

How to use PowerShell Variables

 lazyadmin.nl/powershell/powershell-variables

January 16, 2024

Variables in PowerShell are used to store values, data, or other information you need later in your script. They can also be used to capture the results of a cmdlet. But how do you create variables, and use them outside a function for example?

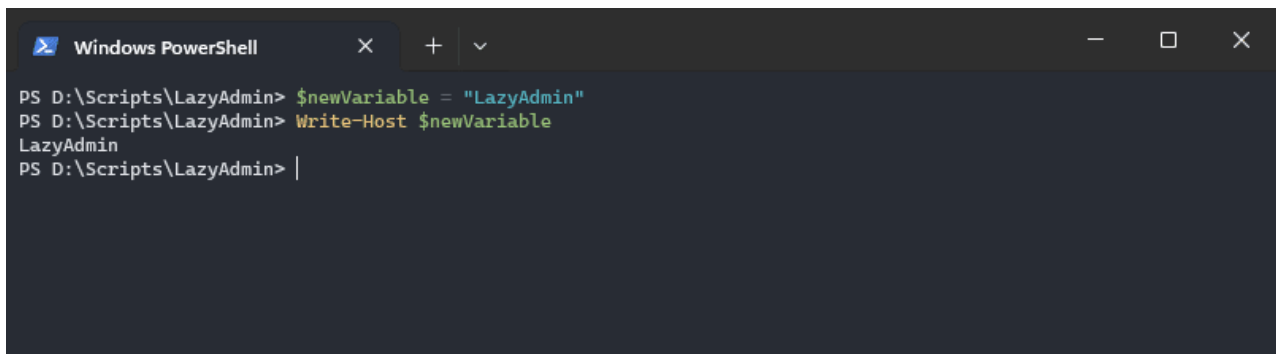
In PowerShell, we have different scopes where variables can belong. Scopes protect the variables from being changed, but they can also prevent you from accessing a variable that you have created in a function.

In this article, we are going to look at how to assign and use variables and use them in different scopes.

PowerShell Variables

Let's start with the basics of PowerShell variables before we are going to look at all the ins and outs. To create a variable in PowerShell, we will be using the dollar sign `$`. The variable name isn't case-sensitive and is commonly written in camelCase (capitalize the first letter of each word *except* the first).

```
$newVariable = "LazyAdmin"
```



```
Windows PowerShell
PS D:\Scripts\LazyAdmin> $newVariable = "LazyAdmin"
PS D:\Scripts\LazyAdmin> Write-Host $newVariable
LazyAdmin
PS D:\Scripts\LazyAdmin> |
```

The variable names can include spaces and special characters, but you really should avoid that. Spaces and special characters will make the code harder to read and will cause a lot of mistyping in your script.

Also good to know is that you don't have to specify the type for your variable. PowerShell variables are loosely typed, which means that a string variable can change into an integer, array, or hashtable for example.

```
$newVariable = "LazyAdmin" # String - System.String
```

```
$newVariable = 123 # Integer - System.Int32
```

```
$newVariable = "Apple", "Grapes", "Kiwi" # Array - Array of System.String
```

What you also need to know is that there are three types of variables in PowerShell. We are not talking here about whether a variable is a string or an integer, but where the variable originates from.

- **User-created variables** – These are the variables as seen in the examples above, variables that are created and maintained by you. User-created variables only exist in the current PowerShell session. As soon as you close the PowerShell window, the variable and its value are gone.

There is one exception though, variables created in your [PowerShell Profile](#) are saved and can be used throughout different PowerShell sessions.

- **Automatic variables** – Automatic variables are created and maintained by PowerShell and store state information. For example, `$error`, `$home`, or `$profile` are some of the automatic variables from PowerShell. You can find the complete list [here](#).
- **Preference variables** – These variables are also created by PowerShell, but the difference is that you can change these variables to customize the behavior of PowerShell. A good example is the variable `$ErrorActionPreference` that allows you to set what PowerShell should do when it runs into an error. You can find all the preference variables [here](#).

FREE EMAIL SERIES!

Level Up with PowerShell

5 Emails, Endless Skills

Create and Set Variables

As already briefly covered in the beginning, creating a variable is pretty straightforward. For a simple, loose typed, variable, you only need to specify the variable name.

```
$newVariable = $null
```

It's also possible to create a variable in PowerShell without specifying any value, even not `$null`. For this, you will have to use the cmdlet `Set-Variable`, which creates the variable with the value `$null` for you.

```
Set-Variable -Name $null
```

The set-variable cmdlet has some advantages when creating a variable, you could for example give a description to your variable or make it read online. But it isn't used a lot generally.

When creating variables you don't have to specify if it's a string or integer in PowerShell, which is really convenient. But when creating a function that expects an integer, then it's a good idea to explicitly specify that the variable is an integer.

To specify the type when creating a variable, you will need to use the cast notation. You do this by placing the type name in brackets before the variable:

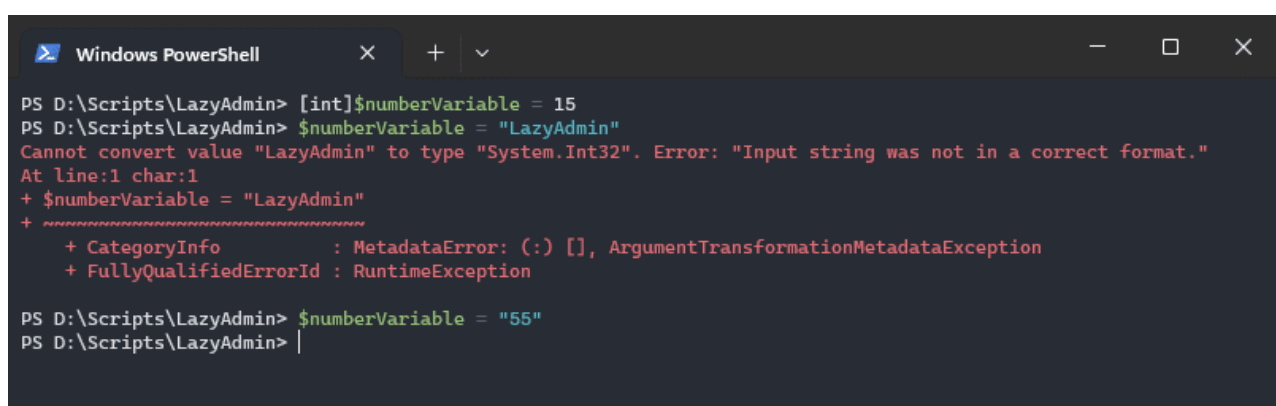
```
[string]$stringVariable = "Hello" # Can only contain a string
[int]$numberVariable = 15 # Can only contain a integer
[datetime]$dateVar = "01/15/24" # Accepts only datetime objects
```

As you can see in the last example (the datetime variable) we don't have to set a datetime object. PowerShell will accept the value, as long as it can convert it to the correct type.

Take the integer variable for example, it won't accept a string with a word but does accept a number in a string because it can cast it to an integer.

```
[int]$numberVariable = 15
$numberVariable = "LazyAdmin"
# Error result
Cannot convert value "LazyAdmin" to type "System.Int32". Error: "Input string was not in a correct format."
At line:1 char:1
+ $numberVariable = "LazyAdmin"
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException

PS D:\Scripts\LazyAdmin> $numberVariable = "55"
```

A screenshot of a Windows PowerShell terminal window. The title bar says "Windows PowerShell". The prompt is "PS D:\Scripts\LazyAdmin>". The user enters "[int]\$numberVariable = 15". The prompt changes to "PS D:\Scripts\LazyAdmin>". The user enters "\$numberVariable = \"LazyAdmin\"". The terminal shows an error: "Cannot convert value \"LazyAdmin\" to type \"System.Int32\". Error: \"Input string was not in a correct format.\"". Below the error, it says "At line:1 char:1" and "+ \$numberVariable = \"LazyAdmin\"". Then it shows the error details: "+ CategoryInfo : MetadataError: (:) [], ArgumentTransformationMetadataException" and "+ FullyQualifiedErrorId : RuntimeException". The user then enters "\$numberVariable = \"55\"". The prompt changes to "PS D:\Scripts\LazyAdmin>". The user enters a pipe character "|". The prompt changes to "PS D:\Scripts\LazyAdmin>".

PowerShell Type of Variable

If you want to know what the type of a variable is in PowerShell, then you will need to use the `GetType()` method. This will return the base type, type name, and a lot of other information about the variable. To only view the type name, select the name property:

```
$numberVariable.GetType().name
```

```
# Result  
Int32
```

Clear PowerShell Variable

To clear a variable in PowerShell you have two options, you can use the `Clear-Variable` cmdlet or just overwrite the variable with a `$null` value. The cmdlet does exactly the same, so the most convenient method is to assign the `$null` value like this.

```
$newVariable = $null  
# Alternative  
Clear-Variable -Name newVariable
```

Removing Variables

Another option is to remove variables in PowerShell. This will remove the variable and its value completely from your PowerShell session. To do this, you will need to use the `Remove-Variable` cmdlet.

When you want to remove or clear a variable in PowerShell for security reasons, then there is something to keep in mind. PowerShell is built on .Net, and .Net uses a garbage collector to clear up memory items that are no longer needed.

But until the garbage collector has passed, the value of your variable will still be available in your computer's memory. The `Clear-Variable` cmdlet actually makes a copy of the variable and marks the old one for collection. Whereas the `Remove-Variable` cmdlet marks the original one for collection.

So if you need to remove a secure string, password, or other sensitive information from the memory, then it's best to the .Net Garbage collector directly after you have changed the variable to `$null`.

```
$newVariable = $null  
[System.GC]::Collect() # Call .Net Garbage Collector
```

Print Variable

To print a variable in PowerShell you can just type the variable name (with the `$` sign) in the console to output the value. In a script, you would commonly use the `Write-Host` cmdlet to output the value of a variable.

```
$numberVariable = "LazyAdmin"  
# Directly in a console  
$numberVariable  
# In a script  
Write-Host $numberVariable
```

```
Windows PowerShell
PS D:\Scripts\LazyAdmin> $numberVariable = "LazyAdmin"
PS D:\Scripts\LazyAdmin> $numberVariable
LazyAdmin
PS D:\Scripts\LazyAdmin> Write-Host $numberVariable
LazyAdmin
PS D:\Scripts\LazyAdmin> |
```

When you want to include the variable inside a string, you will need to make sure that you are using double quotes " ". Only then will the value of the variable be used. If you use a single quote, then the variable name is used as an expression.

```
$numberVariable = "LazyAdmin"
```

```
Write-Host "You are reading $numberVariable"
```

```
# Result
```

```
You are reading LazyAdmin
```

```
Write-Host 'The variable $numberVariable contains a string'
```

```
# Result
```

```
The variable $numberVariable contains a string
```

If the variable contains an object, then you can output the different properties by using the `.` notation. You specify the variable name followed by a dot and the property name. Take the following hashtable:

```
$currentDC = [PSCustomObject]@{
```

```
"HostName" = "DC1"
```

```
"OS" = "Windows Server 2019"
```

```
"OS Version" = "10.0.17763"
```

```
"IP Address" = "192.168.1.1"
```

```
}
```

To print the `HostName` of the `$currentDC` variable, we can write:

```
Write-Host $currentDC.Hostname
```

But when you want to use the same property inside a string, then you will need to wrap this inside parentheses, like so:

```
Write-Host "The hostname is $($currentDC.Hostname)"
```

Variable Scopes

In PowerShell, we have three scopes where a variable can run in:

- **Global** – This is effective in your current PowerShell session. Contains the automatic and preference variables. Also, variables defined in your PowerShell Profile are stored here.
- **Script** – This is created while a script file runs. Command in the script runs inside the script scope.

- **Local** – Current scope, can be the global, script, or a scope inside a function.

Variables are only visible and accessible in their current scope and their child scopes. Variables inside the global scope are always accessible inside the script or local scope.

For example, when you declare a variable inside a function, you can't access it outside that function:

```
function Get-AuthToken {
[CmdletBinding()]
param (
[string] $TenantId
)
# Some code to get a token
$token = Get-MSCloudIdAccessToken -TenantId $TenantId
}
```

Get-AuthToken

The variable \$token would be \$null in this case

Get-MSCloudData -Token \$token -User 'Test'

To access or update variables inside another scope, you will need to use the scope modifier to access it. So for the above example, we can store the **\$token** variable inside the **Global** scope. This way we can use the **\$token** inside other functions as well.

Define the global variable

\$global:token = \$null

```
function Get-AuthToken {
```

```
[CmdletBinding()]
```

```
param (
```

```
[string] $TenantId
```

```
)
```

Some code to get a token

\$global:token = Get-MSCloudIdAccessToken -TenantId \$TenantId

```
}
```

Run the function

Get-AuthToken

\$global:token will now contain the token that we need

Get-MSCloudData -Token \$global:token -User 'Test'

Another common situation is where you need to access a variable outside a ForEach-Object loop or outside an Invoke-Command script block. In those cases, you can access the parent variable with the help of the **\$using:** modifier. In the example below we can access the services inside the Scriptblock with **\$using:services**

```
$services = @(
```

```
'EventSystem',
```

```
'w32time',
```

```
'netlogon'
```

```
)  
$stoppedServices = Invoke-Command -Computername $computername -ScriptBlock  
{Get-Service -Name $using:services | where {$_.Status -eq 'Stopped'}}
```

Try to minimize the use of the global scope as much as possible, because it will make your code harder to read and it can be accessed from anywhere in the script. If you are using global variables in PowerShell, then declare them at the beginning of the script.

List Variables

Have you forgotten a variable name? Or want to know which variables you currently have used? You can look at all variables in your sessions with the `Get-Variable` cmdlet. This will list all global and local variables and their session.

If you run the cmdlet without any parameter, then it will return all variables. To help you find a variable, the `Get-Variables` cmdlet comes with a couple of parameters. For example, we created a variable with the word `number` in it, but don't know the exact name.

To find the variable name you can use the `-name` parameter and the a `*` as a wildcard:

```
Get-Variable -name *number*
```

```
# Result
```

```
Name Value
```

```
----
```

```
numberVariable LazyAdmin
```

You can also use the `-Include` or `-Exclude` parameter to select only the variables with (or without) a particular string. If you only want to view the value of a variable, then use the `-ValueOnly` parameter.

Wrapping Up

Variables are the bread and butter of every programming or scripting language, including PowerShell. They allow you to store information and use it later in your script. Keep your variable names always clear and meaningful.

Global variables have their place but try to minimize their usage. They will make reading your code and debugging harder.

I hope you found this article helpful, if you have any questions, just drop a comment below.

Did you **Liked** this **Article**?

Get the latest articles like this **in your mailbox**
or share this article

I hate spam to, so you can unsubscribe at any time.