PowerShell One-Liners: Variables, Parameters, Properties, and Objects

24 April 2014

by Michael Sorens

PowerShell isn't a conventional language, though it draws inspiration widely. Many people learn it, and use it, best by collecting snippets, or one-liners, and adapting them for use. Michael Sorens provides the second in 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 2

Part 1: Help, Syntax, Display and Files
Part 2: Variables, Parameters, Properties, and Objects,
Part 3: Collections, Hashtables, Arrays and Strings
Part 4: Accessing, Handling and Writing Data

This is part of a multi-part series of PowerShell reference charts. Here you will find details about 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 1 Be sure to review part 1 first, though, which 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. It also examines locations, files, and paths (the basic currency of a shell); key syntactic constructs; and ways to cast your output in list, table, grid, or chart form.

Part 2 this article.

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 fronm the link at the top of the article

Variables Here, There, and Everywhere

Because PowerShell is a shell language you can create complex and powerful operations on the command line. Because PowerShell is a programming language, you can also store that output into variables along the way. Thus, while item 1 demonstrates defining a variable with a simple value, you can use virtually any PowerShell expression for the indicated value. Part 2 will show further examples of variables containing collections and hash tables.

Action	Element	Example	Output
Define variable	[1] $name = value [2] value | Set-Variable –name name	[1] $a = 25; $a [2] 42 | sv a; $a	25 42
Define variable with auto-validation (see Validating Parameter Input at http://bit.ly/MtpW4K)	[constraint]$name = value	[1] [ValidateRange(1,10)][int]$x = 1; $x = 22 [2] [ValidateLength(1,25)][string]$s = ""	--error--
Variable uninterpreted within string	'... variable ...' (single quotes)	$a = 25; '$a not interpolated'	$a not interpolated
Scalar variable interpolated in string	"... variable ..." (double quotes)	$a = 25; "$a interpolated"	25 interpolated
Array variable interpolated in string	"... variable ..." (double quotes)	$arr = "aaa","bbb","x"; "arr is [$arr]"	arr is [aaa bbb x]
Array in string with non-default separator	$OFS='string'; "array-variable"	$arr = "aaa","bbb","x"; $OFS='/'; "arr is [$arr]"	arr is [aaa/bbb/x]
Complex syntactic element interpolated in string	$(…)	$myArray=@(1,2); "first element = $($myArray[0])"	first element = 1
Format output a la printf (see Composite Formatting http://bit.ly/1gawf5H)	formatString -f argumentList	[1] $href="http://foo.com"; $title = "title"; "{1}" -f $href, $title [2] @{a=5;b=25}.GetEnumerator() |%{" {0} => {1}" -f $_.key, $_.value} [3] "{0,-10} = {1,5}" -f "myName", 25	[1] title [2] a => 5`r`nb => 25 [3] myName = 25
Implicit function or loop variable	$PSItem or $_	ls | % { $_.name }	
Private scope (.NET equiv: private) Local scope (.NET equiv: current) Script scope (.NET equiv: internal) Global scope (.NET equiv: public) (Variable scoping in powershell)	$private:name $name **or** $local:name $script:name $global:name		
List all user variables and PowerShell variables	Get-ChildItem variable:	dir variable:	

List all environment variables	Get-ChildItem env:	ls env:	
List specific variables	Get-ChildItem env:wildcardExpr	ls env:HOME*	`HOMEPATH \Users\ms` `HOMEDRIVE C:`
Test if variable exists	Test-Path variable:name	If (!(Test-Path variable:ColorList)) { $ColorList = @() }	

Passing Parameters

Probably the most often-encountered issue with Powershell is not understanding how to pass parameters to a PowerShell cmdlet or function. I suspect most folks start out confused about why it does not work, advance to being sure it is a bug in PowerShell, then finally achieve enlightenment and acceptance of the way it really works. The fundamental rule of passing multiple parameters is simply this: use spaces not commas. The entries below illustrate all the scenarios you would likely need.

Action	Command	Example
Pass multiple parameters inline	cmdlet paramA paramB paramC (spaces—not commas!)	# compare this result with inserting a comma between 5 and 3 function func($a,$b) { "{0}/{1}" -f $a.length, $b.length }; func 5 3
Pass multiple parameters from array (uses splatting operator; see http://stackoverflow.com/a/17198115/115690)	$a = valueA, valueB, valueC; cmdlet @a	# compare this result with using $a instead of @a function func($a,$b) { "{0}/{1}" -f $a.length, $b.length }; $a = 5, 3; func @a
Pass an array of values as a single parameter inline	cmdlet valueA, valueB, valueC	dir prog.exe, prog.exe.config
Pass an array of values as a single parameter in an array	$a = valueA, valueB, valueC; cmdlet $a	$a = "prog.exe", "prog.exe.config"; dir $a
Pass an array of values as a single parameter in a pipe	valueA, valueB, valueC | cmdlet	"prog.exe", "prog.exe.config" | dir

Properties

Properties really take center-stage in PowerShell, perhaps even more so than variables. With PowerShell, you are passing around objects but what you are actually using are their properties. If you invoke, for example, Get-Process, you get a table where each row contains the properties of a returned process. Get-Process by default outputs 8 properties (Handles, Name, etc.). There are actually dozens more, though, and you could show whichever ones you like simply by piping Get-Process into Select-Object. In terse form you might write ps | select -prop Name, StartTime. The entries in this section provide a good grounding in the nature of properties: how to show some or all of them, how to see if one exists, how to add or remove them, and so forth. Possibly the most exciting: if you have worked extensively in .NET you have likely wanted some way to dump complex objects for examination—a non-trivial task requiring either writing your own dumper or using a library. With PowerShell—just one command (entry 22).

Action	Command	Example	Output
Test if property exists	Get-Member -InputObject object -Name propertyName	[1] "{0},{1}" -f [bool](gm -input (1..5) -name count), (1..5).count [2] [bool](gm -input (1..10) -name stuff)	`True,5` `False`
Filter output displaying default properties	any | Where-Object	ps | ? { $_.VM -gt 100MB }	
Filter output displaying selected properties	any | Where-Object | Select-Object	ps | ? { $_.VM -gt 100MB } | select name, vm	
Display default properties in default format	any	[1] Get-Process uses Format-Table [2] Get-WmiObject win32_diskdrive uses Format-List	

Display default properties (table)	any | Format-Table	ps | ? { $_.Name -match "^m" } | ft	
Display default properties (list)	any | Format-List	ps | ? { $_.Name -match "^m" } | fl	
Display default properties (grid)	any |Out-GridView	Get-PsDrive | Out-GridView	
Display all properties (table)	any | Format-Table -force *	ps | ft -force *	
Display all properties (list)	any | Format-List -force *	gwmi win32_diskdrive | fl -force *	
Display all properties (grid)	any | Select-Object *| Out-GridView	Get-PsDrive | Select * | Out-GridView	
Display selected properties (table)	any | Format-Table -Property wildcardExpr	ps | ? { $_. Name -match "^m" } | ft st*	
Display selected properties (list)	any | Format-List -Property wildcardExpr	ps | ? { $_. Name -match "^m" } | fl st*	
Display selected properties (list or table)	any | Select-Object -Property wildcardExpr	ps | ? { $_. Name -match "^m" } | select s* (list) ps | ? { $_. Name -match "^m" } | select start* (table)	
Display calculated property (see Using Calculated Properties)	any | Select-Object @{Name = name;Expression = scriptBlock}	ls . | select Name, @{n="Kbytes";e={ "{0:N0}" -f ($_.Length / 1Kb) }}	Name Kbytes ---- ---- file1.txt 1,088 file2.txt 269 . . .
Add one property to an object	[1] $obj | Add-Member -MemberType NoteProperty -Name name -Value value [2] $obj | Add-Member -NotePropertyName name -NotePropertyValue value	$a = New-Object PSObject; $a | Add-Member "foo" "bar"; $a	foo --- bar
Add multiple properties to a new object	$obj = New-Object PSObject -Property hashTable	$properties = @{name="abc"; size=12; entries=29.5 }; $a = New-Object PSObject -Property $properties; $a | ft -auto	entries name size ------- ---- ---- 29.5 abc 12
Add multiple properties to an existing object	$obj | Add-Member -NotePropertyMembers hashTable	$a | Add-Member -NotePropertyMembers @{ "x"=5;"y"=1}; $a	entries name size x y ------- ---- ---- - - 29.5 abc 12 5 1
Remove a property from one object (Remove a Member from a PowerShell Object?)	$obj.PSObject.Properties.Remove(propertyName)	$a.PSObject.Properties.Remove("name")	entries size ------- ---- 29.5 12
Remove property from a collection of objects (Remove a Member from a PowerShell Object?)	any | Select-Object -Property * -ExcludeProperty propertyName	ls | select -property * -exclude Mode	

List property names of an object type	any | Get-Member -MemberType Property	($PWD | gm -Member Property).Name	Drive Path Provider ProviderPath
List property names with their associated values (really shallow)	[1] any | Format-List [2] any | Select-Object -Property *	[1] $PWD | fl [2] $PWD | select *	Drive : C Provider : Core\FileSystem ProviderPath : C:\usr Path : C:\usr
List property names with their associated values (adjustable shallow to deep)	any | ConvertTo-Json -Depth depth	$PWD | ConvertTo-Json -Depth 1	

(This last snippet, no. 22, is just one-level deeper than the “really shallow” approach in the previous entry (21). But wherever you see type names, there is still room for further expansion—just increase the depth value. Note that the list will get very big very fast—even a depth of 3 is quite voluminous!)

Objects, Types and Casts

This section provides some insights into .NET objects in PowerShell: seeing what type something is or testing if an object is a certain type; accessing .NET enumeration values; casting objects to different types; cloning objects.

Action	Command	Example	Output
Get size of collection	[1] @(any).Count [2] any | Measure-Object	[1] @(Get-Process).Count [2] (Get-Process | Measure-Object).Count	
Get type of non-collection or object array for collection (i.e. does not report base type of array)	object.GetType().FullName	[1] "abc".GetType().FullName [2] (1,2,3).GetType().FullName [3] ("a", "b", "c").GetType().FullName	System.String System.Object[] System.Object[]
Get type of any object or base type of array	object | Get-Member | Select -First 1 |%{$_.TypeName}	[1] 1,2,3 | gm | select -First 1| % { $_.TypeName } [2] 1 | gm | select -First 1| % { $_.TypeName }	System.Int32 System.Int32
Get base type of non-empty array	array[0].GetType().FullName	$myArray[0].GetType().FullName	
Get object hierarchy	object.PsTypeNames	(gci | select -First 1). PsTypeNames	System.IO.DirectoryInfo System.IO.FileSystemInfo System.MarshalByRefObject System.Object
Test type	if (object -is type) . . .	"hello" -is [string]	True
Access .NET enumeration type	[typeName]::enumValue	"/A/B/C//D/E//F/G" .Split("/", [System.StringSplitOptions]::-RemoveEmptyEntries)	
Combine bitwise .NET enum type values	[typeName]::enumValue -bor [typeName]::enumValue	[System.Text.RegularExpressions.-RegexOptions]::Singleline -bor [System.-Text.RegularExpressions.RegexOptions]::-ExplicitCapture	
Cast string to integer (about_Type_Operators: http://bit.ly/1a1aMyp)	string -as [int]	[1] "foo" -as [int] [2] "35.2" -as [int] [3] "0.0" -as [int]	35 0
Test cast string to integer	[bool]($var -as [int] -is [int])	[1] "foo" -as [int] -is [int] [2] "35.2" -as [int] -is [int]	False True
Convert ASCII code to character	[char]integer	[1] [char]48 [2] [char]0x42	0 B

Convert character to ASCII code	[byte][char]character	[byte][char] "A"	65
Convert integer to hexadecimal	"0x{0:x}" -f integer	"0x{0:x}" -f 64	0x40
Convert hexadecimal to integer	hex-value	0x40	64
Test if command exists	Get-Command command -errorAction SilentlyContinue	[1] [bool](gcm Get-ChildItem -ea SilentlyContinue) [2] [bool](gcm Get-MyStuff -ea SilentlyContinue)	True False
Clone object ('How to create new clone instance of PSObject object')	$newObj = $oldObj | Select-Object *		
Clone object except for specific property	$newObj = $oldObj | Select-Object * -except property		
Identify type of each returned object (Example: Get-ChildItem may return DirectoryInfo or FileInfo objects)	any | Select-Object id-field, type-expression	[1] Get-ChildItem | select name, @{n='type';e={$_.GetType().Name}} [2] Get-Alias \| select name, @{n='type';e={$_.ReferencedCommand.GetType().Name}}	

Encapsulation Does a Program Good

Because PowerShell is not just a shell but also a rich scripting language, it supports encapsulation at multiple levels. Scripts provide simple physical separation for your code while modules provide both physical and logical separation. That is, modules let you separate context or scope, so they are well worth the additional effort to set up. In a related vein, it is helpful to be cognizant of command precedence: alias, function, cmdlet, script, application. So, if there is a function and cmdlet of the same name, for example, then the function will be executed when you invoke that name because of precedence rules.

Action	Command	Example
Check permissions for running scripts	Get-ExecutionPolicy	same
Set permissions for running scripts	Set-ExecutionPolicy policy	Set-ExecutionPolicy RemoteSigned
Run a script in current context (dot-sourcing)	. path\script.ps1	echo '$foo = "hello now" ' > tmp\trial.ps1 $foo # empty . tmp\trial # dot-source the file $foo # now contains 'hello now'
Run a script in child context (note that the ampersand is required only if the path or name contains spaces)	& path\script.ps1	echo '$foo = "hello now" ' > tmp\trial.ps1 $foo # empty tmp\trial # execute script $foo # still empty!
Get directory of currently running script (i.e. use this inside a script to know its own path)	$PSScriptRoot (use Split-Path $script:MyInvocation.MyCommand.Path for v2)	$scriptDir = Split-Path $script:MyInvocation.MyCommand.Path $scriptDir = $PSScriptRoot
Load module x.psm1 from standard location	Import-Module module	Import-Module foo
Load module x.psm1 from arbitrary location	Import-Module path\module	Import-Module \usr\ps\mymodules\foo
List cmdlets added by a loaded module	Get-Command -Module module	gcm -Module sqlps

List loaded modules	Get-Module	same
List modules available to load	Get-Module –ListAvailable	same
See what other details to glean about modules	Get-Module | Get-Member	gmo | gm -type property
List modules with custom-specified properties	gmo | Format-Table -p property, property, …	gmo | ft -p name, moduletype, author, version -auto
List contents of a public function	[1] Get-Content function:name [2] (Get-ChildItem function:name).Definition [3] (Get-Command name).ScriptBlock	[1] gc function:Get-Verb [2] (gci function:Get-Verb).definition [3] (gcm Get-Verb).ScriptBlock
List contents of a private function	& (Get-Module module) { Get-Content function:name }	# create a Test module with a function foobar, import it, then run: & (gmo Test) { Get-Content function:foobar }
Determine source of duplicate names (e.g. cmdlet and function imported with the same name)	Get-Command name | select CommandType, Name, ModuleName	# The PS Community extensions has another version of Get-Help: Import-Module pscx; gcm get-help | select CommandType, name, modulename
Trace parameter assignment in cmdlet	Trace-Command -psHost -Name ParameterBinding { expression }	Trace-Command -psHost -Name ParameterBinding { "abc", "Abc" | select -unique }
Trace parameter assignment in own functions	Write-Host $PSBoundParameters	function foo($a, $b) { Write-Host $PSBoundParameters }; foo "one" "two"
Support wildcards passed as parameters (see How to pass a list of files as parameters to a powershell script)	Param ([String[]]$files) $IsWP = [System.Management.Automation.WildcardPattern]::ContainsWildcardCharacters($files) If ($IsWP) { $files = Get-ChildItem $files | % { $_.Name } }	

The Meta-Verse: Profile, History, Version, Prompt

Here you can see how to check what version of PowerShell you are running (even switch to an earlier version if needed); select and run previously used commands by number or by substring; examine any of your numerous profiles (scripts run on PowerShell startup); and change your command prompt.

Action	Command	Example
Display PowerShell version	$PsVersionTable.PSVersion (more reliable than $Host.Version – see How to determine what version of PowerShell is installed?)	$PsVersionTable.PSVersion
Display version and other info of one exe or dll (see get file version in powershell)	(Get-Command path).FileVersionInfo (Get-Item path).VersionInfo | Format-List	
Display version and other info of multiple executables	paths | Get-Command $_.FullName | Select -expand FileVersionInfo	dir *.dll,*.exe | %{gcm $_.FullName} | select -expand File*
Run an earlier version of PowerShell	powershell -Version 2	same
Get complete command history	Get-History	same
Set maximum remembered commands	$MaximumHistoryCount = integer	$MaximumHistoryCount = 1000
Get last n commands from history	Get-History -count n	ghy -Count 25
Get last n commands from history containing substring	Get-History | Select-String string | Select -last n	h | sls child | Select -last 25

Run command from history by command number	Invoke-History integer	r 23
Run command from history by command substring	#commandSubstring	#child (assuming you recently ran e.g. Get-ChildItem); press <tab> to cycle through list of other "child" choices.
View path to profile for [current user, current host]	$PROFILE	same
View path to all profiles (see http://bit.ly/JfgXwO)	$PROFILE | Format-List * -Force	same
Test whether profile exists for [current user, current host]	Test-Path $PROFILE	same
Test whether particular profile exists	Test-Path $PROFILE.profile	Test-Path $PROFILE.CurrentUserCurrentHost
Change your prompt (see http://bit.ly/18LS8Kf)	Define prompt function in your profile	function prompt { . . . }

Running Other Programs

As a shell language supplanting DOS, you will likely want to execute other programs, just like Windows batch files. It is fairly straightforward but the entries here show you a few nuances that you should be aware of. You can also see how to review execution status, see how long something takes to execute, and even limit how much time something may execute.

Action	Command	Example
Execute a program in a separate process	Start-Process program (NB: If using ISE you must use this to execute a program if the program reads from the console - see Why does PowerShell ISE hang on this C# program?)	start tmp\demo.exe
Open Windows Explorer at current directory	[1] Start-Process . [2] explorer .	same
Execute a program in the same process	program	[1] C:\usr\progs\demo.exe [2].\demo.exe
Execute a program with spaces in the name	& "program"	[1] & "C:\Program Files\demo.exe" [2] & "tmp\demo with spaces.exe" [3] & ".\demo in current dir.exe"
Time a command	Measure-Command scriptBlock	Measure-Command { Get-Content stuff.txt }
Time-limit a command (see adding a timeout to batch/powershell)	$j = Start-Job -ScriptBlock { ... } if (Wait-Job $j -Timeout $seconds) { Receive-Job $j } Remove-Job -force $j	
Execution status of last operation—use only for PowerShell commands (see Powershell $LastExitCode=0 but $?=False . Redirecting stderr to stdout gives NativeCommandError)	$?	
Execution status of last external command	$LastExitCode	
Discard output (i.e. run commands for side effects) (see http://bit.ly/1cjmWSk)	[1] any > $null [2] $null = any [3] any | Out-Null [4] [void] (any)	

Parsing and Grouping

While this heading might sound abstruse or obscure and you may be tempted to skip it—don't! Understanding the fundamentals of grouping and command vs. expression parsing in PowerShell can make working in PowerShell much more fruitful. The entries here present a condensed reference—take a look at Keith Hill's Understanding PowerShell Parsing Modes for more details.

Action	Element	Example	Output
Grouping expression (single expression or pipeline)	(command)	(dir C:\).length (dir C:\).name (ps | select -First 1).GetType().Name (ps | select -First 5).GetType().Name ("foo"; "bar"; "baz").GetType().Name ("abc").length ("foo", "bar").length	scalar: a single number array: list of file names Process Object[] --error-- 3 2
Subexpression Multiple statements allowed; single result returns scalar; multiple results return array.	$(command-sequence)	$(ls c:\; ls c:\windows).length $(ps | select -First 1).GetType().Name $(ps | select -First 5).GetType().Name $("foo"; "bar"; "baz").GetType().Name $("abc").length $("foo", "bar").length	scalar: a single number Process Object[] Object[] 3 2
Array subexpression Multiple statements allowed; guarantees array result.	@(command-sequence)	@(ls c:\; ls c:\windows).length @(ps | select -First 1).GetType().Name @(ps | select -First 5).GetType().Name @("foo"; "bar"; "baz").GetType().Name @("abc").length @("foo", "bar").length	scalar: a single number Object[] Object[] Object[] 1 2
Command parsing	Begin with alpha, _, &, ., or \	dir file1.txt	
Expression parsing	Begin with any character other than above	"dir file1.txt"	
Parsing determination: start of command and at start of any subexpression		Write-Host Get-ChildItem vs. Write-Host (Get-ChildItem)	
Run custom cmdlet from batch	powershell -command "import-module M1; cmdlet1"		
Invoke dynamic code	Invoke-Expression string	iex "write-host hello"	hello

Conclusion

That's it for part 2; 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!

Author profile: Michael Sorens

Michael Sorens is passionate about software to be more productive, evidenced by his open source libraries in several languages (see his API bookshelf) as well as SqlDiffFramework (a DB comparison tool for heterogeneous systems including SQL Server, Oracle, and MySql). With degrees in computer science and engineering he has worked the gamut of companies from Fortune 500 firms to Silicon

Valley startups over the last 25 years or so. Current passions include PowerShell, .NET, SQL, and XML technologies (see his full brand page). Spreading the seeds of good design wherever possible, he enjoys sharing knowledge via writing (see his full list of articles), teaching, and StackOverflow. Like what you have read? Connect with Michael on LinkedIn and Google +

Search for other articles by Michael Sorens

Rate this article: Avg rating: ★★★★★ from a total of 16 votes.

Poor OK Good Great Must read SUBMIT

Have Your Say

Do you have an opinion on this article? Then add your comment below:

You must be logged in to post to this forum

Click here to log in.

