

[Michael Sorens](#)

04 April 2014

# PowerShell One-Liners: Help, Syntax, Display and Files

PowerShell is designed to be used by busy IT professionals who want to get things done, and don't necessarily enjoy programming. PowerShell tackles this paradox by providing its own help and command-line intellisense. We aim to make it a bit easier still by providing a series of collections of general-purpose one-liners to cover most of what you'll need to get useful scripting done.

This series is in four parts: This is part 1

- [Part 1: Help, Syntax, Display and Files](#)
- [Part 2: Variables, Parameters, Properties, and Objects,](#)
- [Part 3: Collections, Hashtables, Arrays and Strings](#)
- [Part 4: Accessing, Handling and Writing Data](#)

Well, yes, it would be an onerous task indeed to deliver *everything*. But per the [Pareto principle](#), roughly eighty percent of what you could want to know about PowerShell is here in this series of articles; possibly more! You will tend to learn the other twenty percent as you go, and you may not need it for quite a ways into your PowerShell explorations. I feel compelled to confess that some of the entries are not, strictly speaking, on one physical line, but are written in such a way that, if you really want to run them in one line, you can just remove the line breaks and they will work. (Note that that is not generally true of PowerShell syntax.)

This series of articles evolved out of my own notes on PowerShell as I poked and prodded it to show me more. As my collection burgeoned, I began to organize them until I had one-line recipes for most any simple PowerShell task. *Simple*, though, does not mean *trivial*. You can do quite a lot in one line of PowerShell, such as grabbing the contents of specific elements of a web page or converting a CSV file into a collection of PowerShell objects.

This collection of PowerShell one-liners is organized as follows:

**Part 1** begins by showing you how to have PowerShell itself help you figure out what you need to do to accomplish a task, covering the help system as well as its handy command-line intellisense. The next sections deal with locations, files, and paths: the basic currency of any shell. You are introduced to some basic but key syntactic constructs and then ways to cast your output in list, table, grid, or chart form.

**Part 2** moves into details on variables, parameters, properties, and objects, providing insight into the richness of the PowerShell programming language. [Part 2](#) is rounded out with a few other vital bits on leveraging the PowerShell environment.

**Part 3**

covers the two fundamental data structures of PowerShell: the collection (array) and the hash table (dictionary), examining everything from creating, accessing, iterating, ordering, and selecting. Part 3 also covers converting between strings and arrays, and rounds out with techniques for searching, most commonly applicable to files (searching both directory structures as well as file contents).

**Part 4** is your information source for a variety of input and output techniques: reading and writing files; writing the various output streams; file housekeeping operations; and various techniques related to CSV, JSON, database, network, and XML.

Each part of this series is available as both an online reference here at Simple-Talk.com, [in a wide-form as well](#), and as a downloadable wallchart (from the link at the bottom of the article) in PDF format for those who prefer a printed copy near at hand. Please keep in mind though that this is a quick reference, not a tutorial. So while there are a few brief introductory remarks for each section, there is very little explanation for any given incantation. But do not let that scare you off-jump in and try things! You should find more than a few “aha!” moments ahead of you!

## Notes on using the tables:

- A command will typically use full names of cmdlets but the examples will often use aliases for brevity. Example: Get-Help has aliases *man* and *help*. This has the side benefit of showing you both long and short names to invoke many commands.
- Most tables contain either 3 or 4 columns: a description of an action; the generic command syntax to perform that action; an example invocation of that command; and optionally an output column showing the result of that example where feasible.
- For clarity, embedded newlines (``n`) and embedded return/newline combinations (``r`n`) are highlighted as shown.
- Many actions in PowerShell can be performed in more than one way. The goal here is to show just the simplest which may mean displaying more than one command if they are about equally straightforward. In such cases the different commands are numbered with square brackets (e.g. “[1]”). Multiple commands generally mean multiple examples, which are similarly numbered.
- Most commands will work with PowerShell version 2 and above, though some require at least version 3. So if you are still running v2 and encounter an issue that is likely your culprit.
- The vast majority of commands are built-in, i.e. supplied by Microsoft. There are a few sprinkled about that require loading an additional module or script, but their usefulness makes them worth including in this compendium. These “add-ins” will be demarcated with angle brackets, e.g. <<pscx>> denotes the popular PowerShell Community Extensions (<http://pscx.codeplex.com/>).
- There are many links included for further reading; these are active hyperlinks that you may select if you are working online, but the URLs themselves are also explicitly provided (as in the previous bullet) in case you have a paper copy.

**Note:** Out of necessity, the version of the tables in the articles is somewhat compressed. If you find them hard to read, then there is a wide version of the article [available here](#), and a PDF version is available from the link at the bottom of the article

## What’s What and What’s Where

This is your starting point when you are staring at a PowerShell prompt, knowing not what to do. Find out what commands are available, read help on PowerShell concepts, learn about auto-completion of cmdlets and parameters, see what command an alias refers to, and more. Before even going to the first entry, though, it is useful to learn one thing: PowerShell has help available on both *commands* and *concepts*. You can look up a command simply with, for example, `Get-Help Get-ChildItem` (# 1 below). Or you can search for a command with a substring, e.g. `Get-Help file` (#3). But as with any programming language you have to know syntax, semantics, structures, ... in short all those conceptual items that let you work with commands. If you want to know about variables, for example, you have merely to say `Get-Help about_variables`. All conceptual topics begin with the “`about_`” prefix (#11).

Not at first, but in short order, you will want to be able to find meta-details about commands as well. By that, I mean to answer questions like: What type of objects does a cmdlet return? Does a cmdlet have one or multiple sets of parameters available? Where does a cmdlet come from? All of those meta-details can be seen from entry 20 below.

| Action  | Command  | Example   |
|---|--|---|
| Basic help for x                                | <code>Get-Help cmd</code> (cmd is a full name)                           | <code>help Get-ChildItem</code>   |
| Help in separate window                         | <code>Show-Command cmd</code> ; then select the help icon                | <code>Show-Command Get-ChildItem</code>   |
| List help topics containing x                   | <code>Get-Help string</code> (string is a prefix or uses wildcards)      | [1a] <code>help Get</code><br>[1b] <code>help Get-*</code>  |
| Help for parameter y of cmdlet x                | <code>Get-Help cmd -parameter y</code>                                   | <code>help Get-Date -param month</code>   |
| Help for multiple parameters                    | <code>Get-Help cmd -parameter y*</code> (i.e. use wildcards)             | <code>help Get-Date -param m*</code>  |
| List allowed values for parameter y of cmdlet x | <code>Get-Help cmd -parameter y</code>                                   | <code>help Out-File -parameter Encoding</code>  |
| Intellisense for parameter names [in ISE]       | <code>cmdlet -paramNamePrefix</code>                                     | <code>Out-File -enc</code>  |
| Intellisense for parameter values [in ISE]      | <code>cmdlet -paramName&lt;space&gt; paramValuePrefix</code>             | <code>Out-File -enc&lt;space&gt;</code>   |
| Auto-completion for parameter names             | <code>cmdlet – paramNamePrefix &lt;tab&gt;</code>                        | [1a] <code>Out-File -&lt;tab&gt;</code><br>[1b] <code>Out-File -enc&lt;tab&gt;</code>                               |
| Auto-completion for parameter values            | <code>cmdlet -paramName&lt;space&gt; paramValuePrefix &lt;tab&gt;</code> | [1a] <code>Out-File -enc&lt;space&gt;&lt;tab&gt;</code><br>[1b] <code>Out-File -enc&lt;space&gt;u&lt;tab&gt;</code> |
| List all ‘conceptual’ topics (see text above)   | <code>Get-Help about_*</code>  | same  |
| Filter help output by regex                     | <code>Get-Help topic   Out-String -stream   sls -pattern regex</code>    | <code>help ls   Out-String -Stream   Select-String recurse</code>   |

|   |  |  |
|---|--|--|
| Filter help output by constant                            | Get-Help topic   Out-String -stream   sls -simple text                         | help ls   Out-String -Stream   sls -SimpleMatch "[recurse]"                            |
| Send help text to a file                                  | [1] Get-Help topic  Out-String   Set-Content file<br>[2] Get-Help topic > file | [1] help Get-ChildItem   Out-String   sc help.txt<br>[2] help Get-ChildItem > help.txt |
| List all cmdlets and functions                            | Get-Command  | same   |
| List all cmdlets/functions beginning with characters      | Get-Command string*  | gcm wr*  |
| List all cmdlets/functions filtered by noun               | Get-Command -noun string*  | gcm -noun type*  |
| List all exported items from module x                     | Get-Command -Module module   | Get-Command -Module BitLocker  |
| List properties and methods of cmdlet                     | cmdlet   Get-Member  | Get-ChildItem   gm   |
| List meta-details of cmdlet (see text above)              | Get-Command cmdlet   Select-Object *   | gcm Get-ChildItem   select *   |
| Display module containing cmdlet                          | (Get-Command cmdlet).ModuleName  | (gcm Get-ChildItem).ModuleName   |
| Display assembly containing cmdlet (for compiled cmdlets) | ( Get-Command cmdlet ).dll   | (gcm Get-ChildItem).dll  |
| Display underlying command for alias x                    | Get-Alias -name x  | Get-Alias -name gci  |
| Display aliases for command x                             | Get-Alias -definition x  | Get-Alias -def Get-ChildItem   |
| Get general help for PS Community Extensions              | Import-Module pscx; Get-Help pscx <<pscx>>                                     | same   |
| List all functions in PSCX                                | Get-Command -Module Pscx* -CommandType Function <<pscx>>                       | same   |

Location, Location, Location

See where you are or where you have been and navigate to where you want to be; understand the difference between your PowerShell current location and your Windows working directory; get relative or absolute paths for files or directories.

|        |         |         |
|--------|---------|---------|
| Action | Command | Example |
|--------|---------|---------|

|  |  |   |
|--|--|---|
| Display current location (non-UNC paths)   | [1] Get-Location<br>[2] \$pwd<br>[3] \$pwd.Path  | [1] cd c:\foo; pwd<br>[2] cd c:\foo; \$pwd<br>[3] cd c:\foo; \$pwd.Path   |
| Display current location (UNC paths)   | \$pwd.ProviderPath   | cd \\localhost\c\$; \$pwd.ProviderPath  |
| Change current location (to drive or folder or data store)                                 | Set-Location target  | [1a] cd variable:<br>[1b] sl c:\documents\me<br>[1c] chdir foo\bar  |
| Get absolute path of file in current location  | Resolve-Path file  | Resolve-Path myfile.txt   |
| Get name without path for file or directory  | [1] (Get-Item filespec).Name<br>[2] Split-Path filespec -Leaf  | [1] (Get-Item \users\me\myfile.txt).Name<br>[2] Split-Path \users\me -Leaf  |
| Get parent path for file   | (Get-Item filespec).DirectoryName  | (Get-Item \users\me\myfile.txt).DirectoryName   |
| Get parent path for directory  | (Get-Item filespec).Parent   | (Get-Item \users\me).Parent   |
| Get parent path for file or directory  | Split-Path filespec -Parent  | [1a] Split-Path \users\me\myfile.txt -Parent<br>[1b] Split-Path \users\me -Parent   |
| Get parent name without path for file  | (Get-Item filespec).Directory.Name   | (Get-Item \users\me\myfile.txt).Directory.Name  |
| Get parent name without path for file or directory   | [1] (Get-Item (Split-Path filespec -Parent)).Name<br><br>[2] Split-Path (Split-Path filespec -Parent) -Leaf<br>[3] "filespec".split("\")[-2] | [1] (Get-Item (Split-Path \users\me\myfile.txt -Parent)).Name<br>[2] Split-Path (Split-Path \users\me -Parent) -Leaf<br>[3a] "\users\me\myfile.txt".split("\")[-2]<br>[3b] "\users\me".split("\")[-2] |
| Display working directory (see <a href="http://bit.ly/1j4uomr">http://bit.ly/1j4uomr</a> ) | [Environment]::CurrentDirectory  | same  |
| Change current location and stack it   | Push-Location path   | pushd \projects\stuff   |
| Return to last stacked location  | Pop-Location   | popd  |
| View directory stack   | Get-Location -stack  | same  |
| View directory stack depth   | (Get-Location -stack).Count  | same  |

# Files and Paths and Things

You can list contents of disk folders with Get-ChildItem just as you could use dir from DOS. But Get-ChildItem also lets you examine environment variables, local variables, aliases, registry paths, even database objects with the same syntax! See about\_providers (<http://bit.ly/1ghdcvb>) and PS Provider Help (<http://bit.ly/1dHIC7r>) for more details.

| Action   | Command   | Example  |
|--|---|--|
| List contents of location<br>(location may be on any supported<br>PSDrive-see list datastores below).                            | [1] Get-ChildItem path<br><br>[2] Get-ChildItem psdrive:path<br><br>[3] Get-ChildItem psdrive:  | [1a] Get-ChildItem<br>[1b] Get-ChildItem .<br>[2a] gci<br>c:\users\me\documents<br>[2b] ls SQLSERVER:\SQLÂ-<br>\localhostÂ\SQLEXPRESS\Â-<br>Databases<br>[3a] dir env:<br>[3b] dir variable:<br>[3c] ls alias: |
| List names of files in current directory   | [1] Get-ChildItem   select -ExpandProperty name<br>[2] dir   % { \$_.Name }<br>[3] (dir).Name   | same   |
| List names of files recursively  | [1] dir -Recurse   select -ExpandProperty Name<br>[2] (dir -Recurse).Name   | same   |
| List full paths of files recursively   | [1] dir -Recurse   select -ExpandProperty FullName<br>[2] (dir -Recurse).FullName   | same   |
| List full paths of files recursively with<br>directory marker  | dir -r   % { \$_.FullName + \$(if (\$_.PsisContainer) {'\'}) }  | same   |
| List relative paths of files recursively with<br>directory marker  | dir -r   % { \$_.FullName.substring(\$pwd.Path.length+1) + \$(if (\$_.PsisContainer) {'\'}) }   | same   |
| List file and directory sizes<br>(see<br><a href="http://stackoverflow.com/a/14031005">http://stackoverflow.com/a/14031005</a> ) | dir   % { New-Object PSObject -Property @{ Name = \$_.Name;<br>Size = if(\$_.PsisContainer) { (gci \$_.FullName -Recurse  <br>Measure Length -Sum).Sum } else { \$_.Length }; Type =<br>if(\$_.PsisContainer) {'Directory'} else {'File'} } } | same   |

|  |  |  |
|--|--|--|
| List datastores (regular filesystem drives plus drives from other providers) | [1] Get-PSDrive<br>[2] Get-PSDrive -PSPProvider provider         | [1] Get-PSDrive<br>[2] gdr -PSPProvider<br>FileSystem                |
| List providers (suppliers of datastores)                                     | [1] Get-PSPProvider<br>[2] Get-PSPProvider -PSPProvider provider | [1] Get-PSPProvider<br>[2] Get-PSPProvider -<br>PSPProvider registry |
| List processes   | Get-Process  | same   |

Basic Syntactic Elements

Like with learning most things, you need to learn to crawl before you can learn to run. This section shows you how to do some of the most basic but important things, things that will soon become second nature: knowing what is true and what is false; adding comments; continuing a command across multiple lines or, contrariwise, combining multiple commands on one line; and so forth. Arguably there should be one other important group of items included here: PowerShell operators. But I already published a wallchart on a set of common operators for strings and string arrays. See Harnessing PowerShell’s String Comparison and List-Filtering Features (<http://bit.ly/1c20itX>) for details on -eq, -like, -match, and -contains operators and their variations.

| Action  | Element   | Example  | Output                |
|---|---|--|-----------------------|
| End-of-line comment   | # (octothorp)   | 52 # number of weeks in a year   | 52                    |
| Block comment or documentation comment  | <# ... #>   | <# multi-line<br>comment here #>   |                       |
| Continue command on multiple lines (required unless break after pipe or curly bracket)                                | ` (backquote as last character on line)                                     | “hello ” + `<br>“world”  | hello world           |
| Combine commands on a single line   | ; (semicolon)   | \$a = 25; \$b = -9; “\$a, \$b”   | 25, -9                |
| Escape next character   | ` (backquote)   | \$a = 25; “value of ` \$a is \$a”  | value of \$a<br>is 25 |
| Non-printable characters (newline, tab, etc.)   | `n, `t  | “line one`n line two”  | line one<br>line two  |
| Boolean constant TRUE<br>(see <a href="#">here</a><br>and <a href="http://bit.ly/1iGhXOW">http://bit.ly/1iGhXOW</a> ) | [1] \$TRUE<br>[2] Any string of length > 0<br>[3] Any number not equal to 0 | [1] if (\$TRUE) { “true” } else { “not<br>true” }<br>[2] if (“abc”) { “true” } else { “not | true<br>true<br>true  |

|   |   |   |              |
|---|---|---|--------------|
|   | [4] Array of length > 1   | true" }   | true         |
|   | [5] Array of length 1 whose element is true   | [3] if (-99.5) { "true" } else { "not true" }   | true<br>true |
|   | [6] A reference to any object   | [4] if ((1, 2)) { "true" } else { "not true" }  |              |
|   |   | [5] if ((1)) { "true" } else { "not true" }   |              |
|   |   | [6] \$a = 25; if ([ref]\$a) { "true" } else { "not true" }  |              |
| Boolean constant FALSE<br>(see <a href="#">here</a> )   | [1] \$FALSE   | [1] if (\$FALSE) { "true" } else { "not true" }   | not true     |
|   | [2] Empty string  |   | not true     |
|   | [3] Any number = 0 (e.g. 0, 0.0, 0x0, 0mb, 0D, ...)   | [2] if ("") { "true" } else { "not true" }  | not true     |
|   | [4] Array of length 0   | [3] if (0x0) { "true" } else { "not true" }   | not true     |
|   | [5] Array of length 1 whose element is false  |   | not true     |
|   | [6] \$NULL  | [4] if (@()) { "true" } else { "not true" }   |              |
|   |   | [5] if (("")) { "true" } else { "not true" }  |              |
|   |   | [6] if (\$NULL) { "true" } else { "not true" }  |              |
| null  | \$null  | \$null  |              |
| Iterate each element in a list  | ForEach-Object  | 1..3   % { \$_ * 2 }  | 2<br>4<br>6  |
| Ternary operator, scalar<br>(e.g. result = a > b ? x : y; )   | [1] \$result = switch (boolExpr) { \$true { \$x } \$false { \$y } }                                       | [1] \$result = switch (25 -gt 10) { \$true { "yes" } \$false { "no" } }   | yes<br>yes   |
|   | [2] \$result = if (boolExpr) { \$x } else { \$y }   | [2] \$result = if (25 -gt 10) { "yes" } else { "no" }   |              |
|   | [3] \$result = ?: {boolExpr} {\$x} {\$y}  |   |              |
|   | <<pscx>>  |   |              |
| Coalesce (evaluate 2nd block if 1st block is null)<br>Define Coalesce-Args as:<br>function Coalesce-Args { (@(\$args   ?{\$_}) + \$null)[0] }; Set-Alias ?? Coalesce-Args | [1] Invoke-NullCoalescing block1 block2 <<pscx>>  | [1a] Invoke-NullCoalescing { \$env:dummy } {"usr\tmp"}<br>[1b] ?? { \$env:dummy } {"usr\tmp"}<br>[2a] Coalesce-Args \$env:dummy | \usr\tmp     |
|   | [2] Coalesce-Args block1 block2<br><<code from <a href="http://bit.ly/KhMkwO">http://bit.ly/KhMkwO</a> >> |   |              |



```
"\usr\tmp"  
[2b] ?? $env:dummy "\usr\tmp"
```

## Display Options

According to Using Format Commands to Change Output View (<http://bit.ly/N8b7oe>) each format cmdlet (i.e. Format-Table and Format-List) has default properties that will be used if you do not specify specific properties to display. But the documentation does not reveal what those defaults are or how they vary depending on the type of input object. Also, the default properties differ between Format-Table and Format-List. Finally, keep in mind that if Select-Object is the last cmdlet in your command sequence, it implicitly uses Format-Table or Format-List selecting one or the other based on the number and width of output fields. (And while the included images were not designed for you to make out the actual text, you can discern the representation of the data from those thumbnails.)

| Action   | Command  | Example   |
|--|--|---|
| Format list of objects with each field on a separate line  | any   Format-List  | Is C:\Windows\temp   select name,length   Format-List                             |
| Format list of objects with all fields on one line   | any   Format-Table   | Is C:\Windows\temp   select name,length   Format-Table                            |
| Format list of objects in an interactive grid  | any   Out-GridView   | Is C:\Windows\temp   select name,length   Out-GridView                            |
| Format list as console graph   | any   Out-ConsoleGraph -property quantityPropertyName <<code from <a href="http://bit.ly/1jJN4tG">http://bit.ly/1jJN4tG</a> >>           | Is C:\Windows\temp   select name,length   Out-ConsoleGraph -property length       |
| Format list as gridview graph  | any   Out-ConsoleGraph -GridView -property quantityPropertyName <<code from <a href="http://bit.ly/1jJN4tG">http://bit.ly/1jJN4tG</a> >> | Is C:\Windows\temp   select name,length   Out-ConsoleGraph -property length -grid |
| Send table-formatted output to file  | [1] any   Format-Table -AutoSize   Out-File file -Encoding ascii   | [1] ps   ft -auto   Out-File process.txt -enc ascii                               |
|  | [2] any   Format-Table -AutoSize   Out-String   Set-Content file   | [2] ps   ft -auto   Out-String   sc process.txt                                   |
| Send trimmed table-formatted output to file (removes trailing spaces on fields as well as blank lines) | any   Format-Table -AutoSize   Out-String -Stream   ForEach { \$_.TrimEnd() }   where { \$PSItem }   Set-Content file                    | ps   ft -auto  Out-String -Stream   %{ \$_.TrimEnd() }   ? { \$_ }   sc file      |

## Prompts and Pauses

Some basic interactivity entries, showing how to get input from a user, how to pause and resume, and how to clear the window contents.

| Action   | Command   | Example  |
|--|---|--|
| Prompt user for input  | Read-Host prompt  | \$userValue = Read-Host "Enter name"   |
| Pause for a specific time period   | Start-Sleep seconds   | "before"; Start-Sleep 5; "after"   |
| Pause a script and resume with Enter key                                 | Read-Host -Prompt promptString  | Read-Host -Prompt "Press enter to continue..."   |
| Pause a script and resume with any key (does not work in PowerShell ISE) | Write-Host "Press any key to continue ...";<br>\$x = \$host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown") |  |
| Clear screen   | Clear-Host  |  |
| Display progress bar   | Write-Progress -Activity title -status event -percentComplete percentage                              | for (\$i = 0; \$i -lt \$stuff.Count; \$i++)<br># primary code here<br>Write-Progress -Activity \$title -status \$label[\$i] -percent (( \$i + 1 ) / \$stuff.Count*100) |

## Casts, Type Accelerators, and .NET Classes

You are no doubt familiar with type casting in .NET languages. PowerShell provides the same casting capability but enhances it with a new concept, type accelerators. Type accelerators are simply aliases for .NET framework classes. Note that in PowerShell you can always omit the "System." prefix of a class if it has one to save typing, so you could use, say, DateTime rather than System.DateTime. But with type accelerators you can get a lot more concise than that. For example, you can just use [wmi] instead of System.Management.ManagementObject. As far as casting, in PowerShell you can cast to any .NET type that has either a cast operator or a constructor that accepts the source type (see the Net.IpAddress entry below). Finally, you can use .NET type names to access static members of .NET classes (see the DateTime.Now entry below).

| Action   | Command  | Example | Output |
|--|--|---------|--------|
| List all type accelerators (see <a href="#">here</a> ) | [1] [accelerators]::get <<psc>><br>[2] [psobject].assembly.gettype("System.Management.Automation.TypeAccelerators")::Get | same    |        |

|   |  |   |   |
|---|--|---|---|
| Cast string to IPAddress object             | [System.Net.IpAddress]"ip-address"                                       | [Net.IpAddress]'192.0.0.1'                                | Address : 16777408<br>AddressFamily :<br>InterNetwork<br>...<br>IsIPv4MappedToIPv6 :<br>False<br>IPAddressToString :<br>192.0.0.1 |
| Create new DateTime object with constructor | New-Object -TypeName DateTime<br>-ArgumentList constructor-argument-list | New-Object -TypeName DateTime<br>-ArgumentList 2014,10,10 | Friday, October 10, 2014<br>12:00:00 AM   |
| Cast string to a DateTime object            | [DateTime]"date-time value"  | [DateTime]"10/10/2014"  <br>select Year, DayOfWeek, Ticks | Year DayOfWeek Ticks<br>-- ---- --<br>2014 Friday<br>635484960000000000   |
| Cast string to XML without accelerator      | [System.Xml.XmlDocument] "XML text"                                      | [System.Xml.XmlDocument]<br>"<root>text</root>"           | root<br>--<br>text  |
| Cast string to XML with accelerator         | [xml] "XML text"   | [xml] "<root>text</root>"                                 | root<br>--<br>text  |
| Access static member of .NET class          | [class]::member  | \$currentTime =<br>[datetime]::Now                        |   |
| Cast integer to Boolean                     | # [bool]2 yields True; [bool]0 yields False                              |   |   |

## Conclusion

That’s it for part 1; keep an eye out for more in the near future! While I have been over the recipes presented numerous times to weed out errors and inaccuracies, I think I may have missed one. If you locate it, please share your findings in the comments below!





 **Monthly newsletter**

**Join over 50,000 data professionals**  
Get the latest best practices, insight, and product news from our industry experts

[Get the newsletter](#)

**Products**

- Automate
- Monitor
- Standardize
- Protect & preserve

**Community & Learning**

- Product Learning
- University
- Events & Friends
- Simple Talk
- Books
- Forums

**Support**

- Forums
- Contact product support
- Find my serial numbers
- Download older versions

**Partners**

- SQL Server Central
- Resellers
- Consulting partners

**Solutions**

- Overview
- Maturity Assessment

**Privacy & compliance**

- Privacy and cookies
- License agreement
- Accessibility
- Report security issue
- Modern slavery
- Gender pay gap report
- CCPA - Do not sell my data

**Our Company**

- Careers
- Contact us
- Redgate Blog
- Our values

**Follow us**





Copyright 1999 - 2023 Red Gate Software Ltd

