about_Quoting_Rules - PowerShell | Microsoft Learn

11-14 minutes

- Article
- 09/09/2024
-

In this article

Short description

Describes rules for using single and double quotation marks in PowerShell.

Long description

When parsing, PowerShell first looks to interpret input as an expression. But when a command invocation is encountered, parsing continues in argument mode. Non-numeric arguments without quotes are treated as strings. If you have arguments that contain spaces, such as paths, then you must enclose those argument values in quotes. For more information about argument parsing, see the **Argument mode** section of about_Parsing.

Quotation marks are used to specify a literal string. You can enclose a string in single quotation marks (') or double quotation marks (").

Quotation marks are also used to create a *here-string*. A here-string is a single-quoted or double-quoted string in which quotation marks are interpreted literally. A here-string can span multiple lines. All the lines in a here-string are interpreted as strings, even though they're not enclosed in quotation marks.

In commands to remote computers, quotation marks define the parts of the command that are run on the remote computer. In a remote session, quotation marks also determine whether the variables in a command are interpreted first on the local computer or on the remote computer.

Note

PowerShell treats smart quotation marks, also called typographic or curly quotes, as normal quotation marks for strings. Don't use smart quotation marks to enclose strings. When writing strings that contain smart quotation marks, follow the guidance in the Including quote characters in a string section of this document. For more information about smart quotation marks, see the *Smart Quotes* section in the Wikipedia article Quotation marks in English.

Double-quoted strings

A string enclosed in double quotation marks is an *expandable* string. Variable names preceded by a dollar sign (`$`) are replaced with the variable's value before the string is passed to the command for processing.

For example:

```
$i = 5
"The value of $i is $i."
```

The output of this command is:

```
The value of 5 is 5.
```

Also, in a double-quoted string, expressions are evaluated, and the result is inserted in the string. For example:

```
"The value of $(2+3) is 5."
```

The output of this command is:

```
The value of 5 is 5.
```

Only basic variable references can be directly embedded in an expandable string. Variables references using array indexing or member access must be enclosed in a subexpression. For example:

```
"PS version: $($PSVersionTable.PSVersion)"
```

```
PS version: 7.4.5
```

To separate a variable name from subsequent characters in the string, enclose it in braces (`{}`). This is especially important if the variable name is followed by a colon (`:`). PowerShell considers everything between the `$` and the `:` a scope specifier, typically causing the interpretation to fail. For example, `"$HOME: where the heart is."` throws an error, but `"${HOME}: where the heart is."` works as intended.

To prevent the substitution of a variable value in a double-quoted string, use the backtick character (`` ` ``), which is the PowerShell escape character.

In the following example, the backtick character that precedes the first `$i` variable prevents PowerShell from replacing the variable name with its value. For example:

```
$i = 5
"The value of `$i is $i."
```

The output of this command is:

```
The value of $i is 5.
```

Single-quoted strings

A string enclosed in single quotation marks is a *verbatim* string. The string is passed to the command exactly as you type it. No substitution is performed. For example:

```
$i = 5
'The value of $i is $i.'
```

The output of this command is:

```
The value $i is $i.
```

Similarly, expressions in single-quoted strings aren't evaluated. They're interpreted as string literals. For example:

```
'The value of $(2+3) is 5.'
```

The output of this command is:

```
The value of $(2+3) is 5.
```

Including quote characters in a string

To make double-quotation marks appear in a string, enclose the entire string in single quotation marks. For example:

```
'As they say, "live and learn."'
```

The output of this command is:

```
As they say, "live and learn."
```

You can also enclose a single-quoted string in a double-quoted string. For example:

```
"As they say, 'live and learn.'"
```

The output of this command is:

```
As they say, 'live and learn.'
```

Or, double the quotation marks around a double-quoted phrase. For example:

```
"As they say, ""live and learn."""
```

The output of this command is:

```
As they say, "live and learn."
```

To include a single quotation mark in a single-quoted string, use a second consecutive single quote. For example:

```
'don''t'
```

The output of this command is:

```
don't
```

To force PowerShell to interpret a double quotation mark literally, use a backtick character. This prevents PowerShell from interpreting the quotation mark as a string delimiter. For example:

```
"Use a quotation mark (`") to begin a string."
'Use a quotation mark (`") to begin a string.'
```

Because the contents of single-quoted strings are interpreted literally, the backtick character is treated as a literal character and displayed in the output.

```
Use a quotation mark (") to begin a string.
Use a quotation mark (`") to begin a string.
```

Because PowerShell interprets smart quotation marks, like ‘, ’, “, and ”, as normal quotation marks, smart quotation marks also need to be escaped. For example:

```
"Double ““smart quotation marks`” must be escaped in a double-quoted
string."
'Single ‘‘smart quotation marks’’ must be escaped in a single-quoted
string.'
```

```
Double “smart quotation marks” must be escaped in a double-quoted string.
Single ‘smart quotation marks’ must be escaped in a single-quoted string.
```

Here-strings

The quotation rules for here-strings are slightly different.

A here-string is a single-quoted or double-quoted string surrounded by at signs (@). Quotation marks within a here-string are interpreted literally.

A here-string:

- spans multiple lines
- begins with the opening mark followed by a newline
- ends with a newline followed by the closing mark
- includes every line between the opening and closing marks as part of a single string

Like regular strings, variables are replaced by their values in double-quoted here-strings. In single-quoted here-strings, variables aren't replaced by their values.

You can use here-strings for any text, but they're particularly useful for the following kinds of text:

- Text that contains literal quotation marks
- Multiple lines of text, such as the text in an HTML or XML block
- The Help text for a script or function document

A here-string can have either of the following formats, where `<Enter>` represents the linefeed or newline hidden character that's added when you press the `ENTER` key.

Double-quotes:

```
@"<Enter>
<string> [string] ...<Enter>
"@
```

Single-quotes:

```
@'<Enter>
<string> [string] ...<Enter>
'@
```

Note

The final newline character is part of the closing mark. It's not added to the here-string.

A here-string contains all the text between the opening and closing marks. In the here-string, all quotation marks are interpreted literally. For example:

```
@"
For help, type "get-help"
"@
```

The output of this command is:

```
For help, type "get-help"
```

Using a here-string can simplify using a string in a command. For example:

```
@"
Use a quotation mark, like ' or ", to begin a string.
"@
```

The output of this command is:

```
Use a quotation mark, like ' or ", to begin a string.
```

In single-quoted here-strings, variables are interpreted literally and reproduced exactly. For example:

```
@'
The $profile variable contains the path
of your PowerShell profile.
'@
```

The output of this command is:

```
The $profile variable contains the path
of your PowerShell profile.
```

In double-quoted here-strings, variables are replaced by their values. For example:

```
@"
Even if you have not created a profile,
the path of the profile file is:
$profile.
"@
```

The output of this command is:

```
Even if you have not created a profile,
the path of the profile file is:
C:\Users\User1\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1.
```

Here-strings are typically used to assign multiple lines to a variable. For example, the following here-string assigns a page of XML to the $page variable.

```
$page = [XML] @"
<command:command xmlns:maml="http://schemas.microsoft.com/maml/2004/10"
xmlns:command="http://schemas.microsoft.com/maml/dev/command/2004/10"
xmlns:dev="http://schemas.microsoft.com/maml/dev/2004/10">
<command:details>
        <command:name>
               Format-Table
```

```
        </command:name>
        <maml:description>
            <maml:para>Formats the output as a table.</maml:para>
        </maml:description>
        <command:verb>format</command:verb>
        <command:noun>table</command:noun>
        <dev:version></dev:version>
</command:details>
...
</command:command>
"@
```

Here-strings are also a convenient format for input to the `ConvertFrom-StringData` cmdlet, which converts here-strings to hash tables. For more information, see `ConvertFrom-StringData`.

Note

PowerShell allows double- or single-quoted strings to span multiple lines without using the @ syntax of here-strings. However, full here-string syntax is the preferred usage.

Interpretation of expandable strings

Expanded strings don't necessarily look the same as the default output that you see in the console.

Collections, including arrays, are converted to strings by placing a single space between the string representations of the elements. A different separator can be specified by setting preference variable `$OFS`. For more information, see the $OFS preference variable.

Instances of any other type are converted to strings by calling the `ToString()` method, which may not give a meaningful representation. For example:

```
"hashtable: $(@{ key = 'value' })"
```

```
hashtable: System.Collections.Hashtable
```

To get the same output as in the console, use a subexpression in which you pipe to `Out-String`. Apply the `Trim()` method if you want to remove any leading and trailing empty lines.

```
"hashtable:`n$((@{ key = 'value' } | Out-String).Trim())"
```

```
hashtable:
Name                           Value
----                           -----
key                            value
```

Culture settings affect string interpretation

The `ToString()` methods uses the current configured culture settings to convert values to strings. For example, the culture of the following PowerShell session is set to `de-DE`. When the `ToString()` method converts the value of `$x` to a string it uses a comma (`,`) for the decimal separator. Also, the `ToString()` method converts the date to a string using the appropriate format for the German locale settings.

```
PS> Get-Culture

LCID             Name             DisplayName
----             ----             -----------
1031             de-DE            German (Germany)

PS> $x = 1.2
PS> $x.ToString()
1,2

PS> (Get-Date 2024-03-19).ToString()
19.03.2024 00:00:00
```

However, PowerShell uses the invariant culture when interpreting expandable string expressions.

```
PS? "$x"
1.2

PS> "$(Get-Date 2024-03-19)"
03/19/2024 00:00:00
```

Passing quoted strings to external commands

Some native commands expect arguments that contain quote characters. PowerShell interprets the quoted string before passing it to the external command. This interpretation removes the outer quote characters.

For more information about this behavior, see the about_Parsing article.

See also

- about_Special_Characters
- ConvertFrom-StringData

Previous Chapter Next Chapter