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 [{ [(' "] . *Not* allowed most anywhere else.
; semicolon	statement separator	*Optional* if you always use line breaks after statements; *required* to put multiple statements on one line, e.g. $a = 25; Write-Output $a
$*name* dollar sign	variable prefix	$ followed by letters, numbers, or underscores specifies a variable name, e.g. $width. Letters and numbers are *not* limited to ASCII; some 18,000+ Unicode chars are eligible.
${…}	variable prefix	To embed any *other* characters in a variable name enclose it in braces, e.g ${save-items}. See about_Variables.
${*path*}	path accessor	Special case: ${*drive-qualified path*} lets you, e.g., store to (${C:tmp.txt}=1,2,3) or retrieve from ($data=${C:tmp.txt}) a file. See Provider Paths.
(…)	(a) grouping expression	Wrap any *single* statement (or single command-stream connected by pipes) to override default precedence rules. See the subexpression operator $() for multiple commands. *Group at the front:* access a property from the result of an operation, e.g. (get-process -name win*).name *Group at the end:* pass the result of an operation as an argument: write-output (1,2,3 -join '*')
	(b) grouping operator	Override operator precedence: e.g. 8 + 4 / 2 vs. (8 + 4)/2
	(c) .NET function arg container	Unlike when calling native PowerShell functions, calling .NET functions require parentheses: $hashTable.ContainsKey($x)
$(…)	(a) sub-expression	Wrap *multiple* statements, where the output of each contributes to the total output: $($x=1;$y=2;$x;$y)
	(b) sub-expression inside a string	Interpolate simple variables in a double-quoted string with just $, but complex expressions must be wrapped in a subexpression. Ex: $p = ps | select -first 1 then "proc name is $($p.name)"
@(…) array	array sub-expression	Same as a **sub-expression**, 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. $a = @(ps | where name -like 'foo') See about_Arrays
@{…} hash	hash initializer	Defines a hash table with the format @{ *name1*=*value1*; *name2*=*value2*; ...}. Example: $h = @{abc='hello'; color='green'}. You can then access values by their keys, e.g. $h['color'] or $h.color. See about_Hash_Tables
{…} braces	script block	Essentially an anonymous function. Ex: $sb = {param($color="red"); "color=$color"} then & $sb 'blue'. See about_Script_Blocks
[…] brackets	(a) array indexer	$data[4] returns the 5th element of the $data array.
	(b) hash indexer	$hash['blue'] returns the value associated with key 'blue' in the hash (though you could also use $hash.blue)
	(c) static type	Use to call a static methods, e.g. [Regex]::Escape($x)
	(d) type cast	Cast to a type just like C# ([int]"5.2") but in PS you can *also* cast the variable itself ([xml]$x='<abc/>'). Also applies for function args: function f([int]$i) {...}
	(e) array type designator	Cast to an array type—use with no content inside: function f([int[]] $values) {...}.
$_	pipeline object	This special variable holds the current pipeline object (now with a more friendly alias as well, $PSItem), e.g. ps | where { $_.name -like 'win*' }
@*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 @Args or @PsBoundParameters. See about_Splatting
? question mark	alias for Where-Object	Instead of Get-Stuff | Where-Object { ... } you can write the oft-used cmdlet with the terse alias: Get-Stuff | ? { ... }
%{…}	Alias for ForEach-Object	Instead of 1..5| ForEach-Object { $_ * 2 } you can write the oft-used cmdlet as: 1..5| % { $_ * 2 }
% percent	(a) alias for ForEach-Object	Special case of above for a single property of pipeline input: ls| % name is equivalent to ls| % { $_.name}
	(b) modulo	Returns the remainder of a division e.g. (7 % 2) returns 1.
%=	modulo & store	Common shorthand identical to that in C#: $x %= 5 is shorthand for $x = $x % 5.
: colon	(a) drive designator	Just like conventional Windows drives (dir C:\, etc.) you can use dir alias: to see the contents of the alias drive or $env:path to see the $path variable on the env drive.
	(b) variable scope specifier	An undecorated variable, e.g. $stuff implicitly specifies the current scope. But you can also reference $script:stuff or $global:stuff to specify a different scope. See about_Scopes
:: double colon	static member accessor	Specify a static .NET *method*, e.g. [String]::Join(...) or [System.IO.Path]::GetTempFileName(), or a static *property* [System.Windows.Forms.Keys]::Alt or [int]::MaxValue.
, comma	array builder	Specify an array to feed a pipeline, e.g. 1,3,5,7 | ForEach-Object { $_ * 2 } or specify an array argument, ps -name winword,spoolsv
. period; dot	(a) separator in class path	E.g. System.IO.FileInfo just as in C#
	(b) property / method dereference	Specify property of simple object $myArray.Length or complex one (ps | ? Name -like "win*").name or method $hashTable.ContainsKey($x)
	(c) dot-source operator	Load a PowerShell file into the current scope (e.g. . myScript.ps1) rather than into a subshell.
.. double dot	range operator	Initialize an array (e.g. $a = 1..10) or return an array slice ($a[3..6]).
# octothorp	(a) comment	Everything through the end of the line is a comment.
	(b) history recall	On the command-line, you can type #<tab> to recall the last command for editing. Also, #*string*<tab> recalls the last command containing *string*; subsequent tabs continue through the history stack. (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 Get-ChildItem or & Get-ChildItem do the same thing, "Program Files\stuff.exe" just echoes the string literal, while & "Program Files\stuff.exe" will execute it.
` back tick; grave accent	(a) line continuation	As the last character on a line, lets you continue on the next line where PowerShell would not normally allow a line break. Make sure it is really *last*—no trailing spaces! 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. `t = tab, `r = carriage return, `b = 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. `t). See about_Escape_Characters and about_Special_Characters
@' … '@	literal here-string	A multi-line string with ***no*** 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. ps | select ProcessName
> greater than	divert to file / overwrite	Redirects & overwrites (if file exists) stdout stream to a file (e.g. ps > process_list.txt). See about_Redirection It's a "greater than" symbol but it *doesn't* do comparisons: for algebraic operators use **-gt** or **-lt**, e.g. ($x -lt $y).
***n*>**	divert to file / overwrite	Redirects & overwrites (if file exists) numbered stream (2 thru 5) or all streams (use *) to a file e.g. ps 4> process_list.txt
>>	divert to file / append	Redirects & appends stdout stream to a file, e.g. ps >> process_list.txt. See about_Redirection
***n*>>**	divert to file / append	Redirects & appends numbered stream (2 thru 5) or all streams (use *) to a file, e.g. ps *>> out.txt
***n*>&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: Do-SomethingErrorProne 2>&1
= equals	assignment operator	Assign a value to a variable, e.g. $stuff = 25 or $procs = ps | select -first 5. Use **-eq** or **-ne** for equality operators: ("ab" -eq $x) or ($amt -eq 100).
! exclamation	Logical not	Negates the statement or value that follows. Equivalent to the **-not** operator. if (!$canceled) ...
+ plus	(a) add	Adds numbers, e.g. ($val + 25).
	(b) concatenate	Concatenates strings, arrays, hash tables, e.g. ('hi'+'!').
	(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. [Net.WebRequestMethods+Ftp] See Plus (+) in .NET Class Names
+= compound assignment	add & store	Common shorthand identical to that in C#: $x += 5 is shorthand for $x = $x + 5. Can also be used for concatenation as described under *plus* and concatenation direct to a path: ${c:output.txt) += 'one','two'
- hyphen	(a) negate	Negate a number (-$val).
	(b) subtract	Subtract one number from another ($v2 - 25.1).
	(c) operator prefix	Prefixes lots of operators: logical (-and, -or, -not), comparision (-eq, -ne, -gt, -lt, -le, -ge), bitwise (-bAND, -bOR, -bXOR, -bNOT), and more.
	(d) verb/noun separator	Separates the verb from the noun in every cmdlet, e.g. Get-Process.
-=	subtract & store	Common shorthand identical to that in C#: $x -= 5 is shorthand for $x = $x - 5.
***** asterisk	(a) multiply	Multiply numbers, e.g. ($val * 3.14).
	(b) replicate	Replicate arrays, e.g. ('a','b' * 2).
***=**	multiply & store	Common shorthand identical to that in C#: $x *= 5 is shorthand for $x = $x * 5. Can also be used for replication as described under *asterisk* and replication direct to a path: ${c:output.txt) *= 3
/ virgule	divide	Divide numbers, e.g. ($val / 3.14).
/=	divide & store	Common shorthand identical to that in C#: $x /= 5 is shorthand for $x = $x / 5.
++	increment	Auto-increment a variable: increment then return value (++$v) or return value then increment ($v++).
--	decrement	Auto-decrement a variable: decrement then return value (++$v) or return value then decrement ($v++).
--%	stop parsing or verbatim parameter	Inserted in the midst of a statement, PowerShell treats any arguments after it as literals *except* for DOS-style environment variables (e.g, %PATH%). See about_Parsing
$$		Get the last token in the previous line.
$^		Get the first token in the previous line.
$?		Execution status of the last operation ($true or $false); contrast with $LastExitCode that reports the exit code of the last Windows-based program executed.

References
about_Automatic_Variables, about_Preference_Variables, about_Operators, about_Environment_Variables, about_Quoting_Rules, When to Quote in PowerShell,