CSI3670

(literally the worst meme)

Erik Fredericks
January 22, 2018

Overview

PowerShell Introduction
Examples

What is PowerShell?

“Object-based management engine based on .NET”

PowerShell for Newbies – Jeffery Hicks

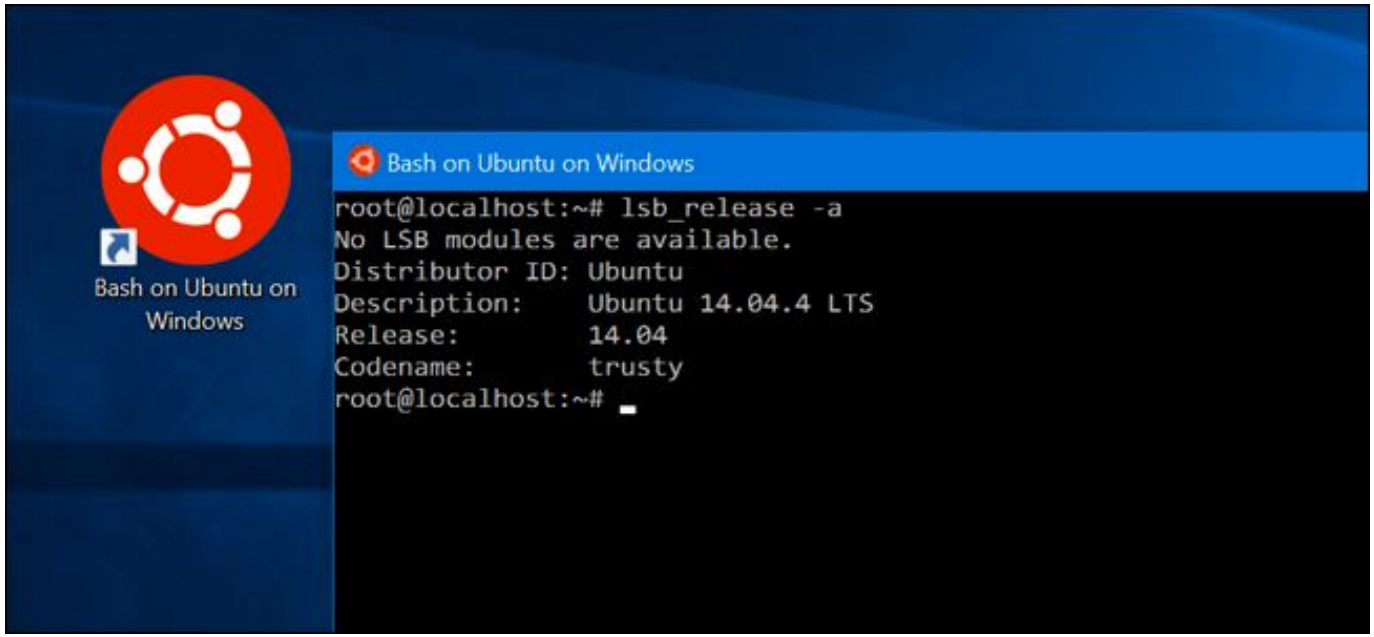
Or…

Bash for Windows!

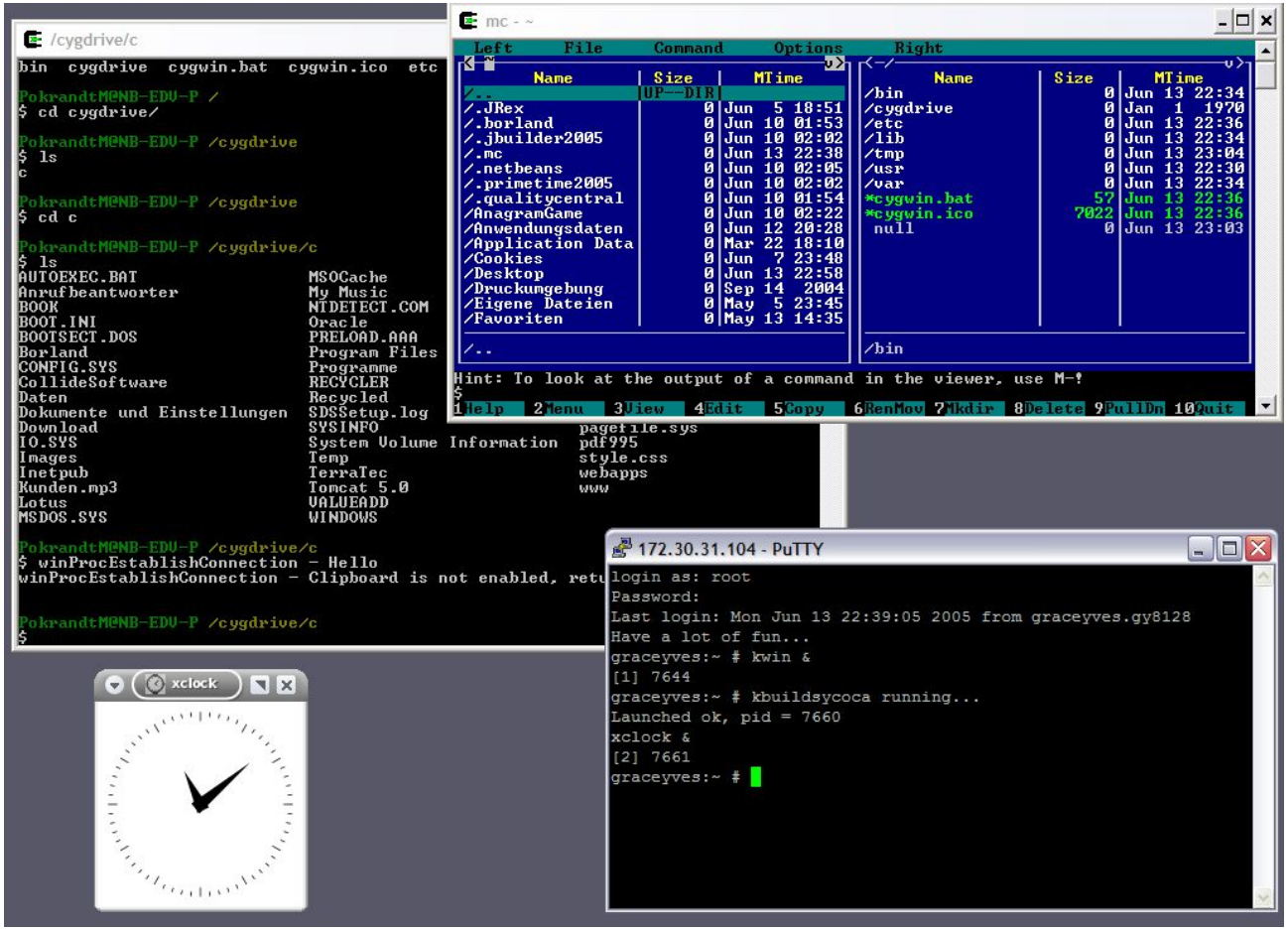
and open source…
https://github.com/PowerShell/PowerShell/releases/

(Or go the other way around…)

http://www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/



(Or go the old way…(more stable))



PowerShell

What it is not

- Just another scripting language (e.g., VBScript)
  - You can script in it, however you can also run commands as is
- A programming language
  - Built on .NET, but not a full-featured language
    - PowerShell SDK, however, can hook in with Visual Studio
- Purely console-driven
  - GUI tools available
    - For better or worse...

Paradigm

What is PowerShell *exactly*?

Think of it less as a text parser and more as dealing with objects in a pipeline

Everything in PowerShell is an **object**

Includes properties, attributes, read-write status, etc.

```
PS C:\> get-service | get-member
```

Objects

PS C:\> get-service | get-member

Object Type

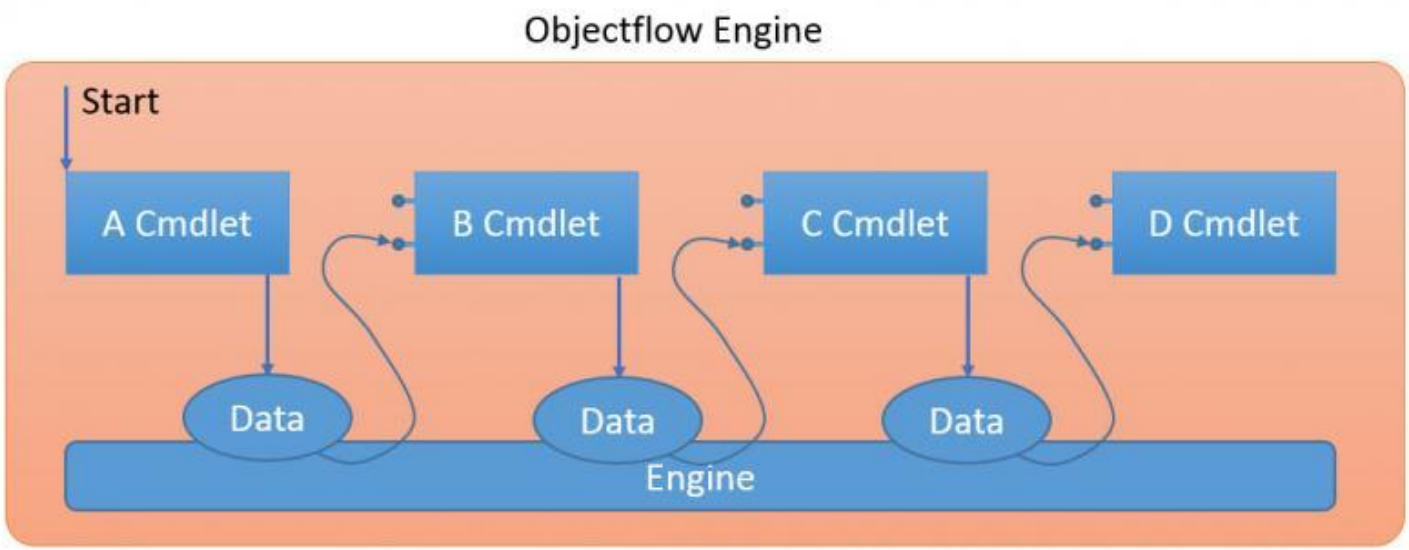
Administrator: Windows PowerShell

```
PS C:\Users\Administrator> get-service | get-member


   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType     Definition
----                      ----------     ----------
Name                      AliasProperty  Name = ServiceName
RequiredServices          AliasProperty  RequiredServices = ServicesDependedOn
Disposed                  Event          System.EventHandler Disposed(System.Object, System.EventArgs)
Close                     Method         void Close()
Continue                  Method         void Continue()
CreateObjRef              Method         System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Dispose                   Method         void Dispose(), void IDisposable.Dispose()
Equals                    Method         bool Equals(System.Object obj)
ExecuteCommand            Method         void ExecuteCommand(int command)
GetHashCode               Method         int GetHashCode()
GetLifetimeService        Method         System.Object GetLifetimeService()
GetType                   Method         type GetType()
InitializeLifetimeService Method         System.Object InitializeLifetimeService()
Pause                     Method         void Pause()
Refresh                   Method         void Refresh()
Start                     Method         void Start(), void Start(string[] args)
Stop                      Method         void Stop()
WaitForStatus             Method         void WaitForStatus(System.ServiceProcess.ServiceControllerStatus desiredStat...
CanPauseAndContinue       Property       bool CanPauseAndContinue {get;}
CanShutdown               Property       bool CanShutdown {get;}
CanStop                   Property       bool CanStop {get;}
Container                 Property       System.ComponentModel.IContainer Container {get;}
DependentServices         Property       System.ServiceProcess.ServiceController[] DependentServices {get;}
DisplayName               Property       string DisplayName {get;set;}
MachineName               Property       string MachineName {get;set;}
ServiceHandle             Property       System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName               Property       string ServiceName {get;set;}
ServicesDependedOn        Property       System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType               Property       System.ServiceProcess.ServiceType ServiceType {get;}
Site                      Property       System.ComponentModel.ISite Site {get;set;}
Status                    Property       System.ServiceProcess.ServiceControllerStatus Status {get;}
ToString                  ScriptMethod   System.Object ToString();
```

The Pipeline



PS C:\> A | B | C | D

The Pipeline

```
function Rx-Output
{
  process { Write-Host $_ -ForegroundColor Green }
}

PS C:\> Write-Output “CIT349 is neat”
CIT349 is neat
PS C:\> Write-Output “CIT349 is neat” | Rx-Output
CIT349 is neat
PS C:\> Write-Host “CIT349 is neat” | Rx-Output
CIT349 is neat
```

The Pipeline

```
PS C:\> get-service | where {$_.status -eq "running"}
```

Each object in get-service piped into the `where` clause
status property checked to see if each object ($_) is running

Monstrosities

Of course, you can chain together commands…

PS C:\> dir “C:\Program Files” -Recurse -File | Group-Object -Property Extension | Select-Object -property Count,Name,@{Name=”Size”; Expression={ ($_.group | Measure-Object -property length -sum).sum}} | Sort-Object -property Size -Descending | Select-Object -first 5 | ft -auto

```
PS C:\Users\Administrator> dir "C:\Program Files" -Recurse -File | Group-Object -Property Extension | Select-Object -pro
perty Count,Name,@{Name="Size";Expression={ ($_.group | Measure-Object -property length -sum).sum}} | Sort-Object -prope
rty Size -Descending | Select-Object -first 5 | Format-Table -AutoSize

Count Name     Size
----- ----     ----
  134 .dll 89867376
   22 .exe  9027248
    6 .txt  2826101
    1 .dat   863824
   64 .mui   779264
```

Or use intermediate variables…

```
# Get files first
PS C:\> $files = dir "C:\Program Files" -Recurse -File

# Then group by extension
PS C:\> $grouped = $files | group extension

# Then select properties we want
PS C:\> $results = $grouped | select
Count,Name,@{Name="Size";Expression={ ($_.group | measure length
-sum).sum}}

# Lastly, sort, truncate list, and format as a table
PS C:\> $results | sort size -desc | select -first 5 | ft -auto
```

Splatting

```
$test = @{'key1'='value1';'key2'='value2'}

$parms = @{'class'='Win32_BIOS';
           'computername'='SERVER-R2';
           'filter'='drivetype=3';
           'credential'='Administrator'
          }


Write-Host $test
Write-Host $test['key1']
```

First and Foremost, HELP

help → documentation on PowerShell built-in

Seem familiar to the **man** pages?

First thing you do after install…

Update the help!

Only minimal help is installed by default → this checks for installed packages or updates

```
PS C:\> update-help
```

(make sure to be running as Administrator/sudo)

Deploying help

What if you're in an enterprise environment?

Don't necessarily want **each** computer downloading

Possible to save help locally

```
PS C:\ Save-Help -DestinationPath \\path\to\destination\v4help -force
```

And then

```
PS C:\ Update-Help -SourcePath \\path\to\destination\v4help -force
```

Using help

PS C:\> get-help <service>

E.g., PS C:\> get-help *service

Can use wildcards

PS C:\> help get-service

PS C:\> help receive-job

Note that PowerShell isn't really case-sensitive

See examples of use:

PS C:\> help receive-job -Examples

Cmdlet (Command-let)

PowerShell's core unit of execution

- Compiled .NET commands
- Follow verb-noun structure
  - E.g., Get-Service, ForEach-Object, Get-Help, Copy-Item, etc.
- Use standard .NET verbs:
  - E.g., Set, Remove, Get
- Noun:
  - Singular "thing" you want
  - E.g., Item, Service, etc.

Parameters

Customize cmdlets

Cmdlet -ParameterName

```
PS C:\> get-service -Name adws -ComputerName emf-dc01
```

Lookup Active Directory Web Services on computer emf-dc01

Alias

```
PS C:\> get-process
PS C:\> ps
```

But not the same as Linux ps...just a shortcut to the Get-Process command

```
PS C:\> get-alias
```

Show all aliases

```
PS C:\> get-alias -name f*
```

Show all aliases starting with f

Variables

```
PS C:\> $a = 10
    PS C:\> echo $a   ///
    PS C:\> Write-Output $a  ///
    PS C:\> $a
```

```
PS C:\> $gs = get-service
PS C:\> $running = $gs | where-object {$_.status -eq
'running'}
```

$_ → current pipeline object (each service within $gs)

Conditionals

Comparison operators

-eq, -ne, -ge, -gt, -in, -notin, -lt, -le, -like, -notlike, -match, -notmatch, -contains, -notcontains, -is, -isnot

Logical operators

-and, -or, -xor, -not

```
$day = "Monday"
if ($day -eq "Monday")
{
  Write-Output "It's Monday!"
}
elseif ($day -eq "Tuesday" -OR $day -eq "Wednesday")
{
  Write-Output "Neat it's actually Tuesday or Wednesday!"
}
else
{ Write-Output "It's some other day" }
```

Comparison Operators

-eq, -ne, -ge, -gt, -lt, -le

General comparison operators

Can't use '=' instead of '-eq'

Or ==

-in, -notin

Check if range/array contains item

```
349 -in 300..400
```

-contains, -notcontains

Check if array contains item (not substring!)

```
$arr = "one","two","3","four"
$arr -contains "two"
```

Comparison Operators

-is, -isnot

Check type

```
if ($var -is "String")
```

-like, -notlike

Substring search with wildcards

```
"cit349" -like "cit*"
```

-match, -notmatch

Find substring

```
"this class is cit349" -match "is"
```

like vs. match

like: matches entire string
match: built-in regex

```
“welcome to cit349” -like “*cit349*”
“welcome to cit349” -like “cit349”
“welcome to cit349” -match “cit349”
“welcome to cit349” -match “*cit349*”
```

Comparison Operators

Want them to be case-sensitive?

Add a “c” prefix to any

-eq → -ceq

--like → clike

(Generally) case-insensitive by default, but to force it:

Prefix with “i”

-eq → -ieq

Switch

```
$day = “Monday”
switch ($day)
{
  { $_ -eq “Monday” } {Write-Output “Mon”; break}
  { $_ -eq “Tuesday” } {Write-Output “Tues”; break}
  default { Write-Output “Other” }
}
```

Loops

for

```
for ($i = 1; $i -le 10; $i++)
{
  Write-Output $i
}
```

foreach

```
foreach ($file in $files)
{
  Write-Output "Length of file: " $file.Length
}
```

Foreach-Object (cmdlet)

```
Get-ChildItem | Foreach-Object { "Length: " + $_.Length }
```

Loops

Enumerator (dictionaries)

```
$test = @{'key1'='value1';'key2'='value2'}
foreach ($h in $test.GetEnumerator()) {
  Write-Host "$($h.name): $($h.value)"
}
```

Loops

while

```
$output = “”;
while ($output -ne “QUIT”)
{
  $output = Read-Host “Type something“
}
```

Loops

```
$op = “”
```

do..while

```
do
{
    $op = Read-Host “Type again”
} while ($op -ne “QUIT”)
```

do..until

```
do
{
    $op = Read-Host “Type again again”
} until ($op -eq “QUIT”)
```

Functions

PowerShell also allows functions

- Acts like a cmdlet but is created within PowerShell
  - i.e., not compiled .NET code

- What functions are loaded?
  - PS C:\> get-command -CommandType function -ListImported

Defining a function

```
function fnName
{
  code...
}
…
fnName
```

(parameters later)

Modules

Package of PowerShell commands

Generally tied to some feature/role

Check installed: `PS C:\> get-module -listavailable`

Available commands: `PS C:\> get-command -module SmbShare`

Import: `PS C:\> import-module SmbShare`

Script Files

PowerShell scripts have a .ps1 extension

Running a script: PS C:\> script1.ps1

```
# script1.ps1
# Prints all files/folders under C:\Program Files
$all = Get-ChildItem "C:\Program Files"
Write-Output $all
```

Questions from last year:

1) What does ISE stand for

Integrated Scripting Environment

2) How to make an alias persist

Need a profile (~\Documents\WindowsPowerShell\profile.ps1)

Then create your alias inside

- Complex aliases require functions

Creating an alias: Set-Alias np c:\windows\notepad.exe

Execution Policy

Can set the Execution Policy for PowerShell

- User preference for security in PowerShell
  - Allowed to load configuration files?
  - Run scripts?
  - Digitally sign scripts?

PS> Set-ExecutionPolicy RemoteSigned

Execution Policies

Options:

Restricted : [default] – does not load configs / run scripts

AllSigned : all scripts/configs signed by verified publisher

RemoteSigned : all downloaded scripts/configs signed

Unrestricted : no restrictions – all scripts can be run

Warning/permission requested upon running downloaded scripts

Bypass : no restrictions – no warnings

Undefined : removes policy from scope

But not those set by Group Policy

Command-line Arguments

Use $args or param

```
#args.ps1
Write-Host “Number of arguments: “ $args.Length;
foreach ($arg in $args)
{
  Write-Host “Argument: $arg”;
}
```

Params

param must be first statement

```
#param.ps1
param([string]$foo = “foo”, [string]$bar = “bar”)
Write-Host “Argument: “ $foo
Write-Host “Argument: “ $bar
```

C:\> param.ps1 -foo fooarg -bar bararg
C:\> param.ps1
C:\> param.ps1 fooarg bararg

Param types

param([Parameter(Mandatory=$true)[type]$p1 = ,
[type]$p2 = , …)

(Also works inside functions)

```
function foo()
{
  param([string]$bar = “bar”)
  Write-Host “the arg: “ $bar
}
```

Param types (args-fxn.ps1)

```
function foo1()
{
  param([string]$bar = "bar")
  Write-Host "the arg, of course, was [" $bar "]"
}

function foo2($first, $second)
{
  Write-Host "This is the first [" $first "] and the second [" $second "]"
}

foo1
foo1 "NOT BAR HAHHA"

foo2
foo2 "first" 2
foo2 -second "THE SECOND"
```

Commands for your toolbelt (AD-related)

All mainly related to AD…so import module (requires AD Directory Services installed)

```
# Import AD module
Import-Module ActiveDirectory
# See what's available
get-command -module ActiveDirectory
```

http://windowsitpro.com/powershell/top-10-active-directory-tasks-solved-powershell

1) Reset user’s password

```
# Create variable with new password (to be encrypted)
$new_passwd = Read-Host “Enter new password” -AsSecureString

# Retrieve account and replace password
Set-AdAccountPassword temp_user -NewPassword $new_passwd

# Make user change at next login
Set-ADUser temp_user -ChangePasswordAtLogon $True
```

2) Disable/Enable an Account

```
PS C:\> Disable-ADAccount tempuser -whatif
  -whatif – simulate running command without running it

Opposite:
PS C:\> Enable-ADAccount tempuser

Enable all users from the 'Dev' department…
PS C:\> Get-ADUser -filter "department -eq 'Dev'" |
Enable-ADAccount
  (-whatif will help show you what will happen on a large scale
  as wel)
```

3/4) Unlock/Delete a user's account

User locks himself out?

```
PS> Unlock-ADAccount fredericks
```

Want to get rid of that user?

```
PS> Remove-ADUser fredericks -whatif
```

Add Members to Group

I want to be added to the cloudcomputing group..

```
PS> Add-ADGroupMember "cloudcomputing" -Members fredericks

Doesn't exist?
PS> New-ADGroup "cloudcomputing" -GroupScope DomainLocal
```

But what if I want to cull empty groups?

```
PS> Get-ADGroup -filter * | where {-Not ($_ |
Get-ADGroupMember)} | Select Name
```

(GetADGroupMember <name> shows users in group)

Demo 1

Let’s add all of you as users to our ActiveDirectory setup

1) Get list of user names from Moodle
2) Pull CSV into Powershell
3) Create user accounts for everybody

…

4) And then delete you all

Demo 2

You can…also write video games in Powershell…

https://blogs.technet.microsoft.com/josebda/2015/03/28/powershell-examples-adventure-house-game/

https://github.com/avdaredevil/SnakeMadNess/blob/master/AP-Snakes.ps1

In-Class Work

Next time, we're going to start off with a little PowerShell in-class assignment

Do a little bit of group coding

http://askubuntu.com/questions/343268/how-to-use-manual-partitioning-during-installation

http://askubuntu.com/questions/221835/installing-ubuntu-alongside-a-pre-installed-windows-with-uefi

https://ubuntuforums.org/showthread.php?t=2293266

PowerShell ISE

Commands at the ready
Easy debugger

Additional Reading

POWERSHELL FOR NEWBIES, Getting started with PowerShell 4.0, Jeffery Hicks
Windows PowerShell Cookbook (Ch4 seems to be free):
https://www.safaribooksonline.com/library/view/windows-powershell-cookbook/9781449359195/ch04.html
10 PowerShell Commands every Windows Admin should Know
http://www.techrepublic.com/blog/10-things/10-powershell-commands-every-windows-admin-should-know/
10 Active Directory Tasks Solved with PowerShell
http://windowsitpro.com/powershell/top-10-active-directory-tasks-solved-powershell
PowerShell Tutorial
https://blog.udemy.com/powershell-tutorial/