# about_Variables - PowerShell | Microsoft Learn

13-16 minutes

---

- Article
- 03/07/2024
-

## In this article

## Short description

Describes how variables store values that can be used in PowerShell.

## Long description

You can store all types of values in PowerShell variables. For example, store the results of commands, and store elements that are used in commands and expressions, such as names, paths, settings, and values.

A variable is a unit of memory in which values are stored. In PowerShell, variables are represented by text strings that begin with a dollar sign (`$`), such as `$a`, `$process`, or `$my_var`.

Variable names aren't case-sensitive, and can include spaces and special characters. But, variable names that include special characters and spaces are difficult to use and should be avoided. For more information, see Variable names that include special characters.

There are several different types of variables in PowerShell.

- User-created variables: User-created variables are created and maintained by the user. By default, the variables that you create at the PowerShell command line exist only while the PowerShell window is open. When the PowerShell windows is closed, the variables are deleted. To save a variable, add it to your PowerShell profile. You can also create variables in scripts with global, script, or local scope.

- Automatic variables: Automatic variables store the state of PowerShell. These variables are created by PowerShell, and PowerShell changes their values as required to maintain their accuracy. Users can't change the value of these variables. For example, the `$PSHOME` variable stores the path to the PowerShell installation directory.

For more information, a list, and a description of the automatic variables, see about_Automatic_Variables.

- Preference variables: Preference variables store user preferences for PowerShell. These variables are created by PowerShell and are populated with default values. Users can change the values of these variables. For example, the `$MaximumHistoryCount` variable determines the maximum number of entries in the session history.

  For more information, a list, and a description of the preference variables, see about_Preference_Variables.

## Working with variables

To create a new variable, use an assignment statement to assign a value to the variable. You don't have to declare the variable before using it. The default value of all variables is `$null`.

To get a list of all the variables in your PowerShell session, type `Get-Variable`. The variable names are displayed without the preceding dollar (`$`) sign that is used to reference variables.

For example:

```
$MyVariable = 1, 2, 3

$Path = "C:\Windows\System32"
```

Variables are useful for storing the results of commands.

For example:

```
$Processes = Get-Process

$Today = (Get-Date).DateTime
```

To display the value of a variable, type the variable name, preceded by a dollar sign (`$`).

For example:

```
$MyVariable
```

```
1
2
3
```

```
$Today
```

```
Tuesday, September 3, 2019 09:46:46
```

To change the value of a variable, assign a new value to the variable.

The following examples display the value of the `$MyVariable` variable, changes the value of the variable, and then displays the new value.

```
$MyVariable = 1, 2, 3
$MyVariable
```

```
1
2
3
```

```
$MyVariable = "The green cat."
$MyVariable
```

```
The green cat.
```

To delete the value of a variable, use the `Clear-Variable` cmdlet or change the value to `$null`.

```
Clear-Variable -Name MyVariable
```

```
$MyVariable = $null
```

To delete the variable, use Remove-Variable or Remove-Item.

```
Remove-Variable -Name MyVariable
```

```
Remove-Item -Path Variable:\MyVariable
```

It is also possible to assign values to multiple variables with one statement. The following examples assigns the same value to multiple variables:

```
$a = $b = $c = 0
```

The next example assigns multiple values to multiple variables.

```
$i,$j,$k = 10, "red", $true    # $i is 10, $j is "red", $k is True
$i,$j = 10, "red", $true       # $i is 10, $j is [object[]], Length 2
```

For more detailed information, see the **Assigning multiple variables** section of about_Assignment_Operators.

## Types of variables

You can store any type of object in a variable, including integers, strings, arrays, and hash tables. And, objects that represent processes, services, event logs, and computers.

PowerShell variables are loosely typed, which means that they aren't limited to a particular type of object. A single variable can even contain a collection, or array, of different types of objects at the same time.

The data type of a variable is determined by the .NET types of the values of the variable. To view a variable's object type, use Get-Member.

For example:

```
$a = 12                         # System.Int32
$a = "Word"                     # System.String
$a = 12, "Word"                 # array of System.Int32, System.String
$a = Get-ChildItem C:\Windows   # FileInfo and DirectoryInfo types
```

You can use a type attribute and cast notation to ensure that a variable can contain only specific object types or objects that can be converted to that type. If you try to assign a value of another type, PowerShell tries to convert the value to its type. If the type can't be converted, the assignment statement fails.

To use cast notation, enter a type name, enclosed in brackets, before the variable name (on the left side of the assignment statement). The following example creates a `$number` variable that can contain only integers, a `$words` variable that can contain only strings, and a `$dates` variable that can contain only **DateTime** objects.

```
[int]$number = 8
$number = "12345"  # The string is converted to an integer.
$number = "Hello"
```

```
Cannot convert value "Hello" to type "System.Int32". Error: "Input
string
was not in a correct format."
At line:1 char:1
+ $number = "Hello"
+ ~~~~~~~~~~~~~~~~~
+ CategoryInfo          : MetadataError: (:) [],
    ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException
```

```
[string]$words = "Hello"
$words = 2        # The integer is converted to a string.
$words += 10      # The plus (+) sign concatenates the strings.
$words
```

```
210
```

```
[datetime] $dates = "09/12/91"  # The string is converted to a DateTime
object.
$dates
```

```
Thursday, September 12, 1991 00:00:00
```

```
$dates = 10    # The integer is converted to a DateTime object.
$dates
```

```
Monday, January 1, 0001 00:00:00
```

## Using variables in commands and expressions

To use a variable in a command or expression, type the variable name, preceded by the dollar (`$`) sign.

If the variable name and dollar sign aren't enclosed in quotation marks, or if they're enclosed in double quotation (`"`) marks, the value of the variable is used in the command or expression.

If the variable name and dollar sign are enclosed in single quotation (`'`) marks, the variable name is used in the expression.

For more information about using quotation marks in PowerShell, see about_Quoting_Rules.

This example gets the value of the `$PROFILE` variable, which is the path to the PowerShell user profile file in the PowerShell console.

```
$PROFILE
```

```
C:\Users\User01\Documents\PowerShell\Microsoft.PowerShell_profile.ps1
```

In this example, two commands are shown that can open the PowerShell profile in **notepad.exe**. The example with double-quote (`"`) marks uses the variable's value.

```
notepad $PROFILE

notepad "$PROFILE"
```

The following examples use single-quote (`'`) marks that treat the variable as literal text.

```
'$PROFILE'
```

```
$PROFILE
```

```
'Use the $PROFILE variable.'
```

```
Use the $PROFILE variable.
```

## Variable names that include special characters

Variable names begin with a dollar (`$`) sign and can include alphanumeric characters and special characters. The variable name length is limited only by available memory.

The best practice is that variable names include only alphanumeric characters and the underscore (`_`) character. Variable names that include spaces and other special characters, are difficult to use and should be avoided.

Alphanumeric variable names can contain these characters:

- Unicode characters from these categories: **Lu**, **Ll**, **Lt**, **Lm**, **Lo**, or **Nd**.
- Underscore (`_`) character.
- Question mark (`?`) character.

The following list contains the Unicode category descriptions. For more information, see UnicodeCategory.

- **Lu** - UppercaseLetter
- **Ll** - LowercaseLetter
- **Lt** - TitlecaseLetter
- **Lm** - ModifierLetter
- **Lo** - OtherLetter
- **Nd** - DecimalDigitNumber

To create or display a variable name that includes spaces or special characters, enclose the variable name with the curly braces (`{}`) characters. The curly braces direct PowerShell to interpret the variable name's characters as literals.

Special character variable names can contain these characters:

- Any Unicode character, with the following exceptions:
  - The closing curly brace (`}`) character (U+007D).

- The backtick (`` ` ``) character (U+0060). The backtick is used to escape Unicode characters so they're treated as literals.

PowerShell has reserved variables such as $, $?, $^, and $_ that contain alphanumeric and special characters. For more information, see about_Automatic_Variables.

For example, the following command creates the variable named `save-items`. The curly braces (`{}`) are needed because variable name includes a hyphen (`-`) special character.

```
${save-items} = "a", "b", "c"
${save-items}
```

```
a
b
c
```

The following command gets the child items in the directory that is represented by the `ProgramFiles(x86)` environment variable.

```
Get-ChildItem ${env:ProgramFiles(x86)}
```

To reference a variable name that includes braces, enclose the variable name in braces, and use the backtick character to escape the braces. For example, to create a variable named `this{value}is` type:

```
${this`{value`}is} = "This variable name uses braces and backticks."
${this`{value`}is}
```

```
This variable name uses braces and backticks.
```

## Variables and scope

By default, variables are only available in the scope in which they're created.

For example, a variable that you create in a function is available only within the function. A variable that you create in a script is available only within the script. If you dot-source the script, the variable is added to the current scope. For more information, see about_Scopes.

You can use a scope modifier to change the default scope of the variable. The following expression creates a variable named `Computers`. The variable has a global scope, even when it's created in a script or function.

```
$Global:Computers = "Server01"
```

For any script or command that executes out of session, you need the `Using` scope modifier to embed variable values from the calling session scope, so that out of session code can access them.

For more information, see about_Remote_Variables.

## Saving variables

Variables that you create are available only in the session in which you create them. They're lost when you close your session.

To create the variable in every PowerShell session that you start, add the variable to your PowerShell profile.

For example, to change the value of the `$VerbosePreference` variable in every PowerShell session, add the following command to your PowerShell profile.

```
$VerbosePreference = "Continue"
```

You can add this command to your PowerShell profile by opening the `$PROFILE` file in a text editor, such as **notepad.exe**. For more information about PowerShell profiles, see about_Profiles.

## The Variable: drive

The PowerShell Variable provider creates a `Variable:` drive that looks and acts like a file system drive, but it contains the variables in your session and their values.

To change to the `Variable:` drive, use the following command:

```
Set-Location Variable:
```

To list the items and variables in the `Variable:` drive, use the `Get-Item` or `Get-ChildItem` cmdlets.

```
Get-ChildItem Variable:
```

To get the value of a particular variable, use file system notation to specify the name of the drive and the name of the variable. For example, to get the `$PSCulture` automatic variable, use the following command.

```
Get-Item Variable:\PSCulture
```

```
Name                           Value
----                           -----
PSCulture                      en-US
```

To display more information about the `Variable:` drive and the PowerShell Variable provider, type:

```
Get-Help Variable
```

## Variable syntax with provider paths

You can prefix a provider path with the dollar ($) sign, and access the content of any provider that implements the IContentCmdletProvider interface.

The following built-in PowerShell providers support this notation:

- about_Environment_Provider
- about_Variable_Provider
- about_Function_Provider
- about_Alias_Provider

## The variable cmdlets

PowerShell includes a set of cmdlets that are designed to manage variables.

To list the cmdlets, type:

```
Get-Command -Noun Variable
```

To get help for a specific cmdlet, type:

```
Get-Help <cmdlet-name>
```

| Cmdlet Name | Description |
|---|---|
| `Clear-Variable` | Deletes the value of a variable. |
| `Get-Variable` | Gets the variables in the current console. |
| `New-Variable` | Creates a new variable. |
| `Remove-Variable` | Deletes a variable and its value. |
| `Set-Variable` | Changes the value of a variable. |

## See also

- about_Automatic_Variables
- about_Environment_Variables
- about_Preference_Variables
- about_Profiles
- about_Quoting_Rules
- about_Remote_Variables
- about_Scopes