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

04 April 2014
by Michael Sorens

> 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

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 head 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 fron the link at the top 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	Get-Help cmd (cmd is a full name)	help Get-ChildItem
Help in separate window	Show-Command cmd; then select the help icon	Show-Command Get-ChildItem
List help topics containing x	Get-Help string (string is a prefix or uses wildcards)	[1a] help Get [1b] help Get-*
Help for parameter y of cmdlet x	Get-Help cmd -parameter y	help Get-Date -param month
Help for multiple parameters	Get-Help cmd -parameter y* (i.e. use wildcards)	help Get-Date -param m*
List allowed values for parameter y of cmdlet x	Get-Help cmd -parameter y	help Out-File -parameter Encoding
Intellisense for parameter names [in ISE]	cmdlet -paramNamePrefix	Out-File -enc
Intellisense for parameter values [in ISE]	cmdlet -paramName<space> paramValuePrefix	Out-File -enc<space>
Auto-completion for parameter names	cmdlet - paramNamePrefix <tab>	[1a] Out-File -<tab> [1b] Out-File -enc<tab>
Auto-completion for parameter values	cmdlet -paramName<space> paramValuePrefix <tab>	[1a] Out-File -enc<space><tab> [1b] Out-File -enc<space>u<tab>
List all 'conceptual' topics (see text above)	Get-Help about_*	same

Filter help output by regex	Get-Help topic | Out-String -stream | sls -pattern regex	help ls | Out-String -Stream | Select-String recurse
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 [2] Get-Help topic > file	[1] help Get-ChildItem | Out-String | sc help.txt [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 [2] $pwd [3] $pwd.Path	[1] cd c:\foo; pwd [2] cd c:\foo; $pwd [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: [1b] sl c:\documents\me [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 [2] Split-Path filespec -Leaf	[1] (Get-Item \users\me\myfile.txt).Name [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 [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 [2] Split-Path (Split-Path filespec -Parent) -Leaf	[1] (Get-Item (Split-Path \users\me\myfile.txt -Parent)).Name [2] Split-Path (Split-Path \users\me -Parent) -Leaf

	[3] "filespec".split("\")[-2]	[3a] "\users\me\myfile.txt".split("\")[-2] [3b] "\users\me".split("\")[-2]
Display working directory (see http://bit.ly/1j4uomr)	[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 (location may be on any supported PSDrive—see list datastores below).	[1] Get-ChildItem path [2] Get-ChildItem psdrive:path [3] Get-ChildItem psdrive:	[1a] Get-ChildItem [1b] Get-ChildItem . [2a] gci c:\users\me\documents [2b] ls SQLSERVER:\SQL\localhost-\SQLEXPRESS\Databases [3a] dir env: [3b] dir variable: [3c] ls alias:
List names of files in current directory	[1] Get-ChildItem | select -ExpandProperty name [2] dir | % { $_.Name } [3] (dir).Name	same
List names of files recursively	[1] dir -Recurse | select -ExpandProperty Name [2] (dir -Recurse).Name	same
List full paths of files recursively	[1] dir -Recurse | select -ExpandProperty FullName [2] (dir -Recurse).FullName	same
List full paths of files recursively with directory marker	dir -r | % { $_.FullName + $(if ($_.PsIsContainer) {'\'}) }	same
List relative paths of files recursively with directory marker	dir -r | % { $_.FullName.substring($pwd.Path.length+1) + $(if ($_.PsIsContainer) {'\'}) }	same
List file and directory sizes (see http://stackoverflow.com/a/14031005)	dir | % { New-Object PSObject -Property @{ Name = $_.Name; Size = if($_.PSIsContainer) { (gci $_.FullName -Recurse \| Measure Length -Sum).Sum } else {$_.Length}; Type = if($_.PSIsContainer) {'Directory'} else {'File'} } }	same
List datastores (regular filesystem drives plus drives from other providers)	[1] Get-PSDrive [2] Get-PSDrive -PSProvider provider	[1] Get-PSDrive [2] gdr -PSProvider FileSystem
List providers (suppliers of datastores)	[1] Get-PSProvider [2] Get-PSProvider -PSProvider provider	[1] Get-PSProvider [2] Get-PSProvider -PSProvider 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 comment here #>	
Continue command on multiple lines (required unless break after pipe or curly bracket)	` (backquote as last character on line)	"hello " + ` "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 is 25
Non-printable characters (newline, tab, etc.)	`n, `t	"line one`n line two"	line one line two
Boolean constant TRUE (see here and http://bit.ly/1iGhXOW)	[1] $TRUE [2] Any string of length > 0 [3] Any number not equal to 0 [4] Array of length > 1 [5] Array of length 1 whose element is true [6] A reference to any object	[1] if ($TRUE) { "true" } else { "not true" } [2] if ("abc") { "true" } else { "not true" } [3] if (-99.5) { "true" } else { "not true" } [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" }	true true true true true true
Boolean constant FALSE (see here)	[1] $FALSE [2] Empty string [3] Any number = 0 (e.g. 0, 0.0, 0x0, 0mb, 0D, ...) [4] Array of length 0 [5] Array of length 1 whose element is false [6] $NULL	[1] if ($FALSE) { "true" } else { "not true" } [2] if ("") { "true" } else { "not true" } [3] if (0x0) { "true" } else { "not true" } [4] if (@()) { "true" } else { "not true" } [5] if (("")) { "true" } else { "not true" } [6] if ($NULL) { "true" } else { "not true" }	not true not true not true not true not true not true
null	$null	$null	
Iterate each element in a list	ForEach-Object	1..3 | % { $_ * 2 }	2 4 6
Ternary operator, scalar (e.g. result = a > b ? x : y;)	[1] $result = switch (boolExpr) { $true { $x } $false { $y } } [2] $result = if (boolExpr) { $x } else { $y } [3] $result = ?: {boolExpr} {$x} {$y} <<pscx>>	[1] $result = switch (25 -gt 10) { $true { "yes" } $false { "no" } } [2] $result = if (25 -gt 10) { "yes" } else { "no" }	yes yes
Coalesce (evaluate 2nd block if 1st block is null) Define Coalesce-Args as: function Coalesce-Args { (@($args \| ?{$_}) + $null)[0] }; Set-Alias ?? Coalesce-Args	[1] Invoke-NullCoalescing block1 block2 <<pscx>> [2] Coalesce-Args block1 block2 <<code from http://bit.ly/KhMkwO>>	[1a] Invoke-NullCoalescing {$env:dummy} {"\usr\tmp"} [1b] ?? {$env:dummy} {"\usr\tmp"} [2a] Coalesce-Args $env:dummy "\usr\tmp" [2b] ?? $env:dummy "\usr\tmp"	\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	ls C:\Windows\temp | select name,length | Format-List
Format list of objects with all fields on one line	any | Format-Table	ls C:\Windows\temp | select name,length | Format-Table
Format list of objects in an interactive grid	any | Out-GridView	ls C:\Windows\temp | select name,length | Out-GridView
Format list as console graph	any | Out-ConsoleGraph -property quantity Property Name <<code from http://bit.ly/1jJN4tG>>	ls C:\Windows\temp | select name,length | Out-ConsoleGraph -property length
Format list as gridview graph	any | Out-ConsoleGraph -GridView -property quantity Property Name <<code from http://bit.ly/1jJN4tG>>	ls 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 [2] any | Format-Table -AutoSize | Out-String | Set-Content file	[1] ps | ft -auto |Out-File process.txt -enc ascii [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 …"; $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++) # primary code here 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 here)	[1] [accelerators]::get <<pscx>> [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 AddressFamily : InterNetwork . . . IsIPv4MappedToIPv6 : False IPAddressToString : 192.0.0.1
Create new DateTime object with constructor	New-Object -TypeName DateTime -ArgumentList constructor-argument-list	New-Object -TypeName DateTime -ArgumentList 2014,10,10	Friday, October 10, 2014 12:00:00 AM
Cast string to a DateTime object	[DateTime]"date-time value"	[DateTime]"10/10/2014" | select Year, DayOfWeek, Ticks	Year DayOfWeek Ticks ---- --------- ----- 2014 Friday 635484960000000000
Cast string to XML without accelerator	[System.Xml.XmlDocument] "XML text"	[System.Xml.XmlDocument] "<root>text</root>"	root ---- text
Cast string to XML with accelerator	[xml] "XML text"	[xml] "<root>text</root>"	root ---- text
Access static member of .NET class	[class]::member	$currentTime = [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!