

PowerShell Scripting – Get started with this Ultimate Guide

Automating daily tasks, extracting information from systems, or managing Microsoft 365? PowerShell scripts really make your daily work a lot easier. When you are working in IT then you can't get around PowerShell.

PowerShell is an advanced command line interface (CLI) and scripting language that can be used on Windows, Linux, and macOS. With the help of cmdlets, we can perform tasks like retrieving users from the Active Directory or testing the network connection of a server. We can combine these tasks and processes into scripts that we can quickly run on multiple computers or schedule as a daily task.

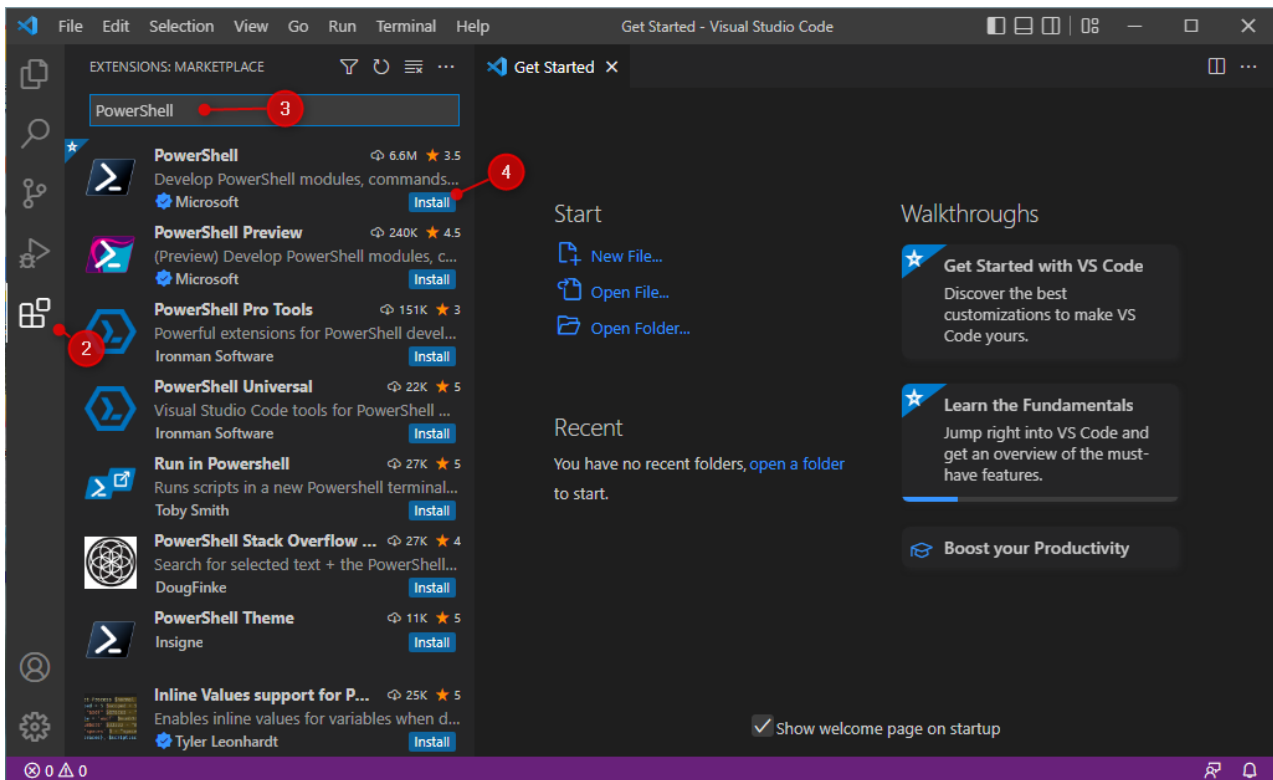
In this article, I will explain how you can create your own PowerShell Scripts. Which tools you can use, how to format your scripts and some general tips to get started. At the end of the article, you will also find a template you can use for your scripts.

PowerShell Editor

Most people that start writing PowerShell scripts use a simple notepad tool, like Notepad++. It works for small scripts, but a good editor makes writing PowerShell scripts much easier. They come with syntax highlighting, autocomplete functions, error detection, etc. And what I like the most is that you can create a project, allowing you to quickly switch between files, and keep your files organized.

For PowerShell, one of the best free editors to start with is Visual Studio Code. This editor, from Microsoft, is completely free and can be used on Windows, Linux, and macOS. To use the editor with PowerShell, you will need to install a plugin (extension).

1. First, **install the Visual Studio Code** using the installer. Simply click next on all screens.
2. Click on **Extensions**
3. Search for **PowerShell**
4. **Install** the PowerShell extension from Microsoft

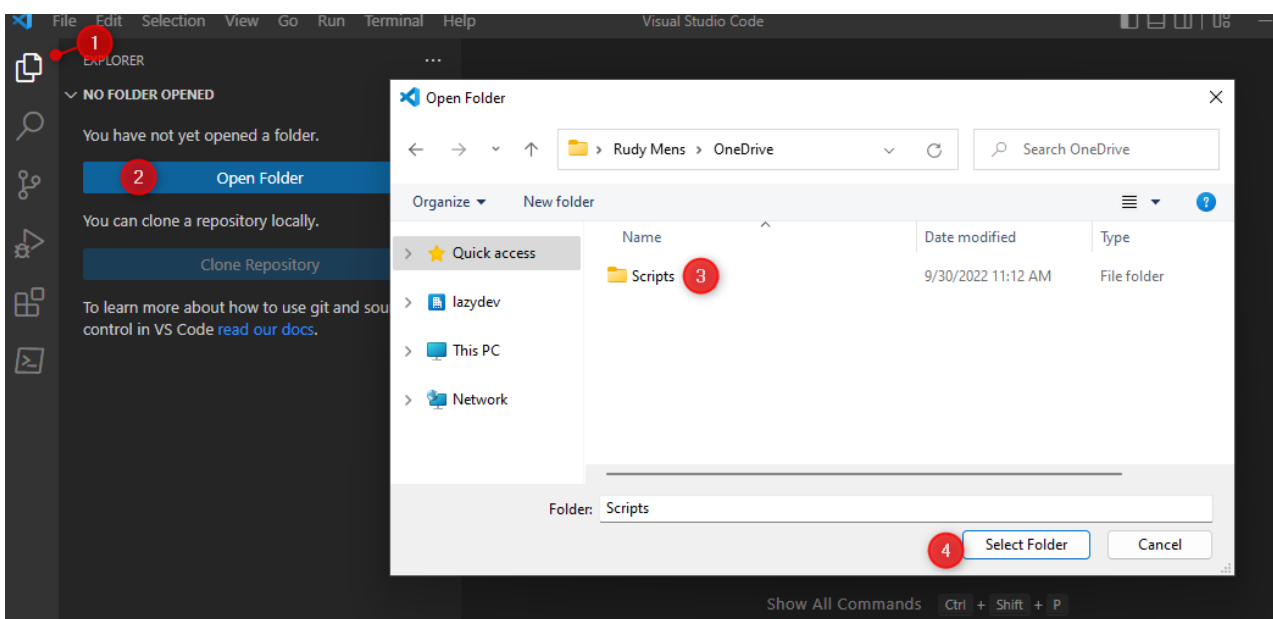


PowerShell Editor – Visual Studio Code

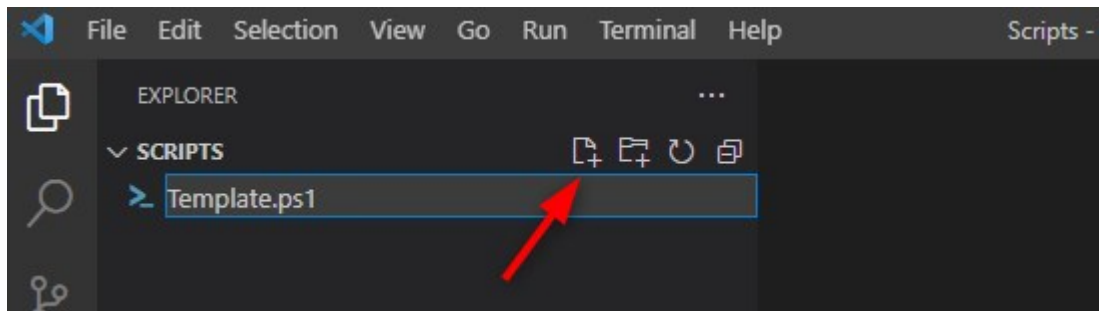
To create our first PowerShell script we will need to create a new file. To keep organized, we will create a new folder “scripts” where we are going to store our files. In Visual Studio Code:

1. Open the **Explorer**
2. Choose **Open Folder**
3. **Create a new folder**, scripts, in your OneDrive for example
4. Click **Select Folder**

You will get a prompt if you trust the files in this folder, make sure you check “Trust the authors of all files in the parent folder” and click **Yes, I Trust the authors**.



To create a new file, right-click in the editor (or click on the New file icon) and create your first PowerShell script. You can also create a subfolder to keep your scripts organized of course.



Running a PowerShell Script

Before we start writing our PowerShell script, it's good to know how you can run or test your scripts. One of the common issues, when you try to run your script, is the error **“Running scripts is disabled on this system”**. To solve this we will have to change the execution permission on your Windows 10 or 11 computer.

Open PowerShell and type the command below to change the execution policy. Read more about the PowerShell [execution policy in this article](#).

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

```
# Verify the setting with
```

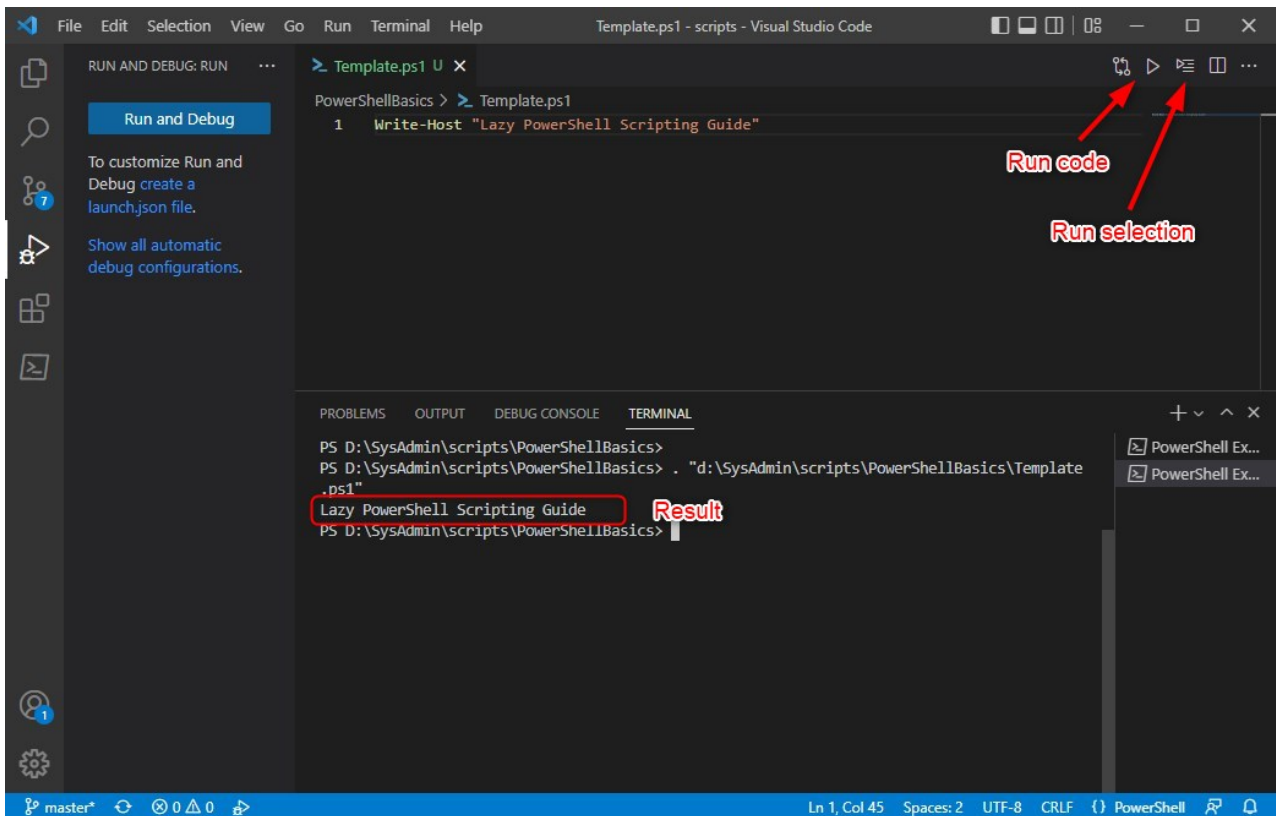
```
Get-ExecutionPolicy
```

With the execution policy set, we can run our scripts without any permissions errors. Now there are multiple options to run a PowerShell script, read more about them [in this article](#), but for this guide, we will focus on the following two:

- Using the built-in terminal in Visual Studio Code
- Run the script in PowerShell

Using Visual Studio Code

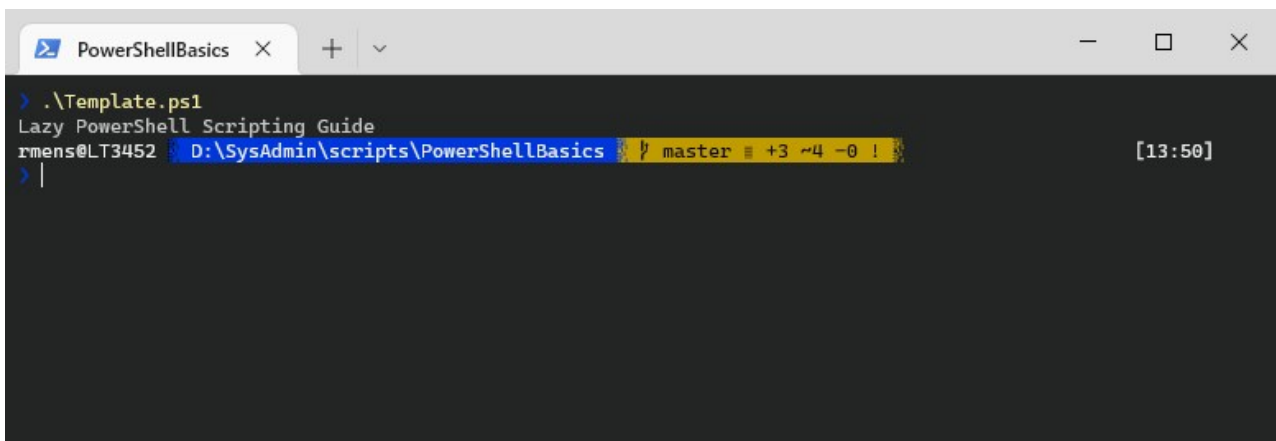
One of the advantages when using Visual Studio Code as an editor for your PowerShell scripts is that you can test and run your scripts in the built-in terminal. You can run the whole script by pressing **F5** or run a selection of lines by pressing **F8**.



Running PowerShell code

Using PowerShell

Another method is using PowerShell itself to run your script. Open Windows PowerShell or Windows Terminal (right-click on Start) and navigate to the folder where your script is saved. Type the filename of your script and press enter.



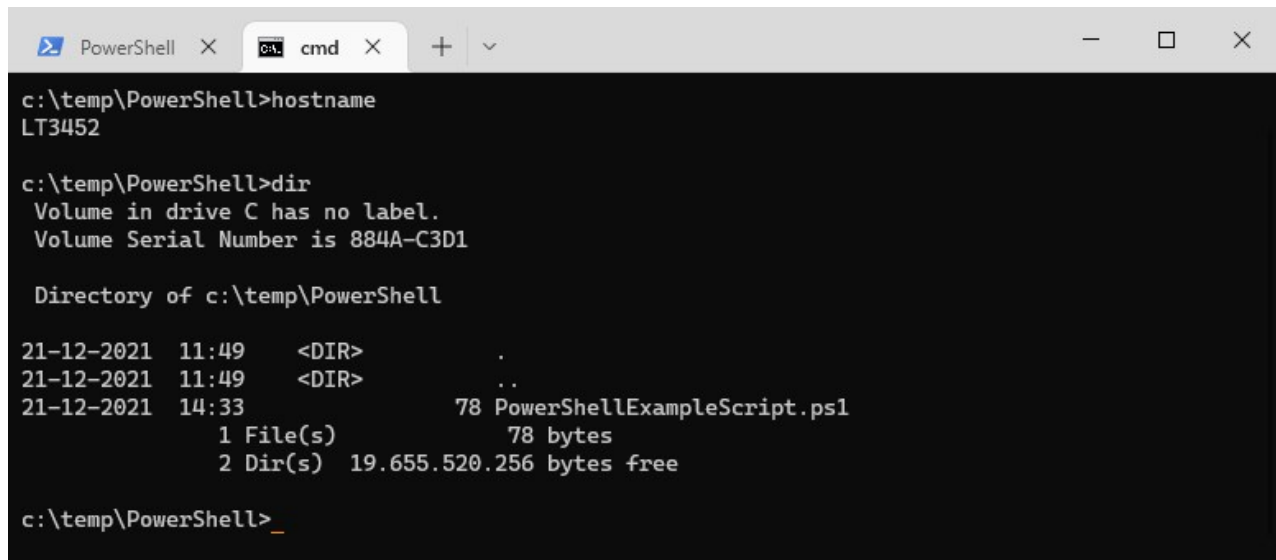
Using Windows Terminal

The Basics

Using Cmdlets in PowerShell

In the command prompt, we have commands, which are built-in functions (commands) that perform a task. For example, if you type `hostname` in the command prompt, it will show your computer name. Or the command `dir` will list the contents of the current

directory.



```
PowerShell x cmd + - □ ×
c:\temp\PowerShell>hostname
LT3452

c:\temp\PowerShell>dir
Volume in drive C has no label.
Volume Serial Number is 884A-C3D1

Directory of c:\temp\PowerShell

21-12-2021  11:49    <DIR>          .
21-12-2021  11:49    <DIR>          ..
21-12-2021  14:33                78 PowerShellExampleScript.ps1
               1 File(s)                78 bytes
               2 Dir(s)  19.655.520.256 bytes free

c:\temp\PowerShell>
```

Command Prompt commands

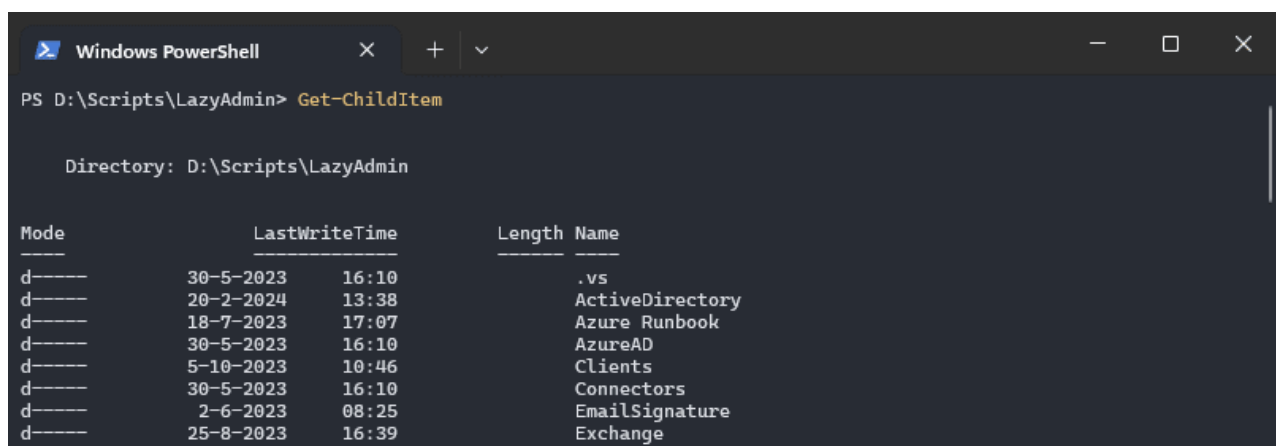
In PowerShell, we have command-lets (cmdlets). You can recognize a cmdlet by its naming because it exists of a *Verb-Noun* pair. This naming convention helps you to understand what a particular cmdlet does.

Note

In some examples below I am using cmdlets from the Active Directory module. Make sure that you have the module installed if you want to follow the steps. Run the following command to install the module:

Add-WindowsCapability -online -Name "Rsat.ActiveDirectory.DS-LDS.Tools~~~~0.0.1.0"

For example, the `Get-ComputerInfo` cmdlet gets all the computer information from your computer. To show the contents of a folder, we can use `Get-ChildItem`, for example.



```
Windows PowerShell x + - □ ×
PS D:\Scripts\LazyAdmin> Get-ChildItem

Directory: D:\Scripts\LazyAdmin

Mode                LastWriteTime         Length Name
----                -
d-----          30-5-2023   16:10             .vs
d-----          20-2-2024   13:38        ActiveDirectory
d-----          18-7-2023   17:07        Azure Runbook
d-----          30-5-2023   16:10        AzureAD
d-----          5-10-2023   10:46        Clients
d-----          30-5-2023   16:10        Connectors
d-----           2-6-2023    08:25        EmailSignature
d-----          25-8-2023   16:39        Exchange
```

Besides the built-in cmdlets, you can also install modules in PowerShell. These modules can, for example, be used to work with Exchange Online or Azure AD. Each module comes with its own cmdlets for its specific task.

Make sure that you also checkout and download the [PowerShell Cheat Sheet](#)

PowerShell Verbs

There are a lot of approved verbs that you can use when you are creating your own PowerShell commands or functions. But the most common verbs that you will come across are:

Verb	Action	Example
Add	Adds or appends resources to another item	Add-MailboxPermission
Clear	Removes all resources from a container, but doesn't delete the container	Clear-Host
Connect	Makes a connection to another system	Connect-AzureAD
Disconnect	Break the connection with the other system	Disconnect-AzureAD
Get	Retrieves data from a resource	Get-ChildItem
New	Creates a new resource	New-Mailbox
Remove	Remove a resource from a container	Remove-Mailbox
Set	Replaces data in an existing resource	Set-Mailbox
Select	Selects a resource in a container	Select-Object

PowerShell Verbs

Variables

When writing scripts you will often need to store data temporarily, so you can use it later in your script. For this we use variables. In PowerShell we don't need to initialize the variables, we can just create them when needed. Variables can not only store data, like strings, and integers, but also the complete output of cmdlets.

Let's start with a simple example, we take the **Get-Computer** cmdlet from before. The cmdlet returns your computer name:

Note

The contents after the # in a PowerShell script is a comment. Comments are used to explain what a function does and to make your code more readable.

hostname

Result

LazyBook

We don't want to just show the computer name, but instead, we want to include the computer name inside a nice string. In this case, we can first store the result of the `Get-Computername` cmdlet into a variable, which we will call `computername`. And then include this variable inside the string:

Store computer name inside the variable

```
$computername = hostname
```

Include the variable inside the string

```
Write-Host "The name of your computer is $computername"
```

Result

The name of your computer is LazyBook

Comparison Operators and If-Else Conditions

In PowerShell, we can use comparison operators to compare or find matching values. By default, the operators are case-insensitive, but you can place a `c` before an operator to make it case-sensitive. For example:

```
'lazyadmin' -eq 'LazyAdmin'
```

Result

True

```
'lazyadmin' -ceq 'LazyAdmin'
```

Result

False

The operators that we can use in PowerShell are:

Operator	Counter-Part operator	Description
-eq	-ne	Equal or not equal
-gt	-lt	Greater or less than
-ge		Great than or equal to
-le		Less than or equal to
-Like	-NotLike	Match a string using * wildcard or not
-Match	-NotMatch	Matches or not the specified regular expression
- Contains	-NotContains	Collection contains a specified value or not
-In	-NotIn	Specified value in collection or not
-Replace		Replace specified value

PowerShell Operators

We can use these operators in combination with for example If-Else statements. If statements allow us to check if a particular comparison is true or not. Depending on the outcome we can execute a piece of code or skip to another part.

Let's take the computer name example again. We have stored the computer name inside the variable. Now let's check if the computer's name is my laptop:

```
$computername = hostname
# Check if the computer name equal LazyBook
if ($computername -eq 'LazyBook') {
Write-Host "This computer is from LazyAdmin"
}else{
Write-host "This is someone else's computer"
}
```

Our computer names start with LT for laptops and PC for desktops. So we could determine if the device is a laptop or not based on the computer name. For this, we are going to use an if statement with an **-like** operator. The like operator accepts a wildcard, so we can, for example, check if a string starts with "LT" in this case

```
$computername = hostname
# Or better is
$computername = $env:COMPUTERNAME
# Check if the computer name start with Lazy
if ($computername -like 'Lazy*') {
Write-Host "This is a laptop"
```



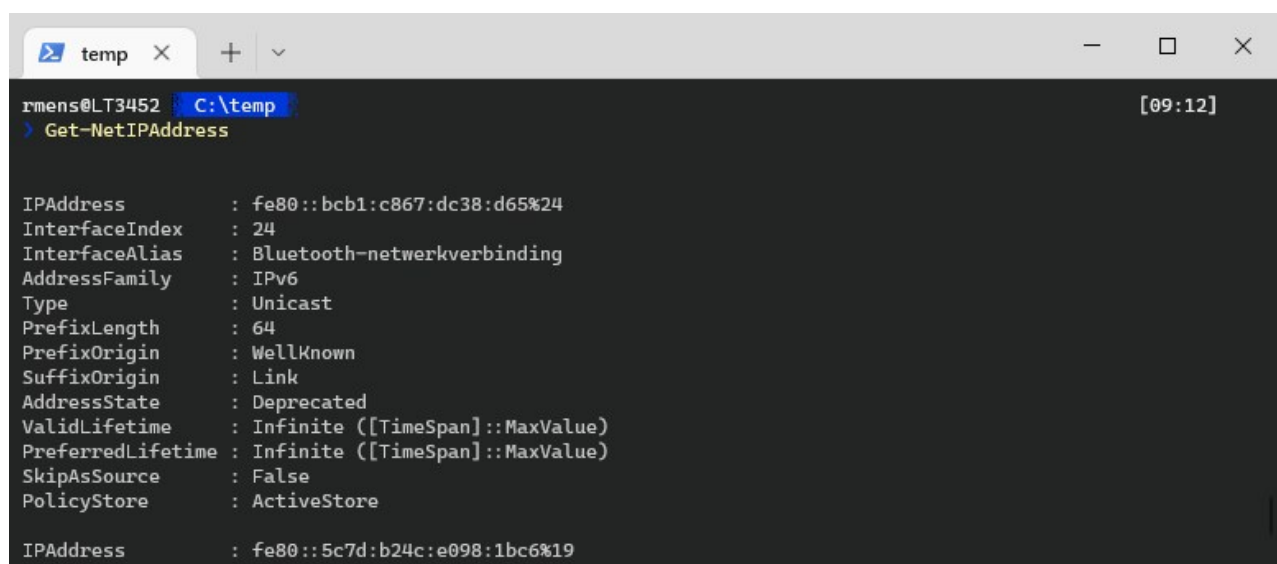
```
}else{  
Write-host "This is something else"  
}
```

Passing true results with Pipelining

Besides variables, there is another way to pass data through to another cmdlet in PowerShell, which is using the pipeline operator `|`. The pipeline operator passes the results of the command to the next command. The most common example of *pipelining* is formatting or selecting the result of a cmdlet.

Let's take the following example, the cmdlet `Get-NetIPAddress` returns the IP Address configuration of all your network interfaces. By default, it will return the results in a list format

Get-NetIPAddress



```
temp x + v  
rmens@LT3452 C:\temp [09:12]  
> Get-NetIPAddress  
  
IPAddress      : fe80::bcb1:c867:dc38:d65%24  
InterfaceIndex : 24  
InterfaceAlias  : Bluetooth-networkverbinding  
AddressFamily   : IPv6  
Type            : Unicast  
PrefixLength    : 64  
PrefixOrigin    : WellKnown  
SuffixOrigin     : Link  
AddressState     : Deprecated  
ValidLifetime   : Infinite ([TimeSpan]::MaxValue)  
PreferredLifetime : Infinite ([TimeSpan]::MaxValue)  
SkipAsSource     : False  
PolicyStore      : ActiveStore  
  
IPAddress      : fe80::5c7d:b24c:e098:1bc6%19
```

To make the results more readable we can format the results into a tablet or select only the properties that we need. We do this by piping the cmdlet `format-table (ft)` behind it or the cmdlet `select-object (select)`:

Get-NetIPAddress | FT

Or select the fields

Get-NetIPAddress | Select InterfaceAlias, IPAddress, PrefixOrigin

```
temp x + v
> Get-NetIPAddress | Select InterfaceAlias, IPAddress, PrefixOrigin

InterfaceAlias      IPAddress      PrefixOrigin
-----
Bluetooth-netwerkverbinding fe80::bcb1:c867:dc38:d65%24 WellKnown
LAN-verbinding* 2    fe80::5c7d:b24c:e098:1bc6%19 WellKnown
LAN-verbinding* 1    fe80::cd08:be20:63f4:76c1%7 WellKnown
Loopback Pseudo-Interface 1 ::1 WellKnown
Ethernet 6          169.254.163.217 WellKnown
Bluetooth-netwerkverbinding 169.254.13.101 WellKnown
LAN-verbinding* 2    169.254.27.198 WellKnown
LAN-verbinding* 1    169.254.118.193 WellKnown
Ethernet            192.168.50.125 Dhcp
Wi-Fi               192.168.1.22 Dhcp
Loopback Pseudo-Interface 1 127.0.0.1 WellKnown

rmens@LT3452 C:\temp [09:23]
> |
```

Now, this is a simple example of piping cmdlets. But let's take a look at a more advanced use of piping cmdlets. We are going to collect all user mailboxes, from each mailbox we are going to look up the mailbox statistics, select only the fields that we need, and export the results to a CSV file. Without the pipe operator, we would need to write a code similar to this:

Note

Get-EXOMailbox cmdlet is part of the Exchange Online module. Make sure that you have installed it if you want to follow the steps below.

```
$mailboxes = Get-EXOMailbox -RecipientTypeDetails UserMailbox
$mailboxes.ForEach({
$Mailboxstats = Get-EXOMailboxStatistics -Identity $_.Identity
$fields = Select-Object -InputObject $Mailboxstats -Property
DisplayName,ItemCount,TotallItemSize
Export-CSV -InputObject $fields -Path c:\temp\file.csv -Append
})
```

But with *piping* in PowerShell we can simply do the following:

```
Get-EXOMailbox -RecipientTypeDetails UserMailbox | Get-EXOMailboxStatistics | Select
DisplayName, ItemCount, TotallItemSize | Export-CSV c:\temp\filename.csv
```

Storing data in Arrays and Hashtables

Arrays and hashtables can be used to store a collection of data. Hashtables use a key value principle where you need to define the key before you can store the value. Arrays use an automatically generated index to store the values.

To create an array we can simply assign multiple values to a variable, separating each of them with a common. For example:

```
# Create an array of fruits
$array = 'apple','raspberry','kiwi'
```

Another option is to first initialize the array and add values to the array later on. To create an empty array we will use the @ symbol followed by parentheses :

```
# Create an empty array
$fruits = @()
# Add content to the array
$fruits += "apple"
```

Hashtables are also known as associative arrays, they are used when you need to store data in a more structured manner. Instead of a numbered index, it's based on a key-value pair, where you need to define the key yourself. To create an empty hashtable you will need to use curly brackets and the @ symbol:

```
# Create empty hashtable
$hashTable = @{}
You could for example use a hashtable to store the server IP Addresses
```

```
$serverIps= @{
'la-srv-lab02' = '192.168.10.2'
'la-srv-db01' = '192.168.10.100'
}
```

Result

Name Value

```
la-srv-lab02 192.168.10.2
la-srv-db01 192.168.10.100
```

Or as a config file inside your script:

```
$mail = @{
SmtpServer = 'smtp.contoso.com'
To = 'johndoe@lazyadmin.nl'
From = 'info@contoso.com'
Subject = 'super long subject goes here'
Body = 'Test email from PowerShell'
Priority = High
}
```

Looping through data with Foreach and Do-While

Looping through data is one of the common tasks in any scripting language. ForEach loops allow you to go through each item in a collection and do something with that item. For example, we take the array of fruits and write each item (fruit) to the console:

```
$fruits = @('apple','pear','banana','lemon','lime','mango')
# Foreach block
Foreach ($fruit in $fruits) {
Write-Host $fruit;
}
```

```
# Shorthand
$fruits.foreach( {
Write-Host $_;
})
```

In the example above we only used a simple array, but you can also use ForEach on objects. Inside the ForEach block you can access each property of the object, for example, if we get all the mailboxes, we can access the display name as follows:

```
$mailboxes = Get-EXOMailbox -RecipientTypeDetails UserMailbox
$mailboxes.ForEach({
# Write the displayname of each mailbox
Write-host $_.DisplayName
})
```

Besides ForEach loops, we can also use **While** and **Do-While** loops. A while loop will only run when a condition is met, the Do-While always runs once and as long as the condition is true.

```
# Do While loop
Do {
Write-Host "Online"
Start-Sleep 5
}
While (Test-Connection -ComputerName 8.8.8.8 -Quiet -Count 1)
Write-Host "Offline"
# While loop
$i = 0;
$path = "C:\temp"
While ($i -lt 10) {
# Do Something
$newFile = "$path\while_test_file_" + $i + ".txt";
New-Item $newFile
$i++;
}
```

Learn more about For loops, ForEach statements, and Do While loops [in this article](#).

Catch errors with Try-Catch

Try-catch blocks are used to handle errors in a proper way. Normally when a function doesn't work or runs into an error, the script will simply stop and throw an error. Sometimes this is fine, but on other occasions, you might want to show a more readable error or simply continue.

For example, when you are updating multiple users in the Active Directory using a ForEach loop. When one of the user accounts doesn't exist, the script will run into an error and stop. But a better solution would be if the script outputs the name of the users it

didn't update and just continues with the next one. This is where Try-Catch blocks come in.

Taking the example above, the following code block will try to find and update the Azure AD user, if an error occurs in the Try block, then the Catch part will show an error.

```
$users.ForEach{
Try{
# Find the user to update
$ADUser = Get-AzureAdUser -SearchString $_.name
# Update the job title
Set-AzureAdUser -ObjectId $ADUser.ObjectId -JobTitle $_.jobtitle
}
Catch{
Write-Host ("Failed to update " + $(($_.name))) -ForegroundColor Red
}
}
```

Now, this is a basic implementation of a try-catch block, it's even possible to use multiple catch blocks on a single Try statement, allowing you to catch different errors. Read more about Try-Catch blocks in this in-depth article.

Creating PowerShell Scripts

You should now have a brief understanding of what tools you can use in PowerShell to create scripts. So let's take a look at how we combine this into true PowerShell scripts. For the examples below we are going to create a small script that creates test files in a given folder.

Below you will find the basic principle of the script. We have a path, hardcoded in the script, an array with the numbers 1 to 10, and a ForEach loop. With the examples below we are going to enhance this script to a true PowerShell script.

```
$path = "C:\temp"
1..10 | ForEach-Object {
$newFile = "$path\test_file_$.txt";
New-Item $newFile
}
```

Documenting and Comments

When creating PowerShell scripts it's always a good idea to add documentation to your script. The documentation is placed at the top of your script in a comment block and describes what the script does, a couple of examples on how to use the script, and a notes block with the author, version, date, etc.

So for our test file script, we can add the following description at the beginning of our file:

<#

.SYNOPSIS

Create test files in given directory

.DESCRIPTION

The script generates an x amount of text file based test file in the given folder. The files don't contain any content.

.EXAMPLE

CreateTestFiles.ps1 -path c:\temp -amount 50

Create 50 files in c:\temp

.NOTES

Version: 1.0

Author: R. Mens - LazyAdmin.nl

Creation Date: 04 oct 2022

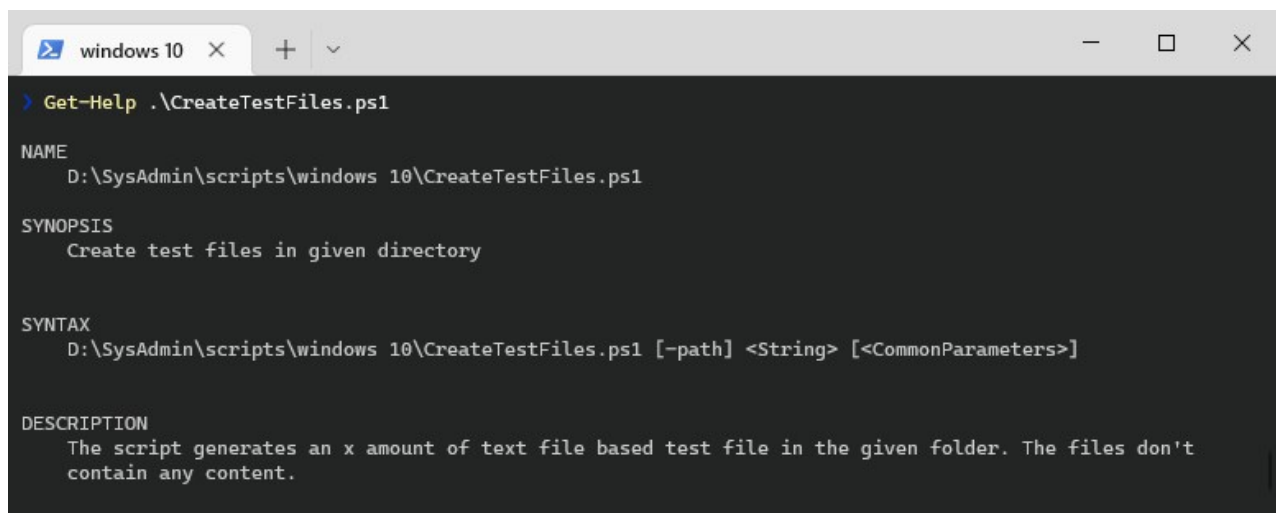
Modified Date:

Purpose/Change: Init

Link: <https://lazyadmin.nl/powershell/powershell-scripting>

#>

If you now run the command **Get-Help** followed by the script name in PowerShell you will get the explanation of your script:

A screenshot of a PowerShell terminal window. The title bar shows 'windows 10' and standard window controls. The command prompt shows the command 'Get-Help .\CreateTestFiles.ps1'. The output is formatted with sections: NAME (D:\SysAdmin\scripts\windows 10\CreateTestFiles.ps1), SYNOPSIS (Create test files in given directory), SYNTAX (D:\SysAdmin\scripts\windows 10\CreateTestFiles.ps1 [-path] <String> [<CommonParameters>]), and DESCRIPTION (The script generates an x amount of text file based test file in the given folder. The files don't contain any content.).

```
> Get-Help .\CreateTestFiles.ps1

NAME
    D:\SysAdmin\scripts\windows 10\CreateTestFiles.ps1

SYNOPSIS
    Create test files in given directory

SYNTAX
    D:\SysAdmin\scripts\windows 10\CreateTestFiles.ps1 [-path] <String> [<CommonParameters>]

DESCRIPTION
    The script generates an x amount of text file based test file in the given folder. The files don't
    contain any content.
```

Get-Help from your script

Read more about comments and documentation in the [PowerShell Best Practice Guide](#)

Cmdlet Parameters

When using PowerShell cmdlets you have probably noticed that most of them come with parameters that we can set. For example, when you want to get a mailbox from Exchange Online, you can specify the username with the parameter **-Identity**. We can do the same in our own PowerShell scripts by defining Parameter attributes.

In the example above we have hard-coded the path variable. If you want to create the test files in a different directory, then you will need to change the script. A better option would be if you could set the path from the command line. So we are going to create a param block, with the parameter path:

```
param(
[Parameter()]
[string]$path
)
# Create files
1..10 | ForEach-Object {
$newFile = "$path\test_file_$_.txt";
New-Item $newFile
}
```

We expect the path to be a string, other options are, for example, a switch where you can choose a value or an integer. If you open PowerShell and type the script file name, then you can also add the -path parameter:



But what if the user of the script doesn't set the path? In this case, the path is mandatory, without it, the test file will be created in the script's root location. To make it mandatory, we add the parameter property **Mandatory** and set it to true. Another option is to set a default value:

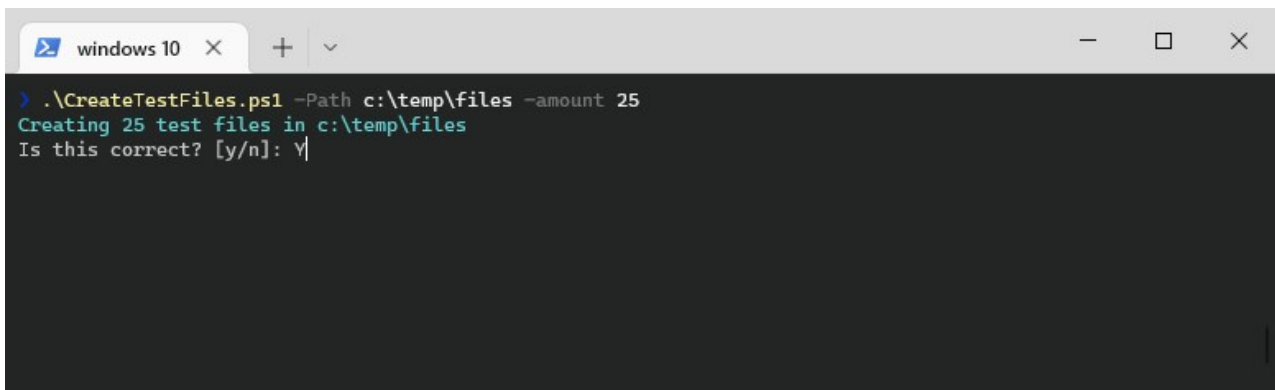
```
param(
[Parameter(
Mandatory = $true,
HelpMessage = "Enter path where test files should be created"
)]
[string]$path,
[Parameter(
HelpMessage = "How many files should be created"
)]
[int]$amount = 10
)
```

Read-Host

Our script is going to create text files in the given folder. If the user makes a typo, then it is possible that the script is going to create 1000 files for example, instead of 100. So before we are going to create the files, it might be a good idea to ask for confirmation. For this, we are going to use the **Read-Host** cmdlet.

Read-Host is used to prompt the user for input. This can be a string or password for example. The results of the input can be validated with a simple comparison. In this case, we are first going to write the console the values that the user has given up through the parameters, and then ask if the user wants to continue:

```
# Ask for confirmation
Write-host "Creating $amount test files in $path" -ForegroundColor Cyan
$reply = Read-Host -Prompt "Is this correct? [Y] Yes [N] No "
# Check the reply and create the files:
if ( $reply -match "[yY]" ) {
# Create files
1..10 | ForEach-Object {
$newFile = "$path\test_file_$_.txt";
New-Item $newFile
}
}
```



Using Read-Host inside PowerShell Script

Functions

Functions are a great way to organize and reuse pieces of code inside your script. A function should perform a single task. You can find a good example of that in [this OneDrive Size Report](#) script. The function **ConvertTo-Gb** takes a value and converts it to Gigabytes and returns the result. If you scroll a bit down in the script you will see that the function **Get-OneDriveStats** gathers all OneDrives and process each of them. Inside the ForEach loop, I convert the values using the ConvertTo-GB function.

The structure of a function is basically a small script. We start with a short description, the synopsis. If we need to pass values to the script, we will use Parameters. You may also notice that some functions have 3 code blocks, a begin, process, and end. The process

block is executed for all values that you pass to the function and the begin and end blocks only once. Now I have to say that I don't use begin and end often.

So we are going to create a simple function for our test files script. I have added the parameter Path in the function as well. It's not really necessary, because the function has also access to the parameters that we have set at the beginning of the script. (this is more to demonstrate the possibilities)

```
Function New-TestFiles{
<#
.SYNOPSIS
Create test files
#>
param(
[Parameter(Mandatory = $true)]
[string]$path
)
1..$amount | ForEach-Object {
$newFile = "$path\test_file_$.txt";
New-Item $newFile
}
}
```

We can now use the function in our script as follows:

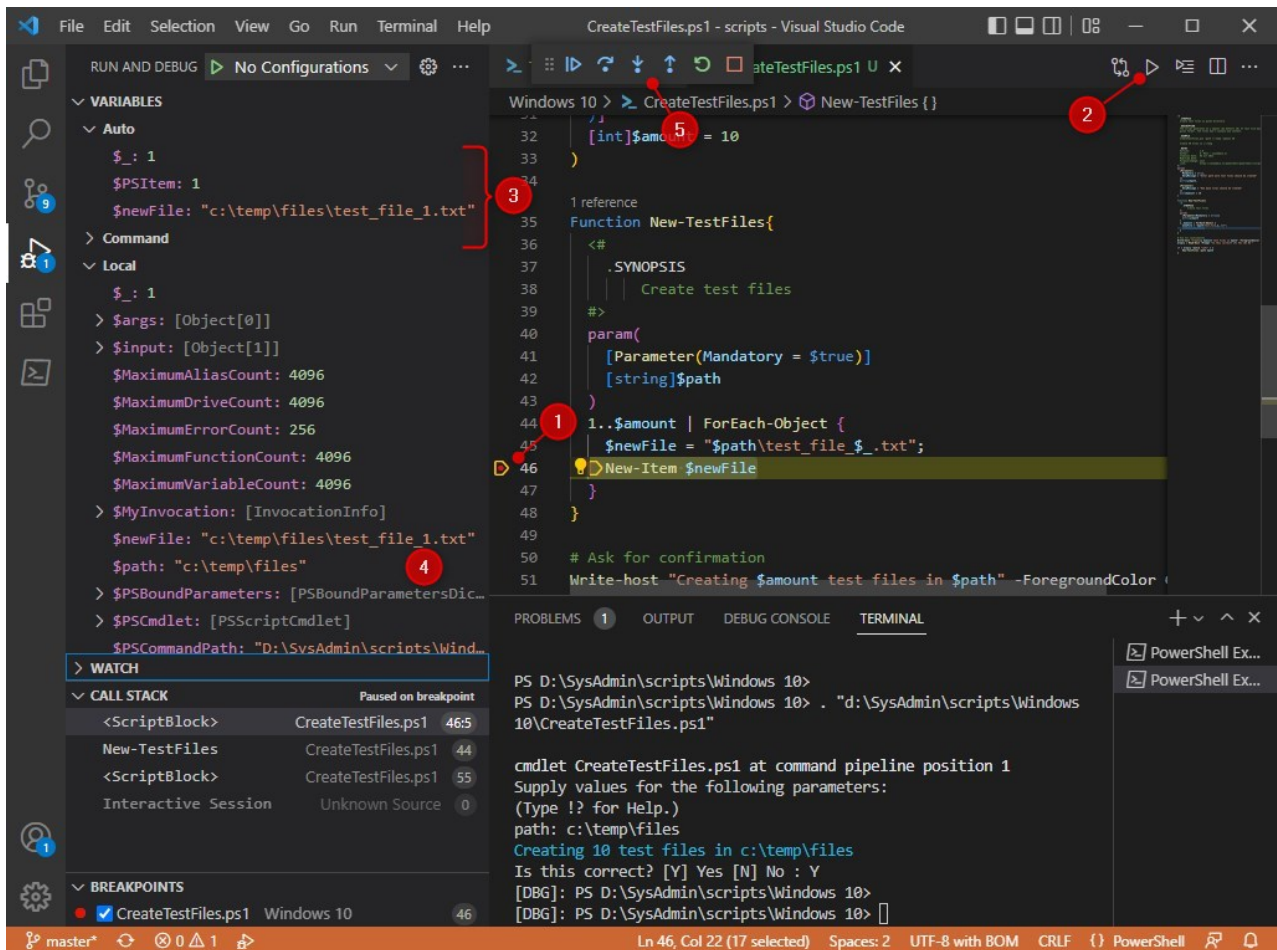
```
if ( $reply -match "[yY]" ) {
New-TestFiles -path $path
}
```

Breakpoints

A typo, a forgotten quote, or a small mistake is easily made when creating a PowerShell script. That is why it's important to constantly test your script when you are writing it. But even then it can sometimes be hard to find the cause of the error in your script.

To help you find errors in your script you can use breakpoints. If you are using Visual Studio Code, then you can use the **built-in Run and Debug** function, allowing you to set line breakpoints and run your script line by line.

In visual studio code, we can set breakpoints by clicking on the red dot just before the line number. In the screenshot below, I have set a breakpoint **(1)** on the `New-Item` cmdlet at line 46. When we click on Run **(2)** the script will be executed and halt on the breakpoint at line 46. At the moment, we can see the variables on the left side **(3)**. This tells us that the `ForEach-Object` loop is at the first item, and the new file will be created in `c:\temp\files` with the name `test_file_1.txt`



Debugging PowerShell in Visual Studio Code

With the buttons at the top **(5)**, we can move to the next step, skip a step, etc.

Using breakpoints in PowerShell

Another option to debug your PowerShell script is to use the cmdlet `Set-PSBreakpoint` in the console. This cmdlet allows you to set breakpoints for any script that you want to run in PowerShell. With the cmdlet, we can set breakpoints on a line number, action, or variable. So let's set the same breakpoint as we did in Visual Studio Code:

Set the breakpoint on line 46 for the script `CreateTestFiles.ps1`

`Set-PSBreakPoint -Script .\CreateTestFiles.ps1 -Line 46`

When we now run the script, it will stop at line 46:

```
PS D:\SysAdmin\scripts\Windows 10> .\CreateTestFiles.ps1 -path c:\temp\files -amount 5
Creating 5 test files in c:\temp\files
Is this correct? [Y] Yes [N] No : Y
[DBG]: PS D:\SysAdmin\scripts\Windows 10> █
```

Debug script from the console

The script is stopped, but it doesn't show anything. So we are going to start by showing the code around our breakpoint. Just type in the letter L in the console:

```
PS D:\SysAdmin\scripts\Windows 10> .\CreateTestFiles.ps1 -path c:\temp\files -amount 5
Creating 5 test files in c:\temp\files
Is this correct? [Y] Yes [N] No : Y
[DBG]: PS D:\SysAdmin\scripts\Windows 10> L

41:     [Parameter(Mandatory = $true)]
42:     [string]$path
43: )
44: 1..$amount | ForEach-Object {
45:     $newFile = "$path\test_file_$.txt";
46:*    New-Item $newFile
47: }
48: }
49:
50: # Ask for confirmation
51: Write-host "Creating $amount test files in $path" -ForegroundColor Cyan
52: $reply = Read-Host -Prompt "Is this correct? [Y] Yes [N] No "
53:
54: if ( $reply -match "[yY]" ) {
55:     New-TestFiles -path $path
56: }
```

Show surrounding code

The asterisk symbol on line 46 indicated the breakpoints that we have set. If we want to know what the path variable is, or the new file name, we can simply type in the variable names in the console and press enter:

```
[DBG]: PS D:\SysAdmin\scripts\Windows 10> $path
c:\temp\files
[DBG]: PS D:\SysAdmin\scripts\Windows 10> $newFile
c:\temp\files\test_file_1.txt
[DBG]: PS D:\SysAdmin\scripts\Windows 10> █
```

Show the contents of the variables

We can view all breakpoints that we have set with the cmdlet Get-PSBreakpoint and remove them with Remove-PSBreakpoint. Combining both will remove all breakpoints:

Get-PSBreakPoint | Remove-PSBreakpoint

Wrapping Up

When creating PowerShell Scripts, always start small and test your script often. Also, make sure that you add comments to your code where necessary. It may be clear now what the codes those, but if you need to edit your script a year later it can sometimes be a puzzle to figure out what the script does and how it works.

If you like to learn more about PowerShell then the book below is really a good read. This is one of the best sellers when it comes to learning PowerShell:

In the examples above we create a small script that creates test files, below you will find the complete script. As mentioned, it's a bit overkill for such a simple task, but it gives you an idea of how to build up your script.

```
<#
.SYNOPSIS
Create test files in given directory
.DESCRPTION
The script generates an x amount (by default 10) of text file based test file in the
given folder. The files don't contain any content.
.EXAMPLE
CreateTestFiles.ps1 -path c:\temp -amount 50
Create 50 files in c:\temp
.NOTES
Version: 1.0
Author: R. Mens - LazyAdmin.nl
Creation Date: 04 oct 2022
Modified Date:
Purpose/Change: Init
Link: https://lazyadmin.nl/powershell/powershell-scripting
#>
param(
[Parameter(
Mandatory = $true,
HelpMessage = "Enter path were test files should be created"
)]
[string]$path,
[Parameter(
HelpMessage = "How many files should be created"
)]
[int]$amount = 10
```

```
)  
Function New-TestFiles{  
<#  
.SYNOPSIS  
Create test files  
#>  
param(  
[Parameter(Mandatory = $true)]  
[string]$path  
)  
1..$amount | ForEach-Object {  
$newFile = "$path\test_file_$_txt";  
New-Item $newFile  
}  
}  
# Ask for confirmation  
Write-host "Creating $amount test files in $path" -ForegroundColor Cyan  
$reply = Read-Host -Prompt "Is this correct? [Y] Yes [N] No "  
if ( $reply -match "[yY]" ) {  
New-TestFiles -path $path  
}  
I hope you found this article useful, if you have any questions, just drop a comment  
below. And please share it if you liked it!
```

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.