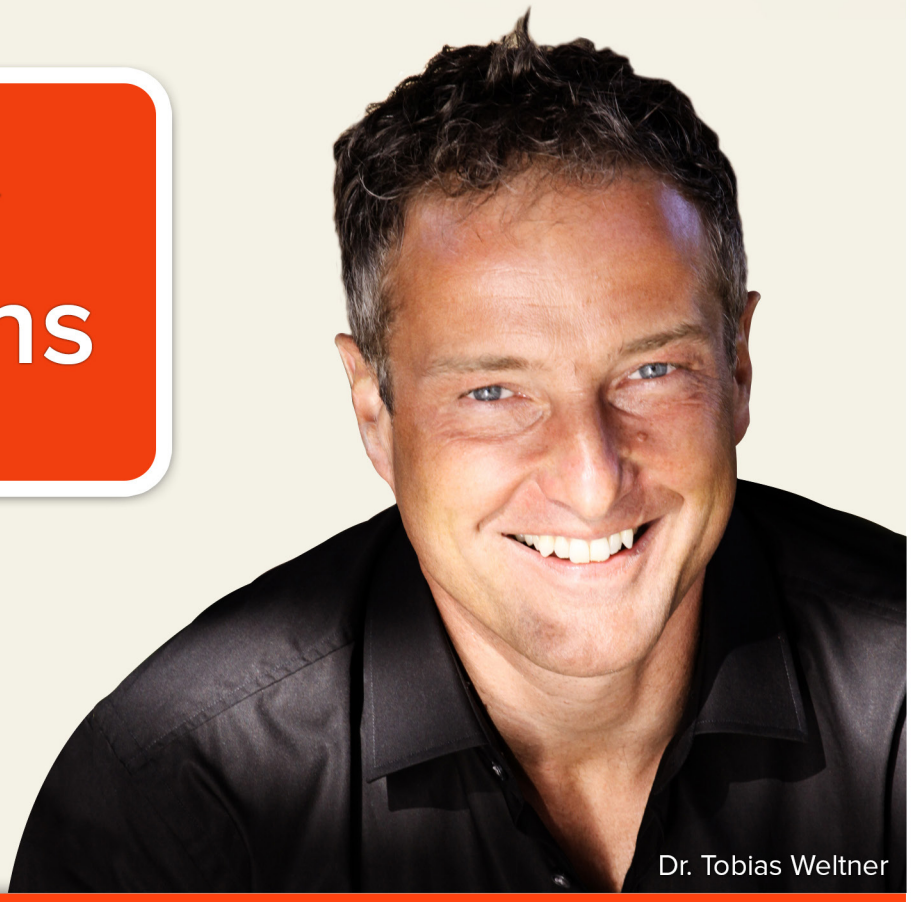
PowerTips MONTHLY

Part of the PowerShell.com reference library, brought to you by **Dr. Tobias Weltner**

Volume 7 December 2013

This Month's Topic:

Functions



Dr. Tobias Weltner

Table of Contents

1. How PowerShell Functions Work

PowerShell functions use the keyword “function” to tie a keyword to a script block. Once the keyword is tied to the script block, whenever a user enters the keyword, the script block is executed:

```
function Test-Function
{
  'Hello World!'
}
```

Curly brackets always mark code that is not executed immediately. Instead, curly brackets are used to “safely transport” PowerShell code to someone else. So, in this scenario, the script code inside the curly brackets is safely attached to the keyword “Test-Function” without executing it. That’s why nothing special happens when you run the function definition above.

Only when you enter the keyword you chose, the attached script block will execute:

```
PS> Test-Function
Hello World!
```

Note that PowerShell functions return anything that your script block “leaves behind”. In the example, a literal text string was returned. PowerShell functions can return anything, though. If it is more than one item, PowerShell automatically wraps the return values into an array.

2. Essential Function Best Practices

When you create your own function, here are some things you should consider:

- Function name: use cmdlet naming convention (Verb-Noun), and for verbs, stick to the list of approved verbs (run Get-Verb to see the list). For the noun part, use a meaningful English term, and use singular, not plural. So, don't call a function 'ListNetworkCards' but rather 'Get-NetworkCard'

- Company Prefix: To avoid name collisions, all public functions should use your very own noun-prefix. So don't call your function "Get-NetworkCard" because this generic name might be used elsewhere, too. Instead, pick a prefix for your company. If you work for, let's say, 'Global International', a prefix could be 'GI', and your function name would be "Get-GINetworkCard".

- Standard Parameter Names: Stick to meaningful standard parameter names. Don't call a parameter -PC. Instead, call it -ComputerName. Don't call it -File. Instead, call it -Path. While there is no official list of approved parameter names, you should get familiar with the parameter names used by the built-in cmdlets to get a feeling for it.

- Optional Parameters: always define at least a default value. If the user omits the parameter, then a default value ensures that the parameter is in a defined state.

- Mandatory Parameters: always define at least a type. If the user omits the parameter and gets prompted, by defining a type you make sure that the user input is converted into the correct type and not treated as a string (unless a string is what your parameter expects anyway).

- Add Comment-Based Help: always add a standard help comment block so your function provides help just like cmdlets do.

All of these best practices are illustrated in a number of examples inside this document.

3. Defining Function Parameters

Function parameters can be defined in two ways. There is a simplified syntax like this:

```
function Test-Function ($Parameter1='Default Value1', $Parameter2='Default Value2')
{
  "You entered $Parameter1 and $Parameter2"
}
```

This syntax is not best practice because PowerShell internally translates this syntax to the official syntax anyway, which looks like this:

```
function Test-Function
{
  param($Parameter1='Default Value1', $Parameter2='Default Value2')

  "You entered $Parameter1 and $Parameter2"
}
```

Basically, the parameter block goes inside the function and is marked with the keyword "param".

Either way, the function returns the parameter values:

```
PS> Test-Function
You entered Default Value1 and Default Value2

PS> Test-Function -Parameter1 'New Text'
You entered New Text and Default Value2
```

4. Picking Standard Parameter Names

While you can call your parameters as you wish, to make things consistent, you should stick to standard parameter names and always capitalize the first letter.

There is no predefined list of standard parameter names, but you can create one by looking at all the parameters from all your cmdlets:

```
Get-Command -CommandType Cmdlet |
  ForEach-Object { $_.Parameters } |
  ForEach-Object { $_.Keys } |
  Group-Object -NoElement |
  Sort-Object Count, Name -Descending |
  Select-Object -Skip 11 |
  Where-Object { $_.Count -gt 1 } |
  Out-GridView
```

The list skips the eleven common parameters as well as parameters that occur only once. The remaining parameters are displayed in a grid view window, and you can then use the full text search to play and find a good parameter name.

5. Using Mandatory Parameters

By default, parameters are optional. To make a parameter mandatory, add this declaration:

```
function Test-Me {
  param
  (
    [Parameter(Mandatory=$true)]
    $name
  )

  "You entered $name."
}
```

The declaration always applies to the following parameter.

Now when you run the function without any parameters, PowerShell prompts for it:

```
PS> Test-Me
cmdlet Test-Me at command pipeline position 1
Supply values for the following parameters:
name:
```

Author Bio

Tobias Weltner is a long-term Microsoft PowerShell MVP, located in Germany. Weltner offers entry-level and advanced PowerShell classes throughout Europe, targeting mid- to large-sized enterprises. He just organized the first German PowerShell Community conference which was a great success and will be repeated next year (more on www.pscommunity.de).. His latest 950-page "PowerShell 3.0 Workshop" was recently released by Microsoft Press.

To find out more about public and in-house training, get in touch with him at tobias.weltner@email.de.

In PowerShell 3.0, the mandatory attribute defaults to $true, so you get away with:

```
param(
  [Parameter(Mandatory)]
  $p
)
```

However, the price you pay for this shortcut is that your function will no longer run in PowerShell 2.0.

6. Add Help Messages to Mandatory Parameters

There are two more things to do to safely use mandatory parameters: first, add a help message so users know better what to input:

```
function Test-Me {
  param
  (
    [Parameter(Mandatory=$true, HelpMessage='Enter your name please')]
    $name
  )

  "You entered $name."
}
```

Now when the user gets prompted, there is also a help prompt offering detailed help on "!?":

```
PS> Test-Me
cmdlet Test-Me at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
name: !?
Enter your name please
```

So when the user enters "!?", your help message appears and assists the user.

7. Strongly-Typed Mandatory Parameters

To safely use mandatory parameters, aside from adding a help message, you should always strongly-type the parameter.

While you can add a data type to any parameter (and any variable in general), this is crucial for mandatory parameters. Here is why: if the user omits the parameter and gets prompted for it, anything entered by the user will be treated as string (text). This can have weird side effects.

Here is a currency calculator that accepts EUROs and returns DOLLARs:

```
function Test-Me {
  param
  (
    [Parameter(Mandatory=$true, HelpMessage='Enter number of EUROs!')]
    $Euro
  )

  $Dollar = $Euro * 1.4
  $Dollar
}
```

When you submit the argument, all is fine. When PowerShell prompts you for it, the function gets the wrong result:

```
PS> Test-Me -Euro 100
140

PS> Test-Me
cmdlet Test-Me at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
Euro: 100
100
```

Reason: in the second case, the user input was treated as string, so PowerShell did a string calculation, repeating the string 1.4 times (which results in one repeated string).

Once you strongly-type the parameter, all is fine:

```
function Test-Me {
  param
  (
    [Parameter(Mandatory=$true, HelpMessage='Enter number of EUROs!')]
    [Double]
    $Euro
  )

  $Dollar = $Euro * 1.4
  $Dollar
}
```

Technical Editor Bio

Aleksandar Nikolic, Microsoft MVP for Windows PowerShell, a frequent speaker at the conferences (Microsoft Sinergija, PowerShell Deep Dive, NYC Techstravaganza, KulenDayz, PowerShell Summit) and the cofounder and editor of the PowerShell Magazine (http://powershellmagazine.com). He is also available for one-on-one online PowerShell trainings. You can find him on Twitter: https://twitter.com/alexandair

8. Masked Mandatory Parameters

If you mark a parameter as mandatory and set its type to "SecureString," PowerShell will automatically prompt for the password with masked characters:

```
function Test-Password {
        param
        (
                [System.Security.SecureString]
                [Parameter(Mandatory=$true)]
                $password
        )


        $plain = (New-Object System.Management.Automation.PSCredential('dummy',$password)).
GetNetworkCredential().Password


"You entered: $plain"
}
```

The parameter will then become a SecureString which you cannot directly read. However, it is fairly easy to reconvert a SecureString back to a plain text using an object called PSCredential as shown above.

9. Using Switch Parameters

Switch parameters work like a switch, so they are either "on" or "off" aka $true or $false.

To add a switch parameter to a function, cast the parameter to [switch] like this:

```
function Test-SwitchParameter {
  param
  (
    [Switch]
    $DoSpecial
  )

  if ($DoSpecial)
  {
    'I am doing something special'
  }
  else
  {
    'I am doing the usual stuff...'
  }
}
```

Here is the result:

```
PS> Test-SwitchParameter
I am doing the usual stuff...

PS> Test-SwitchParameter -DoSpecial
I am doing something special
```

10. Using Parameter Set Magic

PowerShell functions do not support so-called “overloads”, but they support something similar: ParameterSets. So you can define groups of parameters. Users can choose between these parameter groups but cannot mix the parameters from different groups.

This way, it is easy to prevent a user from combining parameters that make no particular sense.

Here is a simple example. The function Add-User either accepts a Name, a SAMAccountName or a DistinguishedName, but not a combination of these:

```
function Add-User
{
	[CmdletBinding(DefaultParameterSetName='A')]
	param
	(
	  [Parameter(ParameterSetName='A',Mandatory=$true)]
	  $Name,

	  [Parameter(ParameterSetName='B',Mandatory=$true)]
	  $SAMAccountName,

	  [Parameter(ParameterSetName='C',Mandatory=$true)]
	  $DN
	)

	$chosen = $PSCmdlet.ParameterSetName
	“You have chosen $chosen parameter set.”
}
```

As you can see, the automatic variable $PSCmdlet tells you which parameter set was chosen. If the user combines parameters from different parameter sets, an error message appears.

```
PS> Add-User -Name test
You have chosen A parameter set.

PS> Add-User -SAMAccountName test
You have chosen B parameter set.

PS> Add-User -DN test
You have chosen C parameter set.

PS> Add-User -DN test -Name test
Add-User : Parameter set cannot be resolved using the specified named parameters.
```

11. Use Parameter Sets to Automatically Bind Data Types

You can also use parameter sets to bind input data automatically to the appropriate parameter set:

```
function Test-Binding {
	[CmdletBinding(DefaultParameterSetName='Name')]
	param(
		[Parameter(ParameterSetName='ID', Position=0, Mandatory=$true)]
		[Int]
		$id,
		[Parameter(ParameterSetName='Name', Position=0, Mandatory=$true)]
		[String]
		$name
	)
```

```
        $set = $PSCmdlet.ParameterSetName
        “You selected $set parameter set”

        if ($set -eq ‘ID’) {
                “The numeric ID is $id”
        } else {
                “You entered $name as a name”
        }
}
```

So now, even if the user does not use parameter names but simply inputs raw arguments, PowerShell still automatically picks the right parameter set, based on the best-matching parameter type:

```
PS> Test-Binding -Name hallo
You selected Name parameter set
You entered hallo as a name

PS> Test-Binding -Id 12
You selected ID parameter set
The numeric ID is 12

PS> Test-Binding hallo
You selected Name parameter set
You entered hallo as a name

PS> Test-Binding 12
You selected ID parameter set
The numeric ID is 12
```

12. Optional and Mandatory at the Same Time

You can use parameters in multiple parameter sets and make them both optional and mandatory, depending on the situation and other parameters entered.

For example, the next function accepts two parameters: -ComputerName and -Credential. -ComputerName is optional, but if the user uses -Credential, then -ComputerName becomes mandatory.

Here’s how you would define this:

```
function Connect-Somewhere
{
      [CmdletBinding(DefaultParameterSetName='A')]
      param
      (

        [Parameter(ParameterSetName='A',Mandatory=$false)]
        [Parameter(ParameterSetName='B',Mandatory=$true)]
        $ComputerName,

        [Parameter(ParameterSetName='B',Mandatory=$false)]
        $Credential
      )

      $chosen = $PSCmdlet.ParameterSetName
      “You have chosen $chosen parameter set.”
}
```

So you can use Connect-Somewhere without any parameters. You can also supply a -ComputerName parameter without the need for -Credential. However, once -Credential is used, -ComputerName also becomes mandatory:

```
PS> Connect-Somewhere
You have chosen A parameter set.

PS> Connect-Somewhere -ComputerName test
You have chosen A parameter set.

PS> Connect-Somewhere -Credential user1
cmdlet Connect-Somewhere at command pipeline position 1
Supply values for the following parameters:
ComputerName: NOWMANDATORY!
You have chosen B parameter set.
```

13. Limiting Number of Arguments

Parameters can receive multiple values when they accept arrays. So this function would accept a single string as well as a comma-separated list of strings:

```
function Add-User
{
    param
    (
        [String[]]
        $UserName
    )

   $UserName | ForEach-Object { “Adding $_” }
}
```

Here is the proof:

```
PS> Add-User -UserName ‘Tobias’
Adding Tobias

PS> Add-User -UserName ‘Tobias’, ‘Nina’, ‘Cofi’
Adding Tobias
Adding Nina
Adding Cofi
```

To make parameters accept multiple arguments but limit the number of arguments, use a validator. This function will accept at most two users:

```
function Add-User
{
    param
    (
        [ValidateCount(1,2)]
        [String[]]
        $UserName
    )

   $UserName | ForEach-Object { “Adding $_” }
}
```

Here is the result:

```
PS> Add-User -UserName 'Tobias', 'Nina'
Adding Tobias
Adding Nina

PS> Add-User -UserName 'Tobias', 'Nina', 'Cofi'
Add-User : Cannot validate argument on parameter 'UserName'. The number of provided
arguments, (3), exceeds the maximum number of allowed arguments (2). Provide fewer than 2
arguments, and then try the command again.
```

14. Rich IntelliSense for Function Arguments

To take advantage of the new PowerShell 3.0 argument completion, make sure you're adding ValidateSet attribute to your function parameters (where appropriate). In previous versions of PowerShell, ValidateSet made sure the user could enter only values listed in the validation set. In PowerShell 3.0, ISE now also uses the list to provide IntelliSense.

Here is a sample:

```
function Select-Color
{
  param(
  [ValidateSet('Red','Green','Blue')]
  $Color
  )

  "You chose $Color"
}
```

So after you run this code, when you enter the line below, in the ISE you will get an IntelliSense menu with the three colors your parameter accepts, and in the powershell.exe console you can press TAB to view the legal values:

```
PS> Select-Color -Color  <- IntelliSense Menu Opens in ISE
```

15. Using Enumeration Types for Parameter IntelliSense

As an alternative to ValidateSet, you can assign an enumeration data type to a parameter. Provided you pick the right data type, you wouldn't even have to put together the list of values:

```
function Select-Color
{
  param(
  [System.ConsoleColor]
  $Color
  )

  "You chose $Color"
}
```

Once you run this code and then in the ISE, enter this, you'll see the new argument completion in action:

```
Select-Color -Color <-ISE opens IntelliSense menu with color values
```

16. Validating Arguments Using Patterns

You can use Regular Expression patterns to validate function parameters:

```
function Get-ZIPCode {
    param(
          [ValidatePattern('^\d{5}$')]
          [String]
          $ZIP
    )
    "Here is the ZIP code you entered: $ZIP"
}
```

You can add a [ValidatePattern()] attribute to the parameter that you want to validate, and specify the RegEx pattern that describes valid arguments.

17. Evaluating User Submitted Parameters

To find out which parameters a user actually submitted to a function, you can use $PSBoundParameters like this:

```
function Get-Parameter {

       param
       (
              $Name,
              $LastName='Default',
              $Age,
              $Id
       )
       $PSBoundParameters
}
```

Here is the result:

```
PS> Get-Parameter

PS> Get-Parameter -Name test -Id 12

Key                                                Value
---                                                -----
Name                                               test
Id                                                 12
```

Actually, $PSBoundParameters behaves like a hash table, so you can also use ContainsKey() to find out whether the user submitted a specific parameter.

18. Splatting Parameters

"Splatting" means that you take a hash table with key/value pairs in it, and then apply it to a function or cmdlet.

All keys are matched with the command parameter names, and all values are submitted to the appropriate parameters.

Sounds complex? It is not. Here is a sample. It first defines a hash table, then submits it to a cmdlet.

```
$hash = @{
  Path = $env:windir
  Filter = '*.ps1'
  Recurse = $true
  ErrorAction = 'SilentlyContinue'
  }

Get-ChildItem @hash
```

Get-ChildItem will take all the parameters from the hash table now, and start searching the entire Windows folder for PowerShell scripts. So the code is equivalent to:

```
Get-ChildItem -Path $env:windir -Filter *.ps1 -Recurse -ErrorAction SilentlyContinue
```

19. Forwarding Parameters

Splatting in combination with $PSBoundParameters can be an excellent way to forward user parameters to other commands.

Here is a sample that gets BIOS information from WMI. The function defines the parameters -ComputerName and -Credential. If the user submits those, they will be forwarded to Get-WmiObject. Else, the local machine is contacted:

```
function Get-BIOS
{
  param
  (
    $ComputerName,
    $Path
  )

  Get-WmiObject -Class Win32_BIOS @PSBoundParameters
}
```

In essence, Get-BIOS now works locally, remotely and remotely with alternate authentication.

$PSBoundParameters automatically contained the hash table with the parameters submitted by the user. If your function defines more parameters, and you would want to exclude some from splatting, remove them from $PSBoundParameters before you splat:

```
function Get-BIOS
{
  param
  (
    $SomethingElse,
    $ComputerName,
    $Path
  )

  $null = $PSBoundParameters.Remove('SomethingElse')
  "The parameter $SomethingElse still exists but will not get splatted"
  Get-WmiObject -Class Win32_BIOS @PSBoundParameters
}
```

Here is the result:

```
PS> Get-BIOS -SomethingElse MyOwnInfo -ComputerName storage1
The parameter MyOwnInfo still exists but will not get splatted


SMBIOSBIOSVersion : P03
Manufacturer      : Phoenix Technologies LTD
Name              : Ver 1.00PARTTBLw
SerialNumber      : 98H340ED2H9300237A30A1
Version           : PTLTD   - 6040000
```

20. Exiting a Function

To exit a function immediately, use the “return” statement.

The next function expects a name (including wildcards) and lists all matching processes. If no name is specified, the function outputs a warning message and exits the function using return:

```
function Get-NamedProcess
{
        param
        ($name=$null)

        if ($name -eq $null)
        {
                Write-Host -ForegroundColor Red ‘Specify a name!’
                return
        }
        Get-Process $name
}
```

Of course, this is just for illustration. There are better ways (including mandatory parameters) to achieve this.

The “return” statement can also be used to return a value to the function caller like this:

```
function ConvertTo-Binary
{
   param($Number)

   return [System.Convert]::ToString($Number, 2)
}
```

With this function, you can convert numbers to binary form:

```
PS> ConvertTo-Binary -Number 123
1111011
```

Note that “return” is not necessary, though, to return values. The function would have returned the value anyway. And if your function had “left behind” other values before, they also would be returned:

```
function ConvertTo-Binary
{
   param($Number)

   “Original Number: $Number”
   [System.Convert]::ToString($Number, 2)
   return
}
```

Here is the result:

```
PS> ConvertTo-Binary -Number 123
Original Number: 123
1111011
```

21. Defining Return Values

Basically, whenever function code "leaves behind" information, PowerShell automatically sends this information to Write-Output. If Write-Output is called multiple times, all results are wrapped in an array. These three examples all work the same:

```
function Test-ReturnValue
{
   Write-Output 1
   Write-Output 'Hello'
   Write-Output (Get-Date)
}

function Test-ReturnValue
{
   1
   'Hello'
   Get-Date
}

function Test-ReturnValue
{
   1
   'Hello'
   return Get-Date
}
```

In either case, the return values are wrapped in an array:

```
PS> Test-ReturnValue
1
Hello

Saturday, November 2, 2013 12:52:13



PS> $result = Test-ReturnValue

PS> $result[0]
1
```

Only results that are outputted directly to the console will not become part of the function return values. So if you want to output a message to the user, use some of the other Write-* cmdlets like Write-Host or Write-Warning:

```
function Test-ReturnValue
{
   Write-Host 'Starting' -ForegroundColor Green
   1
   'Hello'
   Write-Warning 'Almost done...'
   return Get-Date
}
```

Now, the messages will always be visible and never added to the function results:

```
PS> Test-ReturnValue
Starting
1
Hello
WARNING: Almost done...

Saturday, November 2, 2013 12:54:11
```

```
PS> $result = Test-ReturnValue
Starting
WARNING: Almost done...
```

22. Declaring Function Return Type

Functions can return pretty much anything they want which makes it hard for PowerShell development environments and editors to provide rich IntelliSense. Most of the time, IntelliSense only works after you ran a function, so there is data to analyze.

Beginning in PowerShell 3.0, you can declare the output type your function most likely returns. If you do this, editors can pick this up and provide rich IntelliSense even if your function has never run and delivered actual data.

So to get rich IntelliSense in PowerShell ISE 3.0, you should start adding the OutputType attribute to your functions. If you do, then ISE is able to provide IntelliSense inside your code without the need to actually have real values in your variables.

Here is a simple sample:

```
function Test-IntelliSense
{
  [OutputType('System.DateTime')]
  param()

  return Get-Date
}
```

Once you entered this function into ISE 3.0 (do NOT run the code!), try adding these lines:

```
$result = Test-IntelliSense
$result.
```

The moment you type a dot after the variable, ISE provides IntelliSense for the System.DateTime data type. Isn't that nice? No need to run the script all the time to fill variables for real-time IntelliSense.

Attention: if the example does not work for you, and ISE provides different IntelliSense, then that's probably OK. If the variable already has a value, PowerShell will always use the actual data to determine intelliSense options. The hint in your function will only be used if the variable is still empty and undefined.

23. Accepting Pipeline Data in Realtime

To make a function work inside a pipeline and accept information from previous cmdlets or function, at minimum you need one parameter that is marked to accept pipeline information, and place your code inside a process block:

```
function Test-Pipeline {
  param(
    [Parameter(ValueFromPipeline=$true)]
    $InputObject
  )

  process
  {
    "Working with $InputObject"
  }
}
```

Here is the result:

```
PS> 1..4 | Test-Pipeline
Working with 1
Working with 2
Working with 3
Working with 4
```

If you do not mark a parameter, then PowerShell would not know where to put the information that comes from an upstream command. And if you do not put the code inside the process block, then PowerShell would automatically place it into an end block. So your function would only process the last incoming data set.

24. Accepting Pipeline Data as a Block

If your function wants the incoming pipeline data in one chunk and not in real time, then you can use the special variable $Input.

You should convert $Input into an array before you use it because $Input is a one-time use enumerator only, so you can only read data once. By converting it to an array, you are free to use the data as often as you want:

```
function Test-Pipeline {
  $pipelineData = @($Input)

  $Count = $pipelineData.Count
  "Received $Count elements: $pipelineData"
}
```

Here's the result::

```
PS> 1..10 | Test-Pipeline
Received 10 elements: 1 2 3 4 5 6 7 8 9 10
```

If you do not convert $Input to an array before you use it, the results may turn weird:

```
function Test-Pipeline {
  $Count = $Input.Count
  "Received $Count elements: $Input"
}
```

```
PS> 1..10 | Test-Pipeline
Received 1 1 1 1 1 1 1 1 1 1 elements:
```

25. Using Pipeline Filters

You can use the keyword "Filter" instead of "Function" to create a simple pipeline-aware function. Albeit filters are deprecated, they still work and can be an excellent and fast way to create pipeline filters.

Here is a filter that will only display processes that have a "MainWindowTitle" text, thus filtering all running processes and showing only those that have an application window:

```
Filter Test-ApplicationProgram
{
    if ($_.MainWindowTitle -ne '')
    {
      $_
    }
}
```

And here is a sample call that will return only processes that have a main window:

```
PS> Get-Process | Test-ApplicationProgram

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    787      53   110040     102008   319     7,91   7600 chrome
```

```
   1257     144   412284     372672   893   104,19   1272 devenv
    583      61    15792      38608   214     3,39   3288 iexplore
     62       7     1936       7184    78     0,17   1296 notepad
(...)
```

Internally, filters are just functions with a process block, so you could have also written:

```
Function Test-ApplicationProgram
{
  process
  {
    if ($_.MainWindowTitle -ne '')
    {
      $_
    }
  }
}
```

26. Determine Functions Pipeline Position

Assume your function wanted to know whether it is the last element in a pipeline or operating in the middle of it.

For example, if the function is the last element in the pipeline, it may safely assume that it can output the data in a more sophisticated way, whereas if other cmdlets are following, it might decide to just output plain data and leave it to the following cmdlets to process the data.

Here is a way for a function to determine its current pipeline position:

```
function Test-PipelinePosition {
    param
    (
      [Parameter(ValueFromPipeline=$true)]
      $data
    )

    process
    {
      if ($MyInvocation.PipelinePosition -ne $MyInvocation.PipelineLength)
      {
         $data
      }
      else
      {
         Write-Host $data -ForegroundColor Red -BackgroundColor White
      }
    }
}
```

And here's the result:

```
PS> 1..3 | Test-PipelinePosition
1
2
3

PS> 1..3 | Test-PipelinePosition | Sort-Object -Descending
3
2
1
```

As you can see, the formatting changed.

Note, however, that the function would still not know whether the user wanted to store the results in a variable, in which case the formatting would prevent this:

```
PS> $a = 1..3 | Test-PipelinePosition
1
2
3
```

So for a function to safely decide whether it can process data internally or whether someone else picks up its data, the function would also need to know whether it is part of a variable assignment. Here is an example that adds this check by examining the call line by the PowerShell parser:

```
function Test-PipelinePosition {
    param
    (
      [Parameter(ValueFromPipeline=$true)]
      $data
    )

    begin
    {
      $isAssignment = [System.Management.Automation.Language.Parser]::ParseInput($MyInvocation.Line,
[ref]$null, [ref]$null).FindAll({$args[0] -is
[System.Management.Automation.Language.AssignmentStatementAst]}, $true) -ne $null
      $isCmdletFollowing = $MyInvocation.PipelinePosition -ne $MyInvocation.PipelineLength
      $useRaw = $isAssignment -or $isCmdletFollowing
    }
    process
    {
      if ($useRaw)
      {
         $data
      }
      else
      {
         Write-Host $data -ForegroundColor Red -BackgroundColor White
      }
    }
}
```

The function now has two pieces of information: $isAssignment is true if it is part of an assignment, and $isCmdletFollowing is true if a cmdlet follows in the pipeline. However, this really is a never-ending story because the next challenge would be to detect whether the function is an argument to another function or cmdlet.

27. Adding Write Protection to Functions

Functions by default are not write-protected so they can easily be overwritten and redefined. To make a function read-only, first define the function:

```
function Test-ImportantFunction
{
  "You cannot overwrite me!"
}
```

Next, add write protection to it:

```
(Get-Item function:Test-ImportantFunction).Options = 'ReadOnly'
```

Now, the function cannot be changed anymore. If you try, you get an error:

```
Cannot write to function Test-ImportantFunction because it is read-only or constant.
At line:1 char:1
```

You can still remove and then redefine the function, though. This would remove the function, regardless of its read-only status. Simply use the -Force parameter:

```
Remove-Item function:Test-ImportantFunction -Force
```

To make a function “Constant” (and prevent even the deletion as shown a moment ago), you must define the function as Constant in the first place:

```
$code =
{
  “You cannot overwrite me!”
}

New-Item function:Test-ImportantFunction -Options Constant -Value $code
```

Now, the function cannot be changed or removed for as long as PowerShell runs.

28. Spying On Functions and Viewing Source Code

Once you know the name of a PowerShell function, you can always view its source code no matter who defined the function.

For example, in the ISE editor, there is a function called “New-ISESnippet”. This is how you can dump its source code (from within the ISE editor since the function does not exist in a regular PowerShell console):

```
${function:New-IseSnippet}
```

To copy the source code to the clipboard, simply try this:

```
${function:New-IseSnippet} | clip.exe
```

You could also immediately add the source code to the ISE editor:

```
$psISE.CurrentPowerShellTab.Files.Add().Editor.Text = ${function:New-IseSnippet}
```

From here, it’s just a small step to create a function that takes a function name, and then loads the function source code into the ISE editor:

```
function Get-FunctionSourceCode
{
  param($FunctionName)

  $Path = “function:$FunctionName”

  if (Test-Path -Path $Path)
  {
    $sourceCode = Get-Item -Path $Path |
      Select-Object -ExpandProperty Definition

    $psISE.CurrentPowerShellTab.Files.Add().Editor.Text = $sourceCode
  }
  else
  {
    Write-Warning “Function $FunctionName not found.”
  }

}
```

29. Using Common Parameters

While your own function parameters are defined inside the “param” block, your function can also use the standard set of common parameters - just like cmdlets.

Common parameters are automatically added once you use advanced parameter declarations. To make sure your function gets common parameters, add the statement "`[CmdletBinding()]`". It needs to be the first code statement inside your function, just before the `param()` block. When you use this statement, make sure you also use a `param()` block. If you do not want to define own parameters, then leave it blank ("`param()`").

Here is a function that only supplies its own parameters, no common parameter:

```
function Test-Function
{

  param
  (
      $MyParameter
  )
}
```

When you run it, PowerShell will only show the one parameter you defined. And here is the same function with the set of common parameters added:

```
function Test-Function
{
  [CmdletBinding()]
  param
  (
      $MyParameter
  )
}
```

One of the common parameter is -Verbose. If you use Write-Verbose in your function, the user can now control whether or not these messages appear (by default, they are hidden):

```
function Test-Function
{
  [CmdletBinding()]
  param
  (
      $MyParameter
  )

  Write-Verbose "Starting"
  "You entered $MyParameter"
  Write-Verbose "Ending"
}
```

And here is the result:

```
PS> Test-Function -MyParameter ABC
You entered ABC

PS> Test-Function -MyParameter ABC -Verbose
VERBOSE: Starting
You entered ABC
VERBOSE: Ending
```

30. Using Risk Mitigation Parameters

There are two parameters that only appear in cmdlets that can do harm (change things):
-WhatIf
-Confirm
Using these, a user can decide to simulate or get individual confirmations.

Your function can use these parameters as well once it declares that it can actually be harmful.

It can declare “SupportsShouldProcess” to have the risk mitigation parameters become available, and it can declare a “ConfirmImpact” that indicates a severity level. So if a function does something that cannot be undone, it should declare a “high” ConfirmImpact. By default, once a function declares a severity of “High”, PowerShell will automatically ask for confirmation (controlled by $ConfirmPreference).

Here is a sample function that creates a folder for you and supports the risk mitigation parameters:

```
function New-Folder
{
    [CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact='Medium')]
    param
    (
      $Path
    )

    New-Item -Path $Path -ItemType Directory
}
```

And here is the result:

```
PS> New-Folder -Path c:\newfolder1 -WhatIf
What if: Performing the operation “Create Directory” on target “Destination: C:\newfolder1”.

PS> New-Folder -Path c:\newfolder1


    Directory: C:\


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         02.11.2013     15:21            newfolder1
```

Basically, in this simple scenario, the WhatIf and Confirm mode was forwarded to the cmdlets your function uses.

31. Using Custom Risk Mitigation Code

To fully control which parts of your function get executed and which get skipped when a user uses -WhatIf or -Confirm, use this sample:

```
function New-Folder
{
    [CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact='Medium')]
    param
    (
      [Parameter(Mandatory=$true)]
      $Path
    )

    # remove all limitations for remaining code:
    $WhatIfPreference = $false
    $ConfirmPreference = 'None'

    # use limitations only here:
    if ($PSCmdlet.ShouldProcess($env:COMPUTERNAME,"Creating Folder $Path"))
    {
      New-Item -Path $Path -ItemType Directory
    }
    else
    {
      Write-Host 'Just simulating...!' -ForegroundColor Green
    }
}
```

Here is how it works:

```
PS> New-Folder -Path c:\testfolder2 -WhatIf
What if: Performing the operation "Creating Folder c:\testfolder2" on target "TOBIASAIR1".
Just simulating...!
```

The function first resets the WhatIf- and ConfirmPreferences so any code inside the function will run, regardless of the submitted parameters. Next, the function checks what the current simulation mode is (using $PSCmdlet.ShouldProcess()). This way, the function controls the messages sent to the user, and it controls exactly which code will be skipped or replaced.

32. Adding Rich Help

To add rich help information to your function, use a comment block similar to this one:

```
<#
.SYNOPSIS
  Short description
.DESCRIPTION
  Long description
.EXAMPLE
  Example of how to use this cmdlet
.EXAMPLE
  Another example of how to use this cmdlet
#>
```

Place this comment block right before your function (no more than one blank line in between), right inside the function, or at the end of a function. In addition, add comments to each of your parameters.

Now, once you run your function, you get the same detailed help that you know from cmdlets. You can use Get-Help to open help, or inside the ISE editor click your function name and press F1 (note that the help shown inside the ISE editor may be incomplete due to a bug in older PowerShell versions).

```
PS> Get-Help Test-Help -Examples

NAME
    Test-Help

SYNOPSIS
    Short description

    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>Example of how to use this function

    -------------------------- EXAMPLE 2 --------------------------

    C:\PS>Another example of how to use this function

PS> get-help Test-Help -Parameter *

-Parameter1 <Object>
    Documentation for parameter1

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       false
    Accept wildcard characters?  false
```

```
-Parameter2 <Object>
    Documentation for parameter2

    Required?                    false
    Position?                    2
    Default value
    Accept pipeline input?       false
    Accept wildcard characters?  false
```

Here are some examples:

Help above outside:

```
<#
.SYNOPSIS
  Short description
.DESCRIPTION
  Long description
.EXAMPLE
  Example of how to use this function
.EXAMPLE
  Another example of how to use this function
#>
function Test-Help
{
  param
  (
    # Documentation for parameter1
    $Parameter1,

    # Documentation for parameter2
    $Parameter2
  )

  'I am documented'
}
```

Help inside top:

```
function Test-Help
{
<#
.SYNOPSIS
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this function
.EXAMPLE
   Another example of how to use this function
#>
  param
  (
    # Documentation for parameter1
    $Parameter1,

    # Documentation for parameter2
    $Parameter2
  )

  'I am documented'
}
```

Help inside bottom:

```
function Test-Help
{
  param
  (
    # Documentation for parameter1
    $Parameter1,

    # Documentation for parameter2
    $Parameter2
  )

  'I am documented'
  <#
.SYNOPSIS
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this function
.EXAMPLE
   Another example of how to use this function
#>
}
```

And this is the fully-defined comment block you can use:

```
<#
.Synopsis
  Short description
.DESCRIPTION
  Long description
.EXAMPLE
  Example of how to use this function
.EXAMPLE
  Another example of how to use this function
.INPUTS
  Inputs to this function (if any)
.OUTPUTS
  Output from this function (if any)
.NOTES
  General notes
.COMPONENT
  The component this function belongs to
.ROLE
  The role this function belongs to
.FUNCTIONALITY
  The functionality that best describes this function
#>
```