PowerShell One-Liners: Collections, Hashtables, Arrays and Strings

Michael Sorens
13 May 2014

> The way to learn PowerShell is to browse and nibble, rather than to sit down to a formal five-course meal. In his continuing series on Powershell one-liners, Michael Sorens provides Fast Food for busy professionals who want results quickly and aren't too faddy. Part 3 has, as its tasty confections, collections, hashtables, arrays and strings.

This series is in four parts: This is part 3

- **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 a multi-part series of PowerShell reference charts. Here you will details of 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)

Be sure to review parts 1 and 2, though, which begin 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. They also cover locations, files, and paths (the basic currency of a shell); key syntactic constructs; ways to cast your output in list, table, grid, or chart form; and key PowerShell concepts of variables, parameters, properties, and objects.

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.

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.

Each part of this series is available as an online reference here at Simple-Talk.com, a wide version here as well as a downloadable wallchart 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!

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/).

Contents

Collections (Arrays)

Collections are everywhere in PowerShell; they are the most prevalent of all its data structures. Cmdlets and pipes let you pass around objects, but keep in mind that they usually pass around objects (plural), not just an object (singular). So it is important to have a good sense about what you can do with collections. Most of the collections you will encounter, therefore, are generated by some cmdlet. But occasionally you need to create your own, so the first few entries here show you how to do that. This section also presents crucial entries for iterating through collections and comparing collections.

#	Action	Command	Example	Output
1	Initialize literal array with at least 2 elements	[1] @(value, value, value, …) [2] value, value, value, …	[1] $myArray = @(“a”,”b”,”c”,”d”,”e”,”f”,”g”,”h”) [2] $myArray = “a”,”b”,”c”,”d”,”e”,”f”,”g”,”h”	

2	Initialize literal array with one element	[1] @(value) [2] , value	[1] $myArray = @(25) [2] $myArray = ,25	
3	Initialize a strongly-typed array	[typeName[]] $name = values	[int[]] $a = 1,2,3,4	

4	Iterate array/collection by pipeline	$array | ForEach-Object { … $_ … }	1,2,3 | % { “item $_” }	`item 1`
				
				`item 2`
				
				`item 3`

5	Iterate array/collection by non-pipeline	foreach ($var in $array) { commands }	foreach ($item in “a”,”b”) { $item }	`a`
				
				`b`

6	Iterate collection with initialization/finalization	$array | % { beginBlock } { commands } { endBlock }	Return just odd-numbered elements:
			
			‘v1’,’v2’,’v3’,’v4’| foreach {$i=1} { if ($i++ % 2) {$_} } {“done”}
				`v1`
				
				`v3`
				
				`done`

7	Ensure value is an array	[1] @(any) [2] ,any	[1] $a = @(Get-Service | select -first 1) ; $a.length [2] $a = ,(Get-Service | select -first 1) ; $a.length	`1` `1`
8	Fill array with the same value efficiently (see How to fill an array efficiently in PowerShell)		In order from most to least efficient: [1] $a = ,2 * $length [2] [int[]]$a = [System.Linq.Enumerable]::Repeat(2, $length) [3] $a = foreach ($i in 1..$length) { 2 } [4] [int[]]$a = -split “2 ” * $length [5] $a = for ($i = 0; $i -lt $length; $i++) { 2 }	

			[6] $a = 1..$length | %{ 2 } [7] $a = @(); for ($i = 0; $i -lt $length; $i++) { $a += 2 }	

9	Compare arrays (independent of order) returning differences	Compare-Object object1 object2	compare (1..5) (4..1)	`InputObject SideIndicator`
				
				`——— ———-`
				
				`5 <=`
				
				`4 =>`

10	Compare arrays where order is significant	Compare-Object object1 object2 -Sync 0	diff (1..3) (3..1) -Sync 0	`InputObject SideIndicator`
				
				`——— ———-`
				
				`3 =>`
				
				`1 <=`
				
				`1 =>`
				
				`3 <=`

11	Compare arrays returning single Boolean indicating a match or not	@(compare object1 object2).length -eq 0	[1] @(compare (1..5) (5..1)).length -eq 0
			
			[2] @(compare (1..5) (4..1)).length -eq 0
				`True`
				
				`False`

Collection Selection

After iteration, selecting is probably the most common thing to do with a collection. Entries here show how to select one or more elements, contiguous or not, as well as equivalents to the common take and skip operations common to many collection structures. (Note that the output column has been condensed by removing line breaks; the true output will actually show each element on a separate line, except as indicated.)

#	Action	Command	Example	Output
1	Select single element by index	$array[index]	(1,2,3,4,5)[0]	`1`
2	Select multiple specific elements	any | Select-Object -Index m,n	1..10 | select -index 0,4,9	`1 5 10`

3	Select contiguous elements via array notation	$array[m..n]	(1..10)[1..4]	`2 3 4 5`
4	Select contiguous elements	any | Select-Object -Index (m..n)	1..10 | select -index (1..4)	`2 3 4 5`
5	Select first n elements (head)	any | Select-Object -First *n*	$n = 2; 1,2,3,4,5 | select -first $n	`1 2`
6	Select last n elements (tail)	any | Select-Object -Last *n*	$n = 4; 1,2,3,4,5 | select -last $n	`2 3 4 5`
7	Select n elements after skipping m elements	any | Select-Object -First n -Skip m	1..10 | select -skip 3 -first 4	`4 5 6 7`
8	Select all elements except the first n	any | Select-Object -Skip *n*	$n = 2; 1,2,3,4,5 | select -skip $n	`3 4 5`
9	Select all elements except the last n	[1] any | Select-Object -Skip n -Last LargeInt [2] $txt = any; $txt[0..($txt.length-n-1)] [3] any | Skip-Object -Last n <<pscx>>	[1] $n = 2; 1..5 | Select-Object -skip $n -last 10000000 [2] $n = 2; $txt = 1..5; $txt[0..($txt.length-$n-1)] [3] $n = 2; 1..5 | Skip-Object -last $n	`1 2 3`
10	Display all elements on one line	“array-expression”	$a = 3,5,7; “$a”	`3 5 7 #really on one line!`

Collection Union, Intersection, Uniqueness

Entries in this section let you do more complex operations on collections. Note that simple concatenation propagates duplicates whereas union and intersection are strict set operations: they do not include duplicate values. Entries here also show how to obtain just the unique elements in a collection as well as adding to collections.

#	Action	Command	Example	Output

1	Concatenate two collections	$array1 + $array2	@(“apple”,”pear”) + @(“apple”,”orange”)	`apple`
				
				`pear`
				
				`apple`

				`orange`

2	Set union	$array1 + $array2 | select -Unique	@(“apple”,”pear”) + @(“apple”,”orange”) | select -uniq	`apple`
				`pear`
				`orange`

3	Set intersection	$array1 | select -Unique | where { $array2 -contains $_ }	@(1,2,5,9) | select -uniq | ? { @(2,4,9,16) -contains $_ }	`2` `9`
4	Set difference (In PowerShell how can I check if all items from one array exist in a second array?)	$array1 | select -Unique | where { $array2 -notcontains $_ }	1,2,3,4,2,3 | select -uniq |? { 1,3,4,5 -notcontains $_ }	`2`

5	Get unique elements, case-sensitive, sorted	any-sorted | Get-Unique	“abc”, “abc” , “Abc”, “def” | Get-Unique	`abc`
				`Abc`
				`def`

6	Get unique elements, case-sensitive, unsorted	[1] any | Sort-Object -CaseSensitive| Get-Unique
		[2] any | Select-Object -Unique
			[1] “abc”, “Abc”, “def”, “abc” | sort -case | Get-Unique
			[2] “abc”, “Abc”, “def”, “abc” | select -unique
				`abc`
				`Abc`
				`def`

7	Get unique elements, case-insensitive	any | Sort-Object -Unique	“abc”, “Abc”, “def”, “abc” | sort -unique	`abc` `def`
8	Add an element to an array	array += element	$a = 1,2,3; $a += 4; $a	`1 2 3 4`
9	Add an element to multiple arrays (see How to append elements to multiple arrays on the same line?)	arrays = arrays |% {,($_ += element)}	$a = @(“a”,”b”); $b = @(1,2); $a,$b = $a,$b |% {,($_ += ‘foo’)}; “$a — $b”	`a b foo —` `1 2 foo`

Collection Ordering

Once you have a collection chances are you might want to re-order it per the needs of your application. You can do this with derived properties almost as easily as with simple named properties. The last few entries show how to apply sorting to file contents as well.

#	Action	Command	Example	Output
1	Sort collection of strings	any | Sort-Object	“ab12”, “ab1”, “ab103” | sort	ab1 ab103 ab12
2	Sort collection of strings by derived property	any | Sort-Object -property propertyExpression	“ab12”, “ab1”, “ab103” | sort { [int]($_ -replace ‘\D’) }	ab1 ab12 ab103
3	Sort collection of objects by named property	any | Sort-Object -property propertyName	ls \windows\system32\dwm*.dll | sort -property length | select name, length | ft -auto	Name Length —- —— dwmapi.dll 115200 dwmredir.dll 172544 dwmcore.dll 2219520
4	Sort file of specified data type	Get-Content filespec |Sort-Object { [type]$_ }	gc numbers.txt | sort { [int]$_ }	
5	Sort whitespace-delimited text file by first field	Get-Content filespec |Sort-Object { [type](-split $_)[0] }	gc lines.txt | sort { [double](-split $_)[0] }	
6	Sort whitespace-delimited text file by last field	Get-Content filespec |Sort-Object { [type](-split $_)[-1] }	gc lines.txt | sort { [int](-split $_)[-1] }	

Collections and LINQ

If you are used to relying on LINQ-to-Object operators in C# so much that you may are almost compelled to reject PowerShell out of hand, fear not! PowerShell provides an assortment of basic LINQ-equivalent operations out-of-the-box, as detailed in the entries below. Many of them you have already seen if you have read the above sections on collections. Note that the one key thing you do not get with these standard PowerShell operators, though, is lazy evaluation. If you are keen on that, I refer you to Bart DeSmet's LINQ Through PowerShell (http://bit.ly/1j9Y7cS).

#	Action	LINQ Method	PowerShell Cmdlet	Example
1	Projection	Select	Select-Object	Get-Process | Select-Object -Property Name, WorkingSet, StartTime
2	Restriction	Where	Where-Object	Get-ChildItem | Where-Object { $PSItem.Length -gt 1000 }
3	Ordering	OrderBy	Sort-Object	Get-ChildItem | Sort-Object -Property length -Descending
4	Grouping	GroupBy	Group-Object	Get-Service | Group-Object Status
5	Set Operation	Distinct	[1] Get-Unique [2] Sort-Object -Unique [3] Select-Object -Unique	[1] "abc", "def", "abc" | Sort-Object | Get-Unique [2] "abc", "def", "abc" | Sort-Object -unique [3] Get-ChildItem *.cs -r | Select-String "public.*void" | Select-Object -uniq Path
6	Partitioning	Take	Select-Object -First	Get-Process | Select-Object -First 5
7	Partitioning	Skip	Select-Object -Skip	Get-Process | Select-Object -Skip 5
8	Quantifiers	Any	[1] See PowerShell equivalent of LINQ Any()? (JaredPar's solution) [2] See PowerShell equivalent of LINQ Any()? (Paolo Tedesco's solution)	[1] function Test-Any() { begin { $any = $false } process { $any = $true } end { $any } } 1..4 |Where { $_ -gt 5 } \| Test-Any [2] function Test-Any { [CmdletBinding()] param($EvaluateCondition, [Parameter(ValueFromPipeline = $true)] $ObjectToTest) begin { $any = $false } process { if (-not $any -and (& $EvaluateCondition $ObjectToTest)) { $any = $true } } end { $any } }

			1..4 | Test-Any { $_ -gt 5 }
9	Quantifiers	All	function Test-All { [CmdletBinding()] param($EvaluateCondition, [Parameter(ValueFromPipeline = $true)] $ObjectToTest) begin { $all = $true } process { if (!(& $EvaluateCondition $ObjectToTest)) { $all = $false } } end { $all } } 1..4 | Test-Any { $_ -gt 0 }

Hash Tables (Dictionaries)

Hash tables are the other ubiquitous data structure that you will encounter as well as generate yourself. As they are more involved than a simple collection, there are more varied ways to create one. This section provides a synopsis of common techniques for generating hash tables. The next section shows you how to access its members. Hash table values are not strongly typed, as you can see in the first entry, which mixes strings and integers. You can use a standard .NET dictionary, though, for strong typing.

#	Action	Command	Example	Output
1	Initialize literal hash table	@{ label = value; … }	@{ “i1"=”bird” “i2"=256 “i3"=”cat” }	Name Value —- —- i2 256 i3 cat i1 bird
2	Initialize literal hash table (minimal punctuation)		@” i1=bird	Name Value

			i2=256 i3=cat “@ | ConvertFrom-StringData	-- -- i3 cat i2 256 i1 bird
3	Initialize hash table from CSV with header row	Import-Csv $file | foreach { $hash = @{} } { $hash[$_.key] = $_.value}	Assumes headers “first,second” Import-Csv $file | % { $hash = @{} } { $hash[$_.first] = $_.second}	
4	Initialize hash table from CSV without header row (see Convert a 2 columns CSV into a hash table)	(any -replace ‘,’, ‘=’) -join “`n” | ConvertFrom-StringData	$hash = ((Get-Content text.csv) -replace ‘,’, ‘=’) -join “`n” | ConvertFrom-StringData	
5	Initialize hash table from a file where a simple separator is insufficient; specify a regex with two subgroups picking out the key and the value. (How to construct hash table from file using powershell?)	$hash = @{}; Get-Content $file \| foreach { if ($_ -match $regex) { $hash[$matches[1]] = $matches[2] } }	The example matches input lines like “<i1>=<bird>” selecting “i1” as the key and “bird” as the value. $hash = @{} Get-Content $file \| % { if ($_ -match ‘^<(.*)>=<(.*)>’) { $hash[$matches[1]]=$matches[2] } }	
6	Initialize data structure from PS code literal (little need to ever do this; it is just to illustrate what the following entries do from a file)	any | Out-String | Invoke-Expression	“@{ X = ‘x’; Y = ‘y’ }” | Out-String | iex	Name Value -- -- Y y

				X x
7	Initialize hash table from PS code file	variable = filespec.ps1	Assume file contains e.g. @{ X = ‘x’; Y = ‘y’ } $a = .\data.ps1	same as above
8	Initialize hash table from text file	Get-Content filespec | Out-String | Invoke-Expression	Assume file contains e.g. @{ X = ‘x’; Y = ‘y’ } $a = gc .\data.txt | Out-String | iex	same as above
9	Initialize hash of hash tables from INI file	$hash = Get-IniFile file <<code from Get-IniFile>>	“[Install]`nA=640`nB=0x403f`n[Extras]`nOpt=10`nValue=0” | Set-Content test.ini; $ini = Get-IniFile .\test.ini [1] $ini[“Install”][“A”] [2] $ini.Install.B [3] $ini.Extras	[1] 640 [2] 0x403f [3] Name Value —- —- Value 0 Opt 10
10	Initialize a strongly-typed hash	$dict = New-Object ‘System.ÂCollections.Â-Generic.DictionaryÂ-[type,type]’	$dict = New-Object ‘System.ÂCollections.ÂGeneric.Â-DictionaryÂ[string,int]’ $dict.Fred = 25 $dict.Mary = “abc” # runtime error	

Hash Table Access and Iteration

Once you have a hash, there are two things you might want to do with it: do something with a single element or do something with every element. As the first line item shows, there are three different syntaxes possible to access a single element. (Most entries in this section refer to the same simple hash setup in the previous section.)

#	Action	Command	Example	Output
1	Access hash element by key value	[1] $hash[$key] [2] $hash.key [3] $hash.Item($key)	[1] $myHash[“i2”] [2] $myHash.i3 [3] $myHash.Item(“i1”)	256 cat bird
2	Iterate through hash with enumerator	$hash.GetEnumerator() | foreach { … $_.Key … $_.Value …}	$myHash.GetEnumerator() | % { “key={0}, value={1}” -f $_.key, $_.value }	key=i2, value=256 key=i3, value=cat key=i1, value=bird
3	Iterate through hash with keys	$hash.Keys | foreach { … $_ … $hash[$_] … }	[1] $myHash.Keys | % {“k={0}, v={1}” -f $_,$myHash.Item($_) } [2] $myHash.Keys | % {“k={0}, v={1}” -f $_,$myHash[$_] }	k=i2, v=256 k=i3, v=cat k=i1, v=bird
4	Reverse a hash	$hash.Keys \| foreach {$Rhash=@{}} { $Rhash[$hash[$_]] = $_ }	$h = @{ “i1"=”bird”; “i2"=256; “i3"=”cat” }; $h.Keys | % { $Rhash=@{} } { $Rhash[$h[$_]] = $_ } { $Rhash }	Name Value —- —- bird i1 cat i3 256 i2
5	Modify entries with a given value	@($table.GetEnumerator()) \| where {$_.Value -eq oldValue} | foreach { $table[$_.Key] = newValue }	$table = @{ “A1"=3; “A2"=3; “A3"=6; “A4"=12; }; @($table.GetEnumerator()) | ? {$_.Value -eq 3} | % { $table[$_.Key]=4 }	Name Value —- —- A1 4 A2 4 A4 12

```
A3    6
```

Strings to Arrays: Splitting

This section provides an assortment of techniques going in one direction, i.e. splitting up strings into arrays. Here you can see examples of how to split on whitespace, line breaks, simple delimiters, and regular expressions. The next section illustrates how to go back the other direction.

#	Action	Command	Example	Output
1	Split string on whitespace	-split string	# Note that `t = tab and `n = newline: -split “one two`tthree`nfour”	one two three four
2	Split string on simple delimiter (escape any regex metachars with backslash)	string -split delimiter	[1] “one,two,three” -split “,” [2] “one#-#two#-#three” -split “#-#”	one two three
3	Split string on regular expression	[1] [regex]::split(string, regex) [2] string -split regex	[1] [regex]::split(“123#456#apple”, “#(?!\d)”) [2] “123#456#apple” -split “#(?!\d)”	123#456 apple
4	Split string on regular expression with options	[regex]::split(string, regex, options)	[1] [regex]::split(“Apple_aPPle_APple”, “ppl”, “IgnoreCase”) [2] [regex]::split(“Apple_aPPle_APple”, “ppl”, [System.Text.RegularExpressions.Â-RegexOptions]::ÂIgnoreCase)	A e_a e_A e
5	Split string on complex single-char expression (see about_split: about_Split)	string -split scriptBlock	# $_ matches any single character: “Brobdingnag” -split {$_ -eq “n” -or $_ -eq “o”}	Br bdi g ag

6	Split string on complex single-char expression using external criterion	string -split scriptBlock	$i = 5; “a,b#c!d” -split { if ($i -gt 3) {$_ -eq “,”} else {$_ -eq “#”} }	a b#c!d
7	Split pipeline data on Windows line breaks (Out-String uses Environment.NewLine)	string -split “`r`n”	(Get-Content test.txt | Out-String) -split “`r`n”	
8	Split by line, retaining whitespace (here strings use just newline character)	hereString -split “`n”	$data = @” one two three`t`t`t  “@  $b = $data -split “`n”; “<$($b[2])>”	<three >
9	Split by line, trimming whitespace	hereString -split “`n” | % { $_.Trim() }	$b = ($data -split “`n”).Trim(); “<$($b[2])>”	<three>
10	Split by line, retaining empty entries	hereString -split “`n”	“one`n`ntwo`nthree” -split”`n”	one two three
11	Split by line, skipping empty entries	hereString.Split(“`n”, [System.Â-StringSplitOptions]::Â-RemoveEmptyEntries)	“one`n`ntwo`nthree”.Split(“`n”, [System.StringSplitOptions]::Â-RemoveEmptyEntries)	one two three

Arrays to Strings: Joining

Going back the other way-joining array elements together into a string-is simpler than splitting, of course, so this section offers fewer variations than last section, which illustrated how to split up a string.

#	Action	Command	Example	Output
1	Join strings with no delimiter	-join array	[1] -join (“abc”,”def”) [2] $a = “abc”,”def”; -join $a	abcdef
2	Join string array with default delimiter ($OFS) about_Preference_Variables)	“array”	$a = “abc”,”def”; “$a”	abc def
3	Join string array using Windows line breaks	array | Out-String	“abc”,”def”,”ghi”| Out-String	abc`r`ndef`r`nghi
4	Join strings with specified delimiter	[1] $OFS = delimiter; “array” [2] array -join delimiter [3] [string]::join(delimiter, array)	[1] $OFS = “##”; “$(‘abc’, ‘def’, ‘ghi’)” [2] “abc”,”def”,”ghi” -join “##” [3] [string]::join(“##”, “abc”,”def”,”ghi”)	abc##def##ghi

String Search

How can any developer survive without grep? Just take a look at Select-String to find out. It has essentially all the bells and whistles that grep has. You can display context before and after a match (-Context). You can print with or without filenames, line numbers, and other properties (by piping into Select-Object and selecting appropriate properties). You can just print matched files, too, without the matched text; you will find that illustrated in the File Search section next. Here are just a variety of starter recipes to get you thinking about how to fine-tune your searches. Also take a look at Select-StringAligned (available from http://bit.ly/1nlzgrU) that lets you align your matches when you are displaying file names with them instead of having the matches after the ragged right edge of the file names; this reveals patterns in some searches in a startling fashion.

#	Action	Command	Example	Output
1	Replace string in selected files recursively (see PowerShell: Recursively Replace String in Select Sub-files of a Directory)	foreach ($f in gci -r -include pattern) { (gc $f.fullname) | % { $_ -replace regex, replacement } | sc $f.fullname }		

2	Select first occurrence of a pattern on each line	any | Select-String “(pattern)” | foreach { $_.Matches[0].Groups[1].Value }	$a = “abc def”,”foobar”,”12345-,-678";
			$a | sls “([a-z]+)” | % { $_.Matches[0].Groups[1].Value }
				`abc`
				
				`foobar`

3	Select from each line the text after a given pattern (PowerShell – split string & output all text to the right)	any | Select-String ‘(?<=pattern)(.*)’ | select -expa matches | select -expa value | foreach { $_.trim() }	$a = “abc-,-def”,”12345-,-678"; $a | sls ‘(?<=-,-)(.*)’ | select -expa matches | select -expa value | % { $_.trim() }	`def`
				
				`678`

4	Select from each line the text before a given pattern	any | Select-String ‘(.*)(?=pattern)’ | select -expa matches | select -expa value | foreach { $_.trim() }	$a | Select-String ‘(.*)(?=-,-)’ | select -expa matches | select -expa value | % { $_.trim() }	`abc`
				
				`12345`

5	Select from each line text by position (column)	any | % { $_.substring(int,int) }	$a | % { $_.substring(2,3) }	`c-,`
				
				`345`

6	Select from each line one column from CSV	[1] Import-Csv file| select -ExpandProperty name
		
		[2] any | ConvertFrom-Csv -Header nameList |
		select -ExpandProperty name
			$a | ConvertFrom-Csv -Header “V1”, “V2” |
			select -expa V1
				`abc-`
				
				`12345-`

7	Select from each line multiple columns from CSV	[1] Import-Csv file | foreach { formatString -f $_.name1, $_.name2, … }
		
		[2] any | ConvertFrom-Csv -Header names | foreach { formatString -f $_.name1, $_.name2, … }
			$a | ConvertFrom-Csv -Header “V1”, “V2” | % { “{0} / {1}” -f $_.V1, $_.V2 }	`abc- / -def`
				
				`12345- / -678`

8	Select from each line one column from delimited file, no headers	Import-Csv file -Header field-list -Delimiter delimiter | select – ExpandProperty field	import-csv -Header name,id,amt text.csv -Delimiter . | select -expa name	
9	Convert multi-line text input into records (Formatting text in PowerShell)		gc .\test.txt -ReadCount 2 | % {$_ -join ‘,’ } | ConvertFrom-Csv -Header Col1,Col2	
10	Filter out blank lines	any | Where { $_ }	“abc”,””,”def” | ? { $_ }	`abc`

```
def
```

File Search

The previous section, String Search, focused on finding text within files (as well as within collections in general). This section, in contrast, focuses on finding files: files that contain text and files whose names contain text. Get-ChildItem is at the heart of every entry in this section (though I use its ls alias for brevity). Then, depending on the recipe, you typically apply either Select-String or Where-Object to achieve the desired results.

#	Action	Command	Example
1	List file names and lines in multiple files containing a pattern	ls filespec | Select-String pattern	ls . -Recurse *.cs | Select-String “public.*void”
2	List just lines in multiple files containing pattern	(ls filespec | Select-String pattern).Line	(ls . -r *.cs | sls “public.*void”).Line
3	List files containing a pattern, returning strings	ls filespec | Select-String pattern | select -Unique Path	ls *.cs -r | sls “public.*void” | select -uniq Path
4	List files containing a pattern, returning FileInfo objects	ls filespec | Where { Select-String string $_ -Quiet }	ls *.cs -r | ? { sls -quiet “public.*void” $_ }
5	List files not containing a pattern, returning strings	ls filespec | Where { !(Select-String string $_ -Quiet) }.FullName	(ls -r *.xml| ? { !(sls -quiet “home ” $_) }).FullName
6	List files not containing a pattern, returning FileInfo objects	ls filespec | Where { !(Select-String string $_ -Quiet) }	ls -r *.xml| ? { !(sls -quiet “home” $_) }
7	List files with names matching a wildcard pattern	ls -r expression	ls -r *.html
8	List files with names matching a regex pattern	ls . options | Where { $_.Name -match pattern }	ls -r *.xml | ? { $_.name -match “abc{0,3}.*\.xml” }
9	List files with path matching a regex pattern	ls . options | Where { $_ -match pattern }	ls -r *.xml | ? { $_ -match “this\\sub\\path” }
10	Count occurrences of a string per file	any | Select-String pattern | Group path	ls -r *.xml | sls home | group path | select count,name | ft -auto

Conclusion

That's it for part 3; 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!

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 from the link below.