PowerShell equivalents for common Linux/bash commands

mathieubuisson.github.io/powershell-linux-bash/

dsz

September 30, 2015

The majority of my colleagues have more of a Linux background than Windows. So their `cat` and their `grep` are near and dear to their heart and their first reflex when they get into PowerShell is to replicate these commands.

First, this is not always a good approach because bash and PowerShell are fundamentally different. When we run bash commands or external executables in bash, we get plain text. When we run PowerShell cmdlets we get objects.
So quite often, translating the bash way of doing things to PowerShell is the bad way of doing things. Powershell gives us rich objects with properties and methods to easily extract the information we need and/or to manipulate them in all sorts of ways. Use this to your advantage !

Still, I'm going to do this translation exercise for a few basic commands because it can be an interesting learning exercise for bash users coming to PowerShell. Besides, this is an opportunity to illustrate fundamental differences between bash and PowerShell.

pwd :

`Get-Location`
Here is an example :

```
C:\> Get-Location

Path
----
C:\
```

By the way, PowerShell has been designed to be user-friendly, even old-school-Unix-shell-user-friendly, so there are built-in aliases for popular Linux/bash commands which are pointing to the actual cmdlet.
For example, bash users can still let their muscle memory type `pwd`, because it is an alias to the cmdlet `Get-Location`. Other alias to this cmdlet : `gl`.

cd :

`Set-Location`
But you can use the aliases `cd`, `sl` or `chdir` if you have old habits or to save typing when you are using PowerShell interactively at the console.

ls :

`Get-ChildItem`

Conveniently, there is the built-in `ls` alias for those who come from bash and `dir` for those who come from cmd.exe.
A parameter combination which is frequently used is `ls -ltr`, it sorts the items to get the most recently modified files at the end. The PowerShell equivalent would be :

```
C:\> Get-ChildItem $env:USERPROFILE\Desktop | Sort-Object -Property LastWriteTime


    Directory: C:\Users\mbuisson\Desktop


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        18/10/2016     11:18           1998 MediaServerDscConfig.ps1
-a----        22/10/2016     15:37           1567 CopyArtifactsToAzure.ps1
-a----        25/10/2016     13:39           1470 Packaging.ps1
-a----        09/11/2016     13:38           1742 Create-Nuspec.ps1
-a----        21/11/2016     10:42            816 Mount-AzureFileShare.ps1
```

Notice how close to plain English the syntax is. This helps you *"Think it, type it, get it"* as **Jeffrey Snover** likes to say.
Sorting is based on a property, not a column, so you don't need to know the column number, you just need the property name.

find :

`Get-ChildItem`

But this time with the `Include` (or `Filter`) and `Recurse` parameters. For example, a common use case of find is to look recursively for files which have a case-insensitive string in their name :

```
find . -type f -iname "azure"
```

PowerShell is case-insensitive in general, so we have nothing in particular to do in this regard :

```
Get-ChildItem -Filter "*azure*" -Recurse -File
```

cp :

`Copy-Item`

Let's say you want to copy a folder named `Tools` including all its files and sub-directories to your home directory, in bash you run :

```
cp -R Tools ~/
```

In PowerShell on a Windows machine, you would run :

```
C:\> Copy-Item -Path '.\Tools\' -Destination $env:USERPROFILE -Recurse
```

`$env:` is a scope modifier, it tells PowerShell to look for the variable named `USERPROFILE` in a special scope : the environment. This is a convenient way to use environment variables.

In addition, the `Path` and `Destination` parameters are positional. `Path` is at position 1 and `Destination` is at position 2, so the following command accomplishes the same as the previous one :

```
C:\> Copy-Item '.\Tools\' $env:USERPROFILE -Recurse
```

To save even more typing, we could use the aliases `cp` , `copy` or `cpi` .

rm :

`Remove-Item`
Here is the equivalent of the (in)famous `rm -rf` :

```
Remove-Item -Recurse -Force
```

And if you tell me it's too much typing, I'm going to throw aliases at you :
`rm` , `ri` , `rmdir` , `rd` and `del` .

mkdir :

```
C:\> New-Item -ItemType Directory -Name 'MyNewFolder'


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----       28/07/2016     11:23                MyNewFolder
```

It does create the parent if needed without any special parameter, so it works like `mkdir -p` as well.

touch :

`New-Item`
For those who like to “touch” to create a bunch of files, the cmdlet `New-Item` can do it when specifying ‘File’ for the `ItemType` parameter. A nice usage example is :

```
touch MyFile{1..4}
```

This creates 4 files : Myfile1, Myfile2, Myfile3 and Myfile4.
Here is a simple way to do this with PowerShell :

```
C:\> 1..4 | ForEach-Object { New-Item -ItemType File -Name "MyFile$_" }


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        28/07/2016     11:30              0 MyFile1
-a----        28/07/2016     11:30              0 MyFile2
-a----        28/07/2016     11:30              0 MyFile3
-a----        28/07/2016     11:30              0 MyFile4
```

This deserves a few explanations.
`..` is the **range** operator, so `1..4` stands for 1,2,3,4.
`ForEach-Object` is a cmdlet used for iteration. It executes the scriptblock between the `{}` once for every object passed to it via the pipeline.

In case you are wondering what is the `$_` in the example above, it is a representation of the object currently being processed, which was passed from the pipeline. So, in the example above, `$_` stores the value `1`, then it stores the value `2`, then the value `3` and finally the value `4`.

cat :

`Get-Content`
Note that even when we run this cmdlet against a text file, this doesn’t output plain text, this outputs one object of the type `[string]` for each line in the file.
we can use its aliases : `cat`, `gc` or `type`.

tail :

```
tail -n7 ./MyFile1
```

This would output the last 7 lines of the file MyFile1 :

```
C:\> Get-Content -Tail 7 .\MyFile1
Hey there ! I'm line number 2.
Hey there ! I'm line number 3.
Hey there ! I'm line number 4.
Hey there ! I'm line number 5.
Hey there ! I'm line number 6.
Hey there ! I'm line number 7.
Hey there ! I'm line number 8.
```

The `Tail` parameter has an alias : `Last`, this makes this parameter more discoverable for those who `Tail` would not even cross their mind because they don't have a Linux background. This parameter was introduced with PowerShell 3.0.

An exceedingly valuable usage of the `tail` command for troubleshooting is `tail -f` to display any new lines of a log file as they are written to the file. For this, there is the `Wait` parameter, which was introduced in PowerShell 3.0 as well.

grep :

This is the one I get asked about the most :

> "How do you do 'grep' in Powershell ?"

And my answer is :

> "Most of the time, **you don't**."

Think about what you are trying to achieve when you use `grep` : filtering lines of text which contain a specific value, string, or pattern.

In Powershell, most of the time we are dealing with **objects** so this translates to : filtering the objects which have 1 or more value(s) in a property. No text-parsing required here, unless you are dealing with objects of the type `[string]`.

Many cmdlets have built-in parameters which allows to filter the objects, but the generic filtering mechanism is the cmdlet `Where-Object`. For example, to filter the processes which have a working set of more than 100 MB, you would run the following :

```
C:\> Get-Process | Where-Object { $_.WorkingSet -gt 104857600 }

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    392      60   189080     137908   1,669.80    5660   1 chrome
   2578      88   155792     136412   3,480.59    7244   1 chrome
    537      51   271080     135876   1,392.81   10048   1 chrome
    500      64   140452     142732     410.08   10560   1 Code
      0     118   141664     147184   5,991.05    6312   1 ruby
    498      69   261916     148068   1,498.70    2656   1 slack
```

Notice that in the output, the **WorkingSet** property is displayed in KiloBytes, but this is due to the formatting view, the actual value is in bytes.

Since PowerShell version 3.0, `Where-Object` supports a simplified syntax, so the following would do the same as the previous command :

```
Get-Process | Where-Object WorkingSet -gt 104857600
```

By the way, in case we are dealing with strings and we really want to filter objects on a specific string or pattern, we can use `Select-String`.
One use case is working with a plain text log file, for example :

```
C:\> Select-String -Path 'C:\Windows\iis.log' -Pattern 'Failed'

Windows\iis.log:11:[07/24/2017 15:40:08] Failed to create iisCngConfigurationKey key container
(result=0x8009000f)
Windows\iis.log:21:[07/24/2017 15:40:08] Failed to create iisCngWasKey key container
(result=0x8009000f)
Windows\iis.log:76:[07/24/2017 15:40:10] Failed to create iisCngConfigurationKey key container
(result=0x8009000f)
Windows\iis.log:80:[07/24/2017 15:40:10] Failed to create iisCngWasKey key container
(result=0x8009000f)
Windows\iis.log:679:[07/24/2017 15:43:28] < WARNING! > Failed to update the
DynamicIPRestrictionModule globalModule (ignoring failure) (result=0x80070490)
```

And if you are a regular expression nerd, you can feed them to the `Pattern` parameter.

uname :

For example, `uname -a` outputs the OS, the hostname, the kernel version and the architecture.
A good way to get the equivalent information on a Windows machine with PowerShell is to use the Win32_OperatingSystem class from CIM/WMI :

```
C:\> $Properties = 'Caption', 'CSName', 'Version', 'BuildType', 'OSArchitecture'
C:\> Get-CimInstance Win32_OperatingSystem | Select-Object $Properties | Format-Table -AutoSize

Caption                    CSName  Version    BuildType         OSArchitecture
-------                    ------  -------    ---------         --------------
Microsoft Windows 10 Pro DevBox    10.0.15063 Multiprocessor Free 64-bit
```

Yes, I know, it is much longer. Fortunately, there is tab completion for cmdlets, parameters, and even sometimes for parameter values.
The `Format-Table -AutoSize` is just to make the width of the output blog-friendly.

mkfs :

`New-Volume` or `Format-Volume` if the volume already exists.

ping :

Off course, we could run `ping.exe` from PowerShell but if we need an object-oriented `ping` equivalent, there is the cmdlet `Test-Connection` :

```
C:\> Test-Connection 192.168.0.21 | Format-Table -AutoSize

Source      Destination  IPV4Address  IPV6Address Bytes Time(ms)
------      -----------  -----------  ----------- ----- --------
DevBox        192.168.0.21 192.168.0.21          32    1
DevBox        192.168.0.21 192.168.0.21          32    1
DevBox        192.168.0.21 192.168.0.21          32    1
DevBox        192.168.0.21 192.168.0.21          32    1
```

man :

`Get-Help`
The PowerShell help system is extremely... helpful and it would deserve an entire article on its own.
To get the help information for a specific cmdlet, let's say `Stop-Service` :

```
Get-Help Stop-Service -Full
```

The parameter `Full` outputs everything : the description, syntax, information on every parameter, usage examples...
We can also search help information on conceptual topics. Let's say you are a regular expression nerd and you want to know how they work :

```
Get-Help "about_regular*"
```

The help files for conceptual topics have a name starting with "about_".

cut :

This is used to select columns from a text input (usually a file).
Because this is plain text, we need to map the fields to the column numbers and depending on the format of the input we may need to specify a delimiter to define the columns.

In PowerShell, this painful text-parsing is not required because again, we deal with objects. If you want to retain only some properties from some input objects, just use `Select-Object` and specify the property names.
That's it.

```
C:\> Get-ChildItem $env:USERPROFILE\Desktop -Filter "*.ps1" |
>> Select-Object -Property 'Name', 'Length'

Name                        Length
----                        ------
CopyArtifactsToAzure.ps1      1567
Create-Nuspec.ps1             1742
Deploy.ps1                     554
Mount-AzureFileShare.ps1       816
```

The quotes surrounding the property names are not mandatory. They indicate that these are `[string]` values but because the `Property` parameter of `Select-Object` expects strings, PowerShell is kind enough to cast (convert) these values to strings.

There are plenty of other commands and I cannot write on all of them, but if there are popular commands that you think should be there, please let me know and I will add them to this article.