



The Complete Guide to PowerShell Punctuation

- Does *not* include special characters in globs ([about Wildcards](#)) or regular expressions ([about Regular Expressions](#)) as those are separate “languages”.
- **Green** items are placeholders indicating where you insert either a single word/character or, with an ellipsis, a more complex expression.

Symbol	What it is	Explanation
<enter> carriage return	line break	Allowed between statements, within strings, after these separators [, ; =] and—as of V3—these [. : :]. Also allowed after opening tokens [{ [(' "]. <i>Not</i> allowed most anywhere else.
;	statement separator	<i>Optional</i> if you always use line breaks after statements. <i>Required</i> if you put multiple statements on one line, e.g. <code>\$a = 25; Write-Output \$a</code>
\$name dollar sign	variable prefix	\$ followed by letters, numbers, or underscores specifies a variable name, e.g. <code>\$width</code> . Letters and numbers are <i>not</i> limited to ASCII; some 18,000+ Unicode chars are eligible.
{...}	variable prefix	To embed any <i>other</i> characters in a variable name enclose it in braces, e.g. <code>\${save-items}</code> . See about Variables
(...)	(a) grouping expression	Wrap any <i>single</i> statement (or single command-stream connected by pipes) to override default precedence rules. See the subexpression operator <code>()</code> for multiple commands. <i>Group at the front:</i> access a property from the result of an operation, e.g. <code>(get-process -name win*).name</code> <i>Group at the end:</i> pass the result of an operation as an argument: <code>write-output (1,2,3 -join '*')</code>
	(b) grouping operator	Override operator precedence: e.g. <code>8 + 4 / 2</code> vs. <code>(8 + 4)/2</code>
	(c) .NET function arg container	Unlike when calling native PowerShell functions, calling .NET functions require parentheses: <code>\$hashTable.ContainsKey(\$x)</code>
\$(...)	(a) sub-expression	Wrap <i>multiple</i> semicolon-separated statements, where the output of each contributes to the total output: <code>\$(x=1;\$y=2;\$x;\$y)</code>
	(b) sub-expression inside a string	Interpolate simple variables in a double-quoted string with just <code>\$</code> , but complex expressions must be wrapped in a subexpression. Ex: <code>\$p = ps select -first 1</code> then <code>"proc name is \$(\$p.name)"</code>
@(...) array	array sub-expression	Same as a <i>sub-expression</i> , except this returns an array even with zero or one objects. Many cmdlets return a collection of a certain type, say X. If two or more, it is returned as an array of X whereas if you only get one object then it is just an X . Wrapping the call with this operator forces it to always be an array, e.g. <code>\$a = @(ps where name -like 'foo')</code> See about Arrays
@{...} hash	hash initializer	Defines a hash table with the format <code>@{ name1=value1; name2=value2; ... }</code> . Example: <code>\$h = @{abc='hello'; color='green'}</code> . You can then access values by their keys, e.g. <code>\$h['color']</code> or <code>\$h.color</code> . See about Hash Tables
{...} braces	script block	Essentially an anonymous function. Ex: <code>\$sb = {param(\$color="red"); "color=\$color"} then & \$sb 'blue'</code> . See about Script Blocks
[...] brackets	(a) array indexer	<code>\$data[4]</code> returns the 5th element of the <code>\$data</code> array.
	(b) hash indexer	<code>\$hash['blue']</code> returns the value associated with key 'blue' in the hash (though you could also use <code>\$hash.blue</code>)
	(c) static type	Use to call a static methods, e.g. <code>[Regex]::Escape(\$x)</code>
	(d) type cast	Cast to a type just like C# (<code>[int]"5.2"</code>) but in PS you can <i>also</i> cast the variable itself (<code>[xml]\$x='<abc/>'</code>). Also applies for function args: <code>function f([int]\$i) {...}</code>
	(e) array type designator	Cast to an array type—use with no content inside: <code>function f([int[]] \$values) {...}</code> .
\$_	pipeline object	This special variable holds the current pipeline object (now with a more friendly alias as well, <code>\$PSItem</code>), e.g. <code>ps where { \$_.name -like 'win*' }</code>
@name splat	splatting prefix	Allows passing a collection of values stored in a hash table or in an array as parameters to a cmdlet. Particularly useful to forward arguments passed in to another call with <code>@Args</code> or <code>@PsBoundParameters</code> . See about Splatting
? question mark	alias for Where-Object	Instead of <code>Get-Stuff where-Object { ... }</code> you can write the oft-used cmdlet with the terse alias: <code>Get-Stuff ? { ... }</code>
% percent	(a) alias for ForEach-Object	Instead of <code>ls ForEach-Object name</code> you can write the oft-used cmdlet with the terse alias: <code>ls % name</code>
	(b) modulo	Returns the remainder of a division operation e.g. <code>(7 % 2)</code> returns 1.
%=	modulo & store	Common shorthand identical to that in C#: <code>\$x %= 5</code> is shorthand for <code>\$x = \$x % 5</code> .
: colon	(a) drive designator	Just like conventional Windows drives (<code>dir C:\</code> , etc.) you can use <code>dir alias:</code> to see the contents of the alias drive or <code>\$env:path</code> to see the <code>\$path</code> variable on the env drive.
	(b) variable scope specifier	An undecorated variable (<code>\$stuff</code>) implicitly specifies the current scope. Reference <code>\$script:stuff</code> or <code>\$global:stuff</code> for a different scope. See about Scopes
	(c) switch param binder	Switch params are typically present for true (<code>-mySwitch</code>) or absent for false. Can be explicit: <code>-mySwitch:\$false</code>
:: double colon	static member accessor	Specify a static .NET <i>method</i> , e.g. <code>[String]::Join(...)</code> or <code>[System.IO.Path]::GetTempFileName()</code> , or a static <i>property</i> <code>[System.Windows.Forms.Keys]::Alt</code> or <code>[int]::MaxValue</code> .
, comma	array builder	Specify an array to feed a pipeline, e.g. <code>1,3,5,7 ForEach-Object { \$_ * 2 }</code> or specify an array argument, <code>ps -name winword,spoolsv</code>
. period; dot	(a) separator in class path	E.g. <code>System.IO.FileInfo</code> just as in C#
	(b) property / method dereference	Specify property of simple object <code>\$myArray.Length</code> or complex one (<code>ps ? Name -like "win*").name</code> or method <code>\$hashTable.ContainsKey(\$x)</code>
	(c) dot-source operator	Load a PowerShell file into the current scope (e.g. <code>. myScript.ps1</code>) rather than into a subshell.
.. double dot	range operator	Initialize an array (e.g. <code>\$a = 1..10</code>) or return an array slice (<code>\$a[3..6]</code>).
# octothorp	(a) comment	Everything following, through the end of the line, is a comment.
	(b) history recall	On the command-line, you can type <code>#<tab></code> to recall the last command for editing. Also, <code>#string<tab></code> recalls the last command containing <i>string</i> ; subsequent tabs continue through the history stack http://bit.ly/1Jtp4v . (Since V2)

Symbol	What it is	Explanation
<#...> #>	Multi-line comment	Everything between the opening and closing tokens—which may span multiple lines—is a comment.
& ampersand	call operator	Forces the next thing to be interpreted as a command even if it looks like a string. So while either <code>Get-ChildItem</code> or <code>& Get-ChildItem</code> do the same thing, <code>"Program Files\stuff.exe"</code> just echoes the string literal, while <code>& "Program Files\stuff.exe"</code> will execute it.
` back tick; grave accent	(a) line continuation	As the last character on a line, lets you continue to the next line where a line break is not normally allowed. Make sure it is really <i>last</i> —no trailing spaces. Avoid using this whenever possible! See about Escape Characters
	(b) literal character	Precede a dollar sign to avoid interpreting the following characters as a variable name; precede a quote mark inside a string to embed that quote in the string instead of ending the string. See about Escape Characters
	(c) special character	Followed by one of a set of pre-defined characters, allows inserting special characters, e.g. <code>`t</code> = tab, <code>`r</code> = carriage return, <code>`b</code> = backspace. See about Special Characters
'...' single quote	literal string	String with no interpolation; typically used for single-line strings but can be used for multi-line as well.
"..." double quote	interpolated string	String with interpolation of variables, sub-expressions, escapes, and special characters (e.g. <code>`t</code>). See about Escape Characters and about Special Characters
@'...' '@'	literal here-string	A multi-line string with <i>no</i> interpolation; differs from a normal string in that you can embed single quotes within the string without doubling or escaping.
@"..." "@"	interpolated here-string	A multi-line string with interpolation; differs from a normal string in that you can embed double quotes within the string without doubling or escaping.
 pipe	command connector	Pipe output of one command to input of next, e.g. <code>ps select ProcessName</code>
> greater than	divert to file / overwrite	Redirects & overwrites (if file exists) stdout stream to a file (e.g. <code>ps > process_list.txt</code>). See about Redirection It's a “greater than” symbol but it <i>doesn't</i> do comparisons: for algebraic operators use <code>-gt</code> or <code>-lt</code> , e.g. <code>(\$x -lt \$y)</code> .
>>	divert to file / overwrite	Redirects & overwrites (if file exists) numbered stream (2 thru 5) or all streams (use <code>*</code>) to a file e.g. <code>ps 4> process_list.txt</code>
>>>	divert to file / append	Redirects & appends stdout stream to a file, e.g. <code>ps >> process_list.txt</code> . See about Redirection
>>>>	divert to file / append	Redirects & appends numbered stream (2 thru 5) or all streams (use <code>*</code>) to a file, e.g. <code>ps *>> out.txt</code>
>>>&1	output redirect to stdout	Redirects an output stream (2 thru 5) to stdout stream, effectively merging that stream with stdout. Ex: to merge errors with stdout: <code>Do-SomethingErrorProne 2>&1</code>
= equals	assignment operator	Assign a value to a variable, e.g. <code>\$stuff = 25</code> or <code>\$procs = ps select -first 5</code> . Use <code>-eq</code> or <code>-ne</code> for equality operators: <code>("ab" -eq \$x)</code> or <code>(\$amt -eq 100)</code> .
! exclamation	Logical not	Negates the statement or value that follows. Equivalent to the <code>-not</code> operator. <code>if (!\$canceled) ...</code>
+ plus	(a) add	Adds numbers, e.g. <code>(\$val + 25)</code> .
	(b) concatenate	Concatenates strings, arrays, hash tables, e.g. <code>('hi'+ '!')</code> .
	(c) nested class access	Typically best practice says not to have public nested classes but when needed you need a plus to access, e.g. <code>[Net.WebRequestMethod+Ftp]</code> See Plus (+) in .NET Class Names
+= compound assignment	add & store	Common shorthand identical to that in C#: <code>\$x += 5</code> is shorthand for <code>\$x = \$x + 5</code> . Can also be used for concatenation as described under <i>plus</i> .
- hyphen	(a) negate	Negate a number (<code>-\$val</code>).
	(b) subtract	Subtract one number from another (<code>\$v2 - 25.1</code>).
	(c) operator prefix	Prefixes lots of operators: logical (<code>-and</code> , <code>-or</code> , <code>-not</code>), comparison (<code>-eq</code> , <code>-ne</code> , <code>-gt</code> , <code>-lt</code> , <code>-le</code> , <code>-ge</code>), bitwise (<code>-band</code> , <code>-bor</code> , <code>-bxor</code> , <code>-bnot</code>), and more. See about Operators
	(d) verb/noun separator	Separates the verb from the noun in every cmdlet, e.g. <code>Get-Process</code> .
-= asterisk	subtract & store	Common shorthand identical to that in C#: <code>\$x -= 5</code> is shorthand for <code>\$x = \$x - 5</code> .
* multiply	multiply	Multiply numbers, e.g. <code>(\$val * 3.14)</code> .
*= multiply & store	multiply & store	Common shorthand identical to that in C#: <code>\$x *= 5</code> is shorthand for <code>\$x = \$x * 5</code> .
/ virgule	divide	Divide numbers, e.g. <code>(\$val / 3.14)</code> .
/=	divide & store	Common shorthand identical to that in C#: <code>\$x /= 5</code> is shorthand for <code>\$x = \$x / 5</code> .
++	increment	Auto-increment a variable: increment then return value (<code>++\$v</code>) or return value then increment (<code>(\$v++)</code>).
--	decrement	Auto-decrement a variable: decrement then return value (<code>--\$v</code>) or return value then decrement (<code>(\$v--)</code>).
--%	stop parsing or verbatim parameter	Inserted in the midst of a statement, PowerShell treats any arguments after it as literals <i>except</i> for DOS-style environment variables (e.g. <code>%PATH%</code>). See about Parsing
\$\$		Get the last token in the previous line.
\$^		Get the first token in the previous line.
 \$? 		Execution status of the last operation (<code>\$true</code> or <code>\$false</code>); contrast with <code>\$LastExitCode</code> that reports the exit code of the last Windows-based program executed.

References

[about Automatic Variables](#), [about Preference Variables](#), [about Environment Variables](#), [about Quoting Rules](#), [When to Quote in PowerShell](#)