# about_Operators

Article • 09/03/2024

## Short description

Describes the operators that are supported by PowerShell.

## Long description

An operator is a language element that you can use in a command or expression. PowerShell supports several types of operators to help you manipulate values.

## Arithmetic Operators

Use arithmetic operators (`+`, `-`, `*`, `/`, `%`) to calculate values in a command or expression. With these operators, you can add, subtract, multiply, or divide values, and calculate the remainder (modulus) of a division operation.

The addition operator concatenates elements. The multiplication operator returns the specified number of copies of each element. You can use arithmetic operators on any .NET type that implements them, such as: `Int`, `String`, `DateTime`, `Hashtable`, and Arrays.

Bitwise operators (`-band`, `-bor`, `-bxor`, `-bnot`, `-shl`, `-shr`) manipulate the bit patterns in values.

For more information, see about_Arithmetic_Operators.

## Assignment Operators

Use assignment operators (`=`, `+=`, `-=`, `*=`, `/=`, `%=`) to assign, change, or append values to variables. You can combine arithmetic operators with assignment to assign the result of the arithmetic operation to a variable.

For more information, see about_Assignment_Operators.

## Comparison Operators

Use comparison operators (`-eq`, `-ne`, `-gt`, `-lt`, `-le`, `-ge`) to compare values and test conditions. For example, you can compare two string values to determine whether they're

equal.

The comparison operators also include operators that find or replace patterns in text. The (`-match`, `-notmatch`, `-replace`) operators use regular expressions, and (`-like`, `-notlike`) use wildcards `*`.

Containment comparison operators determine whether a test value appears in a reference set (`-in`, `-notin`, `-contains`, `-notcontains`).

Type comparison operators (`-is`, `-isnot`) determine whether an object is of a given type.

For more information, see about_Comparison_Operators.

## Logical Operators

Use logical operators (`-and`, `-or`, `-xor`, `-not`, `!`) to connect conditional statements into a single complex conditional. For example, you can use a logical `-and` operator to create an object filter with two different conditions.

For more information, see about_Logical_Operators.

## Redirection Operators

Use redirection operators (`>`, `>>`, `2>`, `2>>`, and `2>&1`) to send the output of a command or expression to a text file. The redirection operators work like the `Out-File` cmdlet (without parameters) but they also let you redirect error output to specified files. You can also use the `Tee-Object` cmdlet to redirect output.

For more information, see about_Redirection

## Split and Join Operators

The `-split` and `-join` operators divide and combine substrings. The `-split` operator splits a string into substrings. The `-join` operator concatenates multiple strings into a single string.

For more information, see about_Split and about_Join.

## Type Operators

Use the type operators (`-is`, `-isnot`, `-as`) to find or change the .NET type of an object.

For more information, see about_Type_Operators.

# Unary Operators

Use the unary `++` and `--` operators to increment or decrement values and `-` for negation. For example, to increment the variable `$a` from `9` to `10`, you type `$a++`.

For more information, see about_Arithmetic_Operators.

# Special Operators

Special operators have specific use-cases that don't fit into any other operator group. For example, special operators allow you to run commands, change a value's data type, or retrieve elements from an array.

## Grouping operator `( )`

As in other languages, `(...)` serves to override operator precedence in expressions. For example: `(1 + 2) / 3`

However, in PowerShell, there are additional behaviors.

### Grouping result expressions

`(...)` allows you to let output from a *command* participate in an expression. For example:

PowerShell

```
PS> (Get-Item *.txt).Count -gt 10
True
```

### Piping grouped expressions

When used as the first segment of a pipeline, wrapping a command or expression in parentheses invariably causes *enumeration* of the expression result. If the parentheses wrap a *command*, it's run to completion with all output *collected in memory* before the results are sent through the pipeline.

Grouping an expression before piping also ensures that subsequent object-by-object processing can't interfere with the enumeration the command uses to produce its output.

## Grouping assignment statements

Ungrouped assignment statements don't output values. When grouping an assignment statement, the value of the assigned variable is *passed through* and can be used in larger expressions. For example:

PowerShell

```
PS> ($var = 1 + 2)
3
PS> ($var = 1 + 2) -eq 3
True
```

Wrapping the statement in parentheses turns it into an expression that outputs the value of `$var`.

This behavior applies to all the assignment operators, including compound operators like `+=`, and the increment (`++`) and decrement (`--`) operators. However, the order of operation for increment and decrement depends on their position.

PowerShell

```
PS> $i = 0
PS> (++$i) # prefix
1
PS> $i = 0
PS> ($i++) # postfix
0
PS> $i
1
```

In the prefix case, the value of `$i` is incremented before being output. In the postfix case, the value of `$i` is incremented after being output.

You can also use this technique In the context of a conditional statement, such as the `if` statement.

PowerShell

```
if ($textFiles = Get-ChildItem *.txt) {
    $textFiles.Count
}
```

In this example, if no files match, the `Get-ChildItem` command returns nothing and assigns nothing to `$textFiles`, which is considered `$false` in a boolean context. If one

or more **FileInfo** objects are assigned to `$textFiles`, the conditional evaluates to `$true`. You can work with the value of `$textFiles` in the body of the `if` statement.

> ⓘ Note
>
> While this technique is convenient and concise, it can lead to confusion between the assignment operator (`=`) and the equality-comparison operator (`-eq`).

## Subexpression operator `$( )`

Returns the result of one or more statements. For a single result, returns a scalar. For multiple results, returns an array. Use this when you want to use an expression within another expression. For example, to embed the results of command in a string expression.

PowerShell

```
PS> "Today is $(Get-Date)"
Today is 12/02/2019 13:15:20

PS> "Folder list: $((dir c:\ -dir).Name -join ', ')"
Folder list: Program Files, Program Files (x86), Users, Windows
```

## Array subexpression operator `@( )`

Returns the result of one or more statements as an array. The result is always an array of 0 or more objects.

PowerShell

```
PS> $list = @(Get-Process | Select-Object -First 10; Get-Service |
Select-Object -First 10 )
PS> $list.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array

PS> $list.Count
20
PS> $list = @(Get-Service | Where-Object Status -eq Starting )
PS> $list.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

```
PS> $list.Count
0
```

## Hash table literal syntax `@{}`

Similar to the array subexpression, this syntax is used to declare a hash table. For more information, see about_Hash_Tables.

## Call operator `&`

Runs a command, script, or script block. The call operator, also known as the *invocation operator*, lets you run commands that are stored in variables and represented by strings or script blocks. The call operator executes in a child scope. For more about scopes, see about_Scopes. You can use this to build strings containing the command, parameters, and arguments you need, and then invoke the string as if it were a command. The strings that you create must follow the same parsing rules as a command that you type at the command line. For more information, see about_Parsing.

This example stores a command in a string and executes it using the call operator.

```
PS> $c = "get-executionpolicy"
PS> $c
get-executionpolicy
PS> & $c
AllSigned
```

The call operator doesn't parse strings. This means that you can't use command parameters within a string when you use the call operator.

```
PS> $c = "Get-Service -Name Spooler"
PS> $c
Get-Service -Name Spooler
PS> & $c
& : The term 'Get-Service -Name Spooler' is not recognized as the name
of a
cmdlet, function, script file, or operable program. Check the spelling
of
the name, or if a path was included, verify that the path is correct
and
try again.
```

The Invoke-Expression cmdlet can execute code that causes parsing errors when using the call operator.

```
PS> & "1+1"
&: The term '1+1' is not recognized as a name of a cmdlet, function,
script
file, or executable program. Check the spelling of the name, or if a
path was
included, verify that the path is correct and try again.

PS> Invoke-Expression "1+1"
2
```

You can execute a script using its filename. A script file must have a `.ps1` file extension to be executable. Files that have spaces in their path must be enclosed in quotes. If you try to execute the quoted path, PowerShell displays the contents of the quoted string instead of running the script. The call operator allows you to execute the contents of the string containing the filename.

```
PS C:\Scripts> Get-ChildItem

    Directory: C:\Scripts


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        8/28/2018   1:36 PM             58 script name with
spaces.ps1

PS C:\Scripts> ".\script name with spaces.ps1"
.\script name with spaces.ps1
PS C:\Scripts> & ".\script name with spaces.ps1"
Hello World!
```

For more about script blocks, see about_Script_Blocks.

## Background operator `&`

Runs the pipeline before it in the background, in a PowerShell job. This operator acts similarly to the UNIX control operator ampersand (`&`), which runs the command before it asynchronously in subshell as a job.

This operator is functionally equivalent to `Start-Job`. By default, the background operator starts the jobs in the current working directory of the caller that started the parallel tasks. The following example demonstrates basic usage of the background job operator.

PowerShell

```
Get-Process -Name pwsh &
```

That command is functionally equivalent to the following usage of `Start-Job`:

PowerShell

```
Start-Job -ScriptBlock {Get-Process -Name pwsh}
```

Just like `Start-Job`, the `&` background operator returns a `Job` object. This object can be used with `Receive-Job` and `Remove-Job`, just as if you had used `Start-Job` to start the job.

PowerShell

```
$job = Get-Process -Name pwsh &
Receive-Job $job -Wait
```

Output

```
 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      0     0.00     221.16      25.90    6988 988 pwsh
      0     0.00     140.12      29.87   14845 845 pwsh
      0     0.00      85.51       0.91   19639 988 pwsh
```

PowerShell

```
Remove-Job $job
```

The `&` background operator is also a statement terminator, just like the UNIX control operator ampersand (`&`). This allows you to invoke additional commands after the `&` background operator. The following example demonstrates the invocation of additional commands after the `&` background operator.

PowerShell

```
$job = Get-Process -Name pwsh & Receive-Job $job -Wait
```

Output

```
 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      0     0.00     221.16      25.90    6988 988 pwsh
      0     0.00     140.12      29.87   14845 845 pwsh
      0     0.00      85.51       0.91   19639 988 pwsh
```

This is equivalent to the following script:

PowerShell

```
$job = Start-Job -ScriptBlock {Get-Process -Name pwsh}
Receive-Job $job -Wait
```

If you want to run multiple commands, each in their own background process but all on one line, simply place `&` between and after each of the commands.

PowerShell

```
Get-Process -Name pwsh & Get-Service -Name BITS & Get-CimInstance -
ClassName Win32_ComputerSystem &
```

For more information on PowerShell jobs, see about_Jobs.

## Cast operator `[ ]`

Converts or limits objects to the specified type. If the objects can't be converted, PowerShell generates an error.

PowerShell

```
[DateTime] '2/20/88' - [DateTime] '1/20/88' -eq [TimeSpan] '31'
```

A cast can also be performed when a variable is assigned to using cast notation.

## Comma operator `,`

As a binary operator, the comma creates an array or appends to the array being created. In expression mode, as a unary operator, the comma creates an array with just one member. Place the comma before the member.

PowerShell

```
$myArray = 1,2,3
$SingleArray = ,1
Write-Output (,1)
```

Since `Write-Output` expects an argument, you must put the expression in parentheses.

## Dot sourcing operator `.`

Runs a script in the current scope so that any functions, aliases, and variables that the script creates are added to the current scope, overriding existing ones. Parameters declared by the script become variables. Parameters for which no value has been given become variables with no value. However, the automatic variable `$args` is preserved.

PowerShell

```
. c:\scripts\sample.ps1 1 2 -Also:3
```

> **Note**
>
> The dot sourcing operator is followed by a space. Use the space to distinguish the dot from the dot (`.`) symbol that represents the current directory.
>
> In the following example, the Sample.ps1 script in the current directory is run in the current scope.
>
> PowerShell
>
> ```
> . .\sample.ps1
> ```

## Format operator `-f`

Provide access to the .NET composite formatting feature. A composite format string consists of fixed text intermixed with indexed placeholders, called *format items*. These format items correspond to the objects in the list.

Each format item takes the following form and consists of the following components:

`{index[,alignment][:formatString]}`

The matching braces (`{` and `}`) are required.

The formatting operation yields a result string that consists of the original fixed text intermixed with the string representation of the objects in the list. For more information, see Composite Formatting.

Enter the composite format string on the left side of the operator and the objects to be formatted on the right side of the operator.

PowerShell

```
"{0} {1,-10} {2:N}" -f 1,"hello",[math]::pi
```

Output

```
1 hello      3.14
```

You can zero-pad a numeric value with the "0" custom specifier. The number of zeroes following the `:` indicates the maximum width to pad the formatted string to.

PowerShell

```
"{0:00} {1:000} {2:000000}" -f 7, 24, 365
```

Output

```
07 024 000365
```

If you need to keep the curly braces (`{}`) in the formatted string, you can escape them by doubling the curly braces.

PowerShell

```
"{0} vs. {{0}}" -f 'foo'
```

Output

```
foo vs. {0}
```

## Index operator `[ ]`

Selects objects from indexed collections, such as arrays and hash tables. Array indexes are zero-based, so the first object is indexed as `[0]`. You can also use negative indexes to get the last values. Hash tables are indexed by key value.

Given a list of indices, the index operator returns a list of members corresponding to those indices.

```
PS> $a = 1, 2, 3
PS> $a[0]
1
PS> $a[-1]
3
PS> $a[2, 1, 0]
3
2
1
```

PowerShell

```
(Get-HotFix | Sort-Object installedOn)[-1]
```

PowerShell

```
$h = @{key="value"; name="PowerShell"; version="2.0"}
$h["name"]
```

Output

```
PowerShell
```

PowerShell

```
$x = [xml]"<doc><intro>Once upon a time...</intro></doc>"
$x["doc"]
```

Output

```
intro
-----
Once upon a time...
```

When an object isn't an indexed collection, using the index operator to access the first element returns the object itself. Index values beyond the first element return `$null`.

```
PS> (2)[0]
2
PS> (2)[-1]
2
PS> (2)[1] -eq $null
True
PS> (2)[0,0] -eq $null
True
```

## Pipeline operator `|`

Sends ("pipes") the output of the command that precedes it to the command that follows it. When the output includes more than one object (a "collection"), the pipeline operator sends the objects one at a time.

PowerShell

```powershell
Get-Process | Get-Member
Get-Service | Where-Object {$_.StartType -eq 'Automatic'}
```

## Pipeline chain operators `&&` and `||`

Conditionally execute the right-hand side pipeline based on the success of the left-hand side pipeline.

PowerShell

```powershell
# If Get-Process successfully finds a process called notepad,
# Stop-Process -Name notepad is called
Get-Process notepad && Stop-Process -Name notepad
```

PowerShell

```powershell
# If npm install fails, the node_modules directory is removed
npm install || Remove-Item -Recurse ./node_modules
```

For more information, see About_Pipeline_Chain_Operators.

## Range operator `..`

The range operator can be used to represent an array of sequential integers or characters. The values joined by the range operator define the start and end values of the range.

> **Note**
>
> Support for character ranges was added in PowerShell 6.

## Number ranges

```PowerShell
1..10
$max = 10
foreach ($a in 1..$max) {Write-Host $a}
```

You can also create ranges in reverse order.

```PowerShell
10..1
5..-5 | ForEach-Object {Write-Output $_}
```

The start and end values of the range can be any pair of expressions that evaluate to an integer or a character. The endpoints of the range must be convertible to signed 32-bit integers (`[int32]`). Larger values cause an error. Also, if the range is captured in an array, the size of resulting array is limited to `[int]::MaxValue - 56`. This is maximum size of an array in .NET.

For example, you could use the members of an enumeration for your start and end values.

```PowerShell
PS> enum Food {
      Apple
      Banana = 3
      Kiwi = 10
    }
PS> [Food]::Apple..[Food]::Kiwi
0
1
2
3
4
5
6
7
8
9
10
```

> **Important**
>
> The resulting range isn't limited to the values of the enumeration. Instead it represents the range of values between the two values provided. You can't use the range operator to reliably represent the members of an enumeration.

## Character ranges

To create a range of characters, enclose the characters in quotes.

PowerShell

```
PS> 'a'..'f'
a
b
c
d
e
f
```

PowerShell

```
PS> 'F'..'A'
F
E
D
C
B
A
```

If you assign a character range to a string, it's treated the same assigning a character array to a string.

PowerShell

```
PS> [string]$s = 'a'..'e'
$s
a b c d e
$a = 'a', 'b', 'c', 'd', 'e'
$a
a b c d e
```

The characters in the array are joined into a string. The characters are separated by the value of the `$OFS` preference variable. For more information, see about_Preference_Variables.

The order of the characters in the array is determined by the ASCII value of the character. For example, the ASCII values of `c` and `X` are 99 and 88, respectively. That range would be presented in reverse order.

PowerShell

```
PS> 'c'..'X'
c
b
a
`
_
^
]
\
[
Z
Y
X
```

## Member-access operator `.`

Accesses the properties and methods of an object. The member name may be an expression.

PowerShell

```
$myProcess.peakWorkingSet
(Get-Process PowerShell).kill()
'OS', 'Platform' | Foreach-Object { $PSVersionTable. $_ }
```

Starting PowerShell 3.0, when you use the operator on a list collection object that doesn't have the member, PowerShell automatically enumerates the items in that collection and uses the operator on each of them. For more information, see about_Member-Access_Enumeration.

## Static member operator `::`

Calls the static properties and methods of a .NET class. To find the static properties and methods of an object, use the Static parameter of the `Get-Member` cmdlet. The member name may be an expression.

PowerShell

```
[datetime]::Now
```

```
'MinValue', 'MaxValue' | Foreach-Object { [int]:: $_ }
```

## Ternary operator `? <if-true> : <if-false>`

You can use the ternary operator as a replacement for the `if-else` statement in simple conditional cases.

For more information, see about_If.

## Null-coalescing operator `??`

The null-coalescing operator `??` returns the value of its left-hand operand if it isn't null. Otherwise, it evaluates the right-hand operand and returns its result. The `??` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

PowerShell

```
$x = $null
$x ?? 100
```

Output

```
100
```

In the following example, the right-hand operand won't be evaluated.

PowerShell

```
[string] $todaysDate = '1/10/2020'
$todaysDate ?? (Get-Date).ToShortDateString()
```

Output

```
1/10/2020
```

## Null-coalescing assignment operator `??=`

The null-coalescing assignment operator `??=` assigns the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to null. The `??=` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

```
```

PowerShell

```
$x = $null
$x ??= 100
$x
```

Output

```
100
```

In the following example, the right-hand operand won't be evaluated.

PowerShell

```
[string] $todaysDate = '1/10/2020'
$todaysDate ??= (Get-Date).ToShortDateString()
$todaysDate
```

Output

```
1/10/2020
```

## Null-conditional operators `?.` and `?[]`

> **Note**
>
> This feature was moved from experimental to mainstream in PowerShell 7.1.

A null-conditional operator applies a member access, `?.`, or element access, `?[]`, operation to its operand only if that operand evaluates to non-null; otherwise, it returns null.

Since PowerShell allows `?` to be part of the variable name, formal specification of the variable name is required for using these operators. You must use braces (`{}`) around the variable names like `${a}` or when `?` is part of the variable name `${a?}`.

> **Note**
>
> The variable name syntax of `${<name>}` shouldn't be confused with the `$()` subexpression operator. For more information, see Variable name section of about_Variables.

In the following example, the value of **PropName** is returned.

PowerShell

```
$a = @{ PropName = 100 }
${a}?.PropName
```

Output

```
100
```

The following example returns null without trying to access the member name **PropName**.

PowerShell

```
$a = $null
${a}?.PropName
```

In this example, the value of the indexed element is returned.

PowerShell

```
$a = 1..10
${a}?[0]
```

Output

```
1
```

The following example returns null without trying to access indexed element.

PowerShell

```
$a = $null
${a}?[0]
```

# See also

- about_Arithmetic_Operators
- about_Assignment_Operators
- about_Comparison_Operators
- about_Logical_Operators
- about_Operator_Precedence

- about_Member-Access_Enumeration
- about_Type_Operators
- about_Split
- about_Join
- about_Redirection