The Complete Guide to Quoting in PowerShell

Download the latest version of this PowerShell™ wallchart and read the accompanying in-depth article from Simple-Talk at https://www.simple-talk.com/sysadmin/powershell/when-to-quote-in-powershell/.

When to Use Quotes

Guideline	Example

The first non-whitespace character of a *statement* determines parsing mode—command or expression—and dictates whether quotes for that *first* word are needed. Any of these [A..Z _ & . \ ] indicate *command parsing mode*; everything else is *expression parsing mode*. Thus, for the first word to be a string literal, you *must* use quotes. Otherwise, unquoted text at the beginning of a line is interpreted as a command.	Attempts to execute the named item instead of just echoing it (letter => command parsing)
	`PS> hello`
	
	This just echoes the string (quote => expression parsing)
	`PS> "hello"`

A *statement* is not synonymous with a *line*: you can have *multiple* statements on *one* line. Any of these [({ = ; |] begins a new statement and restarts parsing determination. Thus, you can have mixed parsing modes on one line.	The opening quote sets expression mode parsing. The opening parenthesis signals a new statement so it restarts parsing determination and thus **ls** sets command mode parsing `PS> 'abc' + (ls | select -first 1).name`

A *single* statement may also spread over *multiple* lines. The current parsing mode continues until a new statement begins; line breaks in the middle do *not* alter the parsing mode. If in expression mode parsing, each term must be an expression, and an *un*quoted string is *not* an expression, so a parsing error ensues.	The line break after the plus sign does not alter the expression mode parsing. To recognize it as a string it must be quoted; to recognize it as an executable, it must be in parentheses to restart parsing determination.
	`PS> 'abc' +`
	`hello`

If a *command* has embedded spaces, you must enclose it in quotes *and* precede it with the call operator (&), or just escape the spaces (i.e. precede each with a backtick). Letter and ampersand mark command parsing mode. *Without* the call operator, a string is just a string literal because a quote marks expression parsing mode.	Both of these execute stuff.exe:
	`PS> & "C:\Program Files\stuff.exe"`
	``PS> C:\Program` Files\stuff.exe``
	This just echoes the string (that happens to *look* like a program):
	`PS> "C:\Program Files\stuff.exe"`

Arguments to a command (cmdlet, program, etc.) do *not* require quotes because an argument that is not already an expression is treated as though it were double-quoted.	`PS> ls a.txt, b.txt`
If quotes are *not* needed, it does no harm to include them.	Both named files are properly listed: `PS> ls a.txt, "b.txt"`
If an *argument* has embedded spaces, you must enclose it in quotes *or* escape each space with a backtick.	This lists the two files, each with a space in it: ``PS> ls "a file.txt", b` file.txt``

The right-hand side of an operator requires quotes because you're already in expression mode parsing and that parsing mode continues throughout the expression	"foo" is a string literal in both of these:
	`PS> 'aaa' + 'foo'`
	`PS> 'aaa' -eq 'foo'`

While in expression mode parsing, each term must be an expression, and an *un*quoted string is *not* an expression, so a parsing error ensues. Use quotes to make it a string literal, or parentheses to make it an executable.	A parsing error ensues for both of these, because foo is not a valid expression.
	`PS> 'aaa' + foo`
	`PS> 'aaa' -eq foo`

The target of an assignment statement requires quotes because an equals sign triggers a new statement, and thus reassesses parsing mode, so quotes are required to recognize a string literal.	This assigns "hello" to the variable:
	`PS> $item = 'hello'`
	This attempts to execute "hello" and assign it:
	`PS> $item = hello`

Within a hash table you have what *look* like a series of simple assignment statements, *except* the left operand is a name or expression rather than a variable name. Quotes are not needed. On the right, it is a standard target of an assignment; see above.	Except that quotes *are* needed if there is an embedded space: `@{ foo="bar"; "other key"="value" }`

Which Quotes to Use

■ Use single quotes for pure, literal strings; use double quotes for interpolated (interpreted) strings. The table shows double quote interpolation. With single quotes, the output matches the input, character for character.

■ Backticks are highlighted just because they are harder to see.

■ For these examples assume you have **$x = 5** and **$obj = [PsCustomObject] @{ name = 'abc'; type = 25 }.**

Item	Input	Output
Simple variable	"$x is $x"	5 is 5
Simple var, backtick[1]	"`$x is $x"	$x is 5
Simple expression	"Sum of 2 + 3 is $(2+3)"	Sum of 2 + 3 is 5
Simple expr, backtick	"Sum of 2 + 3 is `$(2+3)"	Sum of 2 + 3 is $(2+3)
Complex variable [2]	"Obj is $obj"	Obj is @{name=abc; type=25}
Object property *without* proper decoration[3]	"Obj name is $obj.name"	Obj name is @{name=abc; type=25}.name
Object property as an expression	"Obj name is $($obj.name)"	Obj name is abc
Object property in a here string	@" Obj name is $($obj.name) "@	Obj name is abc
Special characters	"Item1`tItem2`tItem3"	Item1<*tab*>Item2<*tab*>Item3

Notes

[1] The backtick is evaluated—rather than displayed—within double-quotes. Its job is to force the next character to remain a literal, suppressing the normal evaluation of a dollar sign.

[2] Within double quotes a variable, be it simple or complex, is interpolated to its string value, i.e. whatever its ToString method returns. In the case of a PSCustomObject, it is the entire contents of the object as shown. For a .NET type, ToString often defaults to just the name of the type rather than any contents of the instance, but that varies by type.

[3] Within double quotes the string representation of the whole object ($obj) is displayed (again being what its ToString method returns). It is not just the specified property because interpolation stops at the first character that cannot be part of a simple variable name, in this case the period.

Using Quotes within Quotes

Item	Input	Output
Single in double	"one 'two' three"	one 'two' three
Double in single	'one "two" three'	one "two" three
Double in double [1,2]	"one ""two"" three"	one "two" three
Double/double, backtick	"one `"two`" three"	one "two" three
Single in single [3,4]	'one ''two'' three'	one 'two' three
Single/single, backtick[5]	'one `'two`' three'	*error*
Double in double, here string[6]	@" one "two" three "@	one "two" three
Single in single, here string[6]	@' one 'two' three '@	one 'two' three

Notes

[1] Doubling the quote mark results in one double quote mark in the output.

[2] An alternate way to embed a double quote is to escape it with a backtick. The backtick forces the next character to be a literal, even for quotes.

[3] Doubling the quote mark results in one single quote mark in the output. (This is the rare case of a single quoted string *not* matching its input exactly!)

[4]Note that those are two juxtaposed single quotes on either side of "two" in the first line here, even though they look just like double-quotes.

[5] The alternate approach of escaping a quote with a backtick does *not* work for single-quoted strings, because a backtick—like everything else within a single-quoted string—is *not* evaluated.

[6] Within a *here string* you can freely embed quotes just like any other character—no doubling or escaping necessary.

References

PowerShell Language Specification Version 3.0, Understanding PowerShell Parsing Modes, about_Quoting_Rules, about_Special_Characters, Here Strings,