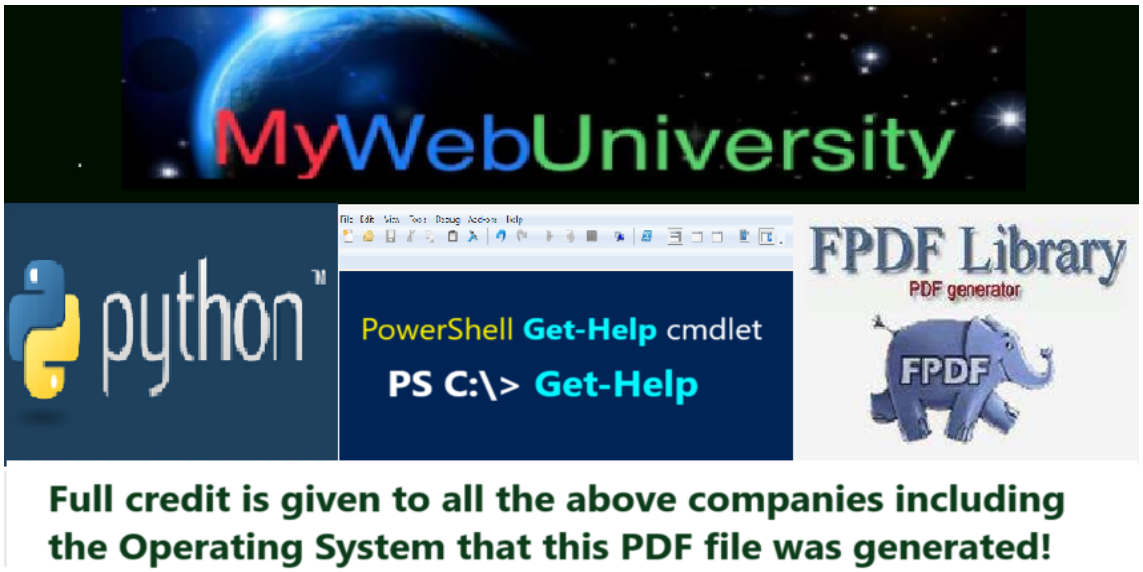

Windows PowerShell Get-Help on Cmdlet 'ForEach-Object'

PS:\>Get-HELP ForEach-Object -Full

NAME

ForEach-Object

SYNOPSIS

Performs an operation against each item in a collection of input objects.

SYNTAX

ForEach-Object [-MemberName] <System.String> [-ArgumentList <System.Object[]>] [-InputObject <System.Management.Automation.PSObject>] [-Confirm] [-WhatIf]

[<CommonParameters>]

ForEach-Object [-Process] <System.Management.Automation.ScriptBlock[]> [-Begin <System.Management.Automation.ScriptBlock>] [-End

<System.Management.Automation.ScriptBlock>] [-InputObject <System.Management.Automation.PSObject>] [-RemainingScripts <System.Management.Automation.ScriptBlock[]>]

[-Confirm] [-WhatIf] [<CommonParameters>]

DESCRIPTION

The `ForEach-Object` cmdlet performs an operation on each item in a collection of input objects. The input objects can be piped to the cmdlet or specified using the

InputObject parameter.

Starting in Windows PowerShell 3.0, there are two different ways to construct a `ForEach-Object` command.

- Script block . You can use a script block to specify the operation. Within the script block, use the `$_` variable to represent the current object. The script

block is the value of the Process parameter. The script block can contain any PowerShell script.

For example, the following command gets the value of the ProcessName property of each process on the computer.

`Get-Process | ForEach-Object {$_.ProcessName}`

`ForEach-Object` supports the `begin`, `process`, and `end` blocks as described in about_functions (about/about_functions.md#piping-objects-to-functions).

> [!NOTE] > The script blocks run in the caller's scope. Therefore, the blocks have access to variables in > that scope and can create new variables that persist

in that scope after the cmdlet completes.

- Operation statement . You can also write an operation statement, which is much more like natural language. You can use the operation statement to specify a

property value or call a method. Operation statements were introduced in Windows PowerShell 3.0.

For example, the following command also gets the value of the ProcessName property of each process on the computer.

`Get-Process | ForEach-Object ProcessName`

PARAMETERS

-ArgumentList <System.Object[]>

Specifies an array of arguments to a method call. For more information about the behavior of ArgumentList , see about_Splatting

(about/about_Splatting.md#splatting-with-arrays).

This parameter was introduced in Windows PowerShell 3.0.

Required? false

Position? named

Default value None

Accept pipeline input? False

Accept wildcard characters? false

-Begin <System.Management.Automation.ScriptBlock>

Specifies a script block that runs before this cmdlet processes any input objects. This script block is only run once for the entire pipeline. For more

information about the `begin` block, see about_Functions (about/about_functions.md#piping-objects-to-functions).

Required? false

Position? named

Default value None

Accept pipeline input? False

Accept wildcard characters? false

-End <System.Management.Automation.ScriptBlock>

Specifies a script block that runs after this cmdlet processes all input objects. This script block is only run once for the entire pipeline. For more information

about the `end` block, see about_Functions (about/about_functions.md#piping-objects-to-functions).

Required? false

Position? named

Default value None

Accept pipeline input? False

Accept wildcard characters? false

-InputObject <System.Management.Automation.PSObject>

Specifies the input objects. `ForEach-Object` runs the script block or operation statement on each input object. Enter a variable that contains the objects, or

type a command or expression that gets the objects.

When you use the InputObject parameter with `ForEach-Object`, instead of piping command results to `ForEach-Object`, the InputObject value is treated as a single

object. This is true even if the value is a collection that's the result of a command, such as `-InputObject (Get-Process)`. Because InputObject can't return

individual properties from an array or collection of objects, we recommend that if you use `ForEach-Object` to perform operations on a collection of objects for

those objects that have specific values in defined properties, you use `ForEach-Object` in the pipeline, as shown in the examples in this topic.

Required? false

Position? named

Default value None

Accept pipeline input? True (ByValue)

Accept wildcard characters? false

-MemberName <System.String>

Specifies the name of the member property to get or the member method to call. The members must be instance members, not static members.

Wildcard characters are permitted, but work only if the resulting string resolves to a unique value. For example, if you run `Get-Process | ForEach -MemberName

*Name`, the wildcard pattern matches more than one member causing the command to fail.

This parameter was introduced in Windows PowerShell 3.0.

Required? true

Position? 0

Default value None

Accept pipeline input? False

Accept wildcard characters? true

-Process <System.Management.Automation.ScriptBlock[]>

Specifies the operation that's performed on each input object. This script block is run for every object in the pipeline. For more information about the `process`

block, see about_Functions (about/about_functions.md#piping-objects-to-functions).

When you provide multiple script blocks to the Process parameter, the first script block is always mapped to the `begin` block. If there are only two script

blocks, the second block is mapped to the `process` block. If there are three or more script blocks, first script block is always mapped to the `begin` block, the

last block is mapped to the `end` block, and the middle blocks are mapped to the `process` block.

Required? true

Position? 0

Default value None

Accept pipeline input? False

Accept wildcard characters? false

-RemainingScripts <System.Management.Automation.ScriptBlock[]>

Specifies all script blocks that aren't taken by the Process parameter.

This parameter was introduced in Windows PowerShell 3.0.

Required? false

Position? named

Default value None

Accept pipeline input? False

Accept wildcard characters? false

-Confirm <System.Management.Automation.SwitchParameter>

Prompts you for confirmation before running the cmdlet.

Required? false
Position? named
Default value False
Accept pipeline input? False
Accept wildcard characters? false

-WhatIf <System.Management.Automation.SwitchParameter>

Shows what would happen if the cmdlet runs. The cmdlet isn't run.

Required? false
Position? named
Default value False
Accept pipeline input? False
Accept wildcard characters? false

<CommonParameters>

This cmdlet supports the common parameters: Verbose, Debug,
ErrorAction, ErrorVariable, WarningAction, WarningVariable,
OutBuffer, PipelineVariable, and OutVariable. For more information, see
about_CommonParameters (https:/go.microsoft.com/fwlink/?LinkID=113216).

INPUTS

System.Management.Automation.PSObject

You can pipe any object to this cmdlet.

OUTPUTS

System.Management.Automation.PSObject

This cmdlet returns objects that are determined by the input.

NOTES

Windows PowerShell includes the following aliases for `ForEach-Object`:

- `%` - `foreach`

The `ForEach-Object` cmdlet works much like the Foreach statement, except that you cannot pipe input to a Foreach statement. For more information about the

Foreach statement, see about_Foreach (./About/about_Foreach.md).

Starting in PowerShell 4.0, `Where` and `ForEach` methods were added for use with collections. You can read more about these new methods here about_arrays

(./About/about_Arrays.md)

---------- Example 1: Divide integers in an array ----------

```
30000, 56798, 12432 | ForEach-Object -Process {$_/1024}

29.296875
55.466796875
12.140625
```

-- Example 2: Get the length of all the files in a directory --

```
Get-ChildItem $PSHOME |
  ForEach-Object -Process {if (!$_.PSIsContainer) {$_.Name; $_.Length / 1024; " " }}
```

If the object isn't a directory, the script block gets the name of the file, divides the value of its Length property by 1024, and adds a space (" ") to separate it

from the next entry. The cmdlet uses the PSISContainer property to determine whether an object is a directory.

----- Example 3: Operate on the most recent System events -----

```
Get-EventLog -LogName System -Newest 1000 |
    ForEach-Object -Begin {Get-Date} -Process {
        Out-File -FilePath Events.txt -Append -InputObject $_.Message
    } -End {Get-Date}
```

`Get-EventLog` gets the 1000 most recent events from the System event log and pipes them to the `ForEach-Object` cmdlet. The Begin parameter displays the current date and time. Next, the Process parameter uses the `Out-File` cmdlet to create a text file that's named events.txt and stores the message property of each of the events in that file. Last, the End parameter is used to display the date and time after all the processing has completed.

-------- Example 4: Change the value of a Registry key --------

```
Get-ItemProperty -Path HKCU:\Network\* |
 ForEach-Object {
  Set-ItemProperty -Path $_.PSPath -Name RemotePath -Value $_.RemotePath.ToUpper()
 }
```

You can use this format to change the form or content of a registry entry value.

Each subkey in the Network key represents a mapped network drive that reconnects at sign on. The RemotePath entry contains the UNC path of the connected drive. For example, if you map the `E:` drive to `\Server\Share`, an E subkey is created in `HKCU:\Network` with the RemotePath registry value set to `\Server\Share`.

The command uses the `Get-ItemProperty` cmdlet to get all the subkeys of the Network key and the `Set-ItemProperty` cmdlet to change the value of the RemotePath registry entry in each key. In the `Set-ItemProperty` command, the path is the value of the PSPath property of the registry key. This is a property of the Microsoft .NET Framework object that represents the registry key, not a registry entry. The command uses the ToUpper() method of the RemotePath value, which is a string REG_SZ .

Because `Set-ItemProperty` is changing the property of each key, the `ForEach-Object` cmdlet is required to access the

property.

--------- Example 5: Use the $null automatic variable ---------

```
1, 2, $null, 4 | ForEach-Object {"Hello"}

Hello
Hello
Hello
Hello
```

Because PowerShell treats `$null` as an explicit placeholder, the `ForEach-Object` cmdlet generates a value for `$null` as it does for other objects piped to it.

---------- Example 6: Get property values ----------

```
Get-Module -ListAvailable | ForEach-Object -MemberName Path
Get-Module -ListAvailable | Foreach Path
```

The second command is equivalent to the first. It uses the `Foreach` alias of the `ForEach-Object` cmdlet and omits the name of the MemberName parameter, which is
optional.

The `ForEach-Object` cmdlet is useful for getting property values, because it gets the value without changing the type, unlike the Format cmdlets or the
`Select-Object` cmdlet, which change the property value type.

------ Example 7: Split module names into component names ------

```
"Microsoft.PowerShell.Core", "Microsoft.PowerShell.Host" |
    ForEach-Object {$_.Split(".")}
"Microsoft.PowerShell.Core", "Microsoft.PowerShell.Host" |
    ForEach-Object -MemberName Split -ArgumentList "."
"Microsoft.PowerShell.Core", "Microsoft.PowerShell.Host" |
    Foreach Split "."
```

Microsoft

PowerShell

Core

Microsoft

PowerShell

Host

The first command uses the traditional syntax, which includes a script block and the current object operator `$_`. It uses the dot syntax to specify the method and

parentheses to enclose the delimiter argument.

The second command uses the MemberName parameter to specify the Split method and the ArgumentList parameter to identify the dot (`.`) as the split delimiter.

The third command uses the Foreach alias of the `ForEach-Object` cmdlet and omits the names of the MemberName and ArgumentList parameters, which are optional.

---- Example 8: Using ForEach-Object with two script blocks ----

1..2 | ForEach-Object { 'begin' } { 'process' }

begin

process

process

Example 9: Using ForEach-Object with more than two script blocks

1..2 | ForEach-Object { 'begin' } { 'process A' } { 'process B' } { 'end' }

begin

process A

process B

process A

process B

end

> [!NOTE] > The first script block is always mapped to the `begin` block, the last block is mapped to the > `end` block, and the two middle blocks are mapped to the

`process` block.

Example 10: Run multiple script blocks for each pipeline item

```
1..2 | ForEach-Object -Begin $null -Process { 'one' }, { 'two' }, { 'three' } -End $null
```

```
one
two
three
one
two
three
```

RELATED LINKS

Online Version: https://learn.microsoft.com/powershell/module/microsoft.powershell.core/foreach-object?view=powershell-5.1&WT.mc_id=ps-gethelp

Compare-Object

Where-Object

Group-Object

Measure-Object

New-Object

Select-Object

Sort-Object

Tee-Object