PowerShell One-Liners: Accessing, Handling and Writing Data

Michael Sorens

05 June 2014

> In the grand finale to Michael Sorens' series of PowerShell one-liners, we come to the handling of data, reading it in and writing it out, whether by files; input/output streams or a database. It shows how it can be done in a variety of formats including CSV, JSON, and XML.

This series is in four parts: This is part 4

- **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, covering a variety of input and output techniques: reading from and writing to files; writing and merging the multitude of output streams available in PowerShell (it's not just `stdout` and `stderr` anymore!); file housekeeping operations (move, copy, create, delete); and various other I/O techniques related to CSV, JSON, database, network, and XML.

Be sure to review the earlier parts, too, though:

Part 1 Help, Syntax, Display and Files 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; ways to cast your output in list, table, grid, or chart form.

Part 2 Variables, Parameters, Properties, and Objects, covers key PowerShell concepts of variables, parameters, properties, and objects. Part 3 explores the two fundamental data structures of PowerShell: the collection (array) and the hash table (dictionary), examining everything from creating, accessing, iterating, ordering, and selecting.

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.

Part 3 Collections, Hashtables, Arrays and Strings 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).

Each part of this series is available as both an online reference here at Simple-Talk.com, and as a special 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

Reading from Files

When reading a file, your first consideration is whether to read it into a collection or into a single string. For the latter choice, carefully consider how you want line endings to be handled; the first few entries illustrate several line ending options. The next few entries consider multiple files together; you can read them into a string collection with either one *file* per slot or one *line* per slot, or into a `MatchInfo` collection, which provides meta-data about each line. The final few entries consider counting what is in files by lines, words, and characters, with several options available. Many of these entries rely on the sample files (f1.txt and f2.txt) shown early on. Note that the fhex (Format-Hex) cmdlet is part of the PowerShell Community Extensions.

#	Action	Command	Example	`Output`

1	Read one file as an array of strings (line endings stripped)	[1] Get-Content filespec [2] ${drive:filespec} # must include a drive designator!	$myFileLines = gc foo.txt $myFileLines = ${c:subdir/foo.txt}	
2	Read one file into a single string (line endings retained, adds final CR/LF)	Get-Content filespec | Out-String	“a b`r`ncd” | sc f1.txt # a 2-line file $a = gc f1.txt |Out-String; $a.Length; $a | fhex -s ascii	9 61 20 62 0D 0A 63 64 0D 0A
3	Read one file into a single string (joining with CR/LF so no final CR/LF)	(Get-Content filespec) -join “`r`n”	$a =(gc f1.txt) -join “`r`n”; $a.Length; $a | fhex -s ascii	7 61 20 62 0D 0A 63 64
4	Read one file into a single string (joining with spaces)	[string] (Get-Content filespec)	$a = [string](gc f1.txt); $a.Length; $a | fhex -s ascii	6 61 20 62 20 63 64
5	Read one file into a single string-faster! (line endings retained, adds final CR/LF)	[1] Get-Content filespec -ReadCount 0 | Out-String [2] [System.IO.File]::ReadAllText(filespec)	[1] $a = Get-Content f1.txt -ReadCount 0 | Out-String; $a.Length; $a | fhex -s ascii [2] $a = [System.IO.File]::ReadAllText(‘f1.txt’); $a.Length; $a | fhex -s ascii	9 61 20 62 0D 0A 63 64 0D 0A
6	Read multiple files as a collection of MatchInfo objects (one per line)	Get-ChildItem filespec | Select-String “.*”	“a b`r`ncd” | sc f1.txt # a 2-line file “one`r`n`r`ntwo`r`n” | sc f2.txt #a 4-line file dir *.txt | sls “.*” | select LineNumber, Filename,Lline	LineNumber Filename Line ———- ——— —- 1 f1.txt a b 2 f1.txt cd 1 f2.txt one

				2 f2.txt 3 f2.txt two 4 f2.txt
7	Read multiple files as a collection of lines (strings)	Get-ChildItem filespec | Get-Content	$a = dir *.txt | gc; $a[0]; $a.Count	a b 6
8	Read multiple files, each file as a single string		$a = dir *.txt | % { gc $_ | out-string}; $a[0]; $a.Count	a b cd 2
9	Include literal text passage (a here string)	@” . . . “@	$passage = @” line one line two “@	
10	Read and execute PowerShell code from text file	Invoke-Expression (“@`”`n” + (gc filespec| Out-String) + ‘”@’)	iex (“@`”`n” + (gc stuff.txt| Out-String) + ‘”@’)	
11	Read with different encoding	Get-Content filespec -Encoding encodingType	gc myfile.txt -encoding UTF8	
12	Count non-empty lines in files	dir files | Get-Content |Measure-Object -Line	(dir *.txt | gc | measure -line).Lines	4
13	Count all lines in files		(dir *.txt | gc).Count	6
14	Count words in files	dir files | Get-Content |Measure-Object -Word	(dir *.txt | gc | measure -word).Words	5

15	Count characters in files, skipping line breaks	dir files | Get-Content |Measure-Object -Char	(dir *.txt | gc | measure -char).Characters	`11`
16	Count characters, skipping all whitespace	dir files | Get-Content Measure-Object -Char -IgnoreWhiteSpace	(dir *.txt | gc | measure -char -ignore).Characters	`10`

Writing to Files

Writing to files seem like it should be one of those very trivial, obvious operations. But because PowerShell's fundamental unit is an object (rather than a string like DOS or Linux) it is not quite so simple. The two main cmdlets `Out-File` and `Set-Content` both deliver content to a file but in different formats and different encodings! See John Cook's *'PowerShell output redirection: Unicode or ASCII?'* (http://bit.ly/1iBa4uv) and Oisin Grehan's *'What's the difference between Set-Content and Out-File?'* (http://bit.ly/1f4zf33) for more details. Then there's the question of streams; the next section covers that topic.

#	Action	Command	Example
1	Write each object with CR/LF line endings (Unicode default; use -Encoding to change) (Works only with FileSystem provider)	[1] any | Out-File -FilePath filespec [2] any > filespec	ls > \Âtmp\Âfile.txt
2	Append to end of file(Unicode default)	[1] any | Out-File -FilePath filespec -append [2] any >> filespec	
3	Write each object.ToString() with CR/LF (ASCII default; use -Encoding to change)	[1] any | Set-Content filespec [2] ${drive:filespec} = any	ls | Set-Content \Âtmp\Âfile.txt ${c:\Âtmp\Âfile.txt} = ls
4	Append string content to end of file (ASCII default; faster than Out-File)	any | Add-Content filespec	
5	Convert Unicode to ASCII without temp file	(Get-Content filespec) | Set-Content filespec	(gc foo.txt) | sc foo.txt
6	Write to file with specified line ending (see http://stackoverflow.com/a/10215589/115690)	$writer = [system.io.file]::CreateText(filespec); $writer.NewLine = lineEndingString;	$writer = [system.io.file]::CreateText("temp.txt"); $writer.NewLine = "`n";

		any | % { $writer.WriteLine($_) }; $writer.Close()	gc file.txt | % { $writer.WriteLine($_) }; $writer.Close()	
7	Detect file encoding	Get-FileEncoding filespec <<code from http://stackoverflow.com/a/9121679/115690>>		

Basic Writing Streams

Unlike DOS or Linux, PowerShell goes beyond the basic `stdout` and `stderr` streams to introduce several more that may be written to or redirected as desired. See *'About redirection'* (http://bit.ly/1aVRAQK) for details. The three redirection columns act as follows: **Write** creates or overwrites a file; **Append** creates or appends to a file; **Merge** joins the specified stream with `stdout`.

#	Action	Command	Write	Append	Merge	Example
1	Write to console (bypasses stdout) – cannot be redirected!	[1] Write-Host string [2] [console]::WriteLine(string)	NA	NA	NA	write-host "hello world" [console]::WriteLine("hello world")
2	Write to output stream (i.e. stdout) This is implicit if omitted, so usually not needed.	Write-Output	>	>>	NA	write-output "hello world" echo "hello world" "hello world"
3	Write to file (Unicode) and to stdout	Tee-Object	NA	NA	NA	ps | tee output.txt
4	Write to error stream (i.e. stderr)	Write-Error	2>	2>>	2>&1	write-error "problem with …"
5	Write to warning stream	Write-Warning	3>	3>>	3>&1	write-warning "NB: null detected…"
6	Write to verbose stream Requires $VerbosePreference or -Verbose to manifest	Write-Verbose	4>	4>>	4>&1	write-verbose "parameters: a, b, c"
7	Write to debug stream	Write-Debug	5>	5>>	5>&1	write-debug "xyz"

	Requires $DebugPreference or -Debug to manifest				
8	Redirect all streams	NA	*>	*>>	*>&1

Move, Copy, Create, Delete

#	Action	Command	Example
1	Copy file or folder	Copy-Item source target	cp -Recurse C:\Âtest \Â\Âremotesys\Âtest
2	Copy files from a tree and flatten hierarchy	Get-ChildItem spec -Recurse | Copy-Item -dest target	gci *.txt -recurse | Copy-Item -destination c:\Âtarget
3	Copy files from a tree and maintain hierarchy	Copy-Item -Recurse -Filter spec source target	Copy-Item -Recurse -Filter *.txt -path c:\Âsource -dest c:\Âtarget
4	Rename file or folder	Rename-Item oldName newName	rni foo.txt bar.txt
5	Move file (same drive or between drives)	Move-Item source destination	mv c:\Âfoo.txt d:\Âtemp
6	Move directory on the same drive	Move-Item source destination	mv c:\Âtmp\Âsomedir c:\Âother\Âsomedir
7	Move directory to a different drive	Copy-Item – recurse source target; Remove-Item -recurse source	cp -r c:\Âsome\Âdir d:\Âsome\Âdir; rm -r c:\Â-some\Âdir
8	Create file	[1] New-Item path – ItemType file -Value “text” [2] echo ” text ” | Set-Content path	[1] ni foo.txt – ItemType file -Value “Hello world” [2] echo “Hello world” | Set-Content foo.txt
9	Create empty directory	New-Item path -ItemType directory	mkdir foo.txt – ItemType directory
10	Delete file	Remove-Item path	rm foo.txt
11	Delete folder	Remove-Item path -recurse	rmdir -r .\Âsome\Âtempdir
12	Empty a file	Clear-Content path	clc foo.txt

13	Empty a folder	Remove-Item path\Â*.*	

Convert, Combine, and More

Beyond basic reading and writing, here are some notions on other file manipulations. Note that processing large files deserves special consideration-for example, while `Get-Content` can be very fast (with `-ReadCount 0)`, the PowerShell pipeline is not as peppy, so avoid it-see '*Speeding Up Your Scripts!*' (http://bit.ly/1mLI8Bp).

#	Action	Command	Example
1	Convert line endings (Windows to Unix)	ConvertTo-UnixLineEnding <<pscx>>	
2	Convert line endings (Unix to Windows)	ConvertTo-WindowsLineEnding <<pscx>>	
3	Append to start of file	. { any; cat filespec } | Set-Content newFilespec	.{ cat addstuff.txt; cat stuff.txt } | sc newstuff.txt
4	Append to end of file	Add-Content filespec -value text	ac -path *.txt -value "END"
5	Concatenate multiple files	Get-Content filespec-array	[1] cat example1.txt, example2.txt > examples.txt [2] cat example*.txt > allexamples.txt [3] gci filespec | Get-Content | Set-Content .\Âall.txt
6	Concatenate 2 files	[1] Get-Content filespecB |Add-Content -Path filespecA [2] Add-Content -path filespecA -value (Get-Content filespecB)	gc .\Âfile2.txt| ac -Path .\Â-file1.txt
7	Read a large file (see http://bit.ly/1mLI8Bp)	Get-Content filespec -ReadCount 0	
8	Retrieve first n lines of a large file (see http://stackoverflow.com/a/11369924/115690)	Get-Content filespec -TotalCount n	

9	Remove first n lines of a file(see http://stackoverflow.com/a/2076557/115690)	[1] ${C:filespec} = ${C:filespec } | select -skip count [2] (gc filespec | select -Skip count) | sc filespec	
10	Display new input from end of a file	Get-FileTail -Wait filespec <<pscx>>	tail -Wait c:\Âusr\Âtmp\Â-logfile.txt
11	Trim all lines in a file	(Get-Content filespec)| % { $_.trim() } | Set-Content filespec	(gc $myFile)\| % {$_.trim()} | sc $ myFile
12	Write compressed archive	[1] Write-Zip <<pscx>> [2] Write-Gzip <<pscx>> [3] Write-Tar <<pscx>>	
13	Hex Dump	Format-Hex filespec <<pscx>>	fhex c:\Âusr\Âtmp\Âfile.dat

Patterned Data: CSV and More

For an in-depth treatment on getting data in and out of PowerShell, peruse my article *'PowerShell Data Basics: File-Based Data'* (http://bit.ly/19W5Cnn) covering all the items below plus fixed-width fields, ragged-right text, multi-line record input and more.

#	Action	Command	Example	Output
1	Convert CSV data to objects	[1] Import-Csv filespec [2] any | ConvertTo-Csv	@' Shape,Color,Count Square,Green,4 Rectangle,,12 '@ | ConvertFrom-Csv	`Shape Color Count` `--- ----- -----` `Square Green 4` `Rectangle 12`
2	Convert CSV data to objects with delimiter	[1] Import-Csv file -Delimiter char	@' Shape+Color+Count Square+Green+4	`Shape Color Count` `--- ----- -----` `Square Green 4`

		[2] any | ConvertFrom-Csv -Delimiter char	Trapezoid+Black+100 ‘@ | ConvertFrom-Csv -Delimiter ‘+’	`Trapezoid Black 100`
3	Convert CSV data to objects with unprintable delimiter	[1] Import-Csv file -Delimiter “$([char]hexCode)” [2] any \| ConvertFrom-Csv -Delimiter “$([char]hexCode)”	@” Shape`tColor`tCount Square`tGreen`t4 Trapezoid`tBlack`t100 “@ | ConvertFrom-Csv -Delimiter “$([char]0x09)”	`Shape Color Count` `—— —— ——` `Square Green 4` `Trapezoid Black 100`
4	Convert CSV data to objects with multi-character delimiter	Get-Content file| foreach { $_ -replace delimiter, [char]uncommonChar } | ConvertFrom-Csv -Delimiter uncommonChar	Get-Content file.dat| % { $_ -replace “.1234.”,[char]0x06 } | ConvertFrom-Csv -Delimiter 0x06	
5	Convert CSV data to objects with external headers	[1] Import-Csv file -Header field1, field2, … [2] any | ConvertTo-Csv -Header field1, field2, …	@’ Square,Green,4 Trapezoid,Black,100 ‘@ | ConvertFrom-Csv -Header Shape, Color, Count	`Shape Color Count` `—— —— ——` `Square Green 4` `Trapezoid Black 100`
6	Convert objects to CSV data (see http://bit.ly/1aZfgDN)	Export-CSV or ConvertTo-Csv	Get-Date | Select Hour,Minute,Second | ConvertTo-Csv	`#TYPE System.DateTime` `“Hour”,”Minute”,”Second”` `“19”,”5”,”27”`
7	Convert patternable data to objects (i.e. not regular enough to simply specify a single delimiter but definable with regex)	ImportWith-Regex <<code from http://bit.ly/19W5Cnn>>	#Assume file.dat contains: george jetson 5 warren buffett 123	`Id FName LName` `— —— ——` `5 george jetson`

			horatioalger -99 $regex = “^(?<FName>.{7})(?<LName>.{10})(?<Id>.{3})$” ImportWith-Regex file.dat $regex	123 warren buffett -99 horatio alger
8	Convert objects to JSON	any | ConvertTo-Json	Get-Date | Select-Object -Property Hour,Minute,Second | ConvertTo-Json	{ “Hour”: 19, “Minute”: 0, “Second”: 44 }
9	Convert from JSON to objects	json | ConvertFrom-Json	‘{ “Hour”:19, “Minute”:0, “Second”:44 }’ | ConvertFrom-Json	Hour Minute Second —- —— —— 19 0 44

Network Basics

From simple network (UNC) paths to full-blown web-scraping and file downloading, here are the key cmdlets to get you going.

#	Action	Command	Example
1	Display current location (non-UNC paths only)	[1] Get-Location [2] $pwd [3] $pwd.Path	same
2	Display current location (UNC or non-UNC paths)	$pwd.ProviderPath	cd \Â\Âlocalhost\Âc$; $pwd.ProviderPat
3	Display UNC path of current location	$pwd.Drive.DisplayRoot	same
4	List all mapped drive ids and their UNC paths	Get-WmiObject win32_logicaldisk -filter “drivetype=4” | select DeviceId, ProviderName	same

5	Display UNC path for a mapped drive	(gwmi win32_logicaldisk -filter “deviceid=’drive’ “) .ProviderName	(gwmi win32_logicaldisk -filter “deviceid= “).ProviderName
6	Display server name of mapped drive	(gwmi win32_logicaldisk -filter “deviceid=’drive’ “) .ProviderName.Split(‘\Â’)[2]	(gwmi win32_logicaldisk -filter “deviceid=).ProviderName.Split(‘\Â’)[2]
7	Determine if a file system drive is a mapped drive	(gwmi win32_logicaldisk -filter “deviceid=’drive’ “) -ne $null	(gwmi win32_logicaldisk -filter “deviceid= $null
8	List all free drive letters (see http://stackoverflow.com/a/17548038/115690)	ls function:[a-z]: -name | where {-not (get-psdrive $_[0] -ea 0)}	same
9	Open URL in default browser	Start-Process -FilePath url	Start-Process http://www.cnn.com
10	Encode HTML	[System.Web.HttpUtility]::HtmlEncode(string)	[System.Web.HttpUtility]::HtmlEncode(‘s <something else>’)
11	Decode HTML	[System.Web.HttpUtility]::HtmlDecode(string)	[System.Web.HttpUtility]::HtmlDecode(‘s <something else>’)
12	Get web page content (a la wget or webget)	(Invoke-WebRequest uri).Content	$page = (Invoke-WebRequest http://www.xyz.com).Content
13	Get multiple files from the web	Start-BitsTransfer -Source uri -Destination path	Start-BitsTransfer -Source http://s01/tes Destination c:\Âtestdir\Â
14	Web scraping <<Html Agility Pack http://bit.ly/1kuhBHM>>	Add-Type -Path .\ÂHtmlAgilityPack.dll; $doc = New-Object HtmlAgilityPack.HtmlDocument; $page = (Invoke-WebRequest uri).Content; $status = $doc.LoadHtml($page); $items = $doc.DocumentNode.SelectNodes(xpath)	Specify a value for uri and for xpath.

Database

Install the `sqlps` module (installed automatically with SS2012), then import the module (recommend using `–DisableNameChecking` due to non-standard names). Once installed you will have a SQL Server provider (`Get-PSProvider`) and a SQL Server data store (`Get-PSDrive`) available, allowing navigating the data space with familiar commands.

The last two entries in the table below are perhaps the most intriguing: converting between PowerShell and SQL Server. From SQL Server to PowerShell is simple; the other way is less so. Chad Miller provides an outstanding foundation with his cmdlets `Out-DataTable`, `Add-SqlTable`, and `Write-DataTable` (see references). I added a few bells and whistles and provided a convenience wrapper function that composes all three cmdlets into one: the `Out-SqlTable` which I reference below.

- Hierarchy of SQL Server objects – http://msdn.microsoft.com/en-us/library/cc281947.aspx
- Canonical aliases for navigating SS provider – http://msdn.microsoft.com/en-us/library/hh213536
- Using SMO Methods and Properties – http://msdn.microsoft.com/en-us/library/hh213689
- Comparing Invoke-Sqlcmd and sqlcmd utility options – http://msdn.microsoft.com/en-us/library/cc281720
- Practical PowerShell for SQL Server Developers and DBAs
- Inserting data into a DB table with ADO (Richard Siddaway) – http://bit.ly/1eVZokD
- Use PowerShell to Write to SQL (Chad Miller) – http://bit.ly/1e5B2mo

#	Action	Command	Example
1	Go to root of SQL Server data store	Set-Location SQLSERVER:\Â	same
2	Go to root of DB objects	Set-Location SQLSERVER:\ÂSQL	same
3	List instance names on machine	Get-ChildItem SQLSERVER:\ÂSQL\Â-machine	ls SQLSERVER:\ÂSQL\Âlocalhost
4	List databases on selected instance of machine	Get-ChildItem SQLSERVER:\ÂÂSQL\Â-machine\ÂÂinstance\ÂDatabases	ls SQLSERVER:\ÂÂSQL\Âlocalhost\ÂÂ-SQLEXPRESS\ÂDatabases
5	List tables in selected database	Get-ChildItem SQLSERVER:\ÂSQL\Â-machine\Âinstance\ÂDatabases\Â-database\ÂTables	ls SQLSERVER: \ÂSQL\Âlocalhost\ÂDEFAULT\Â-Databases\Âsandbox\ÂTables
6	Create mapped drive for shortcut path	New-PSDrive -Name name -PSProvider SQLSERVER -Root root	mount -name sandboxDB -PSProvider SQLSERVER -Root SQLSERVER:\ÂSQL\Âlocalhost\ÂDEFAULT\Â-Databases\Âsandbox
7	List tables in selected database with shortcut	Get-ChildItem mappedDrive:Tables	ls sandboxDB:\ÂTables
8	Get properties of DB object	Get-Item databasePath | Get-Member -Type Properties	gi sandboxDB:\ÂTables | gm -type properties

9	List subset of tables using SMO property		gci sandboxDB:\ÂTables | where {$_.Schema -eq “dbo”}
10	Generate create scripts for all tables using SMO method		gci sandboxDB:\ÂTables | % { $_.Script() }
11	Create script for particular table		(gci sandboxDB:\ÂTables | ? { $_.name -eq “temp1” }).script()
12	Generate create scripts for each table using SMO method sending output to one file		gci sandboxDB:\ÂTables | % { $_.Script() | Out-File C:\Âtmp\ÂCreateTables.sql -append }
13	Invoke query (with default context)	cd SQLSERVER:\ÂSQL\Âmachine\Âinstance (or lower) Invoke-Sqlcmd -Query tsqlCommandString	Invoke-Sqlcmd -Query “SELECT DB_NAME()”
14	Invoke query with server specified	Invoke-Sqlcmd -Query tsqlCommandString -Server machine\Âinstance	
15	Convert DB data to PS objects	[1] Invoke-Sqlcmd -Query tsqlCommandString |Out-GridView [2] Invoke-Sqlcmd -Query tsqlCommandString |Format-Table	[1] Invoke-Sqlcmd “select * from Table1” | ogv [2] Invoke-Sqlcmd “select * from Table1” | ft -auto
16	Convert PS objects to DB data	any | Out-SqlTable options <<code from http://bit.ly/1bjlFe1>>	ps | select ProcessName, Handle | Out-SqlTable -TableName “processes” -DropExisting -RowId “MyId”

XML Data

There is much to say about XML data access in PowerShell. One very nice feature is that you can use either XPath notation or object notation to access nodes! This topic does not really lend itself to table entries here, though, so see *‘PowerShell Data Basics: XML‘* (http://bit.ly/1f54D2T) for the full story.

#	Action	Command	Example	Output
1	Convert literal text to XML	[1] [xml] variable = any [2] variable = [xml] any	[xml]$xml=@” <root>	root first

			<first><more>foobar</more></first> <second>data</second> </root> “@; $xml.SelectNodes(“//*”) | Select -Expand Name	more second
2	Convert file contents to XML	[xml]$variable = Get-Content filespec	[xml]$xml = Get-Content mystuff.xml	
3	Access XML nodes with XPath	variable.SelectNodes(xpath)	$xml.SelectNodes(“//first[more]”)	more —- foobar
4	Access XML nodes with object notation	variable.nodeName.nodeName…	$xml.root.first	more —- foobar
5	Pretty-print XML	[1] any | Format-Xml <<pscx>> [2] Format-Xml filespec <<pscx>>		
6	Transform XML with XSLT	Convert-Xml xmlFilespec xsltFilespec <<pscx>>		
7	Test XML for well-formedness and validity	Test-Xml filespec		

Conclusion

You made it to the end of part 4-almost 400 recipes later!-which is the end of the series (at least for now). As usual, I will conclude with my tongue-in-cheek disclaimer: 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. And enjoy your PowerShell adventures!