remove File" on Target "C:\temp2.txt".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All
[?] Help (default is "Y"):
```

The following example shows the effect of changing the value of `$ConfirmPreference` to **Medium**. Because most cmdlets and function are medium risk, they're automatically confirmed. To suppress the confirmation prompt for a single command, use the **Confirm** parameter with a value of `$false`.

```
$ConfirmPreference = "Medium"
Remove-Item -Path C:\temp2.txt
```

```
Confirm
Are you sure you want to perform this action?
Performing operation "Remove File" on Target "C:\temp2.txt".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All
[?] Help (default is "Y"):
```

```
Remove-Item -Path C:\temp3.txt -Confirm:$false
```

Determines how PowerShell responds to debugging messages generated by a script, cmdlet or provider, or by a `Write-Debug` command at the command line.

The `$DebugPreference` variable takes one of the `ActionPreference` enumeration values: **SilentlyContinue**, **Stop**, **Continue**, **Inquire**, **Ignore**, **Suspend**, or **Break**.

Some cmdlets display debugging messages, which are typically technical messages designed for programmers and technical support professionals. By default, debugging messages aren't displayed, but you can display debugging messages by changing the value of `$DebugPreference`.

You can use the **Debug** common parameter of a cmdlet to display or hide the debugging messages for a specific command. For more information, see about_CommonParameters.

The valid values are as follows:

- **Break** - Enter the debugger when an error occurs or when an exception is raised.
- **Stop**: Displays the debug message and stops executing. Writes an error to the console.
- **Inquire**: Displays the debug message and asks you whether you want to continue.
- **Continue**: Displays the debug message and continues with execution.
- **SilentlyContinue**: (Default) No effect. The debug message isn't displayed and execution continues without interruption.

Adding the **Debug** common parameter to a command, when the command is configured to generate a debugging message, changes the value of the `$DebugPreference` variable to **Continue**.

The following examples show the effect of changing the values of `$DebugPreference` when a `Write-Debug` command is entered at the command line. The change affects all debugging messages, including messages generated by cmdlets and scripts. The examples show the **Debug** parameter, which displays or hides the debugging messages related to a single command.

This example shows the effect of the `$DebugPreference` variable's default value, **SilentlyContinue**. By default, the `Write-Debug` cmdlet's debug message isn't displayed and processing continues. When the **Debug** parameter is used, it overrides the preference for a single command. The debug message is displayed.

```
$DebugPreference
```

```
SilentlyContinue
```

```
Write-Debug -Message "Hello, World"
```

```
Write-Debug -Message "Hello, World" -Debug
```

```
DEBUG: Hello, World
```

This example shows the effect of `$DebugPreference` with the **Continue** value. The debug message is displayed and the command continues to process.

```
 $DebugPreference = "Continue"
Write-Debug -Message "Hello, World"
```

```
 DEBUG: Hello, World
```

This example uses the **Debug** parameter with a value of `$false` to suppress the message for a single command. The debug message isn't displayed.

```
 Write-Debug -Message "Hello, World" -Debug:$false
```

This example shows the effect of `$DebugPreference` being set to the **Stop** value. The debug message is displayed and the command is stopped.

```
 $DebugPreference = "Stop"
Write-Debug -Message "Hello, World"
```

```
 DEBUG: Hello, World
Write-Debug : The running command stopped because the preference variable
 "DebugPreference" or common parameter is set to Stop: Hello, World
At line:1 char:1
+ Write-Debug -Message "Hello, World"
```

This example uses the **Debug** parameter with a value of `$false` to suppress the message for a single command. The debug message isn't displayed and processing isn't stopped.

```
 Write-Debug -Message "Hello, World" -Debug:$false
```

This example shows the effect of `$DebugPreference` being set to the **Inquire** value. The debug message is displayed and the user is prompted for confirmation.

```
 $DebugPreference = "Inquire"
Write-Debug -Message "Hello, World"
```

```
 DEBUG: Hello, World

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [?] Help (default is "Y"):
```

This example uses the **Debug** parameter with a value of `$false` to suppress the message for a single command. The debug message isn't displayed and processing continues.

```
 Write-Debug -Message "Hello, World" -Debug:$false
```

Determines how PowerShell responds to a non-terminating error, an error that doesn't stop the cmdlet processing. For example, at the command line or in a script, cmdlet, or provider, such as the errors generated by the `Write-Error` cmdlet.

The `$ErrorActionPreference` variable takes one of the `ActionPreference` enumeration values: **SilentlyContinue**, **Stop**, **Continue**, **Inquire**, **Ignore**, **Suspend**, or **Break**.

You can use a cmdlet's **ErrorAction** common parameter to override the preference for a specific command.

The valid values are as follows:

- **Break** - Enter the debugger when an error occurs or when an exception is raised.
- **Continue**: (Default) Displays the error message and continues executing.
- **Ignore**: Suppresses the error message and continues to execute the command. Unlike **SilentlyContinue**, **Ignore** doesn't add the error message to the `$Error` automatic variable. The **Ignore** value is also valid for `$ErrorActionPreference`, where it suppresses both non-terminating and statement-terminating errors. However, **Ignore** only prevents `$Error` recording for non-terminating errors. Terminating errors that are suppressed by `Ignore` are still recorded in `$Error`.
- **Inquire**: Displays the error message and asks you whether you want to continue.
- **SilentlyContinue**: No effect. The error message isn't displayed and execution continues without interruption.
- **Stop**: Displays the error message and stops executing. In addition to the error generated, the **Stop** value generates an ActionPreferenceStopException object to the error stream.
- **Suspend**: Automatically suspends a workflow job to allow for further investigation. After investigation, the workflow can be resumed. The **Suspend** value is intended for per-command use, not for use as saved preference. **Suspend** isn't a valid value for the `$ErrorActionPreference` variable.

`$ErrorActionPreference` applies to **both** non-terminating and statement-terminating errors. Unlike the `-ErrorAction` parameter (which only affects non-terminating errors), the preference variable can also suppress or escalate errors generated by `$PSCmdlet.ThrowTerminatingError()`. When set to **Stop**, it escalates non-terminating errors to script-terminating errors. For more information about error categories, see about_Error_Handling. For more information about the **ErrorAction** common parameter, see about_CommonParameters.

Many native commands write to `stderr` as an alternative stream for additional information. This behavior can cause confusion when looking through errors or the additional output information can be lost to the user if `$ErrorActionPreference` is set to a state that mutes the output.

Beginning in PowerShell 7.2, error records redirected from native commands, like when using redirection operators (`2>&1`), aren't written to the `$Error` variable and the preference variable `$ErrorActionPreference` doesn't affect the redirected output.

PowerShell 7.4 added a feature that allows you to control how messages written to `stderr` are handled. For more information, see $PSNativeCommandUseErrorActionPreference.

These examples show the effect of the different values of the `$ErrorActionPreference` variable. The **ErrorAction** parameter is used to override the `$ErrorActionPreference` value.

This example shows the `$ErrorActionPreference` default value, **Continue**. A non-terminating error is generated. The message is displayed and processing continues.

```
 # Change the ErrorActionPreference to 'Continue'
$ErrorActionPreference = 'Continue'
# Generate a non-terminating error and continue processing the script.
Write-Error -Message  'Test Error' ; Write-Host 'Hello World'
```

```
 Write-Error: Test Error
Hello World
```

This example shows the `$ErrorActionPreference` default value, **Inquire**. An error is generated and a prompt for action is shown.

```
 # Change the ErrorActionPreference to 'Inquire'
$ErrorActionPreference = 'Inquire'
Write-Error -Message 'Test Error' ; Write-Host 'Hello World'
```

```
 Confirm
Test Error
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default is "Y"):
```

This example shows the `$ErrorActionPreference` set to **SilentlyContinue**. The error message is suppressed.

```
 # Change the ErrorActionPreference to 'SilentlyContinue'
$ErrorActionPreference = 'SilentlyContinue'
# Generate an error message
Write-Error -Message 'Test Error' ; Write-Host 'Hello World'
# Error message is suppressed and script continues processing
```

```
 Hello World
```

This example shows the `$ErrorActionPreference` set to **Stop**. It also shows the extra object generated to the `$Error` variable.

```
 # Change the ErrorActionPreference to 'Stop'
$ErrorActionPreference = 'Stop'
# Error message is generated and script stops processing
Write-Error -Message 'Test Error' ; Write-Host 'Hello World'

# Show the ActionPreferenceStopException and the error generated
$Error[0]
$Error[1]
```

```
 Write-Error: Test Error

ErrorRecord                : Test Error
```

```
WasThrownFromThrowStatement : False
TargetSite                  : System.Collections.ObjectModel.Collection`1[System.Management.Automation.PSObject]
                              Invoke(System.Collections.IEnumerable)
StackTrace                  :    at System.Management.Automation.Runspaces.PipelineBase.Invoke(IEnumerable input)
                                 at Microsoft.PowerShell.Executor.ExecuteCommandHelper(Pipeline tempPipeline,
                              Exception& exceptionThrown, ExecutionOptions options)
Message                     : The running command stopped because the preference variable "ErrorActionPreference" or
                              common parameter is set to Stop: Test Error
Data                        : {System.Management.Automation.Interpreter.InterpretedFrameInfo}
InnerException              :
HelpLink                    :
Source                      : System.Management.Automation
HResult                     : -2146233087

Write-Error: Test Error
```

Determines the display format of error messages in PowerShell.

The `$ErrorView` variable takes one of the `ErrorView` enumeration values: **NormalView**, **CategoryView**, or **ConciseView**.

The valid values are as follows:

- **ConciseView**: (Default) Provides a concise error message and a refactored view for advanced module builders. As of PowerShell 7.2, if the error is from the command line or a script module the output is a single line error message. Otherwise, you receive a multiline error message that contains the error and a pointer to the error showing where it occurs in that line. If the terminal supports Virtual Terminal, then ANSI color codes are used to provide color accent. The Accent color can be changed at `$Host.PrivateData.ErrorAccentColor`. Use `Get-Error` cmdlet for a comprehensive detailed view of the fully qualified error, including inner exceptions.

  ConciseView was added in PowerShell 7.

- **NormalView**: A detailed view designed for most users. Consists of a description of the error and the name of the object involved in the error.

- **CategoryView**: A succinct, structured view designed for production environments. The format is as follows:

  `{Category}: ({TargetName}:{TargetType}):[{Activity}], {Reason}`

For more information about the fields in **CategoryView**, see ErrorCategoryInfo class.

This example shows how an error appears when the value of `$ErrorView` is the default, **ConciseView**. `Get-ChildItem` is used to find a non-existent directory.

```
Get-ChildItem -Path 'C:\NoRealDirectory'
```

```
Get-ChildItem: Can't find path 'C:\NoRealDirectory' because it doesn't exist.
```

This example shows how an error appears when the value of `$ErrorView` is the default, **ConciseView**. `Script.ps1` is run and throws an error from `Get-Item` statement.

```
./Script.ps1
```

```
Get-Item: C:\Script.ps1
Line |
  11 | Get-Item -Path .\stuff
     | ^ Can't find path 'C:\demo\stuff' because it doesn't exist.
```

This example shows how an error appears when the value of `$ErrorView` is changed to **NormalView**. `Get-ChildItem` is used to find a non-existent file.

```
Get-ChildItem -Path C:\nofile.txt
```

```
Get-ChildItem : Can't find path 'C:\nofile.txt' because it doesn't exist.
At line:1 char:1
+ Get-ChildItem -Path C:\nofile.txt
```

This example shows how the same error appears when the value of `$ErrorView` is changed to **CategoryView**.

```
$ErrorView = "CategoryView"
Get-ChildItem -Path C:\nofile.txt
```

```
ObjectNotFound: (C:\nofile.txt:String) [Get-ChildItem], ItemNotFoundException
```

This example demonstrates that the value of `$ErrorView` only affects the error display. It doesn't change the structure of the error object that's stored in the `$Error` automatic variable. For information about the `$Error` automatic variable, see about_Automatic_Variables.

The following command takes the **ErrorRecord** object associated with the most recent error in the error array, **element 0**, and formats the properties of object in a list.

```
$Error[0] | Format-List -Property * -Force
```

```
PSMessageDetails      :
Exception             : System.Management.Automation.ItemNotFoundException:
                          Cannot find path 'C:\nofile.txt' because it does
                          not exist.
                        at System.Management.Automation.SessionStateInternal.
                          GetChildItems(String path, Boolean recurse, UInt32
                          depth, CmdletProviderContext context)
                        at System.Management.Automation.ChildItemCmdlet
                          ProviderIntrinsics.Get(String path, Boolean
```

```
                          recurse, UInt32 depth, CmdletProviderContext context)
                        at Microsoft.PowerShell.Commands.GetChildItemCommand.
                          ProcessRecord()
TargetObject          : C:\nofile.txt
CategoryInfo          : ObjectNotFound: (C:\nofile.txt:String) [Get-ChildItem],
                          ItemNotFoundException
FullyQualifiedErrorId : PathNotFound,
                          Microsoft.PowerShell.Commands.GetChildItemCommand
ErrorDetails          :
InvocationInfo        : System.Management.Automation.InvocationInfo
ScriptStackTrace      : at <ScriptBlock>, <No file>: line 1
PipelineIterationInfo : {0, 1}
```

Determines how many enumerated items are included in a display. This variable doesn't affect the underlying objects, only the display. When the value of `$FormatEnumerationLimit` is fewer than the number of enumerated items, PowerShell adds an ellipsis (`...`) to indicate items not shown.

Valid values: Integers (`Int32`)

Default value: 4

This example shows how to use the `$FormatEnumerationLimit` variable to improve the display of enumerated items.

The command in this example generates a table that lists all the services running on the computer in two groups: one for **running** services and one for **stopped** services. It uses a `Get-Service` command to get all the services, and then sends the results through the pipeline to the `Group-Object` cmdlet, which groups the results by the service status.

The result is a table that lists the status in the **Name** column, and the processes in the **Group** column. To change the column labels, use a hash table, see about_Hash_Tables. For more information, see the examples in Format-Table.

Find the current value of `$FormatEnumerationLimit` .

```
$FormatEnumerationLimit
```

```
4
```

List all services grouped by **Status**. There are a maximum of four services listed in the **Group** column for each status because `$FormatEnumerationLimit` has a value of **4**.

```
Get-Service | Group-Object -Property Status
```

```
 Count  Name       Group
-----  ----       -----
60     Running    {AdtAgent, ALG, Ati HotKey Poller, AudioSrv...}
41     Stopped    {Alerter, AppMgmt, aspnet_state, ATI Smart...}
```

To increase the number of items listed, increase the value of `$FormatEnumerationLimit` to **1000**. Use `Get-Service` and `Group-Object` to display the services.

```
 $FormatEnumerationLimit = 1000
Get-Service | Group-Object -Property Status
```

```
 Count  Name       Group
-----  ----       -----
60     Running    {AdtAgent, ALG, Ati HotKey Poller, AudioSrv, BITS, CcmExec...
41     Stopped    {Alerter, AppMgmt, aspnet_state, ATI Smart, Browser, CiSvc...
```

Use `Format-Table` with the **Wrap** parameter to display the list of services.

```
 Get-Service | Group-Object -Property Status | Format-Table -Wrap
```

```
 Count  Name       Group
-----  ----       -----
60     Running    {AdtAgent, ALG, Ati HotKey Poller, AudioSrv, BITS, CcmExec,
                  Client for NFS, CryptSvc, DcomLaunch, Dhcp, dmserver,
                  Dnscache, ERSvc, Eventlog, EventSystem, FwcAgent, helpsvc,
                  HidServ, IISADMIN, InoRPC, InoRT, InoTask, lanmanserver,
                  lanmanworkstation, LmHosts, MDM, Netlogon, Netman, Nla,
                  NtLmSsp, PlugPlay, PolicyAgent, ProtectedStorage, RasMan,
                  RemoteRegistry, RpcSs, SamSs, Schedule, seclogon, SENS,
                  SharedAccess, ShellHWDetection, SMT PSVC, Spooler,
                  srservice, SSDPSRV, stisvc, TapiSrv, TermService, Themes,
                  TrkWks, UMWdf, W32Time, W3SVC, WebClient, winmgmt, wscsvc,
                  wuauserv, WZCSVC, zzInterix}

41     Stopped    {Alerter, AppMgmt, aspnet_state, ATI Smart, Browser, CiSvc,
                  ClipSrv, clr_optimization_v2.0.50727_32, COMSysApp,
                  CronService, dmadmin, FastUserSwitchingCompatibility,
                  HTTPFilter, ImapiService, Mapsvc, Messenger, mnmsrvc,
                  MSDTC, MSIServer, msvsmon80, NetDDE, NetDDEdsdm, NtmsSvc,
                  NVSvc, ose, RasAuto, RDSessMgr, RemoteAccess, RpcLocator,
                  SCardSvr, SwPrv, SysmonLog, TlntSvr, upnphost, UPS, VSS,
                  WmdmPmSN, Wmi, WmiApSrv, xmlprov}
```

The `$InformationPreference` variable lets you set information stream preferences that you want displayed to users. Specifically, informational messages that you added to commands or scripts by adding the Write-Information cmdlet. If the **InformationAction** parameter is used, its value overrides the value of the `$InformationPreference` variable. `Write-Information` was introduced in PowerShell 5.0.

The `$InformationPreference` variable takes one of the `ActionPreference` enumeration values: **SilentlyContinue**, **Stop**, **Continue**, **Inquire**, **Ignore**, **Suspend**, or **Break**.

The valid values are as follows:

- **Break** - Enter the debugger when you write to the Information stream.
- **Stop**: Stops a command or script at an occurrence of the `Write-Information` command.
- **Inquire**: Displays the informational message that you specify in a `Write-Information` command, then asks whether you want to continue.
- **Continue**: Displays the informational message, and continues running.
- **SilentlyContinue**: (Default) No effect. The informational messages aren't displayed, and the script continues without interruption.

The **Log*Event** preference variables determine which types of events are written to the PowerShell event log in Event Viewer. By default, only engine and provider events are logged. But, you can use the **Log*Event** preference variables to customize your log, such as logging events about commands.

The **Log*Event** preference variables are as follows:

- `$LogCommandHealthEvent` : Logs errors and exceptions in command initialization and processing. The default is `$false` (not logged).
- `$LogCommandLifecycleEvent` : Logs the starting and stopping of commands and command pipelines and security exceptions in command discovery. The default is `$false` (not logged).
- `$LogEngineHealthEvent` : Logs errors and failures of sessions. The default is `$true` (logged).
- `$LogEngineLifecycleEvent` : Logs the opening and closing of sessions. The default is `$true` (logged).
- `$LogProviderHealthEvent` : Logs provider errors, such as read and write errors, lookup errors, and invocation errors. The default is `$true` (logged).
- `$LogProviderLifecycleEvent` : Logs adding and removing of PowerShell providers. The default is `$true` (logged). For information about PowerShell providers, see about_Providers.

To enable a **Log*Event**, type the variable with a value of `$true` , for example:

```
$LogCommandLifecycleEvent = $true
```

To disable an event type, type the variable with a value of `$false` , for example:

```
$LogCommandLifecycleEvent = $false
```

The events that you enable are effective only for the current PowerShell console. To apply the configuration to all consoles, save the variable settings in your PowerShell profile. For more information, see about_Profiles.

Determines how many commands are saved in the command history for the current session.

Valid values: 1 - 32768 (`Int32`)

Default: 4096

To determine the number of commands current saved in the command history, type:

```
(Get-History).Count
```

To see the commands saved in your session history, use the `Get-History` cmdlet. For more information, see about_History.

The Output Field Separator (OFS) specifies the character that separates the elements of an array that's converted to a string.

Valid values: Any string.

Default: Space

By default, the `$OFS` variable doesn't exist and the output file separator is a space, but you can add this variable and set it to any string. You can change the value of `$OFS` in your session, by typing `$OFS="<value>"`.

Note

If you're expecting the default value of a space (`" "`) in your script, module, or configuration output, be careful that the `$OFS` default value hasn't been changed elsewhere in your code.

This example shows that a space is used to separate the values when an array is converted to a string. In this case, an array of integers is stored in a variable and then the variable is cast as a string.

```
 $array = 1,2,3,4
[string]$array
```

```
 1 2 3 4
```

To change the separator, add the `$OFS` variable by assigning a value to it. The variable must be named `$OFS`.

```
 $OFS = "+"
[string]$array
```

```
 1+2+3+4
```

To restore the default behavior, you can assign a space (`" "`) to the value of `$OFS` or delete the variable. The following commands delete the variable and then verify that the separator is a space.

```
 Remove-Variable OFS
[string]$array
```

```
 1 2 3 4
```

Determines the character encoding method that PowerShell uses when piping data into native applications.

Note

In the majority of scenarios, the value for `$OutputEncoding` should align to the value of `[Console]::InputEncoding`.

The valid values are as follows: Objects derived from an Encoding class, such as **ASCIIEncoding**, **UTF7Encoding**, **UTF8Encoding**, **UTF32Encoding**, and **UnicodeEncoding**.

Default: UTF8Encoding object.

The first command finds the value of `$OutputEncoding`. Because the value is an encoding object, display only its **EncodingName** property.

```
$OutputEncoding.EncodingName
```

The remaining examples use the following PowerShell script saved as `hexdump.ps1` to illustrate the behavior of `$OutputEncoding`.

```
$inputStream = [Console]::OpenStandardInput()
try {
    $buffer = [byte[]]::new(1024)
    $read = $inputStream.Read($buffer, 0, $buffer.Length)
    Format-Hex -InputObject $buffer -Count $read
} finally {
    $inputStream.Dispose()
}
```

The following example shows how the string value `café` is encoded to bytes when piped into `hexdump.ps1` created above. It demonstrates that the string value is encoded using the **UTF8Encoding** scheme.

```
'café' | pwsh -File ./hexdump.ps1
```

```
   Label: Byte[] (System.Byte[]) <28873E25>

         Offset Bytes                                           Ascii
                00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
         ------ ----------------------------------------------- -----
0000000000000000 63 61 66 C3 A9 0D 0A                           cafÃ©��
```

The following example shows how the bytes change when changing the encoding to **UnicodeEncoding**.

```
$OutputEncoding = [System.Text.Encoding]::Unicode
'café' | pwsh -File ./hexdump.ps1
```

```
   Label: Byte[] (System.Byte[]) <515A7DC3>

         Offset Bytes                                           Ascii
                00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
         ------ ----------------------------------------------- -----
0000000000000000 FF FE 63 00 61 00 66 00 E9 00 0D 00 0A 00      ÿþc a f é ♪ ♪
```

Determines how PowerShell responds to progress updates generated by a script, cmdlet, or provider, such as the progress bars generated by the Write-Progress cmdlet. The `Write-Progress` cmdlet creates progress bars that show a command's status.

The `$ProgressPreference` variable takes one of the `ActionPreference` enumeration values: **SilentlyContinue**, **Stop**, **Continue**, **Inquire**, **Ignore**, **Suspend**, or **Break**.

The valid values are as follows:

- **Break** - Enter the debugger when you write to the Progress stream.
- **Stop**: Doesn't display the progress bar. Instead, it displays an error message and stops executing.
- **Inquire**: Doesn't display the progress bar. Prompts for permission to continue. If you reply with `Y` or `A`, it displays the progress bar.
- **Continue**: (Default) Displays the progress bar and continues with execution.
- **SilentlyContinue**: Executes the command, but doesn't display the progress bar.

Specifies default values for the parameters of cmdlets and advanced functions. The value of `$PSDefaultParameterValues` is a hash table where the key consists of the cmdlet name and parameter name separated by a colon (`:`). The value is a custom default value that you specify.

`$PSDefaultParameterValues` was introduced in PowerShell 3.0.

For more information about this preference variable, see about_Parameters_Default_Values.

Specifies the default email server that's used to send email messages. This preference variable is used by cmdlets that send email, such as the Send-MailMessage cmdlet.

Enables and disables automatic importing of modules in the session. The `$PSModuleAutoLoadingPreference` variable doesn't exist by default. The default behavior when the variable isn't defined is the same as `$PSModuleAutoLoadingPreference = 'All'`.

To automatically import a module, get or use a command contained in the module.

The `$PSModuleAutoLoadingPreference` variable takes one of the `PSModuleAutoLoadingPreference` enumeration values:

- `All` : Modules are imported automatically on first-use.
- `ModuleQualified` : Modules are imported automatically only when a user uses the module-qualified name of a command in the module. For example, if the user types `MyModule\MyCommand`, PowerShell imports the **MyModule** module.
- `None` : Disables the automatic importing of modules. To import a module, use the `Import-Module` cmdlet.

For more information about automatic importing of modules, see about_Modules.

PowerShell 7.3 changed the way it parses the command line for native commands. The new `$PSNativeCommandArgumentPassing` preference variable controls this behavior.

Caution

The new behavior is a **breaking change** from the previous behavior. This may break scripts and automation that work around the various issues when invoking native applications.

The automatic variable `$PSNativeCommandArgumentPassing` allows you to select the behavior at runtime. The valid values are `Legacy`, `Standard`, and `Windows`. `Legacy` is the historic behavior.

The `$PSNativeCommandArgumentPassing` variable is defined by default but the value is platform specific.

- On Windows, the preference is set to `Windows`.
- On non-Windows platforms, the preference is set to `Standard`.
- If you have removed the `$PSNativeCommandArgumentPassing` variable, PowerShell uses the `Standard` behavior.

The behavior of `Windows` and `Standard` mode are the same except, in `Windows` mode, PowerShell uses the `Legacy` behavior of argument passing when you run the following files.

- `cmd.exe`
- `cscript.exe`
- `find.exe`
- `sqlcmd.exe`
- `wscript.exe`
- Files ending with:
  - `.bat`
  - `.cmd`
  - `.js`
  - `.vbs`
  - `.wsf`

If the `$PSNativeCommandArgumentPassing` is set to either `Legacy` or `Standard`, the parser doesn't check for these files. For examples of the new behavior, see about_Parsing.

PowerShell 7.3 also added the ability to trace parameter binding for native commands. For more information, see Trace-Command.

When `$PSNativeCommandUseErrorActionPreference` is `$true`, native commands with non-zero exit codes issue errors according to `$ErrorActionPreference`.

Some native commands, like robocopy use non-zero exit codes to represent information other than errors. In these cases, you can temporarily disable the behavior and prevent non-zero exit codes from issuing errors.

```
& {
    # Disable $PSNativeCommandUseErrorActionPreference for this scriptblock
    $PSNativeCommandUseErrorActionPreference = $false
    robocopy.exe D:\reports\operational "\\reporting\ops" CY2022Q4.md
    if ($LASTEXITCODE -gt 8) {
        throw "robocopy failed with exit code $LASTEXITCODE"
    }
}
```

In this example, the `$PSNativeCommandUseErrorActionPreference` variable is changed inside a scriptblock. The change is local to the scriptblock. When the scriptblock exits, the variable reverts to its previous value.

Specifies the default application name for a remote command that uses Web Services for Management (WS-Management) technology. For more information, see About Windows Remote Management.

The system default application name is `WSMAN`, but you can use this preference variable to change the default.

The application name is the last node in a connection URI. For example, the application name in the following sample URI is `WSMAN`.

```
http://Server01:8080/WSMAN
```

The default application name is used when the remote command doesn't specify a connection URI or an application name.

The **WinRM** service uses the application name to select a listener to service the connection request. The parameter's value should match the value of the **URLPrefix** property of a listener on the remote computer.

To override the system default and the value of this variable, and select a different application name for a particular session, use the **ConnectionURI** or **ApplicationName** parameters of the New-PSSession, Enter-PSSession, or Invoke-Command cmdlets.

The `$PSSessionApplicationName` preference variable is set on the local computer, but it specifies a listener on the remote computer. If the application name that you specify doesn't exist on the remote computer, the command to establish the session fails.

Specifies the default session configuration that's used to create new sessions in the current session.

This preference variable is set on the local computer, but it specifies a session configuration that's located on the remote computer.

The value of the `$PSSessionConfigurationName` variable is a fully qualified resource URI.

The default value `http://schemas.microsoft.com/PowerShell/microsoft.PowerShell` indicates the **Microsoft.PowerShell** session configuration on the remote computer.

If you specify only a configuration name, the following schema URI is prepended:

```
http://schemas.microsoft.com/PowerShell/
```

You can override the default and select a different session configuration for a particular session by using the **ConfigurationName** parameter of the `New-PSSession`, `Enter-PSSession`, or `Invoke-Command` cmdlets.

You can change the value of this variable at any time. When you do, remember that the session configuration that you select must exist on the remote computer. If it doesn't, the command to create a session that uses the session configuration fails.

This preference variable doesn't determine which local session configurations are used when remote users create a session that connects to this computer. However, you can use the permissions for the local session configurations to determine which users may use them.

Establishes the default values for advanced user options in a remote session. These option preferences override the system default values for session options.

The `$PSSessionOption` variable contains a **PSSessionOption** object. For more information, see System.Management.Automation.Remoting.PSSessionOption. Each property of the object represents a session option. For example, the **NoCompression** property turns of data compression during the session.

By default, the `$PSSessionOption` variable contains a **PSSessionOption** object with the default values for all options, as shown below.

```
 MaximumConnectionRedirectionCount : 5
NoCompression                     : False
NoMachineProfile                  : False
ProxyAccessType                   : None
ProxyAuthentication               : Negotiate
ProxyCredential                   :
SkipCACheck                       : False
SkipCNCheck                       : False
SkipRevocationCheck               : False
OperationTimeout                  : 00:03:00
NoEncryption                      : False
UseUTF16                          : False
IncludePortInSPN                  : False
OutputBufferingMode               : None
Culture                           :
UICulture                         :
MaximumReceivedDataSizePerCommand :
MaximumReceivedObjectSize         : 209715200
ApplicationArguments              :
OpenTimeout                       : 00:03:00
CancelTimeout                     : 00:01:00
IdleTimeout                       : -00:00:00.0010000
```

For descriptions of these options and more information, see New-PSSessionOption. For more information about remote commands and sessions, see about_Remote and about_PSSessions.

To change the value of the `$PSSessionOption` preference variable, use the `New-PSSessionOption` cmdlet to create a **PSSessionOption** object with the option values you prefer. Save the output in a variable called `$PSSessionOption`.

```
$PSSessionOption = New-PSSessionOption -NoCompression
```

To use the `$PSSessionOption` preference variable in every PowerShell session, add a `New-PSSessionOption` command that creates the `$PSSessionOption` variable to your PowerShell profile. For more information, see about_Profiles.

You can set custom options for a particular remote session. The options that you set take precedence over the system defaults and the value of the `$PSSessionOption` preference variable.

To set custom session options, use the `New-PSSessionOption` cmdlet to create a **PSSessionOption** object. Then, use the **PSSessionOption** object as the value of the **SessionOption** parameter in cmdlets that create a session, such as `New-PSSession`, `Enter-PSSession`, and `Invoke-Command`.

As of PowerShell 7.2 you can now access the `$PSStyle` automatic variable to view and change the rendering of ANSI string output. `$PSStyle` is an instance of the **PSStyle** class. The members of this class define strings containing ANSI escape sequences that control the rendering of text in the terminal.

The base members return strings of ANSI escape sequences mapped to their names. The values are settable to allow customization. The property names make it easier for you to create decorated strings using tab completion. For example:

```
"$($PSStyle.Background.BrightCyan)Power$($PSStyle.Underline)$($PSStyle.Bold)Shell$($PSStyle.Reset)"
```

The **Background** and **Foreground** members also have a `FromRgb()` method to specify 24-bit color.

For more information about `$PSStyle`, see about_ANSI_Terminals.

Used by `Start-Transcript` to specify the name and location of the transcript file. If you don't specify a value for the **Path** parameter, `Start-Transcript` uses the path in the value of the `$Transcript` global variable. If you haven't created this variable, `Start-Transcript` stores the transcripts in the following location using the default name:

- On Windows: `$HOME\Documents`
- On Linux or macOS: `$HOME`

The default filename is: `PowerShell_transcript.<computername>.<random>.<timestamp>.txt`.

Determines how PowerShell responds to verbose messages generated by a script, cmdlet, or provider, such as the messages generated by the Write-Verbose cmdlet. Verbose messages describe the actions performed to execute a command.

By default, verbose messages aren't displayed, but you can change this behavior by changing the value of `$VerbosePreference`.

The `$VerbosePreference` variable takes one of the `ActionPreference` enumeration values: **SilentlyContinue**, **Stop**, **Continue**, **Inquire**, **Ignore**, **Suspend**, or **Break**.

The valid values are as follows:

- **Break** - Enter the debugger when you write to the Verbose stream.
- **Stop**: Displays the verbose message and an error message and then stops executing.
- **Inquire**: Displays the verbose message and then displays a prompt that asks you whether you want to continue.
- **Continue**: Displays the verbose message and then continues with execution.
- **SilentlyContinue**: (Default) Doesn't display the verbose message. Continues executing.

You can use the **Verbose** common parameter of a cmdlet to display or hide the verbose messages for a specific command. For more information, see about_CommonParameters.

These examples show the effect of the different values of `$VerbosePreference` and the **Verbose** parameter to override the preference value.

This example shows the effect of the **SilentlyContinue** value, that's the default. The command uses the **Message** parameter, but doesn't write a message to the PowerShell console.

```
Write-Verbose -Message "Verbose message test."
```

When the **Verbose** parameter is used, the message is written.

```
Write-Verbose -Message "Verbose message test." -Verbose
```

```
VERBOSE: Verbose message test.
```

This example shows the effect of the **Continue** value. The `$VerbosePreference` variable is set to **Continue** and the message is displayed.

```
$VerbosePreference = "Continue"
Write-Verbose -Message "Verbose message test."
```

```
VERBOSE: Verbose message test.
```

This example uses the **Verbose** parameter with a value of `$false` that overrides the **Continue** value. The message isn't displayed.

```
Write-Verbose -Message "Verbose message test." -Verbose:$false
```

This example shows the effect of the **Stop** value. The `$VerbosePreference` variable is set to **Stop** and the message is displayed. The command is stopped.

```
$VerbosePreference = "Stop"
Write-Verbose -Message "Verbose message test."
```

```
VERBOSE: Verbose message test.
Write-Verbose : The running command stopped because the preference variable
  "VerbosePreference" or common parameter is set to Stop: Verbose message test.
At line:1 char:1
+ Write-Verbose -Message "Verbose message test."
```

This example uses the **Verbose** parameter with a value of `$false` that overrides the **Stop** value. The message isn't displayed.

```
Write-Verbose -Message "Verbose message test." -Verbose:$false
```

This example shows the effect of the **Inquire** value. The `$VerbosePreference` variable is set to **Inquire**. The message is displayed and the user is prompted for confirmation.

```
 $VerbosePreference = "Inquire"
Write-Verbose -Message "Verbose message test."
```

```
 VERBOSE: Verbose message test.

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [?] Help (default is "Y"):
```

This example uses the **Verbose** parameter with a value of `$false` that overrides the **Inquire** value. The user isn't prompted and the message isn't displayed.

```
 Write-Verbose -Message "Verbose message test." -Verbose:$false
```

Determines how PowerShell responds to warning messages generated by a script, cmdlet, or provider, such as the messages generated by the Write-Warning cmdlet.

By default, warning messages are displayed and execution continues, but you can change this behavior by changing the value of `$WarningPreference`.

The `$WarningPreference` variable takes one of the `ActionPreference` enumeration values: **SilentlyContinue**, **Stop**, **Continue**, **Inquire**, **Ignore**, **Suspend**, or **Break**.

The valid values are as follows:

- **Break** - Enter the debugger when a warning message is written.
- **Stop**: Displays the warning message and an error message and then stops executing.
- **Inquire**: Displays the warning message and then prompts for permission to continue.
- **Continue**: (Default) Displays the warning message and then continues executing.
- **SilentlyContinue**: Doesn't display the warning message. Continues executing.

You can use the **WarningAction** common parameter of a cmdlet to determine how PowerShell responds to warnings from a particular command. For more information, see about_CommonParameters.

These examples show the effect of the different values of `$WarningPreference`. The **WarningAction** parameter overrides the preference value.

This example shows the effect of the default value, **Continue**.

```
 $m = "This action can delete data."
Write-Warning -Message $m
```

```
 WARNING: This action can delete data.
```

This example uses the **WarningAction** parameter with the value **SilentlyContinue** to suppress the warning. The message isn't displayed.

```
 $m = "This action can delete data."
Write-Warning -Message $m -WarningAction SilentlyContinue
```

This example changes the `$WarningPreference` variable to the **SilentlyContinue** value. The message isn't displayed.

```
 $WarningPreference = "SilentlyContinue"
$m = "This action can delete data."
Write-Warning -Message $m
```

This example uses the **WarningAction** parameter to stop when a warning is generated.

```
 $m = "This action can delete data."
Write-Warning -Message $m -WarningAction Stop
```

```
 WARNING: This action can delete data.
Write-Warning : The running command stopped because the preference variable
  "WarningPreference" or common parameter is set to Stop:
    This action can delete data.
At line:1 char:1
+ Write-Warning -Message $m -WarningAction Stop
```

This example changes the `$WarningPreference` variable to the **Inquire** value. The user is prompted for confirmation.

```
 $WarningPreference = "Inquire"
$m = "This action can delete data."
Write-Warning -Message $m
```

```
 WARNING: This action can delete data.

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [?] Help (default is "Y"):
```

This example uses the **WarningAction** parameter with the value **SilentlyContinue**. The command continues to execute and no message is displayed.

```
 $m = "This action can delete data."
Write-Warning -Message $m -WarningAction SilentlyContinue
```

This example changes the `$WarningPreference` value to **Stop**.

```
 $WarningPreference = "Stop"
$m = "This action can delete data."
Write-Warning -Message $m
```

```
 WARNING: This action can delete data.
Write-Warning : The running command stopped because the preference variable
  "WarningPreference" or common parameter is set to Stop:
    This action can delete data.
At line:1 char:1
+ Write-Warning -Message $m
```

This example uses the **WarningAction** with the **Inquire** value. The user is prompted when a warning occurs.

```
 $m = "This action can delete data."
Write-Warning -Message $m -WarningAction Inquire
```

```
 WARNING: This action can delete data.

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [?] Help (default is "Y"):
```

Determines whether **WhatIf** is automatically enabled for every command that supports it. When **WhatIf** is enabled, the cmdlet reports the expected effect of the command, but doesn't execute the command.

The valid values are as follows:

- **False** (**0**, not enabled): (Default) **WhatIf** isn't automatically enabled. To enable it manually, use the cmdlet's **WhatIf** parameter.
- **True** (**1**, enabled): **WhatIf** is automatically enabled on any command that supports it. Users can use the **WhatIf** parameter with a value of **False** to disable it manually, such as `-WhatIf:$false`.

These examples show the effect of the different values of `$WhatIfPreference`. They show how to use the **WhatIf** parameter to override the preference value for a specific command.

This example shows the effect of the `$WhatIfPreference` variable set to the default value, **False**. Use `Get-ChildItem` to verify that the file exists. `Remove-Item` deletes the file. After the file is deleted, you can verify the deletion with `Get-ChildItem`.

```
 Get-ChildItem -Path .\test.txt
Remove-Item -Path ./test.txt
```

```
    Directory: C:\Test


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---           9/13/2019    10:53             10 test.txt
```

```
 Get-ChildItem -Path .\test.txt
```

```
 Get-ChildItem : Cannot find path 'C:\Test\test.txt' because it does not exist.
At line:1 char:1
+ Get-ChildItem -File test.txt
```

This example shows the effect of using the **WhatIf** parameter when the value of `$WhatIfPreference` is **False**.

Verify that the file exists.

```
 Get-ChildItem -Path .\test2.txt
```

```
    Directory: C:\Test

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---           2/28/2019    17:06             12 test2.txt
```

Use the **WhatIf** parameter to determine the result of attempting to delete the file.

```
 Remove-Item -Path .\test2.txt -WhatIf
```

```
 What if: Performing the operation "Remove File" on target "C:\Test\test2.txt".
```

Verify that the file wasn't deleted.

```
 Get-ChildItem -Path .\test2.txt
```

```
    Directory: C:\Test

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---           2/28/2019    17:06             12 test2.txt
```

This example shows the effect of the `$WhatIfPreference` variable set to the value, **True**. When you use `Remove-Item` to delete a file, the file's path is displayed, but the file isn't deleted.

Attempt to delete a file. A message is displayed about what would happen if `Remove-Item` was run, but the file isn't deleted.

```
 $WhatIfPreference = "True"
Remove-Item -Path .\test2.txt
```

```
 What if: Performing the operation "Remove File" on target "C:\Test\test2.txt".
```

Use `Get-ChildItem` to verify that the file wasn't deleted.

```
Get-ChildItem -Path .\test2.txt
```

```
    Directory: C:\Test

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---           2/28/2019    17:06             12 test2.txt
```

This example shows how to delete a file when the value of `$WhatIfPreference` is **True**. It uses the **WhatIf** parameter with a value of `$false`. Use `Get-ChildItem` to verify the file was deleted.

```
Remove-Item -Path .\test2.txt -WhatIf:$false
Get-ChildItem -Path .\test2.txt
```

```
Get-ChildItem : Cannot find path 'C:\Test\test2.txt' because it does not exist.
At line:1 char:1
+ Get-ChildItem -Path .\test2.txt
```

The following are examples of the `Get-Process` cmdlet that doesn't support **WhatIf** and `Stop-Process` that does support **WhatIf**. The `$WhatIfPreference` variable's value is **True**.

`Get-Process` doesn't support **WhatIf**. When the command is executed, it displays the **Winword** process.

```
Get-Process -Name Winword
```

```
 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
------    -----      -----     ------      --  -- -----------
   130   119.84     173.38       8.39   15024   4 WINWORD
```

`Stop-Process` does support **WhatIf**. The **Winword** process isn't stopped.

```
Stop-Process -Name Winword
```

```
What if: Performing the operation "Stop-Process" on target "WINWORD (15024)".
```

You can override the `Stop-Process` **WhatIf** behavior by using the **WhatIf** parameter with a value of `$false`. The **Winword** process is stopped.

```
Stop-Process -Name Winword -WhatIf:$false
```

To verify that the **Winword** process was stopped, use `Get-Process`.

```
Get-Process -Name Winword
```

```
Get-Process : Cannot find a process with the name "Winword".
 Verify the process name and call the cmdlet again.
At line:1 char:1
+ Get-Process -Name Winword
```

- about_Automatic_Variables
- about_CommonParameters
- about_Environment_Variables
- about_Error_Handling
- about_Profiles
- about_Remote
- about_Scopes
- about_Variables

Source: https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_preference_variables?view=powershell-7.3#debugpreference