Learn Discover Product documentation Development languages Topics Sign in

PowerShell Overview DSC PowerShellGet Utility modules Module Browser API Browser Resources Download PowerShell

Version

PowerShell 5.1

Filter by title

about_ANSI_Terminals
about_Arithmetic_Operators
about_Arrays
about_Assignment_Operators
about_Automatic_Variables
about_Booleans
about_Break
about_Built-in_Functions
about_Calculated_Properties
about_Case-Sensitivity
about_Character_Encoding
about_CimSession
about_Classes
about_Classes_Constructors
about_Classes_Inheritance
about_Classes_Methods
about_Classes_Properties
about_Command_Precedence
about_Command_Syntax
about_Comments
about_Comment_Based_Help
about_CommonParameters
about_Comparison_Operators
about_Continue
about_Core_Commands
about_Data_Files
about_Data_Sections
about_Debuggers
about_DesiredStateConfiguration
about_Do
about_Enum
about_Environment_Provider
about_Environment_Variables
about_Eventlogs
about_Execution_Policies
about_FileSystem_Provider
about_For
about_Foreach
about_Format.ps1xml
about_Functions
about_Functions_Advanced

Download PDF

Learn / PowerShell /

about_Arithmetic_Operators

Article • 04/06/2024 • 14 contributors

Feedback

In this article

Short description
Long description
Operator precedence
Division and rounding
Show 6 more

Short description

Describes the operators that perform arithmetic in PowerShell.

Long description

Arithmetic operators calculate numeric values. You can use one or more arithmetic operators to add, subtract, multiply, and divide values, and to calculate the remainder (modulus) of a division operation.

The addition operator (`+`) and multiplication operator (`*`) also operate on strings, arrays, and hashtables. The addition operator concatenates the input. The multiplication operator returns multiple copies of the input. You can even mix object types in an arithmetic statement. The method that's used to evaluate the statement is determined by the type of the leftmost object in the expression.

Beginning in PowerShell 2.0, all arithmetic operators work on 64-bit numbers.

Beginning in PowerShell 3.0, the `-shr` (shift-right) and `-shl` (shift-left) are added to support bitwise arithmetic in PowerShell. The bitwise operators only work on integer types.

PowerShell supports the following arithmetic operators:

- Addition (`+`) - Adds numbers, concatenates strings, arrays, and hash tables

PowerShell Copy

```
6 + 2                        # result = 8
"file" + "name"              # result = "filename"
@(1, "one") + @(2.0, "two")  # result = @(1, "one", 2.0, "two")
@{"one" = 1} + @{"two" = 2}  # result = @{"one" = 1; "two" = 2}
```

- Subtraction (`-`) - Subtracts or negates numbers

PowerShell Copy

```
6 - 2   # result = 4
- -6    # result = 6
(Get-Date).AddDays(-1) # Yesterday's date
```

- Multiplication (`*`) - Multiply numbers or copy strings and arrays the specified number of times

  ```PowerShell
  6 * 2       # result = 12
  @("!") * 4  # result = @("!","!","!","!")
  "!" * 3     # result = "!!!"
  ```

- Division (`/`) - Divides numbers

  ```PowerShell
  6 / 2  # result = 3
  ```

- Modulus (`%`) - returns the remainder of a division operation.

  ```PowerShell
  7 % 2  # result = 1
  ```

- Bitwise AND (`-band`)

  ```PowerShell
  5 -band 3  # result = 1
  ```

- Bitwise NOT (`-bnot`)

  ```PowerShell
  -bnot 5  # result = -6
  ```

- Bitwise OR (`-bor`)

  ```PowerShell
  5 -bor 0x03  # result = 7
  ```

- Bitwise XOR (`-bxor`)

  ```PowerShell
  5 -bxor 3   # result = 6
  ```

- Shifts bits to the left (`-shl`)

  ```PowerShell
  102 -shl 2  # result = 408
  ```

- Shifts bits to the right (`-shr`)

  ```PowerShell
  102 -shr 2  # result = 25
  ```

Operator precedence

PowerShell processes arithmetic operators in the following order:

Expand table

Precedence	Operator	Description
1	`()`	Parentheses
2	`-`	For a negative number or unary operator
3	`*`, `/`, `%`	For multiplication and division
4	`+`, `-`	For addition and subtraction
5	`-band`, `-bnot`	For bitwise operations
5	`-bor`, `-bxor`	For bitwise operations
5	`-shr`, `-shl`	For bitwise operations

PowerShell processes the expressions from left to right according to the precedence rules. The following examples show the effect of the precedence rules:

PowerShell Copy

```
3+6/3*4    # result = 11
3+6/(3*4)  # result = 3.5
(3+6)/3*4  # result = 12
```

The order in which PowerShell evaluates expressions might differ from other programming and scripting languages that you have used. The following example shows a complicated assignment statement.

PowerShell Copy

```
$a = 0
$b = @(1,2)
$c = @(-1,-2)

$b[$a] = $c[$a++]
```

In this example, the expression `$a++` is evaluated before `$b[$a]`. Evaluating `$a++` changes the value of `$a` after it's used in the statement `$c[$a++]`, but before it's used in `$b[$a]`. The variable `$a` in `$b[$a]` equals `1`, not `0`. Therefore, the statement assigns a value to `$b[1]`, not `$b[0]`.

The code above is equivalent to:

PowerShell Copy

```
$a = 0
$b = @(1,2)
$c = @(-1,-2)

$tmp = $c[$a]
$a = $a + 1
$b[$a] = $tmp
```

Division and rounding

When the quotient of a division operation is an integer, PowerShell rounds the value to the nearest integer. When the value is `.5`, it rounds to the nearest even integer.

The following example shows the effect of rounding to the nearest even integer.

PowerShell Copy

```
PS> [int]( 5 / 2 )  # Result is rounded down
2

PS> [int]( 7 / 2 )  # Result is rounded up
4
```

You can use the `[Math]` class to get different rounding behavior.

PowerShell Copy

```
PS> [int][Math]::Round(5 / 2,[MidpointRounding]::AwayFromZero)
3

PS> [int][Math]::Ceiling(5 / 2)
3

PS> [int][Math]::Floor(5 / 2)
2
```

For more information, see the Math.Round method.

Type conversion to accommodate result

PowerShell automatically selects the .NET numeric type that best expresses the result without losing precision. For example:

PowerShell Copy

```
2 + 3.1
(2).GetType().FullName
(2 + 3.1).GetType().FullName
```

Output Copy

```
5.1
System.Int32
System.Double
```

If the result of an operation is too large for the type, the type of the result is widened to accommodate the result, as in the following example:

PowerShell Copy

```
(512MB).GetType().FullName
(512MB * 512MB).GetType().FullName
```

Output Copy

```
System.Int32
System.Double
```

The type of the result isn't always the same as one of the operands. In the following example, the negative value can't be cast to an unsigned integer, and the unsigned integer is too large to be cast to `Int32`:

PowerShell Copy

```
([int32]::minvalue + [uint32]::maxvalue).GetType().FullName
```

Output Copy

```
System.Int64
```

In this example, `Int64` can accommodate both types.

The `System.Decimal` type is an exception. If either operand has the **Decimal** type, the result is **Decimal** type. Any result too large for the **Decimal** value is an error.

PowerShell Copy

```
PS> [Decimal]::maxvalue
79228162514264337593543950335

PS> [Decimal]::maxvalue + 1
RuntimeException: Value was either too large or too small for a Decimal.
```

Potential loss of precision

Anytime you have a result that exceeds the range of the type, you risk losing precision due to type conversion. For example, adding a sufficiently large `[long]` and `[int]` results in the operands being converted to `[double]`. In this example, `9223372036854775807` is the maximum value of a `[long]` integer. Adding to value overflows the range of `[long]`.

PowerShell Copy

```
PS> (9223372036854775807 + 2).GetType().FullName
System.Double
```

Casting the result to `[ulong]` yields an inaccurate result, because the operands were coerced to `[double]` first.

PowerShell Copy

```
PS> [ulong](9223372036854775807 + 2)
9223372036854775808
```

Defining the larger value as `[ulong]` first avoids the problem and produces the correct result.

PowerShell Copy

```
PS> 9223372036854775807ul + 2
9223372036854775809
```

However, exceeding the range of `[ulong]` results in a `[double]`.

PowerShell Copy

```
PS> ([ulong]::MaxValue + 1).GetType().FullName
System.Double
```

Bigint arithmetic

When you perform arithmetic operations on `[bigint]` numbers, PowerShell uses converts all operands to `[bigint]`, which results in truncation of non-integer values. For example, the `[double]` value `1.9` is truncated to `1` when converted to `[bigint]`.

PowerShell Copy

```
PS> [bigint]1 / 1.9
1
PS> 1 / [bigint]1.9
1
```

This behavior is different from the behavior of other numeric types. In this example, an `[int]` divided by a `[double]` results in a `[double]`. Casting `1.9` to an `[int]` rounds the value up to `2`.

PowerShell Copy

```
PS> 1 / 1.9
0.526315789473684
PS> 1 / [int]1.9
0.5
```

Adding and multiplying non numeric types

You can add numbers, strings, arrays, and hash tables. And, you can multiply numbers, strings, and arrays. However, you can't multiply hash tables.

When you add strings, arrays, or hash tables, the elements are concatenated. When you concatenate collections, such as arrays or hash tables, a new object is created that contains the objects from both collections. If you try to concatenate hash tables that have the same key, the operation fails.

For example, the following commands create two arrays and then add them:

PowerShell Copy

```
$a = 1,2,3
$b = "A","B","C"
$a + $b
```

Output Copy

```
1
2
3
A
B
C
```

You can also perform arithmetic operations on objects of different types. The operation that PowerShell performs is determined by the Microsoft .NET type of the leftmost object in the operation. PowerShell tries to convert all the objects in the operation to the .NET type of the first object. If it succeeds in converting the objects, it performs the operation appropriate to the .NET type of the first object. If it fails to convert any of the objects, the operation fails.

The following examples demonstrate the use of the addition and multiplication operators in operations that include different object types.

PowerShell Copy

```
$array = 1,2,3
$red = [ConsoleColor]::Red
$blue = [ConsoleColor]::Blue

"file" + 16      # result = "file16"
$array + 16      # result = 1,2,3,16
$array + "file"  # result = 1,2,3,"file"
$array * 2       # result = 1,2,3,1,2,3
"file" * 3       # result = "filefilefile"
$blue + 3        # result = Red
$red - 3         # result = Blue
$blue - $red     # result = -3
+ '123'          # result = 123
```

Because the method that's used to evaluate statements is determined by the leftmost object, addition and multiplication in PowerShell aren't strictly commutative. For example, `(a + b)` doesn't always equal `(b + a)`, and `(ab)` doesn't always equal `(ba)`.

The following examples demonstrate this principle:

PowerShell Copy

```
PS> "file" + 16
file16

PS> 16 + "file"
InvalidArgument: can't convert value "file" to type "System.Int32". Error:
"Input string wasn't in a correct format."
```

Hash tables are a slightly different case. You can add hash tables to another hash table, as long as, the added hash tables don't have duplicate keys.

The following example show how to add hash tables to each other.

PowerShell Copy

```
$hash1 = @{a=1; b=2; c=3}
$hash2 = @{c1="Server01"; c2="Server02"}
$hash1 + $hash2
```

Output Copy

```
Name                           Value
----                           -----
c2                             Server02
a                              1
b                              2
c1                             Server01
c                              3
```

The following example throws an error because one of the keys is duplicated in both hash tables.

PowerShell Copy

```
$hash1 = @{a=1; b=2; c=3}
$hash2 = @{c1="Server01"; c="Server02"}
$hash1 + $hash2
```

Output Copy

```
OperationStopped:
Line |
   3 |  $hash1 + $hash2
     |  ~~~~~~~~~~~~~~~
     | Item has already been added. Key in dictionary: 'c'  Key being added: 'c'
```

Also, you can add a hash table to an array; and, the entire hash table becomes an item in the array.

PowerShell Copy

```
$array1 = @(0, "Hello World", [datetime]::Now)
$hash1 = @{a=1; b=2}
$array2 = $array1 + $hash1
$array2
```

Output Copy

```
0
Hello World

Monday, June 12, 2017 3:05:46 PM

Key   : a
Value : 1
Name  : a

Key   : b
Value : 2
Name  : b
```

However, you can't add any other type to a hash table.

PowerShell Copy

```
$hash1 + 2
```

Output Copy

```
InvalidOperation: A hash table can only be added to another hash table.
```

Although the addition operators are very useful, use the assignment operators to add elements to hash tables and arrays. For more information see about_assignment_operators. The following examples use the `+=` assignment operator to add items to an array:

PowerShell Copy

```
$array = @()
(0..2).foreach{ $array += $_ }
$array
```

Output Copy

```
0
1
2
```

Arithmetic operators and variables

You can also use arithmetic operators with variables. The operators act on the values of the variables. The following examples demonstrate the use of arithmetic operators with variables:

PowerShell Copy

```
PS> $intA = 6
PS> $intB = 4
PS> $intA + $intB
10

PS> $a = "Power"
PS> $b = "Shell"
PS> $a + $b
PowerShell
```

Arithmetic operators and commands

Typically, you use the arithmetic operators in expressions with numbers, strings, and arrays. However, you can also use arithmetic operators with the objects that commands return and with the properties of those objects.

The following examples show how to use the arithmetic operators in expressions with PowerShell commands:

PowerShell Copy

```
(Get-Date) + (New-TimeSpan -day 1)
```

The parenthesis operator forces the evaluation of the `Get-Date` cmdlet and the evaluation of the `New-TimeSpan -Day 1` cmdlet expression, in that order. Both results are then added using the `+` operator.

PowerShell Copy

```
Get-Process | Where-Object { ($_.ws * 2) -gt 50mb }
```

Output Copy

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1896      39    50968      30620   264 1,572.55   1104 explorer
  12802      78   188468      81032   753 3,676.39   5676 OUTLOOK
    660       9    36168      26956   143    12.20    988 PowerShell
    561      14     6592      28144   110 1,010.09    496 services
   3476      80    34664      26092   234 ...45.69    876 svchost
    967      30    58804      59496   416   930.97   2508 WINWORD
```

In the above expression, each process working space (`$_.ws`) is multiplied by `2`; and, the result, compared against `50mb` to see if it's greater than that.

Bitwise operators

PowerShell supports the standard bitwise operators, including bitwise-AND (`-band`), the inclusive and exclusive bitwise-OR operators (`-bor` and `-bxor`), and bitwise-NOT (`-bnot`).

Beginning in PowerShell 2.0, all bitwise operators work with 64-bit integers.

Beginning in PowerShell 3.0, the `-shr` (shift-right) and `-shl` (shift-left) are introduced to support bitwise arithmetic in PowerShell.

PowerShell supports the following bitwise operators.

Expand table

Operator	Description	Expression	Result
`-band`	Bitwise AND	`10 -band 3`	2
`-bor`	Bitwise OR (inclusive)	`10 -bor 3`	11
`-bxor`	Bitwise OR (exclusive)	`10 -bxor 3`	9
`-bnot`	Bitwise NOT	`-bNot 10`	-11
`-shl`	Shift-left	`102 -shl 2`	408
`-shr`	Shift-right	`102 -shr 1`	51

Bitwise operators act on the binary format of a value. For example, the bit structure for the number 10 is 00001010 (based on 1 byte), and the bit structure for the number 3 is 00000011. When you use a bitwise operator to compare 10 to 3, the individual bits in each byte are compared.

In a bitwise AND operation, the resulting bit's set to 1 only when both input bits are 1.

Copy

```
1010      (10)
0011      ( 3)
--------------  bAND
0010      ( 2)
```

In a bitwise OR (inclusive) operation, the resulting bit's set to 1 when either or both input bits are 1. The resulting bit's set to 0 only when both input bits are set to 0.

Copy

```
1010      (10)
0011      ( 3)
--------------  bOR (inclusive)
1011      (11)
```

In a bitwise OR (exclusive) operation, the resulting bit's set to 1 only when one input bit's 1.

Copy

```
1010      (10)
0011      ( 3)
--------------  bXOR (exclusive)
1001      ( 9)
```

The bitwise NOT operator is a unary operator that produces the binary complement of the value. A bit of 1 is set to 0 and a bit of 0 is set to 1.

For example, the binary complement of 0 is -1, the maximum unsigned integer (0xFFFFFFFF), and the binary complement of -1 is 0.

PowerShell Copy

```
-bNot 10
```

Output Copy

```
-11
```

Copy

```
0000 0000 0000 1010  (10)
------------------------- bNOT
1111 1111 1111 0101  (-11, 0xFFFFFFF5)
```

In a bitwise shift-left operation, all bits are moved "n" places to the left, where "n" is the value of the right operand. A zero is inserted in the ones place.

Expand table

Expression	Result	Binary Result
`21 -shl 0`	21	0001 0101
`21 -shl 1`	42	0010 1010
`21 -shl 2`	84	0101 0100

In a bitwise shift-right operation, all bits are moved "n" places to the right, where "n" is specified by the right operand. The shift-right operator (`-shr`) copies the sign bit to the left-most place when shifting a signed value. For unsigned values, a zero is inserted in the left-most position.

⛶ Expand table

Expression	Result	Binary	Hex
`21 -shr 0`	21	00010101	0x15
`21 -shr 1`	10	00001010	0x0A
`21 -shr 2`	5	00000101	0x05
`21 -shr 31`	0	00000000	0x00
`21 -shr 32`	21	00010101	0x15
`21 -shr 64`	21	00010101	0x15
`21 -shr 65`	10	00001010	0x0A
`21 -shr 66`	5	00000101	0x05
`[int]::MaxValue -shr 1`	1073741823	00111111111111111111111111111111	0x3FFFFFFF
`[int]::MinValue -shr 1`	-1073741824	11000000000000000000000000000000	0xC0000000
`-1 -shr 1`	-1	11111111111111111111111111111111	0xFFFFFFFF
`(-21 -shr 1)`	-11	11111111111111111111111111110101	0xFFFFFFF5
`(-21 -shr 2)`	-6	11111111111111111111111111111010	0xFFFFFFF4

See also

- about_Arrays
- about_Hash_Tables
- about_Operators
- about_Assignment_Operators
- about_Comparison_Operators
- about_Variables
- Get-Date
- New-TimeSpan

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback