Powershell-Function

What is Function?   
A function is a block of code that has a name and it has a property that it is reusable. Like in other programming language we can create function in powershell as well, it is basically groups a number of program statements into a unit and function is given name to it.   
This unit can be invoked from other parts of a program.

**Function Syntax:**

With parameter

**function** **FunctionName** (parameters)

{

#script block

}

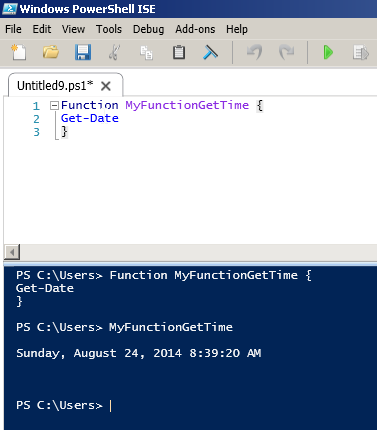
Without parameter

**Function** MyFunctionGetTime

{

Get-**Date**

}

To get execute function just give the function name. PS c:\>MyFunctionGetTime   
   
By use of fuction we can do the script organization. We can reuse the code.   
Use of function give you ability to call script blocks multiple times. 

**function** **Addtion** ([int] $x, [int] $y) {

$z = $x+ $y

#"Function is running"

**return** $z

}

#someFunction (5, 4) --Wrong

$b = someFunction 5 4 --right

$b # sum of tow valies

$z # Default value zero here as it is out of scope

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Clone | CompareTo | Contains | CopyTo | EndsWith |
| Equals | GetEnumerator | GetHashCode | GetType | GetTypeCode |
| IndexOf | IndexOfAny | Insert | IsNormalized | LastIndexOf |
| LastIndexOfAny | Normalize | PadLeft | PadRight | Remove |
| Replace | Split | StartsWith | Substring | ToBoolean |
| ToByte | ToChar | ToCharArray | ToDateTime | ToDecimal |
| ToDouble | ToInt16 | ToInt32 | ToInt64 | ToLower |
| ToLowerInvariant | ToSByte | ToSingle | ToString | ToType |
| ToUInt16 | ToUInt32 | ToUInt64 | ToUpper | ToUpperInvariant |
| Trim | TrimEnd | TrimStart |  |  |

Concatenating strings

However, some tasks require the use of operators. For example, PowerShell doesn’t have a concat function. Instead, you can just concatenate two or more strings with the plus operator:



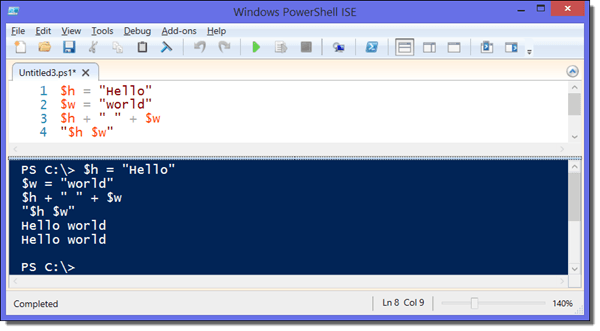
|  |  |
| --- | --- |
| 1  2  3 | $h = "Hello"  $w = "world"  $h + " " + $w |

Alternatively, you can expand the two strings within double quotes:



|  |  |
| --- | --- |
| 1 | "$h $w" |

In the example, the space guarantees that the two words don’t stick together. However, you could insert any other delimiter.

[](https://4sysops.com/wp-content/uploads/2015/05/Concatenating-strings.-in-PowerShell.png)

*Concatenating strings in PowerShell*

This method is also suitable for concatenating the elements of a string array:



|  |  |
| --- | --- |
| 1  2 | $st = @("Say hello", " world")  "$st" |

PowerShell automatically inserts a space here between the linked array elements.

Another option for concatenating strings is the *join* operator. Its usage is perhaps a bit counterintuitive. Normally, you place the operator in front of the string unless you want to separate the strings with a delimiter after the merge:



|  |  |
| --- | --- |
| 1 | -join($h,$w) |

If you don’t like the result “Helloworld” in the example and you want to add a space between the two words, you have to call *-join* this way:



|  |  |
| --- | --- |
| 1 | $h,$w -join " " |

Splitting strings

For the opposite task (that is, for splitting strings), you can use either the *split* method or the *split* operator. The former is simpler and only allows for use of explicit delimiters; the operator, on the other hand, supports regular expressions.

In the following example, the string is split at the double “ll” and at the space. The delimiter itself falls by the wayside:



|  |  |
| --- | --- |
| 1 | ("Hello world").split("ll"" ") |

You can specify the maximum number of substrings in which the string is split. PowerShell will then produce an array with the corresponding number of elements:



|  |  |
| --- | --- |
| 1 | ("Hello world").split("ll"" ", 2) |

The result array in the above example has two elements.

Extracting substrings ^

The method *Substring* has a similar purpose because it can extract a part of the string. However, instead of using delimiters, you have to specify the position and length of the string that you want to extract:



|  |  |
| --- | --- |
| 1 | ("Hello world").Substring(2,5) |

This command results in “llo w”, which corresponds to the substring with a length of five characters that begins at the third character (counting begins at 0!).

The counterpart method to *Substring* is *Remove*. It deletes the specified range from the string:



|  |  |
| --- | --- |
| 1 | ("Hello world").Remove(2,3) |

In this example, “llo” is removed from “Hello world.”

To eliminate leading or trailing spaces, you can use *TrimStart*, *TrimEnd*, or *Trim*.

Searching and replacing characters

PowerShell knows a variety of techniques to find and replace substrings. For the more demanding tasks, regular expressions are available, which can be applied with the *-match*or *-replace* operators.

In addition, the string object offers several methods for this task. Of course, each of these methods has its specific purpose. *Replace* is the simplest of these methods and doesn’t support regular expressions.



|  |  |
| --- | --- |
| 1 | ("Hello World").Replace("Hello","New") |

A counterpart to the *-match* operator doesn’t exist. However, PowerShell supports several methods that specialize on a particular search type. For instance, *StartsWith* and *EndsWith*determine whether a string begins or ends with a certain character or string, respectively. Likewise, *Contains* tells you if a string contains a certain substring:



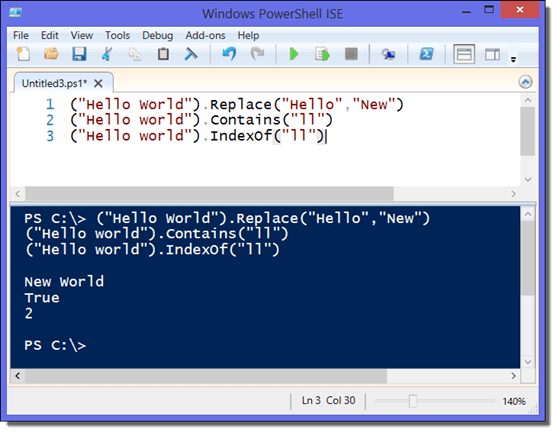
|  |  |
| --- | --- |
| 1 | ("Hello world").Contains("ll") |

The above command results in *True.* To calculate the position of a character or substring in the string, *IndexOf* is at your disposal:



|  |  |
| --- | --- |
| 1 | ("Hello world").IndexOf("ll") |

Don’t forget that counting starts at 0!

[](https://4sysops.com/wp-content/uploads/2015/05/Searching-and-replacing-characters-in-PowerShell.png)

*Searching and replacing characters in PowerShell*

Comparing strings ^

In general, you can work with the same comparison operators as for numerical values to determine differences between strings. Primarily, this includes *-eq* and *-ne*, as well as *-like*, which supports wildcards.

String objects also offer methods for this purpose. If the first string is “bigger” than the second string (that is, if it comes first in the sort order), the cmdlet returns 1; if the first string is smaller, the result is ‑1.



|  |  |
| --- | --- |
| 1 | ("Hello world").CompareTo("Hello" + " " + "world") |

In the above example, *CompareTo* returns 0 because the strings are identical. In contrast, the comparable call with *Equals* returns *True*:



|  |  |
| --- | --- |
| 1 | ("Hello world").Equals("Hello" + " " + "world") |

$MyName = "Boe Prox"

"Hello $($MyName), today is $((Get-Date).DayOfWeek)"

**[https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig2_s.ashx](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig2.ashx)[Click on image for larger view.]**  ***Figure 2.*** Evaluating variables in string output.

Now I have taken my variable (or a command) and will evaluate it and display it as a string within my text. This is very useful if you wanted to display some verbose or debug messages as well as display your own errors if needed.

Notice also how I have each of these encased in a $(). This ensures that I properly evaluate the variable or command. Not doing that will result in something like the following image.

**[https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig3_s.ashx](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig3.ashx)[Click on image for larger view.]**  ***Figure 3.*** String output not being properly displayed.

I've mentioned double quotes and given an example, but what about single quotes? Usually you will want to use these when you are only planning on displaying text without looking to evaluate any variables or commands within the text. Using my example above, you will see that everything is treated as a literal text and nothing gets evaluated.

$MyName = 'Boe Prox'

'Hello $($MyName), today is $((Get-Date).DayOfWeek)'

**[https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig4_s.ashx](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig4.ashx)[Click on image for larger view.]**  ***Figure 4.*** Showing literal values in a string.

It is definitely important to know the difference between these two so you are providing the proper output.

If you wanted to know just how many characters are in a string, you can use the Length property to find this out.

$MyName = "Boe Prox"

("Hello $($MyName), today is $((Get-Date).DayOfWeek)").Length

**[https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig5_s.ashx](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig5.ashx)[Click on image for larger view.]**  ***Figure 5.*** Displaying the length of a string.

**Here-Strings** are a great technique to use if you want to have a lot of text that cover several lines. Using HTML code is a great example of this.

@"

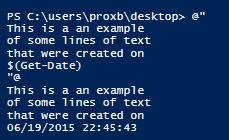
This is a an example

of some lines of text

that were created on

$(Get-Date)

"@

**[](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig6.ashx)*Figure 6.*** Example of using a Here-String.

In this example I used double quotes, but you can also use single quotes if you are not planning on evaluating variables or commands.

One last thing on this topic. Did you know that you can format these strings? It is quite simple using the –f operator to perform the formatting. Given, some of these may seem a little odd, but once you start using them, it becomes like second nature. A couple of examples are seen below.

#Show Day of the Week

"Today is {0:dddd}" -f (Get-Date)

#Percentage

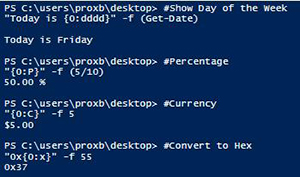
"{0:P}" -f (5/10)

#Currency

"{0:C}" -f 5

#Convert to Hex

"0x{0:x}" -f 55

**[](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig7.ashx)[Click on image for larger view.]**  ***Figure 7.*** Using .Net formatting in string output.

**Manipulating Strings**  
Now that we have spent some time with formatting strings and displaying ways to display them, we should take a look at manipulating those strings.

If for some reason you wanted to put all of the text in all caps, you can make use of the ToUpper() method that is available in the string.

$Text = "Hello there!"

$Text.ToUpper()

**[https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig8_s.ashx](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig8.ashx)[Click on image for larger view.]**  ***Figure 8.*** Making all characters upper case.

The same can be done with ToLower() to ensure that nothing is capitalized.

$Text = "THIS IS ALL CAPS"

$Text.ToLower()

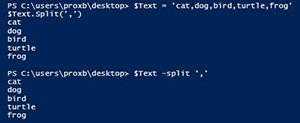
**[https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig9_s.ashx](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig9.ashx)[Click on image for larger view.]**  ***Figure 9.*** Making all characters lower case.

Let's say that you have a string that is comma separated, but you wanted to turn this into an array of items instead. We can make use of the .Split() method or use the –Split operator to perform this action. The important thing to remember is that using the .Split() method is a literal approach and is case sensitive. So what you ask for is what you get. Using –Split assumes that you are using Regular Expressions instead, so you need to make sure that you have an understanding of this technique. Although in most cases with the –Split, you can just use the similar character, such as a comma in this case.

$Text = 'cat,dog,bird,turtle,frog'

$Text.Split(',')

$Text -split ','

**[](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig10.ashx)[Click on image for larger view.]**  ***Figure 10.*** Splitting a string into an array by character.

We can also find the location of a character in a string by looking at IndexOf() and supplying the character. In this case, we can see that the letter "y" is the 37th character in the following string.

$Text = 'The quick brown fox jumps over the lazy dog'

$Index = $Text.IndexOf('y')

$Index

**[https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig11_s.ashx](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig11.ashx)[Click on image for larger view.]**  ***Figure 11.*** Determining the index of a character.

I can actually use this knowledge with SubString() to display all of the characters up until this character. With Substring, I can specify a starting index and then display all of the text up until the end or until it goes to a specified index that is specified. Note that if you specify an index that goes beyond the length of the string, it will throw an error.

$Text = 'The quick brown fox jumps over the lazy dog'

$Index = $Text.IndexOf('y')

$Text.Substring(0,$Index)

**[https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig12_s.ashx](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig12.ashx)[Click on image for larger view.]**  ***Figure 12.*** Displaying a specific set of characters using SubString.

The last thing that I will show today is how you can replace things in a string using both the .Replace() method and the Regular Expression approach using the –Replace operator. Much like the .Split() operator, it requires a literal value that is case sensitive while the regular expression provides some more flexibility with what you can replace.

An example of how case sensitivity can play into using .Replace()

$Text = "This is Boe, but boe isn't around at the moment"

$Text.Replace('Boe','Jeff')

**[https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig13_s.ashx](https://mcpmag.com/articles/2015/07/23/~/media/ECG/mcpmag/Images/2015/07/PSStrings_Fig13.ashx)[Click on image for larger view.]**  ***Figure 13.*** Replacing literal text using a method.

This example shows how you can make use of regular expressions with –Split to provide some more flexibility in what you can replace. In this case, I want to replace the first two octets in a IP address with xxx so it is not completely known.

$IPs = "168.192.1.1, 185.25.26.59, 10.15.1.12, 158.128.2.45"

$IPs -replace '\d{1,3}\.\d{1,3}\.','xxx.xxx.'

Date Properties:

$a = (get-date).day  
$a = (get-date).dayofweek  
$a = (get-date).dayofyear  
$a = (get-date).hour  
$a = (get-date).millisecond  
$a = (get-date).minute  
$a = (get-date).month  
$a = (get-date).second  
$a = (get-date).timeofday  
$a = (get-date).year

To see all the properties and methods of the DateTime object, type get-date get-member

If you specify a value that is greater than the number of days in the month, PowerShell adds the number of days to the month and displays the result. For example, "get-date -month 2 -day 31" displays "March 3", not "February 31".

**Examples**

Retrieve the current date and time, but display only the date:

PS C:\> get-date -DisplayHint date  
  
Retrieve the current date and time:

PS C:\> $now=Get-Date -format "dd-MMM-yyyy HH:mm"

Retrieve the current date and time, display as a General short date/time:

PS C:\> get-date -format g

Display the day of the year:

PS C:\> (get-date).dayofyear

Get the day of the week as an integer (1=Monday, 7=Sunday):

PS C:\> [Int]$dow = Get-Date | Select-Object -ExpandProperty DayOfWeek  
PS C:\> $dow

Display yesterdays date, using the .AddDays method:

PS C:\> (Get-Date).AddDays(-1)

Display daylight savings and UTC:

PS C:\>$a = get-date  
$a.IsDaylightSavingTime()  
$a.ToUniversalTime()

Display the bios date of a remote machine using WMI:

PS C:\>$a = get-wmiobject win32\_bios -computer SERVER64  
$a | format-list -property Name, @{Label="BIOS Date "; `  
Expression={$\_.ConvertToDateTime($\_.ReleaseDate)}}