Working with Files and Folders

### Listing All the Files and Folders Within a Folder

You can get all items directly within a folder by using **Get-ChildItem**. Add the optional **Force** parameter to display hidden or system items. For example, this command displays the direct contents of Windows PowerShell Drive C (which is the same as the Windows physical drive C):

Copy

Get-ChildItem -Force C:\

The command lists only the directly contained items, much like using Cmd.exe's **DIR** command or **ls** in a UNIX shell. In order to show contained items, you need to specify the **-Recurse** parameter as well. (This can take an extremely long time to complete.) To list everything on the C drive:

Copy

Get-ChildItem -Force C:\ -Recurse

**Get-ChildItem** can filter items with its **Path**, **Filter**, **Include**, and **Exclude**parameters, but those are typically based only on name. You can perform complex filtering based on other properties of items by using **Where-Object**.

The following command finds all executables within the Program Files folder that were last modified after October 1, 2005 and which are neither smaller than 1 megabyte nor larger than 10 megabytes:

Copy

Get-ChildItem -Path $env:ProgramFiles -Recurse -Include \*.exe | Where-Object -FilterScript {($\_.LastWriteTime -gt "2005-10-01") -and ($\_.Length -ge 1m) -and ($\_.Length -le 10m)}

### Copying Files and Folders

Copying is done with **Copy-Item**. The following command backs up C:\boot.ini to C:\boot.bak:

Copy

Copy-Item -Path c:\boot.ini -Destination c:\boot.bak

If the destination file already exists, the copy attempt fails. To overwrite a pre-existing destination, use the Force parameter:

Copy

Copy-Item -Path c:\boot.ini -Destination c:\boot.bak -Force

This command works even when the destination is read-only.

Folder copying works the same way. This command copies the folder C:\temp\test1 to the new folder c:\temp\DeleteMe recursively:

Copy

Copy-Item C:\temp\test1 -Recurse c:\temp\DeleteMe

You can also copy a selection of items. The following command copies all .txt files contained anywhere in c:\data to c:\temp\text:

Copy

Copy-Item -Filter \*.txt -Path c:\data -Recurse -Destination c:\temp\text

You can still use other tools to perform file system copies. XCOPY, ROBOCOPY, and COM objects, such as the **Scripting.FileSystemObject,** all work in Windows PowerShell. For example, you can use the Windows Script Host **Scripting.FileSystem COM** class to back up C:\boot.ini to C:\boot.bak:

Copy

(New-Object -ComObject Scripting.FileSystemObject).CopyFile("c:\boot.ini", "c:\boot.bak")

### Creating Files and Folders

Creating new items works the same on all Windows PowerShell providers. If a Windows PowerShell provider has more than one type of item—for example, the FileSystem Windows PowerShell provider distinguishes between directories and files—you need to specify the item type.

This command creates a new folder C:\temp\New Folder:

Copy

New-Item -Path 'C:\temp\New Folder' -ItemType "directory"

This command creates a new empty file C:\temp\New Folder\file.txt

Copy

New-Item -Path 'C:\temp\New Folder\file.txt' -ItemType "file"

### Removing All Files and Folders Within a Folder

You can remove contained items using **Remove-Item**, but you will be prompted to confirm the removal if the item contains anything else. For example, if you attempt to delete the folder C:\temp\DeleteMe that contains other items, Windows PowerShell prompts you for confirmation before deleting the folder:

Copy

Remove-Item C:\temp\DeleteMe

Confirm

The item at C:\temp\DeleteMe has children and the -recurse parameter was not

specified. If you continue, all children will be removed with the item. Are you

sure you want to continue?

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help

(default is "Y"):

If you do not want to be prompted for each contained item, specify the **Recurse**parameter:

Copy

Remove-Item C:\temp\DeleteMe -Recurse

### Mapping a Local Folder as a Windows Accessible Drive

You can also map a local folder, using the **subst** command. The following command creates a local drive P: rooted in the local Program Files directory:

Copy

subst p: $env:programfiles

Just as with network drives, drives mapped within Windows PowerShell using **subst** are immediately visible to the Windows PowerShell shell.

### Reading a Text File into an Array

One of the more common storage formats for text data is in a file with separate lines treated as distinct data elements. The **Get-Content** cmdlet can be used to read an entire file in one step, as shown here:

Copy

PS> Get-Content -Path C:\boot.ini

[boot loader]

timeout=5

default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS

[operating systems]

multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"

/noexecute=AlwaysOff /fastdetect

multi(0)disk(0)rdisk(0)partition(1)\WINDOWS=" Microsoft Windows XP Professional

with Data Execution Prevention" /noexecute=optin /fastdetect

**Get-Content** already treats the data read from the file as an array, with one element per line of file content. You can confirm this by checking the **Length** of the returned content:

Copy

PS> (Get-Content -Path C:\boot.ini).Length

6

# Working with file paths

We are going to start this off by showing you the commands for working with file paths.

## Test-Path

Test-Path is one of the more well known commands when you start working with files. It allows you to test for a folder or a file before you try to use it.

If( Test-Path -Path $Path )

{

Do-Stuff -Path $Path

}

## Split-Path

Split-Path will take a full path to a file and gives you the parent folder path.

PS:> Split-Path -Path 'c:\users\kevin.marquette\documents'

c:\users\kevin.marquette

If you need the file or folder at the end of the path, you can use the -Leaf argument to get it.

PS:> Split-Path -Path 'c:\users\kevin.marquette\documents' -Leaf

documents

## Join-Path

Join-Path can join folder and file paths together.

PS:> Join-Path -Path $env:temp -ChildPath testing

C:\Users\kevin.marquete\AppData\Local\Temp\testing

I use this anytime that I am joining locations that are stored in variables. You don’t have to worry about how to handle the backslash becuse this takes care of it for you. If your variables both have backslashes in them, it sorts that out too.

## Resolve-Path

Resolve-Path will give you the full path to a location. The important thing is that it will expand wildcard lookups for you. You will get an array of values if there are more than one matche.

Resolve-Path -Path 'c:\users\\*\documents'

Path

----

C:\users\kevin.marquette\Documents

C:\users\Public\Documents

That will enumerate all the local users document folders.

I commonly use this on any path value that I get as user input into my functions that accept multiple files. I find it as an easy way to add wildcard support to parameters.

# Saving and reading data

Now that we have all those helper CmdLets out of the way, we can talk about the options we have for saving and reading data.

I use the $Path and $Data variables to represent your file path and your data in these examples. I do this to keep the samples cleaner and it better reflects how you would use them in a script.

## Basic redirection with Out-File

PowerShell was introduced with Out-File as the way to save data to files. Here is what the help on that looks like.

Get-Help Out-File

<#

SYNOPSIS

Sends output to a file.

DESCRIPTION

The Out-File cmdlet sends output to a file. You can use this cmdlet instead of the redirection operator (>) when you need to use its parameters.

#>

For anyone coming from batch file, Out-File is the basic replacement for the redirection operator >. Here is a sample of how to use it.

'This is some text' | Out-File -FilePath $Path

It is a basic command and we have had it for a long time. Here is a second example that shows some of the limitations.

Get-ChildItem |

Select-Object Name, Length, LastWriteTime, Fullname |

Out-File -FilePath $Path

The resulting file looks like this when executed from my temp folder:

Name

Length LastWriteTime FullName

----

------ ------------- --------

3A1BFD5A-88A6-487E-A790-93C661B9B904 9/6/2016 10:38:54 AM C:\Users\kevin.marqu...

acrord32\_sbx 9/4/2016 10:18:18 AM C:\Users\kevin.marqu...

TCD789A.tmp 9/8/2016 12:27:29 AM C:\Users\kevin.marqu...

You can see that the last column of values are cut short. Out-File is processing objects for the console but redirects the output to a file. All the issues you have getting something to format in the console will show up in your output file. The good news is that we have lots of other options for this that I will cover below.

## Save text data with Add-Content

I personally don’t use Out-File and prefer to use the Add-Content and Set-Contentcommands. There is also a Get-Content command that goes with them to read file data.

$data | Add-Content -Path $Path

Get-Content -Path $Path

Add-Content will create and append to files. Set-Content will create and overwrite files.

These are good all-purpose commands as long as performance is no a critical factor in your script. They are great for individual or small content requests. For large sets of data where performance matters more than readability, we can turn to the .Net framework. I will come back to this one.

## Import data with Get-Content -Raw

Get-Content is the goto command for reading data. By default, this command will read each line of the file. You end up with an array of strings. This also passes each one down the pipe nicely.

The -Raw parameter will bring the entire contents in as a multi-line string. This also performs faster because fewer objects are getting created.

Get-Content -Path $Path -Raw

## Save column based data with Export-CSV

If you ever need to save data for Excel, Export-CSV is your starting point. This is good for storing an object or basic structured data that can be imported later. The CSV format is comma separated values in a text file. Excel is often the default viewer for CSV files.

If you want to import Excel data in PowerShell, save it as a CSV and then you can use Import-CSV. There are other ways to do it but this is by far the easiest.

$data | Export-CSV -Path $Path

Import-CSV -Path $Path

### -NoTypeInformation

Export-CSV will insert type information into the first line of the CSV. If you don’t want that, then you can specify the -NoTypeInformation parameter.

$data | Export-CSV -Path $Path -NoTypeInformation

## Save rich object data with Export-CliXml

The Export-CliXml command is used to save full objects to a file and then import them again with Import-CliXml. This is for objects with nested values or complex datatypes. The raw data will be a verbose serialized object in XML. The nice thing is that you can save a an object to the file and when you import it, you will get that object back.

Get-Date | Export-Clixml date.clicml

$date = Import-Clixml .\date.clicml

$date.GetType()

IsPublic IsSerial Name BaseType

-------- -------- ---- --------

True True DateTime System.ValueType

This serialized format is not intened for be viewd or edited directly. Here is what the date.clixml file looks like:

<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">

<Obj RefId="0">

<DT>2017-03-17T00:00:00.3522798-07:00</DT>

<MS>

<Obj N="DisplayHint" RefId="1">

<TN RefId="0">

<T>Microsoft.PowerShell.Commands.DisplayHintType</T>

<T>System.Enum</T>

<T>System.ValueType</T>

<T>System.Object</T>

</TN>

<ToString>DateTime</ToString>

<I32>2</I32>

</Obj>

</MS>

</Obj>

</Objs>

Don’t worry about trying to understand it. You are not intended to be digging into it.

This is another command that I don’t find myself using often. If I have a nested or hierarchical dataset, then JSON is my goto way to save that information.

## Save structured data with ConvertTo-Json

When my data is nested and I may want to edit it by hand, then I use ConvertTo-Json to convert it to JSON. ConvertFrom-Json will convert it back into an object. These commands do not save or read from files on their own. You will have to turn to Get-Content and Set-Content for that.

$Data = @{

Address = @{

Street = '123 Elm'

State = 'California'

}

}

$Data | ConvertTo-Json | Add-Content -Path $Path

$NewData = Get-Content -Path $Path -Raw | ConvertFrom-Json

$NewData.Address.State

There is one important thing to note on this example. I used a [hashtable] for my $Databut ConvertFrom-Json returns a [PSCustomObject] instead. This may not be a problem but there is not an easy fix for it.

Also note the use of the Get-Content -Raw in this example. ConvertFrom-Json expects one string per object.

Here is the contents of the JSON file from above:

{

"Address": {

"State": "California",

"Street": "123 Elm"

}

}

You will notice that this is similar the original hashtable. This is why JSON is a popular format. It is easy to read, understand and edit if needed. I use this all the time for configuration files in my own projects.

# Other options and details

All of those CmdLets are easy to work with. We also have some other parameters and access to .Net for more options.

## Get-Content -ReadCount

The -ReadCount parameter on Get-Content defines how many lines that Get-Content will read at once. There are some situations where this can improve the memory overhead of working with larger files.

This generally includes piping the results to something that can process them as they come in and don’t need to keep the input data.

$dataset = @{}

Get-Content -Path $path -ReadCount 15 |

Where-Object {$PSItem -match 'error'} |

ForEach-Object {$dataset[$PSItem] += 1}

This example will count how many times each error shows up in the $Path. This pipeline can process each line as it is read from the file.

You may not have code that leverages this often but this is a good option to be aware of.

## Faster reads with System.IO.File

That ease of use that the CmdLets provide can come at a small cost in raw performance. It is small enough that you will not notice it for most of the scripting that you do. When that day comes that you need more speed, you will find yourself turning to the native .Net commands. Thankfully they are easy to work with.

[System.IO.File]::ReadAllLines( ( Resolve-Path $Path ) )

This is just like Get-Content -Path $Path in that you will end up with a collection full of strings. You can also read the data as a multi-line string.

[System.IO.File]::ReadAllText( ( Resolve-Path $Path ) )

The $Path must be the full path or it will try to save the file to your C:\Windows\System32folder. This is why I use Resolve-Path in this example.

Here is an example CmdLet that I built around these .Net calls: Import-Content

## Writes with System.IO.StreamWriter

On that same note, we can also use System.IO.StreamWriter to save data. It is not always faster than the native Cmdlets. This one clearly falls into the rule that if performance matters, test it.

System.IO.StreamWriter is also a little bit more complicated than the native CmdLets.

try

{

$stream = [System.IO.StreamWriter]::new( $Path )

$data | ForEach-Object{ $stream.WriteLine( $\_ ) }

}

finally

{

$stream.close()

}

We have to open a StreamWriter to a $path. Then we walk the data and save each line to the StreamWriter. Once we are done, we close the file. This one also requires a full path.

I had to add error handling around this one to make sure the file was closed when we were done. You may want to add a catch in there for custom error handling.

This should work very well for string data. If you have issues, you may want to call the .ToString() method on the object you are writing to the stream. If you need more flexibility, just know that you have the whole .Net framework available at this point.

## Saving XML

If you are working with XML files, you can call the Save() method on the XML object.

$Xml = [xml]"<r><data/></r>"

$Path = (join-path $pwd 'File.xml')

$Xml.Save($Path)

Just like the other .Net methods in System.IO, you need to specify the full path to the file. I use $pwd in this example because it is an automatic variable that contains the result of Get-Location (local path).

## Quick note on encoding

The file encoding is the way the data is transformed into binary when saved to disk. Most of the time it just works unless you do a lot of cross platform work.

If you are running into issues with encoding, most of the CmdLets support specifying the encoding. If you want to default the encoding for each command, you can use the $PSDefaultParameterValues hashtable like this:

# Create default values for any parameter

# $PSDefaultParameterValues["Function:Parameter"] = $value

# Set the default file encoding

$PSDefaultParameterValues["Out-File:Encoding"] = "UTF8"

$PSDefaultParameterValues["Set-Content:Encoding"] = "UTF8"

$PSDefaultParameterValues["Add-Content:Encoding"] = "UTF8"

$PSDefaultParameterValues["Export-CSV:Encoding"] = "UTF8"

Read data from file

foreach($line in Get-Content .\file.txt) {

if($line -match $regex){

# Work here

}

}

First, let's start by defining a variable containing a path to our directory as well as another variable, an object that is an array list (this is what *New-Object System.Collections.ArrayList* does), to hold our findings.

$fileDirectory = "c:\scripts\reports";  
$parse\_results = New-Object System.Collections.ArrayList;

Now we'll need a foreach loop combined with a Get-ChildItem cmdlet call to get a list of all the files in the directory.

# Use a foreach to loop through all the files in a directory.  
# This method allows us to easily track the file name so we can report   
# our findings by file.  
foreach($file in Get-ChildItem $fileDirectory)  
{  
 # Processing code goes here  
}

Get-ChildItem "C:\Users\gerhardl\Documents\My Received Files" -Filter \*.log |

Foreach-Object {

$content = Get-Content $\_.FullName

#filter and save content to the original file

$content | Where-Object {$\_ -match 'step[49]'} | Set-Content $\_.FullName

#filter and save content to a new file

$content | Where-Object {$\_ -match 'step[49]'} | Set-Content ($\_.BaseName + '\_out.log')

}