about_Functions - PowerShell

sdwheeler ⋮ 22-28 minutes

---

about_Functions

- Article
- 06/26/2024
-

In this article

1. Short description
2. Long description
3. Syntax
4. Input processing methods
5. Simple functions
6. Function Names
7. Functions with Parameters
8. Using Splatting to Represent Command Parameters
9. Piping Objects to Functions
10. Filters
11. Function Scope
12. Finding and Managing Functions Using the Function: Drive
13. Reusing Functions in New Sessions
14. Writing Help for Functions
15. See also

Short description

Describes how to create and use functions in PowerShell.

Long description

A function is a list of PowerShell statements that has a name that you assign. When you run a function, you type the function name. The statements in the list run as if you had typed them at the command prompt.

Functions can be as simple as:

```
function Get-PowerShellProcess { Get-Process pwsh }
```

Once a function is defined, you can use it like the built-in cmdlets. For example, to call the newly defined `Get-PowerShellProcess` function:

```
Get-PowerShellProcess
```

```
 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
    110    78.72     172.39      10.62   10936   1 pwsh
```

A function can also be as complex as a cmdlet or an application.

Like cmdlets, functions can have parameters. The parameters can be named, positional, switch, or dynamic parameters. Function parameters can be read from the command line or from the pipeline.

Functions can return values that can be displayed, assigned to variables, or passed to other functions or cmdlets. You can also specify a return value using the `return` keyword. The `return` keyword doesn't affect or suppress other output returned from your function. However, the `return` keyword exits the function at that line. For more information, see about_Return.

The function's statement list can contain different types of statement lists with the keywords `begin`, `process`, end, and `clean`. These statement lists handle input from the pipeline differently.

The filter keyword is used to create a type of function that runs on each object in the pipeline. A filter resembles a function with all its statements in a `process` block.

Functions can also act like cmdlets. You can create a function that works just like a cmdlet without using C# programming. For more information, see about_Functions_Advanced.

Important

Within script files and script-based modules, functions must be defined before they can be called.

Syntax

The following are the syntax for a function:

```
function [<scope:>]<name> [([type]$parameter1[,[type]$parameter2])]
{
  begin {<statement list>}
  process {<statement list>}
  end {<statement list>}
  clean {<statement list>}
}
```

```
function [<scope:>]<name>
{
  param([type]$parameter1 [,[type]$parameter2])
  dynamicparam {<statement list>}
  begin {<statement list>}
  process {<statement list>}
  end {<statement list>}
  clean {<statement list>}
}
```

A function includes the following items:

- A `function` keyword
- A scope (optional)
- A name that you select
- Any number of named parameters (optional)
- One or more PowerShell commands enclosed in braces `{}`

For more information about the `dynamicparam` keyword and dynamic parameters in functions, see about_Functions_Advanced_Parameters.

Input processing methods

The methods described in this section are referred to as the input processing methods. For functions, these three methods are represented by the `begin`, `process`, and `end` blocks of the function. PowerShell 7.3 adds a `clean` block process method.

You aren't required to use any of these blocks in your functions. If you don't use a named block, then PowerShell puts the code in the `end` block of the function. However, if you use any of these named blocks, or define a `dynamicparam` block, you must put all code in a named block.

The following example shows the outline of a function that contains a `begin` block for one-time preprocessing, a `process` block for multiple record processing, and an `end` block for one-time post-processing.

```
Function Test-ScriptCmdlet
{
[CmdletBinding(SupportsShouldProcess=$True)]
    Param ($Parameter1)
    begin{}
    process{}
    end{}
}
```

`begin`

This block is used to provide optional one-time preprocessing for the function. The PowerShell runtime uses the code in this block once for each instance of the function in the pipeline.

process

This block is used to provide record-by-record processing for the function. You can use a `process` block without defining the other blocks. The number of `process` block executions depends on how you use the function and what input the function receives.

The automatic variable `$_` or `$PSItem` contains the current object in the pipeline for use in the `process` block. The `$input` automatic variable contains an enumerator that's only available to functions and script blocks. For more information, see about_Automatic_Variables.

- Calling the function at the beginning, or outside of a pipeline, executes the `process` block once.
- Within a pipeline, the `process` block executes once for each input object that reaches the function.
- If the pipeline input that reaches the function is empty, the `process` block **does not** execute.
  - The `begin`, `end`, and `clean` blocks still execute.

Important

If a function parameter is set to accept pipeline input, and a `process` block isn't defined, record-by-record processing will fail. In this case, your function will only execute once, regardless of the input.

end

This block is used to provide optional one-time post-processing for the function.

clean

The `clean` block was added in PowerShell 7.3.

The `clean` block is a convenient way for users to clean up resources that span across the `begin`, `process`, and `end` blocks. It's semantically similar to a `finally` block that covers all other named blocks of a script function or a script cmdlet. Resource cleanup is enforced for the following scenarios:

1. when the pipeline execution finishes normally without terminating error
2. when the pipeline execution is interrupted due to terminating error
3. when the pipeline is halted by `Select-Object -First`
4. when the pipeline is being stopped by `Ctrl+c` or `StopProcessing()`

The clean block discards any output that's written to the **Success** stream.

Caution

Adding the `clean` block is a breaking change. Because `clean` is parsed as a keyword, it prevents users from directly calling a command named `clean` as the first statement in a script block. However, it's not likely to be a problem. The command can still be invoked using the call operator (`& clean`).

Simple functions

Functions don't have to be complicated to be useful. The simplest functions have the following format:

```
function <function-name> {statements}
```

For example, the following function starts PowerShell with the **Run as Administrator** option.

```
function Start-PSAdmin {Start-Process PowerShell -Verb RunAs}
```

To use the function, type: `Start-PSAdmin`

To add statements to the function, type each statement on a separate line, or use a semicolon `;` to separate the statements.

For example, the following function finds all `.jpg` files in the current user's directories that were changed after the start date.

```
function Get-NewPix
{
  $start = Get-Date -Month 1 -Day 1 -Year 2010
  $allpix = Get-ChildItem -Path $env:UserProfile\*.jpg -Recurse
  $allpix | Where-Object {$_.LastWriteTime -gt $Start}
}
```

You can create a toolbox of useful small functions. Add these functions to your PowerShell profile, as described in about_Profiles and later in this topic.

Function Names

You can assign any name to a function, but functions that you share with others should follow the naming rules that have been established for all PowerShell commands.

Functions names should consist of a verb-noun pair where the verb identifies the action that the function performs and the noun identifies the item on which the cmdlet performs its action.

Functions should use the standard verbs that have been approved for all PowerShell commands. These verbs help us to keep our command names consistent and easy for users to understand.

For more information about the standard PowerShell verbs, see Approved Verbs.

Functions with Parameters

You can use parameters with functions, including named parameters, positional parameters, switch parameters, and dynamic parameters. For more information about dynamic parameters in functions, see about_Functions_Advanced_Parameters.

Named Parameters

You can define any number of named parameters. You can include a default value for named parameters, as described later in this topic.

You can define parameters inside the braces using the `param` keyword, as shown in the following sample syntax:

```
function <name> {
  param ([type]$parameter1 [,[type]$parameter2])
```

```
  <statement list>
}
```

You can also define parameters outside the braces without the `Param` keyword, as shown in the following sample syntax:

```
function <name> [([type]$parameter1[,[type]$parameter2])] {
  <statement list>
}
```

Below is an example of this alternative syntax.

```
function Add-Numbers([int]$one, [int]$two) {
    $one + $two
}
```

While the first method is preferred, there is no difference between these two methods.

When you run the function, the value you supply for a parameter is assigned to a variable that contains the parameter name. The value of that variable can be used in the function.

The following example is a function called `Get-SmallFiles`. This function has a `$Size` parameter. The function displays all the files that are smaller than the value of the `$Size` parameter, and it excludes directories:

```
function Get-SmallFiles {
  Param($Size)
  Get-ChildItem $HOME | Where-Object {
    $_.Length -lt $Size -and !$_.PSIsContainer
  }
}
```

In the function, you can use the `$Size` variable, which is the name defined for the parameter.

To use this function, type the following command:

```
Get-SmallFiles -Size 50
```

You can also enter a value for a named parameter without the parameter name. For example, the following command gives the same result as a command that names the **Size** parameter:

```
Get-SmallFiles 50
```

To define a default value for a parameter, type an equal sign and the value after the parameter name, as shown in the following variation of the `Get-SmallFiles` example:

```
function Get-SmallFiles ($Size = 100) {
  Get-ChildItem $HOME | Where-Object {
    $_.Length -lt $Size -and !$_.PSIsContainer
  }
}
```

If you type `Get-SmallFiles` without a value, the function assigns 100 to `$size`. If you provide a value, the function uses that value.

Optionally, you can provide a brief help string that describes the default value of your parameter, by adding the **PSDefaultValue** attribute to the description of your parameter, and specifying the **Help** property of **PSDefaultValue**. To provide a help string that describes the default value (100) of the **Size** parameter in the `Get-SmallFiles` function, add the **PSDefaultValue** attribute as shown in the following example.

```
function Get-SmallFiles {
  param (
      [PSDefaultValue(Help = '100')]
      $Size = 100
  )
  Get-ChildItem $HOME | Where-Object {
    $_.Length -lt $Size -and !$_.PSIsContainer
  }
}
```

For more information about the **PSDefaultValue** attribute class, see PSDefaultValue Attribute Members.

Positional Parameters

A positional parameter is a parameter without a parameter name. PowerShell uses the parameter value order to associate each parameter value with a parameter in the function.

When you use positional parameters, type one or more values after the function name. Positional parameter values are assigned to the `$args` array variable. The value that follows the function name is assigned to the first position in the `$args` array, `$args[0]`.

The following `Get-Extension` function adds the `.txt` filename extension to a filename that you supply:

```
function Get-Extension {
  $name = $args[0] + ".txt"
  $name
}
```

```
Get-Extension myTextFile
```

```
myTextFile.txt
```

Switch Parameters

A switch is a parameter that doesn't require a value. Instead, you type the function name followed by the name of the switch parameter.

To define a switch parameter, specify the type `[switch]` before the parameter name, as shown in the following example:

```
function Switch-Item {
  param ([switch]$on)
  if ($on) { "Switch on" }
  else { "Switch off" }
}
```

When you type the 0n switch parameter after the function name, the function displays `Switch on`. Without the switch parameter, it displays `Switch off`.

```
Switch-Item -on
```

```
Switch on
```

```
Switch-Item
```

```
Switch off
```

You can also assign a **Boolean** value to a switch when you run the function, as shown in the following example:

```
Switch-Item -on:$true
```

```
Switch on
```

```
Switch-Item -on:$false
```

```
Switch off
```

Using Splatting to Represent Command Parameters

You can use splatting to represent the parameters of a command. This feature is introduced in Windows PowerShell 3.0.

Use this technique in functions that call commands in the session. You don't need to declare or enumerate the command parameters, or change the function when command parameters change.

The following sample function calls the `Get-Command` cmdlet. The command uses `@Args` to represent the parameters of `Get-Command`.

```
function Get-MyCommand { Get-Command @Args }
```

You can use all the parameters of `Get-Command` when you call the `Get-MyCommand` function. The parameters and parameter values are passed to the command using `@Args`.

```
Get-MyCommand -Name Get-ChildItem
```

```
CommandType     Name                ModuleName
-----------     ----                ----------
Cmdlet          Get-ChildItem       Microsoft.PowerShell.Management
```

The `@Args` feature uses the `$Args` automatic parameter, which represents undeclared cmdlet parameters and values from remaining arguments.

For more information, see about_Splatting.

Piping Objects to Functions

Any function can take input from the pipeline. You can control how a function processes input from the pipeline using `begin`, `process`, `end`, and `clean` keywords. The following sample syntax shows these keywords:

The `process` statement list runs one time for each object in the pipeline. While the `process` block is running, each pipeline object is assigned to the `$_` automatic variable, one pipeline object at a time.

The following function uses the `process` keyword. The function displays values from the pipeline:

```
function Get-Pipeline
{
  process {"The value is: $_"}
}

1,2,4 | Get-Pipeline
```

```
The value is: 1
The value is: 2
The value is: 4
```

If you want a function that can take pipeline input or input from a parameter, then the `process` block needs to handle both cases. For example:

```
function Get-SumOfNumbers {
    param (
        [int[]]$Numbers
    )

    begin { $retValue = 0 }

    process {
        if ($null -ne $Numbers) {
           foreach ($n in $Numbers) {
               $retValue += $n
           }
        } else {
           $retValue += $_
        }
    }

    end { $retValue }
}

PS> 1,2,3,4 | Get-SumOfNumbers
10
PS> Get-SumOfNumbers 1,2,3,4
10
```

When you use a function in a pipeline, the objects piped to the function are assigned to the `$input` automatic variable. The function runs statements with the `begin` keyword before any objects come from the pipeline. The function runs statements with the `end` keyword after all the objects have been received from the pipeline.

The following example shows the `$input` automatic variable with `begin` and `end` keywords.

```
function Get-PipelineBeginEnd {
    begin   { "Begin: The input is $input" }
    end     { "End:   The input is $input" }
}
```

If this function is run using the pipeline, it displays the following results:

```
1,2,4 | Get-PipelineBeginEnd
```

```
Begin: The input is
End:   The input is 1 2 4
```

When the `begin` statement runs, the function doesn't have the input from the pipeline. The `end` statement runs after the function has the values.

If the function has a `process` keyword, each object in `$input` is removed from `$input` and assigned to `$_`. The following example has a `process` statement list:

```
function Get-PipelineInput
{
    process {"Processing:  $_ " }
    end     {"End:   The input is: $input" }
}
```

In this example, each object that's piped to the function is sent to the `process` statement list. The `process` statements run on each object, one object at a time. The `$input` automatic variable is empty when the function reaches the `end` keyword.

```
1,2,4 | Get-PipelineInput
```

```
Processing:  1
Processing:  2
Processing:  4
End:   The input is:
```

For more information, see Using Enumerators

PowerShell 7.3 added the `clean` block. The `clean` block is a convenient way for users to clean up resources created and used in the `begin`, `process`, and `end` blocks. It's semantically similar to a `finally` block that covers all other named blocks of a script function or a script cmdlet. Resource cleanup is enforced for the following scenarios:

1. when the pipeline execution finishes normally without terminating error
2. when the pipeline execution is interrupted due to terminating error
3. when the pipeline is halted by `Select-Object -First`
4. when the pipeline is being stopped by `Ctrl+C` or `StopProcessing()`

Caution

Adding the `clean` block is a breaking change. Because `clean` is parsed as a keyword, it prevents users from directly calling a command named `clean` as the first statement in a script block. However, it's not likely to be a problem. The command can still be invoked using the call operator (`& clean`).

Filters

A filter is a type of function that runs on each object in the pipeline. A filter resembles a function with all its statements in a `process` block.

The syntax of a filter is as follows:

```
filter [<scope:>]<name> {<statement list>}
```

The following filter takes log entries from the pipeline and then displays either the whole entry or only the message portion of the entry:

```
filter Get-ErrorLog ([switch]$Message)
{
  if ($Message) { Out-Host -InputObject $_.Message }
  else { $_ }
}
```

It can be used as follows:

```
Get-WinEvent -LogName System -MaxEvents 100 | Get-ErrorLog -Message
```

Function Scope

A function exists in the scope in which it's created.

If a function is part of a script, the function is available to statements within that script. By default, a function in a script isn't available outside of that script.

You can specify the scope of a function. For example, the function is added to the global scope in the following example:

```
function global:Get-DependentSvs {
  Get-Service | Where-Object {$_.DependentServices}
}
```

When a function is in the global scope, you can use the function in scripts, in functions, and at the command line.

Functions create a new scope. The items created in a function, such as variables, exist only in the function scope.

For more information, see about_Scopes.

Finding and Managing Functions Using the Function: Drive

All the functions and filters in PowerShell are automatically stored in the `Function:` drive. This drive is exposed by the PowerShell **Function** provider.

When referring to the `Function:` drive, type a colon after **Function**, just as you would do when referencing the `C` or `D` drive of a computer.

The following command displays all the functions in the current session of PowerShell:

```
Get-ChildItem function:
```

The commands in the function are stored as a script block in the definition property of the function. For example, to display the commands in the Help function that comes with PowerShell, type:

```
(Get-ChildItem function:help).Definition
```

You can also use the following syntax.

```
$function:help
```

For more information about the `Function:` drive, see the help topic for the **Function** provider. Type `Get-Help Function`.

Reusing Functions in New Sessions

When you type a function at the PowerShell command prompt, the function becomes part of the current session. The function is available until the session ends.

To use your function in all PowerShell sessions, add the function to your PowerShell profile. For more information about profiles, see about_Profiles.

You can also save your function in a PowerShell script file. Type your function in a text file, and then save the file with the `.ps1` filename extension.

Writing Help for Functions

The `Get-Help` cmdlet gets help for functions, as well as for cmdlets, providers, and scripts. To get help for a function, type `Get-Help` followed by the function name.

For example, to get help for the `Get-MyDisks` function, type:

```
Get-Help Get-MyDisks
```

You can write help for a function using either of the two following methods:

- Comment-Based Help for Functions

  Create a help topic using special keywords in the comments. To create comment-based help for a function, the comments must be placed at the beginning or end of the function body or on the lines preceding the function keyword. For more information about comment-based help, see about_Comment_Based_Help.

- XML-Based Help for Functions

Create an XML-based help topic, such as the type that's typically created for cmdlets. XML-based help is required if you are localizing help topics into multiple languages.

To associate the function with the XML-based help topic, use the `.EXTERNALHELP` comment-based help keyword. Without this keyword, `Get-Help` can't find the function help topic and calls to `Get-Help` for the function return only autogenerated help.

For more information about the `.EXTERNALHELP` keyword, see about_Comment_Based_Help. For more information about XML-based help, see How to Write Cmdlet Help.

See also

- about_Automatic_Variables
- about_Comment_Based_Help
- about_Function_Provider
- about_Functions_Advanced
- about_Functions_Advanced_Methods
- about_Functions_Advanced_Parameters
- about_Functions_CmdletBindingAttribute
- about_Functions_OutputTypeAttribute
- about_Parameters
- about_Profiles
- about_Scopes
- about_Script_Blocks

In this article