Understanding PowerShell

Required: PowerShell 3.0 or greater
(Some cmdlets are in PowerShell 2.0)
Windows Server and Active Directory Preferred

Peter McEldowney

Contents

Expanding Functionality and Finding cmdlets 2
Getting help in PowerShell and referencing syntax 5
Understanding Syntax to use Get-Help outputs 6
Working with the Event Log 7
Managing Windows Features and Roles in Windows Server 10
Managing Your Network Adapter 11
Working with variables 14
Working with text [strings and arrays] 15
Using a basic for loop 16
Understanding the ActiveDirectory module 18
    Managing the Default Domain Password Policy 18
    Creating new users 19
    Creating New Organizational Units 21
    Getting Information about AD objects 23

Objective: Introduce a basic and fundamental understanding of common PowerShell usage.

What is PowerShell?

PowerShell is Command Prompt (cmd) on steroids. It is designed to enable complex scripts that can manipulate your environment precisely. Not only can you use legacy executables that are native to Windows (like ipconfig or the net commands), but there are now modules that can be added to enhance capability and management potential. The individual commands within these modules, called cmdlets, will prove to be the easiest and most robust means to configure, automate, and monitor systems.

For more information on PowerShell, please check out Windows Powershell Cmdlets
http://technet.microsoft.com/en-us/scriptcenter/dd772285.aspx

For videos that demonstrate more PowerShell, check out Scripting with Windows PowerShell
http://technet.microsoft.com/en-us/scriptcenter/powershell.aspx

For Best Results: Manually type all commands into your PowerShell console.
This gets you comfortable navigating PowerShell.

[Anything after **Ex:** that is in *italics* can be entered into a PowerShell console window]

To learn how to run PowerShell efficiently using the keyboard, check out:

Ex: *Get-Help about_Line_Editing*

For an easier to read version, pipe the output to the *more* command. This will be explained later:
Ex: *Get-Help about_Line_Editing | more*

For more detailed and advanced information on advanced PowerShell functionality, check out the embedded help documentation. All of the help sections can be found using:

Ex: *Get-Help about*

Expanding Functionality and Finding cmdlets

PowerShell allows users to manipulate outputs to suit current needs. Take the *get-command* cmdlets; we can use it to find command-lets (cmdlets):

Ex: *Get-Command*

Notice how we used a verb and a noun/action. We can get the available verbs in PowerShell with:
[Notice how there is never a space between the verb and the noun]

Ex: *Get-Verb*

We can manipulate the output of the *Get-Command* cmdlet by specifying a verb. The next example shows how we can find all the cmdlets we have available to us that contain the verb *get*:

Ex: *Get-Command –Verb Get*

Notice how long the output is. We can use a command (actually an old command) called more. This command is pipe | capable (sometimes referred to as | piping and can typically be found below the backspace key with a shift). This means that it can take the output of another command as its input. Using this input, the command creates a scrolling document that can be incremented 1 line at a time or 1 page at a time. We will be using this frequently for making outputs easier to read because more is pipe capable.

Ex: *Get-Command –Verb Get | more*

We can narrow our results down even more by specifying a module. Take NetTCPIP:

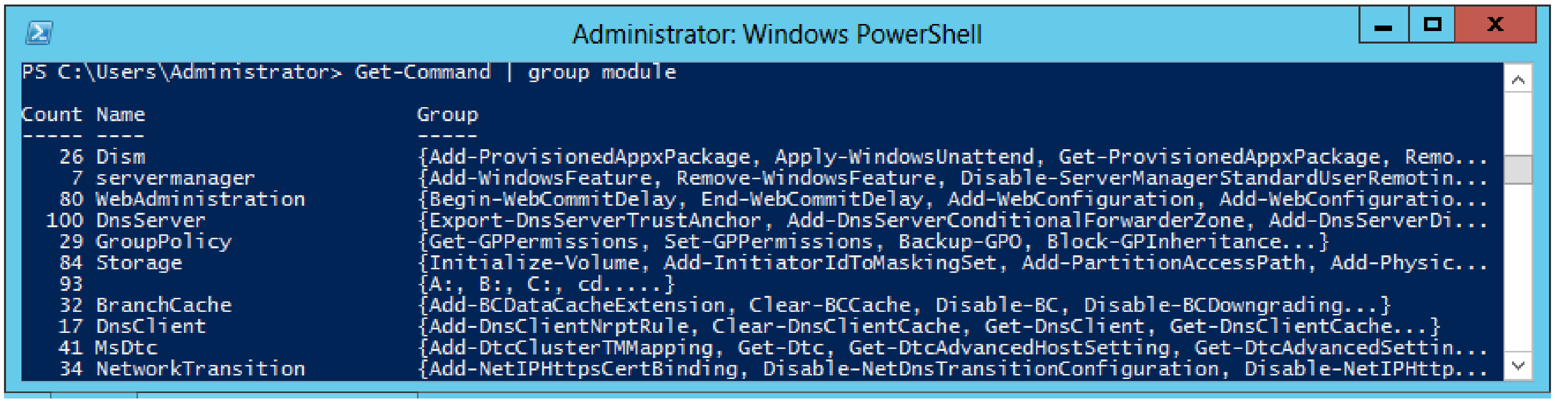
Ex: *Get-Command –Verb Get –Module NetTCPIP*

These arguments (or switches or options) are crucial. We can cycle through the available arguments by pressing *tab* after we enter a dash. This can be done for most cmdlet but if the module is not loaded that contains the cmdlets, then autofill will not work until the module is loaded (covered next).

Ex: *Get-Command* –[press tab immediately following the dash]

Similar to the legacy more command, many cmdlets are pipe capable. This is identified by being able to 'Accept Pipeline Input.' In the next example, we are finding a specific column and grouping objects by the text that is in that particular column. This will also tell us how many commands are available in each module.

Ex: *Get-Command |Group-Object module*



```
PS C:\Users\Administrator> Get-Command | group module

Count Name                      Group
----- ----                      -----
   26 Dism                      {Add-ProvisionedAppxPackage, Apply-WindowsUnattend, Get-ProvisionedAppxPackage, Remo...
    7 servermanager             {Add-WindowsFeature, Remove-WindowsFeature, Disable-ServerManagerStandardUserRemotin...
   80 WebAdministration         {Begin-WebCommitDelay, End-WebCommitDelay, Add-WebConfiguration, Add-WebConfiguratio...
  100 DnsServer                 {Export-DnsServerTrustAnchor, Add-DnsServerConditionalForwarderZone, Add-DnsServerDi...
   29 GroupPolicy               {Get-GPPermissions, Set-GPPermissions, Backup-GPO, Block-GPInheritance...}
   84 Storage                   {Initialize-Volume, Add-InitiatorIdToMaskingSet, Add-PartitionAccessPath, Add-Physic...
   93                           {A:, B:, C:, cd.....}
   32 BranchCache               {Add-BCDataCacheExtension, Clear-BCCache, Disable-BC, Disable-BCDowngrading...}
   17 DnsClient                 {Add-DnsClientNrptRule, Clear-DnsClientCache, Get-DnsClient, Get-DnsClientCache...}
   41 MsDtc                     {Add-DtcClusterTMMapping, Get-Dtc, Get-DtcAdvancedHostSetting, Get-DtcAdvancedSettin...
   34 NetworkTransition         {Add-NetIPHttpsCertBinding, Disable-NetDnsTransitionConfiguration, Disable-NetIPHttp...
```

[We can also use the *group* alias instead of typing the full cmdlet name *group-object*]

Now since we can find the cmdlets available in a module, we should know how to import modules into PowerShell so that we can do more. All the modules that are on our local machines have already been imported but look at /n software inc as an example, you can import SSH, SFTP, HTTP, web publishing, Instant Messaging, and more modules. These can be used for advanced functionality but if you install these cmdlets, you will notice that in the default Profile that is loaded with PowerShell, it adds a line that states *import-module netcmdlets* so that the module loads at when PowerShell loads. These cmdlets are not discussed beyond this section in this tutorial. [http://www.netcmdlets.com/download/]
The default Profile is loaded when PowerShell starts. This is located in your documents folder under the WindowsPowerShell folder in a file called Microsoft.PowerShell_profile.ps1
This file is loaded every time PowerShell loads and can be configured to alter prompts or set up aliases or however you want to configure your PowerShell console. This makes migrating settings from one PowerShell console to any PowerShell console simple because this file can be backed up.

Next, we should work with the idea that the same task can be accomplished many ways. For example, if we want to check for the servermanager module (that we will be using later), we can do this with:

Ex: *Get-Module –ListAvailable | where name –match servermanager*

One nice aspect of using *get-module* to find your cmdlets is that you can find out way more information about the module. Let us use another pipe capable cmdlet to help us called *format-list.*

Ex: *Get-Module –ListAvailable | where name –match servermanager | Format-List*

There are many other ways to format and since format is a verb, we can find all the commands that begin with format by using:

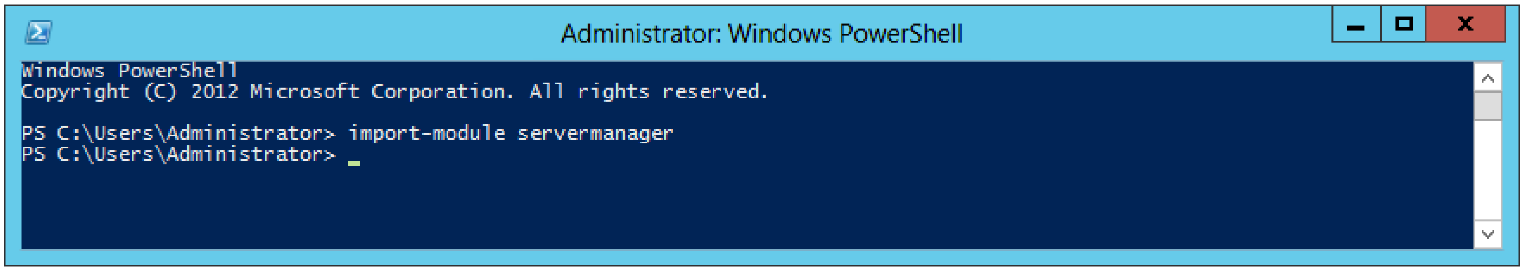
Ex: *Get-Command –Verb Format*

We will be using various *Format-* cmdlets throughout this introduction. We can view all the commands using that are available for a particular module with *get-command*:

Ex: *Get-Command –Module servermanager*

This module allows us to manage roles and features of Server from the PowerShell command line. If the servermanager module is not installed (ie. no output), you can import it by using:

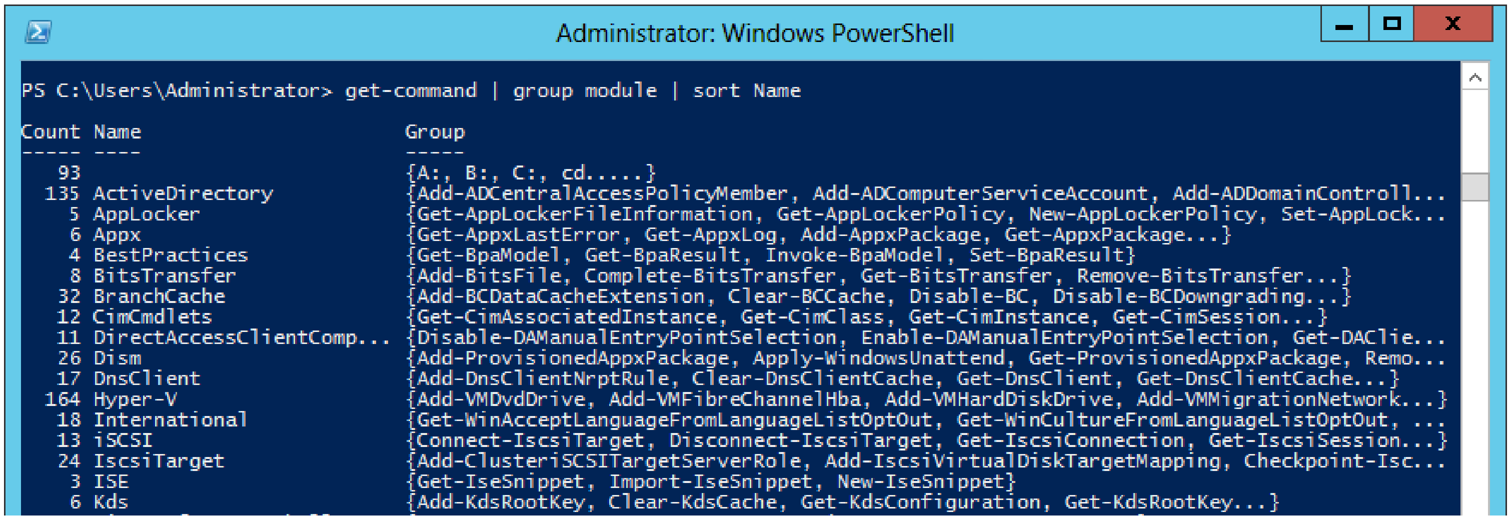
Ex: *Import-Module ServerManager*



```
Windows PowerShell
Copyright (C) 2012 Microsoft Corporation. All rights reserved.

PS C:\Users\Administrator> import-module servermanager
PS C:\Users\Administrator> _
```

The last command demonstrated how we can use the *get-command* cmdlet to find all of cmdlets that are associated with the servermanager module. The cool part about this is that we can use this for any module. For example, if you have the management tools installed with your roles, it should include PowerShell management cmdlets. Like earlier, let us take the output of a command and perform formatting by piping the output to pipe compatible cmdlets like the *sort-object* cmdlet.

Ex: *Get-Command –All| group module | sort name*



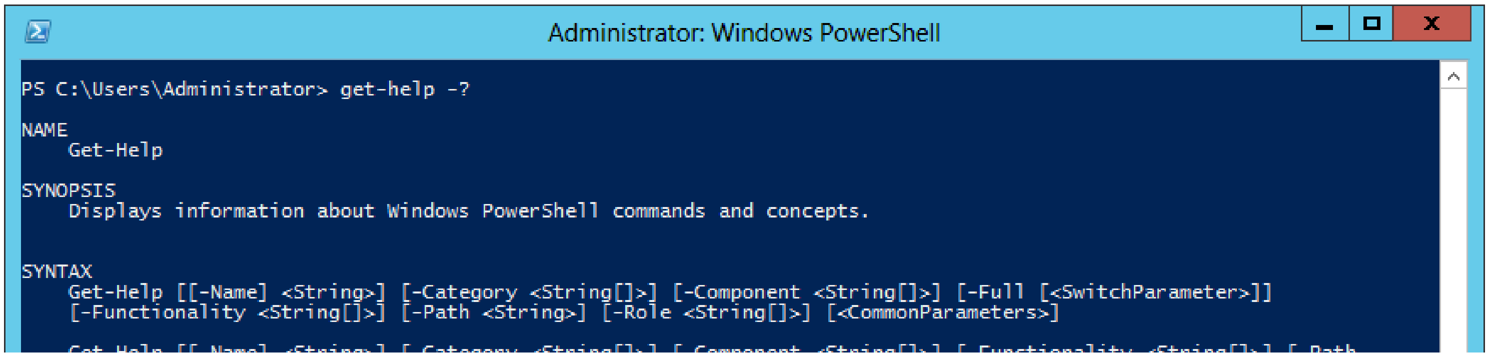
```
PS C:\Users\Administrator> get-command | group module | sort Name

Count Name                      Group
----- ----                      -----
   93                           {A:, B:, C:, cd.....}
  135 ActiveDirectory           {Add-ADCentralAccessPolicyMember, Add-ADComputerServiceAccount, Add-ADDomainControll...
    5 AppLocker                 {Get-AppLockerFileInformation, Get-AppLockerPolicy, New-AppLockerPolicy, Set-AppLock...
    6 Appx                      {Get-AppxLastError, Get-AppxLog, Add-AppxPackage, Get-AppxPackage...}
    4 BestPractices             {Get-BpaModel, Get-BpaResult, Invoke-BpaModel, Set-BpaResult}
    8 BitsTransfer              {Add-BitsFile, Complete-BitsTransfer, Get-BitsTransfer, Remove-BitsTransfer...}
   32 BranchCache               {Add-BCDataCacheExtension, Clear-BCCache, Disable-BC, Disable-BCDowngrading...}
   12 CimCmdlets                {Get-CimAssociatedInstance, Get-CimClass, Get-CimInstance, Get-CimSession...}
   11 DirectAccessClientComp... {Disable-DAManualEntryPointSelection, Enable-DAManualEntryPointSelection, Get-DAClie...
   26 Dism                      {Add-ProvisionedAppxPackage, Apply-WindowsUnattend, Get-ProvisionedAppxPackage, Remo...
   17 DnsClient                 {Add-DnsClientNrptRule, Clear-DnsClientCache, Get-DnsClient, Get-DnsClientCache...}
  164 Hyper-V                   {Add-VMDvdDrive, Add-VMFibreChannelHba, Add-VMHardDiskDrive, Add-VMMigrationNetwork...}
   18 International             {Get-WinAcceptLanguageFromLanguageListOptOut, Get-WinCultureFromLanguageListOptOut, ...
   13 iSCSI                     {Connect-IscsiTarget, Disconnect-IscsiTarget, Get-IscsiConnection, Get-IscsiSession...}
   24 IscsiTarget               {Add-ClusteriSCSITargetServerRole, Add-IscsiVirtualDiskTargetMapping, Checkpoint-Isc...
    3 ISE                       {Get-IseSnippet, Import-IseSnippet, New-IseSnippet}
    6 Kds                       {Add-KdsRootKey, Clear-KdsCache, Get-KdsConfiguration, Get-KdsRootKey...}
```

Notice how we used the *–All* switch. There are other switches we can use too. We can find these extra switches by using the following command (switches are just options specified after the command to customize our commands). In legacy command prompt, the order of the switches matters. In PowerShell, the switches only matter if the switch parameter is required [in other words, the text of the switch is in square brackets] but what the switch needs as an input (aka the data type) is not surrounded by [square brackets]. You can also find out required switch by using the *–full* switch with the get-help cmdlet or checking out the *–online* TechNet for a cmdlet, explained on the next page.

Getting help in PowerShell and referencing syntax

Just like in the old days of command prompt, *-? (dash- question mark?)* is available for quick syntax reference. This will bring up the *get-help* for *get-help* (ie. *Get-Help Get-Help*).

Ex: *Get-Help -?*


Administrator: Windows PowerShell

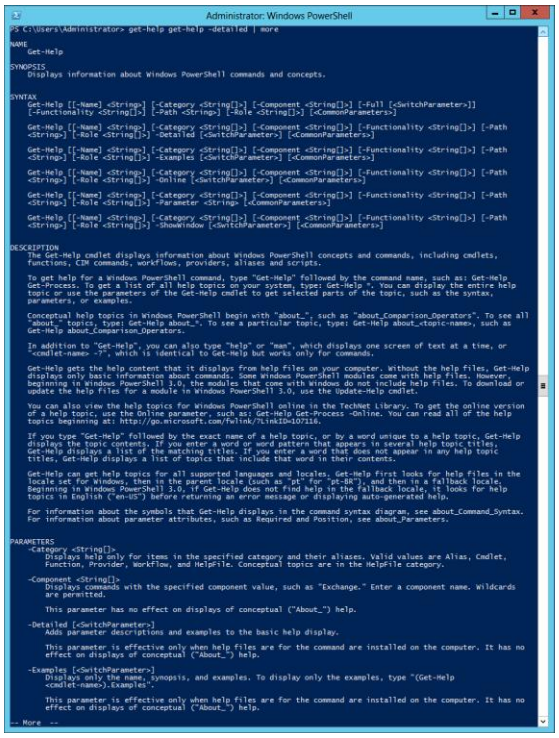
```
PS C:\Users\Administrator> get-help -?

NAME
    Get-Help

SYNOPSIS
    Displays information about Windows PowerShell commands and concepts.


SYNTAX
    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>] [-Full [<SwitchParameter>]]
    [-Functionality <String[]>] [-Path <String>] [-Role <String[]>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>] [-Functionality <String[]>] [-Path
```

If we check out the help of the *get-command* cmdlet, we can see that there are no required parameters. If we look at the *get-help* output, we can see that the *get-help* cmdlets does require a string.

Ex: *Get-Help Get-Command*

Whenever you do not know the syntax for a command, you can use the *get-help* cmdlet. The original *get-help* cmdlet tends to have a lots of switches and can be very confusing, luckily the *–detailed* argument will explain the function of each argument. These can be your best friend or worst enemy, depending on how much you like reading. Some can be longer and short are short and sweet.
You cannot get to the detailed help section using the *-?* so if you're going for speed, create an alias.

Ex: *Get-Help Get-Command –detailed | more*


Administrator: Windows PowerShell

```
PS C:\Users\Administrator> get-help get-help -detailed | more

NAME
    Get-Help

SYNOPSIS
    Displays information about Windows PowerShell commands and concepts.


SYNTAX
    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>] [-Full [<SwitchParameter>]]
    [-Functionality <String[]>] [-Path <String>] [-Role <String[]>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>] [-Functionality <String[]>] [-Path
    <String>] [-Role <String[]>] -Detailed [<SwitchParameter>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>] [-Functionality <String[]>] [-Path
    <String>] [-Role <String[]>] -Examples [<SwitchParameter>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>] [-Functionality <String[]>] [-Path
    <String>] [-Role <String[]>] -Online [<SwitchParameter>] [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>] [-Functionality <String[]>] [-Path
    <String>] [-Role <String[]>] -Parameter <String> [<CommonParameters>]

    Get-Help [[-Name] <String>] [-Category <String[]>] [-Component <String[]>] [-Functionality <String[]>] [-Path
    <String>] [-Role <String[]>] -ShowWindow [<SwitchParameter>] [<CommonParameters>]


DESCRIPTION
    The Get-Help cmdlet displays information about Windows PowerShell concepts and commands, including cmdlets,
    functions, CIM commands, workflows, providers, aliases and scripts.

    To get help for a Windows PowerShell command, type "Get-Help" followed by the command name, such as: Get-Help
    Get-Process. To get a list of all help topics on your system, type: Get-Help *. You can display the entire help
    topic or use the parameters of the Get-Help cmdlet to get selected parts of the topic, such as the syntax,
    parameters, or examples.

    Conceptual help topics in Windows PowerShell begin with "about_", such as "about_Comparison_Operators". To see all
    "about_" topics, type: Get-Help about_*. To see a particular topic, type: Get-Help about_<topic-name>, such as
    Get-Help about_Comparison_Operators.

    In addition to "Get-Help", you can also type "help" or "man", which displays one screen of text at a time, or
    "<cmdlet-name> -?", which is identical to Get-Help but works only for commands.

    Get-Help gets the help content that it displays from help files on your computer. Without the help files, Get-Help
    displays only basic information about commands. Some Windows PowerShell modules come with help files. However,
    beginning in Windows PowerShell 3.0, the modules that come with Windows do not include help files. To download or
    update the help files for a module in Windows PowerShell 3.0, use the Update-Help cmdlet.

    You can also view the help topics for Windows PowerShell online in the TechNet Library. To get the online version
    of a help topic, use the Online parameter, such as: Get-Help Get-Process -Online. You can read all of the help
    topics beginning at: http://go.microsoft.com/fwlink/?LinkID=107116.

    If you type "Get-Help" followed by the exact name of a help topic, or by a word unique to a help topic, Get-Help
    displays the topic contents. If you enter a word or word pattern that appears in several help topic titles,
    Get-Help displays a list of the matching titles. If you enter a word that does not appear in any help topic
    titles, Get-Help displays a list of topics that include that word in their contents.

    Get-Help can get help topics for all supported languages and locales. Get-Help first looks for help files in the
    locale set for Windows, then in the parent locale (such as "pt" for "pt-BR"), and then in a fallback locale.
    Beginning in Windows PowerShell 3.0, if Get-Help does not find help in the fallback locale, it looks for help
    topics in English ("en-US") before returning an error message or displaying auto-generated help.

    For information about the symbols that Get-Help displays in the command syntax diagram, see about_Command_Syntax.
    For information about parameter attributes, such as Required and Position, see about_Parameters.


PARAMETERS
    -Category <String[]>
        Displays help only for items in the specified category and their aliases. Valid values are Alias, Cmdlet,
        Function, Provider, Workflow, and HelpFile. Conceptual topics are in the HelpFile category.

    -Component <String[]>
        Displays commands with the specified component value, such as "Exchange." Enter a component name. Wildcards
        are permitted.

        This parameter has no effect on displays of conceptual ("About_") help.

    -Detailed [<SwitchParameter>]
        Adds parameter descriptions and examples to the basic help display.

        This parameter is effective only when help files are for the command are installed on the computer. It has no
        effect on displays of conceptual ("About_") help.

    -Examples [<SwitchParameter>]
        Displays only the name, synopsis, and examples. To display only the examples, type "(Get-Help
        <cmdlet-name>).Examples".

        This parameter is effective only when help files are for the command are installed on the computer. It has no
        effect on displays of conceptual ("About_") help.

-- More  --
```

Want a little more information directly from TechNet about the cmdlet that you are inquiring about? Use the *–online* argument. This is especially nice for understanding exactly what a cmdlet can have passed to it (whether by pipe or just what the cmdlet is expecting after that specific argument.
[It even opens in another window]

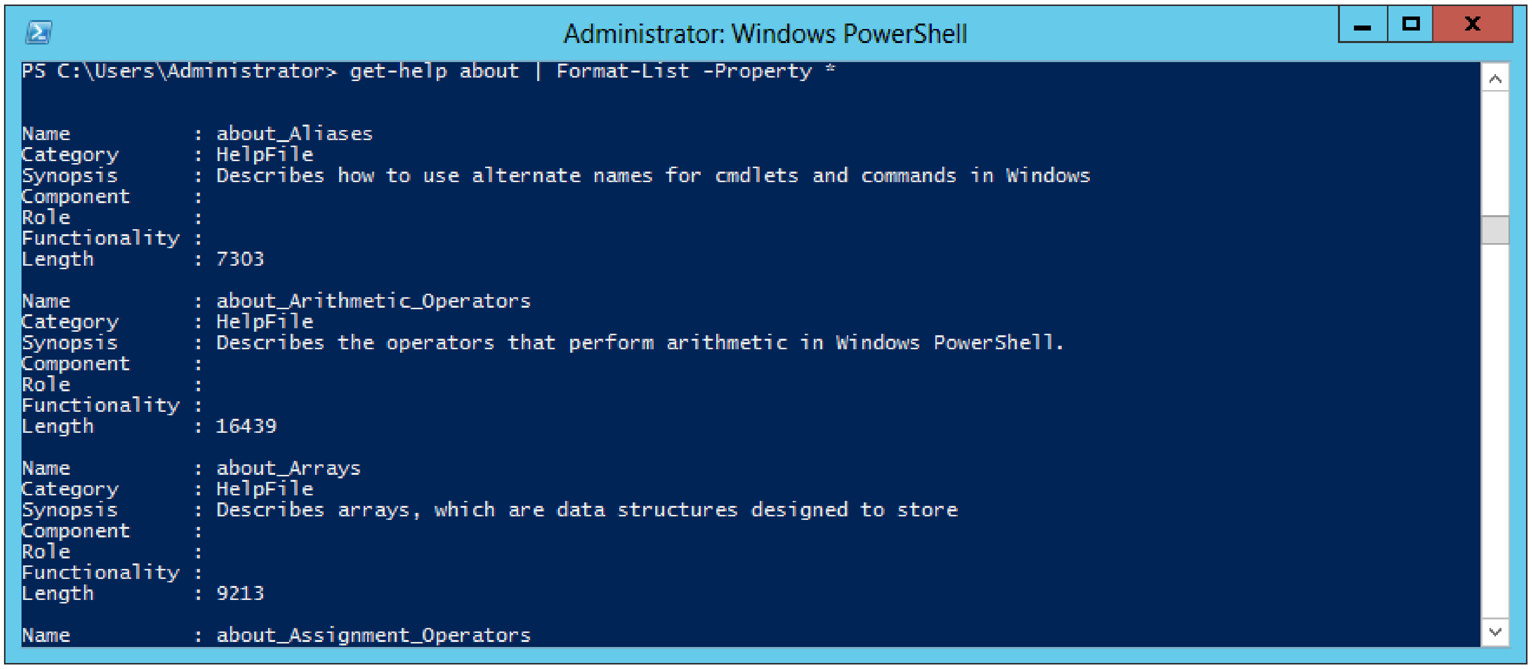
Ex: *Get-Help Get-Module -online*

Another awesome part about PowerShell is the simplicity of finding comprehensive explanations of everything scripting. Check out some of the available about sections:

Ex: *Get-Help about | more*

If you would like a detailed reading of this page, please use:

Ex: *Get-Help about | Format-List –Property **



```
PS C:\Users\Administrator> get-help about | Format-List -Property *


Name          : about_Aliases
Category      : HelpFile
Synopsis      : Describes how to use alternate names for cmdlets and commands in Windows
Component     :
Role          :
Functionality :
Length        : 7303

Name          : about_Arithmetic_Operators
Category      : HelpFile
Synopsis      : Describes the operators that perform arithmetic in Windows PowerShell.
Component     :
Role          :
Functionality :
Length        : 16439

Name          : about_Arrays
Category      : HelpFile
Synopsis      : Describes arrays, which are data structures designed to store
Component     :
Role          :
Functionality :
Length        : 9213

Name          : about_Assignment_Operators
```

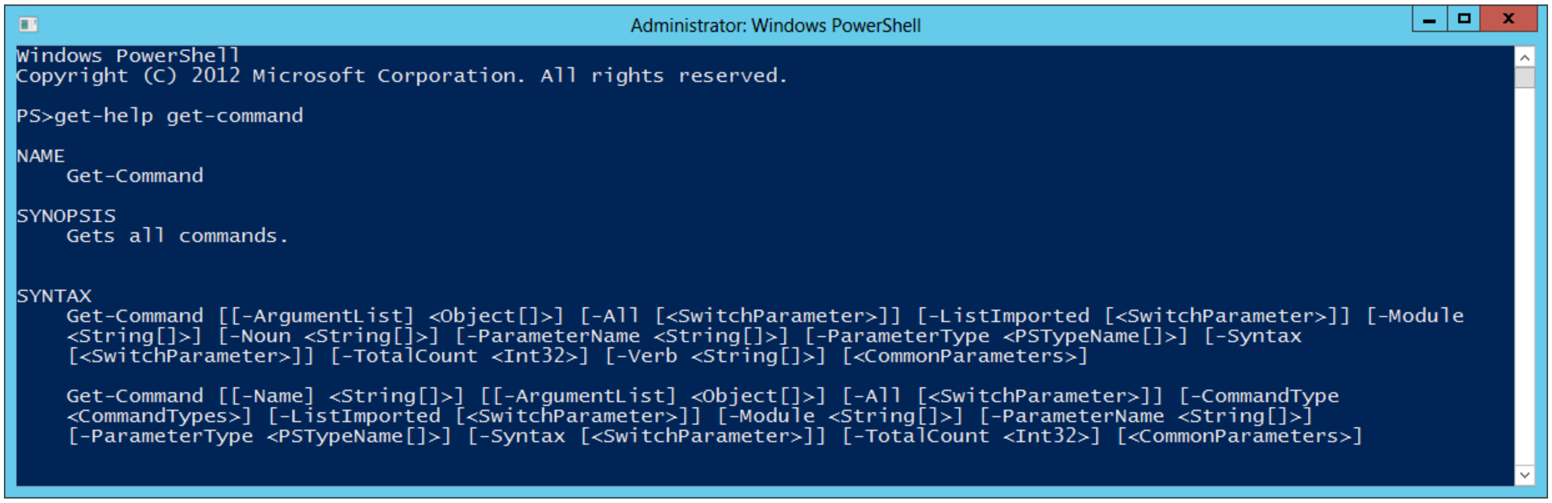
Notice the immense number of entries. Each entry is detailed and very informative. Check out this entry about Aliases. Within the first paragraph or 2, it explains how to make your own alias. [An example of an alias, you could type *ip* to get to *ipconfig* with the alias below the help documentation]

Ex: *get-help about_Alias | more*

Ex: *New-Alias –Name ip –Value ipconfig*

Understanding Syntax to use Get-Help outputs

When viewing the *get-help* documentation that is associated with many cmdlets, you may notice a section titled Syntax. Here is what the *get-help get-command* Syntax output looks like:

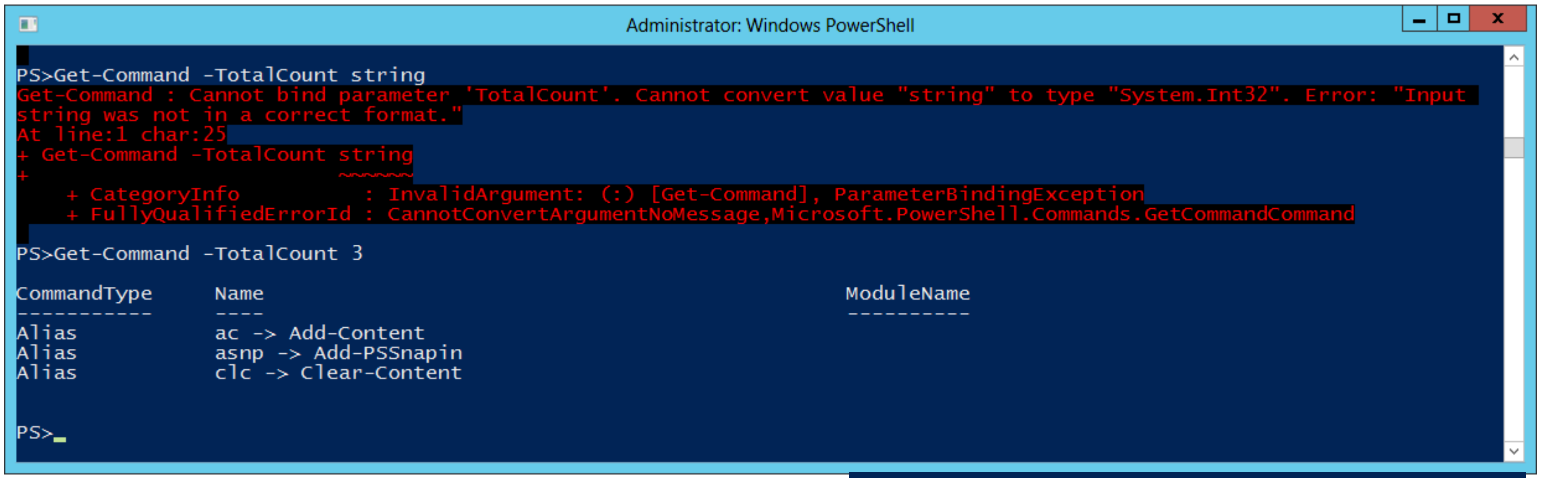


```
Windows PowerShell
Copyright (C) 2012 Microsoft Corporation. All rights reserved.

PS>get-help get-command

NAME
    Get-Command

SYNOPSIS
    Gets all commands.


SYNTAX
    Get-Command [[-ArgumentList] <Object[]>] [-All [<SwitchParameter>]] [-ListImported [<SwitchParameter>]] [-Module
    <String[]>] [-Noun <String[]>] [-ParameterName <String[]>] [-ParameterType <PSTypeName[]>] [-Syntax
    [<SwitchParameter>]] [-TotalCount <Int32>] [-Verb <String[]>] [<CommonParameters>]

    Get-Command [[-Name] <String[]>] [[-ArgumentList] <Object[]>] [-All [<SwitchParameter>]] [-CommandType
    <CommandTypes>] [-ListImported [<SwitchParameter>]] [-Module <String[]>] [-ParameterName <String[]>]
    [-ParameterType <PSTypeName[]>] [-Syntax [<SwitchParameter>]] [-TotalCount <Int32>] [<CommonParameters>]
```

Anything enclosed in [Square Brackets] means optional. In other words, the cmdlet will work without that argument passed/typed to the cmdlet, but you can tell the cmdlet to run in a particular manner, for a particular item, or even formatting outputs with specific arguments passed to (or typed after) the cmdlet.

As well, with most of the arguments, there is a data type that is enclosed in a less than <and a greater than> symbol. Data types are the formatting of the data that is expected by the cmdlet. For example, <Int32> means that a whole number (or integer) is expected. <String> means that basic text is expected. Strings are best passed with quotation marks around them. This is because a space is interpreted as another argument being passed where quotation marks mean that it is all part of the same string.

This means that we can ensure that the data being passed to the cmdlet is syntactically and structurally correct. If the data type is incorrect, you will receive an error telling you so, like the example below.

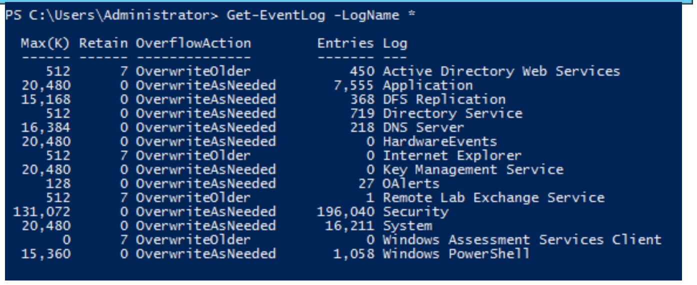


```
PS>Get-Command -TotalCount string
Get-Command : Cannot bind parameter 'TotalCount'. Cannot convert value "string" to type "System.Int32". Error: "Input
string was not in a correct format."
At line:1 char:25
+ Get-Command -TotalCount string
+                         ~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-Command], ParameterBindingException
    + FullyQualifiedErrorId : CannotConvertArgumentNoMessage,Microsoft.PowerShell.Commands.GetCommandCommand

PS>Get-Command -TotalCount 3

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           ac -> Add-Content
Alias           asnp -> Add-PSSnapin
Alias           clc -> Clear-Content


PS>_
```

Working with the Event Log

Try the following command.

Ex: *Get-EventLog –LogName **



```
PS C:\Users\Administrator> Get-EventLog -LogName *

  Max(K) Retain OverflowAction        Entries Log
  ------ ------ --------------        ------- ---
     512      7 OverwriteOlder            450 Active Directory Web Services
  20,480      0 OverwriteAsNeeded       7,555 Application
  15,168      0 OverwriteAsNeeded         368 DFS Replication
     512      0 OverwriteAsNeeded         719 Directory Service
  16,384      0 OverwriteAsNeeded         218 DNS Server
  20,480      0 OverwriteAsNeeded           0 HardwareEvents
     512      7 OverwriteOlder              0 Internet Explorer
  20,480      0 OverwriteAsNeeded           0 Key Management Service
     128      0 OverwriteAsNeeded          27 OAlerts
     512      7 OverwriteOlder              1 Remote Lab Exchange Service
 131,072      0 OverwriteAsNeeded     196,040 Security
  20,480      0 OverwriteAsNeeded      16,211 System
       0      7 OverwriteOlder              0 Windows Assessment Services Client
  15,360      0 OverwriteAsNeeded       1,058 Windows PowerShell
```

The previous command says to get all log names in the event log. This output will tell you what Logs are available for you to access, the size of the log, and the behavior of the log as it acquires entries as the log reaches its maximum size limitations. Next, we can try:

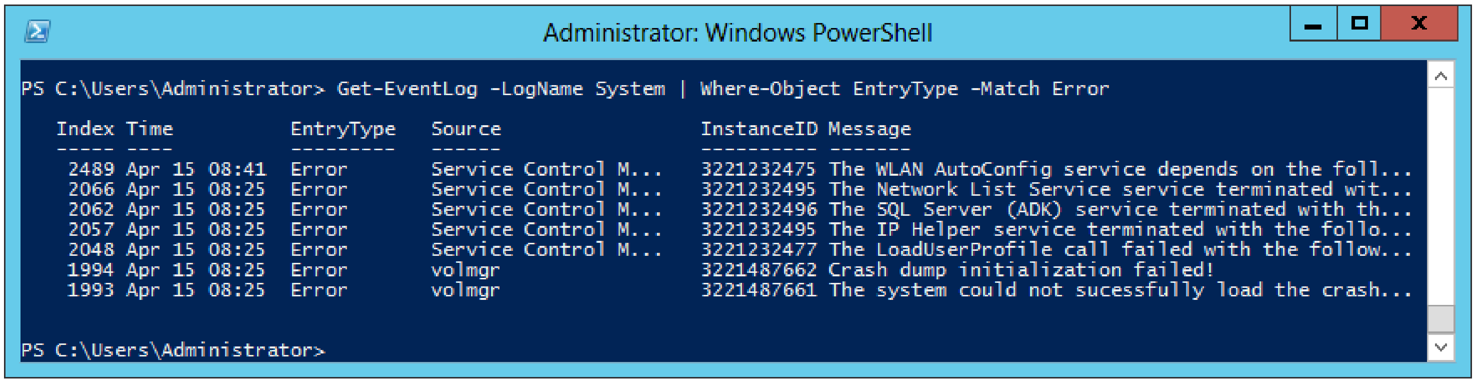
Ex: *Get-EventLog –LogName System*

As we can see, we get a significant amount of information. Looking at the *get-help*, we can find ways to limit the results based on particular items. To find an entry after a particular date, use the *–After mm/dd/yy* argument. Visit *get-help get-eventlog –detailed* for more information.

Ex: *Get-EventLog –LogName System –After 6/21/13*

This can generate a very large output. However, we can also limit our results by creating a stipulation that the value in the column must match in order to be displayed.
[Note: You must press enter on a blank line to run the command.]

Ex: *Get-EventLog –LogName System –After 6/21/13 `*
| Where-Object EntryType –Match Error



Administrator: Windows PowerShell

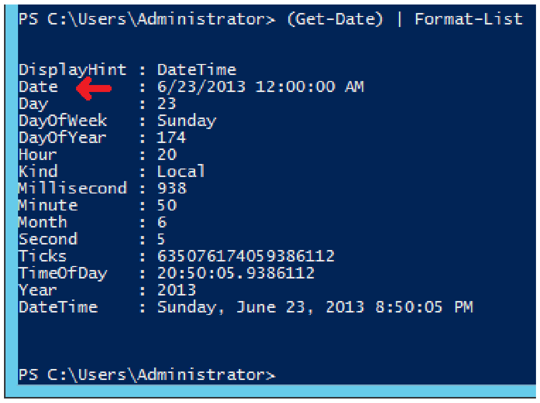
```
PS C:\Users\Administrator> Get-EventLog -LogName System | Where-Object EntryType -Match Error

   Index Time          EntryType   Source                 InstanceID Message
   ----- ----          ---------   ------                 ---------- -------
    2489 Apr 15 08:41  Error       Service Control M...   3221232475 The WLAN AutoConfig service depends on the foll...
    2066 Apr 15 08:25  Error       Service Control M...   3221232495 The Network List Service service terminated wit...
    2062 Apr 15 08:25  Error       Service Control M...   3221232496 The SQL Server (ADK) service terminated with th...
    2057 Apr 15 08:25  Error       Service Control M...   3221232495 The IP Helper service terminated with the follo...
    2048 Apr 15 08:25  Error       Service Control M...   3221232477 The LoadUserProfile call failed with the follow...
    1994 Apr 15 08:25  Error       volmgr                 3221487662 Crash dump initialization failed!
    1993 Apr 15 08:25  Error       volmgr                 3221487661 The system could not sucessfully load the crash...


PS C:\Users\Administrator>
```

Now we can better analyze events.
Where-Object has an alias of *where* native to PowerShell.

Say we wanted to clean up the output of the event log. We can pipe the entire output to the *select-object* cmdlets, where it will select individual columns for us. Try the following cmdlets to manipulate your output for the information you want.

[Check out *get-help select-object –detailed* for more information]. Just like before, we can shorten *select-object* to *select*. We can do this for a couple of pipe capable cmdlets, but not all.

Ex: *Get-EventLog –LogName System –After (Get-Date).Date `*
| Where EntryType –match Error `
| Select Source, Message `
| Format-List



```
PS C:\Users\Administrator> (Get-Date) | Format-List

DisplayHint : DateTime
Date        : 6/23/2013 12:00:00 AM
Day         : 23
DayOfWeek   : Sunday
DayOfYear   : 174
Hour        : 20
Kind        : Local
Millisecond : 938
Minute      : 50
Month       : 6
Second      : 5
Ticks       : 635076174059386112
TimeOfDay   : 20:50:05.9386112
Year        : 2013
DateTime    : Sunday, June 23, 2013 8:50:05 PM


PS C:\Users\Administrator>
```

There are a couple of important items to note with these last examples. First, we have tick marks at the ` end of each row. This means that the same line continues on the next line. This is great for scripting because it makes reading the script significantly easier.

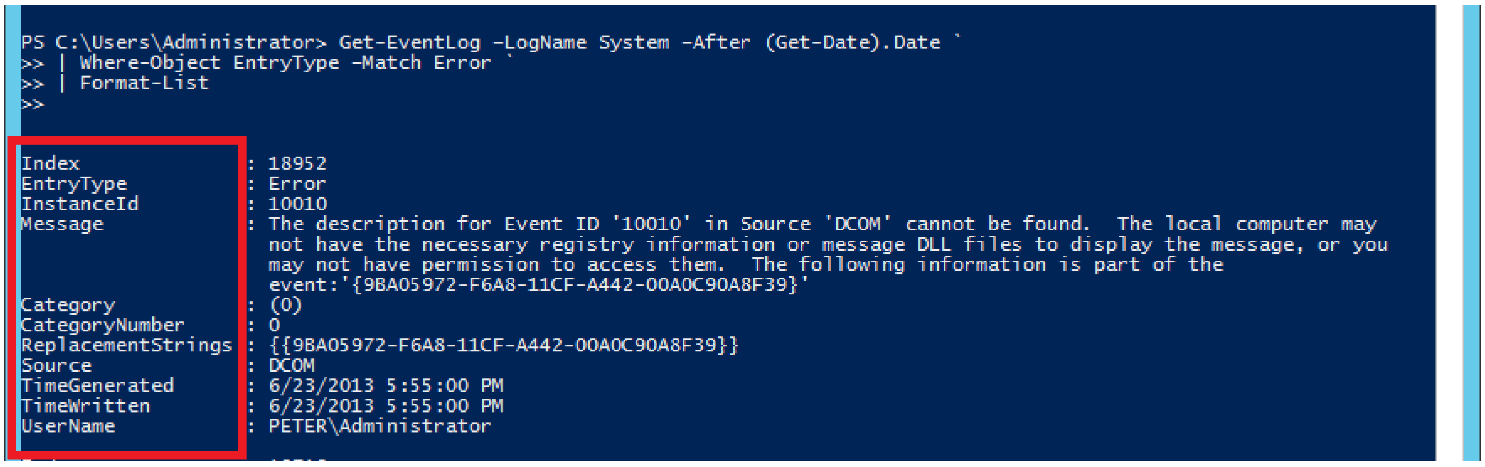
Another item to note is the *(Get-Date).Date* portion of the command. If you run this, you will see that this pull just the date from the *Get-Date* cmdlet. If we take a closer look at the *Get-Date* cmdlet, we can see that it contains valuable information. The Date.

Ex: *Get-Date | Format-List*

If we want to pull a specific value from the formatted output of *Get-Date*, we can do that by appending a period. to *(Get-Date).* and specify the item title. We want todays date with no time so we use *(Get-Date).Date*
We could use any name that is in the left hand column. This goes for the output of any command that we surround in (parenthesis).

We can also use *Select-Object* similarly to that. The only difference is that *Select-Object* displays a column title where the method (previously).mentioned does not store column names.

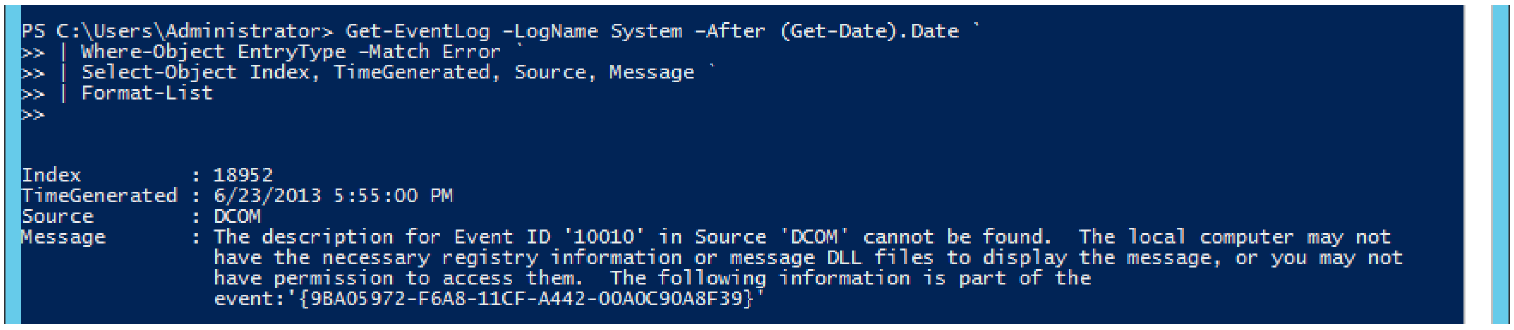
We can *select* any of the strings surrounded by the red box:



```
PS C:\Users\Administrator> Get-EventLog -LogName System -After (Get-Date).Date `
>> | Where-Object EntryType -Match Error `
>> | Format-List
>>

Index              : 18952
EntryType          : Error
InstanceId         : 10010
Message            : The description for Event ID '10010' in Source 'DCOM' cannot be found.  The local computer may
                     not have the necessary registry information or message DLL files to display the message, or you
                     may not have permission to access them.  The following information is part of the
                     event:'{9BA05972-F6A8-11CF-A442-00A0C90A8F39}'
Category           : (0)
CategoryNumber     : 0
ReplacementStrings : {{9BA05972-F6A8-11CF-A442-00A0C90A8F39}}
Source             : DCOM
TimeGenerated      : 6/23/2013 5:55:00 PM
TimeWritten        : 6/23/2013 5:55:00 PM
UserName           : PETER\Administrator
```

Ex:

Get-EventLog –LogName System –After (Get-Date).Date `
| Where-Object EntryType –Match Error `
| Select-Object Index, TimeGenerated, Source, Message `
| Format-List



```
PS C:\Users\Administrator> Get-EventLog -LogName System -After (Get-Date).Date `
>> | Where-Object EntryType -Match Error `
>> | Select-Object Index, TimeGenerated, Source, Message `
>> | Format-List
>>

Index         : 18952
TimeGenerated : 6/23/2013 5:55:00 PM
Source        : DCOM
Message       : The description for Event ID '10010' in Source 'DCOM' cannot be found.  The local computer may not
                have the necessary registry information or message DLL files to display the message, or you may not
                have permission to access them.  The following information is part of the
                event:'{9BA05972-F6A8-11CF-A442-00A0C90A8F39}'
```

What the previous command says is that we want to get all entries in the Event Log named 'System' and anywhere where the 'EntryType' has a match to the word 'Error,' we are going to select (or display) the Index, TimeGenerated, Source, and Message columns. We are then going to format it as a list.

For a more detailed and advanced EventLog usage documentation, check out:

Ex: *Get-Help about_EventLogs | more*

Managing Windows Features and Roles in Windows Server

Microsoft Windows Server almost always includes the servermanager module for Windows but I have encountered machines where it is not native. If server manager is not installed, use the second **Ex:** below to import it. The first **Ex:** will return available cmdlets.

Ex: *get-command –module servermanager*

Ex: *import-module servermanager*

The commands that the servermanager module includes are all that you need to manage Windows features and roles (yes, this includes in Server Core, if you have PowerShell installed).

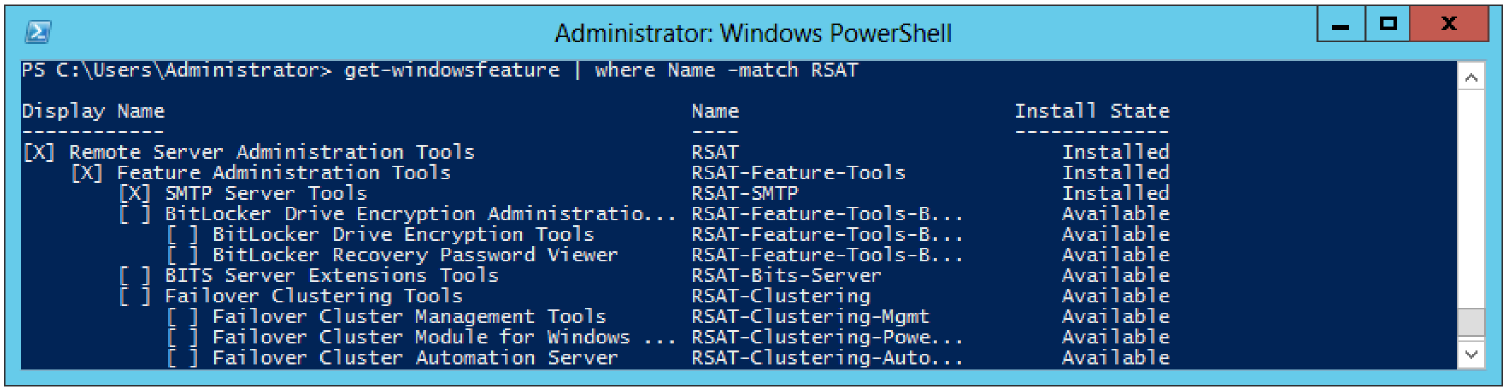
Try the following command to list all of the Windows Features that are available.

Ex: *Get-WindowsFeature*

Using this will allow you to see all of the many available features on your installation of Windows Server. To narrow your search, try something along the lines of:

Ex: *Get-WindowsFeature | where Name –match RSAT*

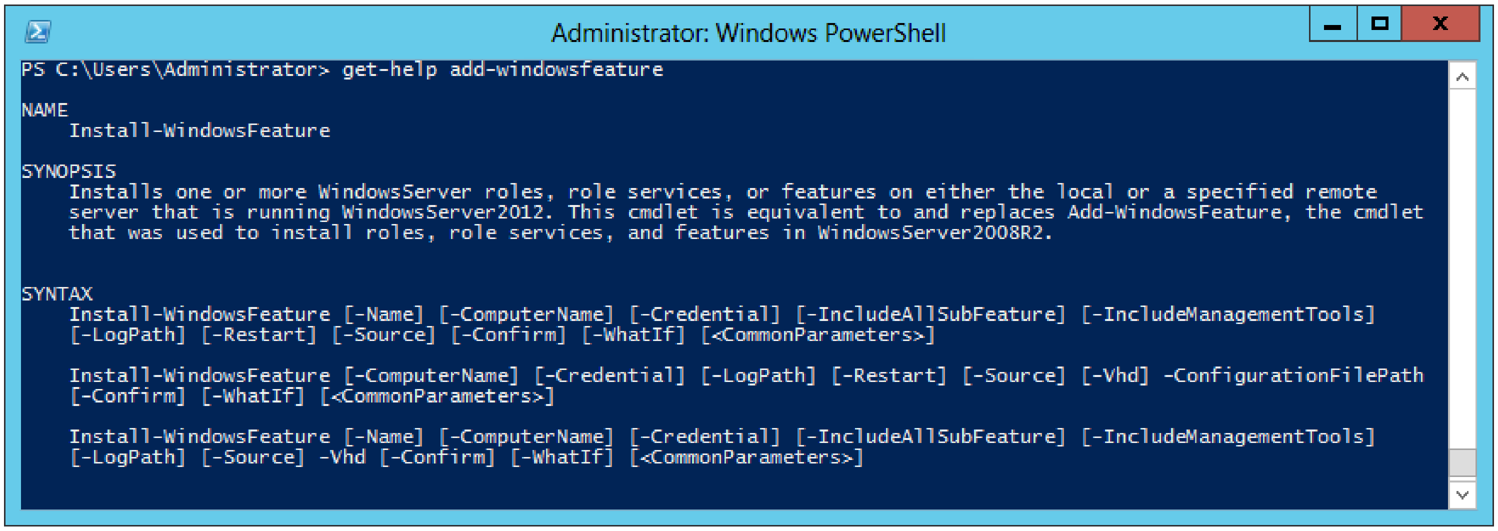
This is telling Windows to list all available features (installed and not installed) that contain a match to the letters RSAT.



```
PS C:\Users\Administrator> get-windowsfeature | where Name -match RSAT

Display Name                                            Name                       Install State
------------                                            ----                       -------------
[X] Remote Server Administration Tools                  RSAT                           Installed
    [X] Feature Administration Tools                    RSAT-Feature-Tools             Installed
        [X] SMTP Server Tools                           RSAT-SMTP                      Installed
        [ ] BitLocker Drive Encryption Administratio... RSAT-Feature-Tools-B...        Available
            [ ] BitLocker Drive Encryption Tools        RSAT-Feature-Tools-B...        Available
            [ ] BitLocker Recovery Password Viewer      RSAT-Feature-Tools-B...        Available
        [ ] BITS Server Extensions Tools                RSAT-Bits-Server               Available
        [ ] Failover Clustering Tools                   RSAT-Clustering                Available
            [ ] Failover Cluster Management Tools       RSAT-Clustering-Mgmt           Available
            [ ] Failover Cluster Module for Windows ... RSAT-Clustering-Powe...        Available
            [ ] Failover Cluster Automation Server      RSAT-Clustering-Auto...        Available
```

To add any of the features found with the *Get-WindowsFeature* cmdlets, you can use the *Add-WindowsFeature* cmdlets (or *Install-WindowsFeature*, since *Add-WindowsFeature* is simply an Alias). We can prove this by using the following command:

Ex: *Get-Help Add-WindowsFeature*



```
PS C:\Users\Administrator> get-help add-windowsfeature

NAME
    Install-WindowsFeature

SYNOPSIS
    Installs one or more WindowsServer roles, role services, or features on either the local or a specified remote
    server that is running WindowsServer2012. This cmdlet is equivalent to and replaces Add-WindowsFeature, the cmdlet
    that was used to install roles, role services, and features in WindowsServer2008R2.


SYNTAX
    Install-WindowsFeature [-Name] [-ComputerName] [-Credential] [-IncludeAllSubFeature] [-IncludeManagementTools]
    [-LogPath] [-Restart] [-Source] [-Confirm] [-WhatIf] [<CommonParameters>]

    Install-WindowsFeature [-ComputerName] [-Credential] [-LogPath] [-Restart] [-Source] [-Vhd] -ConfigurationFilePath
    [-Confirm] [-WhatIf] [<CommonParameters>]

    Install-WindowsFeature [-Name] [-ComputerName] [-Credential] [-IncludeAllSubFeature] [-IncludeManagementTools]
    [-LogPath] [-Source] -Vhd [-Confirm] [-WhatIf] [<CommonParameters>]
```

The *Add-WindowsFeature* cmdlet has an abundance of handy switches. For example, we can append *–IncludeAllSubFeature* to install everything that is indented under a particular feature. We can also use *–IncludeManagementTools* to make sure that the MMC snap-in and the scripting cmdlets are available. We can also use this if the MMC was not installed during the initial installation of the Role or Feature. For example, to install Hyper-V with the cmdlets and mmc, you can use:

Ex: *Add-WindowsFeature Hyper-V –IncludeManagementTools*

As well, if we wanted to add or remove (*remove-windowsfeature*) features like the GUI, we can do it by using the following command:

Ex: *Add-WindowsFeature User-Interfaces-Infra –IncludeAllSubFeature*

This will also install Desktop-Experience, which will make the GUI aspect of Windows Server more like Windows Desktop by allow much more customization (like, right click on the Desktop and selecting Personalize). As well, you could take a Full GUI installation of Windows, configure it to your liking, and remove the GUI to improve server efficiency by using the following command:

Ex: *Remove-WindowsFeature User-Interfaces-Infra –IncludeManagementTools*

Note how I included the *–IncludeManagementTools* switch but not the *–IncludeAllSubFeature* switch. This is because the *–IncludeAllSubFeature* is implied when removing roles but sometimes you might not want to remove the management tools so that they can be used to manage remote machines.

Managing Your Network Adapter

The first aspect of managing your Network Adapter requires knowing what is available for you to configure. There is a large variety of modules included in PowerShell 3.0 to help you configure your Network:

Ex: *Get-Module –ListAvailable | Where name –match "net"*

The best way to determine what a module can perform is to explore the commands.

Ex: *Get-Command –Module NetAdapter*
Get-Command –Module NetTCPIP
Get-Command –Module NetworkTransition

There is an abundance of advanced functionality that PowerShell 3.0 has added so check it out and if you need help with syntax, use the *Get-Help* cmdlets. For example, the *Get-NetAdapter* cmdlets, that is part of the *NetAdapter* module can pull information about the physical medium of the device, link speed, interface index, and much more.

Ex: *Get-NetAdapter*

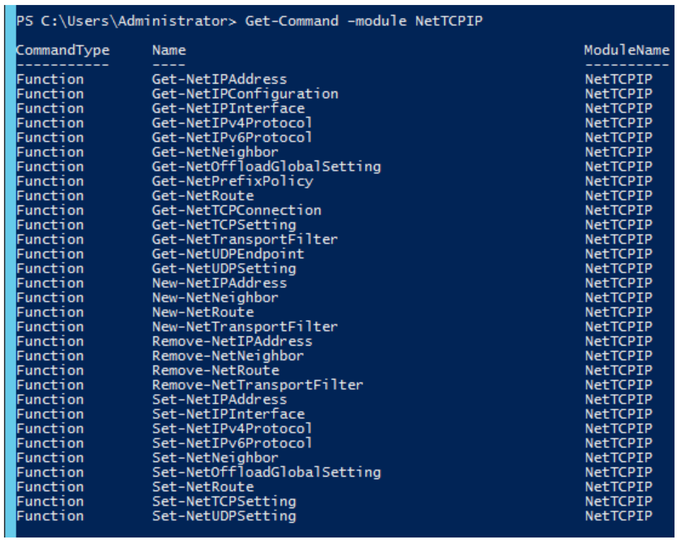
For the actually configuring a network adapter, many cmdlets will allow you to specify an interface index so that you do not have to remember the name of the interface. The *Get-NetAdapter* cmdlet will display the ifIndex (or InterfaceIndex) for the installed Network Adapters.

As well, there are many switches we can use to manipulate the *Get-NetAdapter* cmdlet. For example, we can use the *–Physical* or the *–IncludeHidden* switches to filter by devices.

Ex: *Get-NetAdapter –Physical*
Ex: *Get-NetAdapter –IncludeHidden*

We can also use the *Format-List* cmdlets so that the output has more room to display.

Ex: *Get-NetAdapter | Format-List*



```
PS C:\Users\Administrator> Get-Command -module NetTCPIP

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Get-NetIPAddress                                   NetTCPIP
Function        Get-NetIPConfiguration                             NetTCPIP
Function        Get-NetIPInterface                                 NetTCPIP
Function        Get-NetIPv4Protocol                                NetTCPIP
Function        Get-NetIPv6Protocol                                NetTCPIP
Function        Get-NetNeighbor                                    NetTCPIP
Function        Get-NetOffloadGlobalSetting                        NetTCPIP
Function        Get-NetPrefixPolicy                                NetTCPIP
Function        Get-NetRoute                                       NetTCPIP
Function        Get-NetTCPConnection                               NetTCPIP
Function        Get-NetTCPSetting                                  NetTCPIP
Function        Get-NetTransportFilter                             NetTCPIP
Function        Get-NetUDPEndpoint                                 NetTCPIP
Function        Get-NetUDPSetting                                  NetTCPIP
Function        New-NetIPAddress                                   NetTCPIP
Function        New-NetNeighbor                                    NetTCPIP
Function        New-NetRoute                                       NetTCPIP
Function        New-NetTransportFilter                             NetTCPIP
Function        Remove-NetIPAddress                                NetTCPIP
Function        Remove-NetNeighbor                                 NetTCPIP
Function        Remove-NetRoute                                    NetTCPIP
Function        Remove-NetTransportFilter                          NetTCPIP
Function        Set-NetIPAddress                                   NetTCPIP
Function        Set-NetIPInterface                                 NetTCPIP
Function        Set-NetIPv4Protocol                                NetTCPIP
Function        Set-NetIPv6Protocol                                NetTCPIP
Function        Set-NetNeighbor                                    NetTCPIP
Function        Set-NetOffloadGlobalSetting                        NetTCPIP
Function        Set-NetRoute                                       NetTCPIP
Function        Set-NetTCPSetting                                  NetTCPIP
Function        Set-NetUDPSetting                                  NetTCPIP
```

Once we know which network adapter we want to configure, let us look at what we can do through the NetTCPIP module.

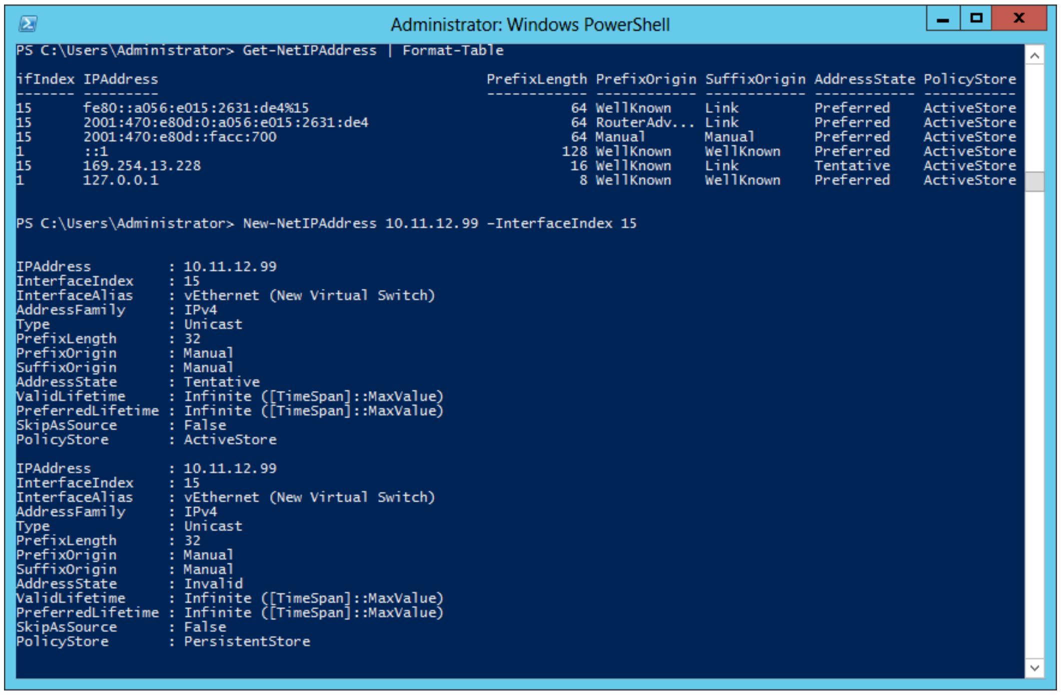
Ex: *Get-Command –module NetTCPIP*

These are the cmdlets to manage the local network setting of the machine. We can get, set, or create a new routing table entry; we can get, set, or create a new IP address for an adapter. We can also remove any of these. The best way to use these is to reference the embedded help documentation.

Ex: *New-NetIPAddress -?*

The only required field for the *New-NetIPAddress* cmdlets is an IP address and an Interface Alias or Interface Index (since you can use either, they have 2 separate syntax listings). The Interface Alias is just the name of the adapter. Let's try using the Interface Index (remember, this is machine specific).

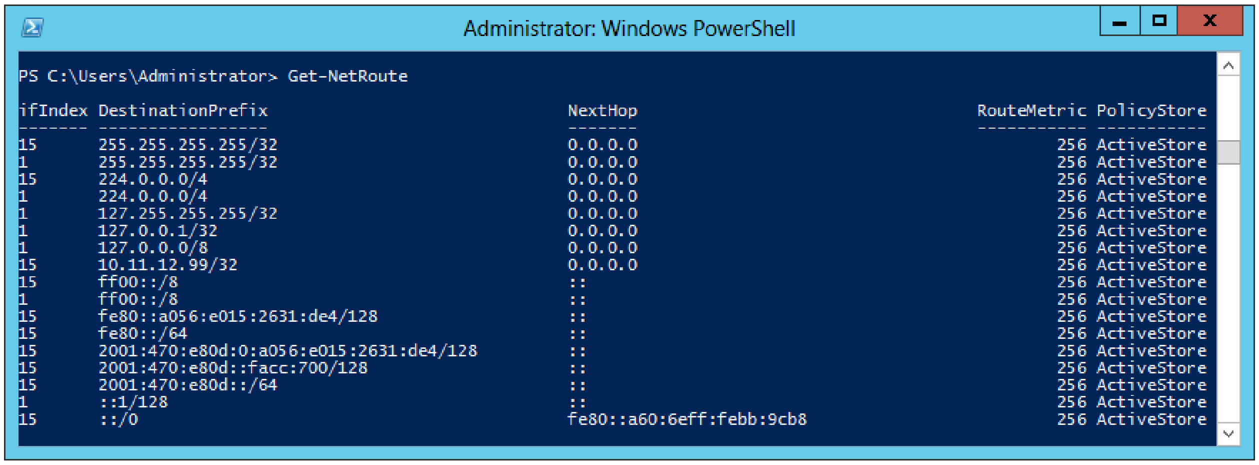
Ex: *New-NetIPAddress 10.11.12.99 –InterfaceIndex 15*



```
PS C:\Users\Administrator> Get-NetIPAddress | Format-Table

ifIndex IPAddress                                       PrefixLength PrefixOrigin SuffixOrigin AddressState PolicyStore
------- ---------                                       ------------ ------------ ------------ ------------ -----------
15      fe80::a056:e015:2631:de4%15                               64 WellKnown    Link         Preferred    ActiveStore
15      2001:470:e80d:0:a056:e015:2631:de4                        64 RouterAdv... Link         Preferred    ActiveStore
15      2001:470:e80d::facc:700                                   64 Manual       Manual       Preferred    ActiveStore
1       ::1                                                      128 WellKnown    WellKnown    Preferred    ActiveStore
15      169.254.13.228                                            16 WellKnown    Link         Tentative    ActiveStore
1       127.0.0.1                                                  8 WellKnown    WellKnown    Preferred    ActiveStore


PS C:\Users\Administrator> New-NetIPAddress 10.11.12.99 -InterfaceIndex 15


IPAddress         : 10.11.12.99
InterfaceIndex    : 15
InterfaceAlias    : vEthernet (New Virtual Switch)
AddressFamily     : IPv4
Type              : Unicast
PrefixLength      : 32
PrefixOrigin      : Manual
SuffixOrigin      : Manual
AddressState      : Tentative
ValidLifetime     : Infinite ([TimeSpan]::MaxValue)
PreferredLifetime : Infinite ([TimeSpan]::MaxValue)
SkipAsSource      : False
PolicyStore       : ActiveStore

IPAddress         : 10.11.12.99
InterfaceIndex    : 15
InterfaceAlias    : vEthernet (New Virtual Switch)
AddressFamily     : IPv4
Type              : Unicast
PrefixLength      : 32
PrefixOrigin      : Manual
SuffixOrigin      : Manual
AddressState      : Invalid
ValidLifetime     : Infinite ([TimeSpan]::MaxValue)
PreferredLifetime : Infinite ([TimeSpan]::MaxValue)
SkipAsSource      : False
PolicyStore       : PersistentStore
```

Notice how it creates 2 entries. An ActiveStore entry and a PersistentStore entry. The Persistent means that it will persist across reboots where the Active means that it will only store that entry for the active windows session (ie. a reboot will remove that entry).

PowerShell also makes reading and managing the Routing table much easier.

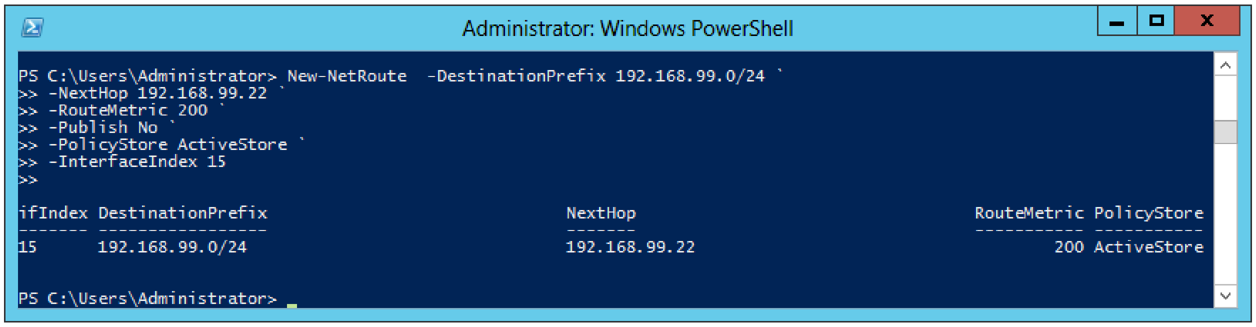


```
PS C:\Users\Administrator> Get-NetRoute

ifIndex DestinationPrefix                                NextHop                             RouteMetric PolicyStore
------- -----------------                                -------                             ----------- -----------
15      255.255.255.255/32                               0.0.0.0                                     256 ActiveStore
1       255.255.255.255/32                               0.0.0.0                                     256 ActiveStore
15      224.0.0.0/4                                      0.0.0.0                                     256 ActiveStore
1       224.0.0.0/4                                      0.0.0.0                                     256 ActiveStore
1       127.255.255.255/32                               0.0.0.0                                     256 ActiveStore
1       127.0.0.1/32                                     0.0.0.0                                     256 ActiveStore
1       127.0.0.0/8                                      0.0.0.0                                     256 ActiveStore
15      10.11.12.99/32                                   0.0.0.0                                     256 ActiveStore
15      ff00::/8                                         ::                                          256 ActiveStore
1       ff00::/8                                         ::                                          256 ActiveStore
15      fe80::a056:e015:2631:de4/128                     ::                                          256 ActiveStore
15      fe80::/64                                        ::                                          256 ActiveStore
15      2001:470:e80d:0:a056:e015:2631:de4/128           ::                                          256 ActiveStore
15      2001:470:e80d::facc:700/128                      ::                                          256 ActiveStore
15      2001:470:e80d::/64                               ::                                          256 ActiveStore
1       ::1/128                                          ::                                          256 ActiveStore
15      ::/0                                             fe80::a60:6eff:febb:9cb8                    256 ActiveStore
```

If we quickly reference the Syntax with *Set-NetRoute -?* we can see that we do not need any configuration options to run the cmdlets successfully. However, if we do not specify anything, there will be no entry made into the routing table.
[Since this command has so many configurable options, we will use a tick ` again to span multiple lines]

Ex:

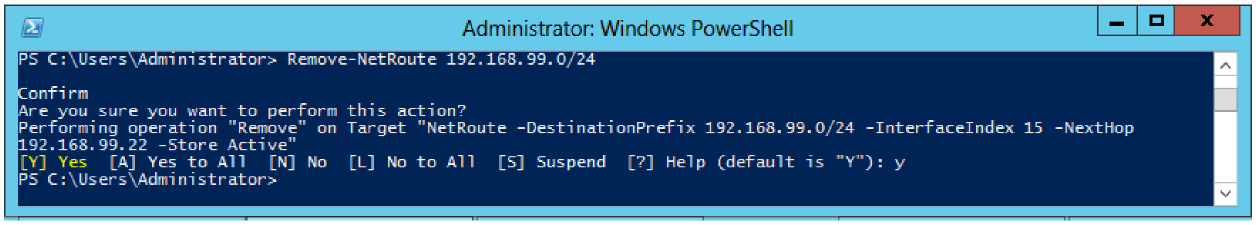
```
New-NetRoute  -DestinationPrefix 192.168.99.0/24 `
              -NextHop 192.168.99.22 `
              -RouteMetric 200 `
              -Publish No `
              -PolicyStore ActiveStore `
              -InterfaceIndex 15
```



```
PS C:\Users\Administrator> New-NetRoute  -DestinationPrefix 192.168.99.0/24 `
>> -NextHop 192.168.99.22 `
>> -RouteMetric 200 `
>> -Publish No `
>> -PolicyStore ActiveStore `
>> -InterfaceIndex 15
>>

ifIndex DestinationPrefix                              NextHop                                  RouteMetric PolicyStore
------- -----------------                              -------                                  ----------- -----------
15      192.168.99.0/24                                192.168.99.22                                    200 ActiveStore

PS C:\Users\Administrator> _
```

Removing a Routing Table entry is really easy. You just need *Remove-NetRoute* [Prefix to remove].

Ex: *Remove-NetRoute 192.168.99.0/24*



```
PS C:\Users\Administrator> Remove-NetRoute 192.168.99.0/24

Confirm
Are you sure you want to perform this action?
Performing operation "Remove" on Target "NetRoute -DestinationPrefix 192.168.99.0/24 -InterfaceIndex 15 -NextHop
192.168.99.22 -Store Active"
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
PS C:\Users\Administrator>
```

The best way to learn the NetTCPIP module for PowerShell is to use it. Remember the module and you can always search for Syntax.

Working with variables

Variables are indicated by a $before a word. We use variables to store information. For example, say we wanted to store a bunch of text (a string of text) for a particular variable.

Ex: *$a = 'bob,patricia,robert,brenda,pony,what,neverman,test,netboot'*

Now if we type in *$a* into PowerShell, it will return our list. We can store anything we want in variable. This can be very handy if we can to prompt a user for credentials and store it in a variable. Try this:

Ex: *$cred = Get-Credential*

Say we want to open an elevated Command Prompt with our stored credentials. If we entered our credentials correctly, we will successfully open an elevated cmd window.

Ex: *Start-Process cmd –Credential $cred*

If you want to learn more about the properties of the variables that we have defined, use the *Get-Member* cmdlet:

Ex: *$cred | Get-Member –MemberType Property*

This shows that we have 2 things stored in the properties of this variable, a string for the username and a SecureString for the password.

Working with text [strings and arrays]

Say we have this variable defined and we want to call a specific letter from the string. We can do this by using a means of calling items from an array of items that we have defined:

Ex: *$a = "bob,patricia,robert,brenda,pony,what,neverman,test,netboot"*

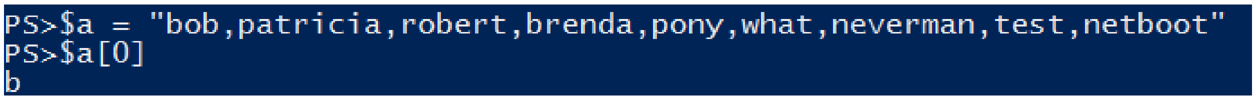
We can also measure the length of strings by appending .Length to any defined variable or a string.

Ex: *$a = "bob,patricia,robert,brenda,pony,what,neverman,test,netboot"*
$a.Length

or *"bob,patricia,robert,brenda,pony,what,neverman,test,netboot".Length*

We can also call the letters of a string based on the position in the string. For example, if we wanted to get the letter in the very first position (position index starts at 0), we can get b by using:

Ex: *$a[0]*



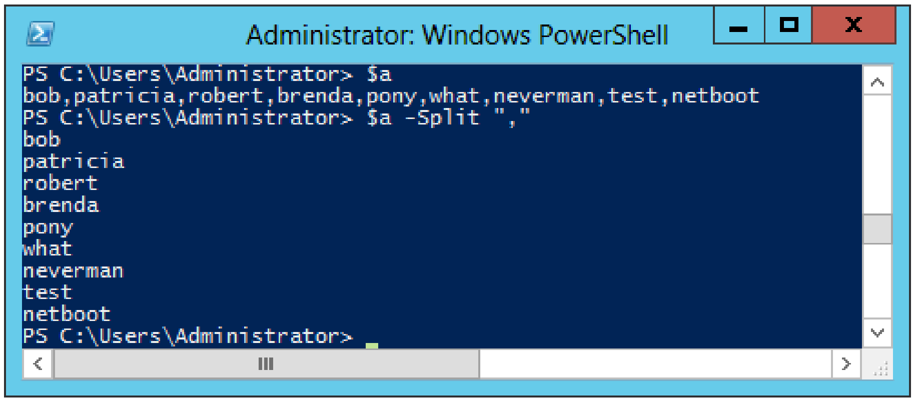
```
PS>$a = "bob,patricia,robert,brenda,pony,what,neverman,test,netboot"
PS>$a[0]
b
```

PowerShell has the ability to manipulate strings. For example, say we want to format the text defined earlier as *$a*.

Ex: *$a –Split ","*

Or we can replace our variable *$a* with this newly formatted list (which is now an array). All we need to do is redefine $a with our split switch.

Ex: *$a = $a –Split ","*



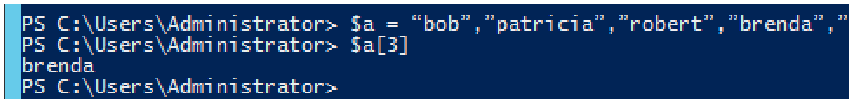
```
PS C:\Users\Administrator> $a
bob,patricia,robert,brenda,pony,what,neverman,test,netboot
PS C:\Users\Administrator> $a -Split ","
bob
patricia
robert
brenda
pony
what
neverman
test
netboot
PS C:\Users\Administrator> _
```

Now if we call *$a* we can see that our list is formatted better. We have also now transformed our 1 string into multiple strings. This is called an array. We can also accomplish this by inputting each string surrounded by quotation marks, separated by a comma.

Ex: *$a = "bob","patricia","robert","brenda","pony","what","neverman","test","netboot"*

This is called an array because we have multiple terms (strings) defined under the same variable. Now if we call the variable, they appear on their respective lines. We can also call a specific string from our array of strings, based on its position. For example, if we wanted to print the name Brenda on the screen, we could use *$a[3]* because Brenda is the 4th name on the list (we start counting at 0):

Ex: *$a[3]*



```
PS C:\Users\Administrator> $a = "bob","patricia","robert","brenda","
PS C:\Users\Administrator> $a[3]
brenda
PS C:\Users\Administrator>
```

If we wanted to pull up the 1st item on our list, it would resemble:

Ex: *$a[0]*

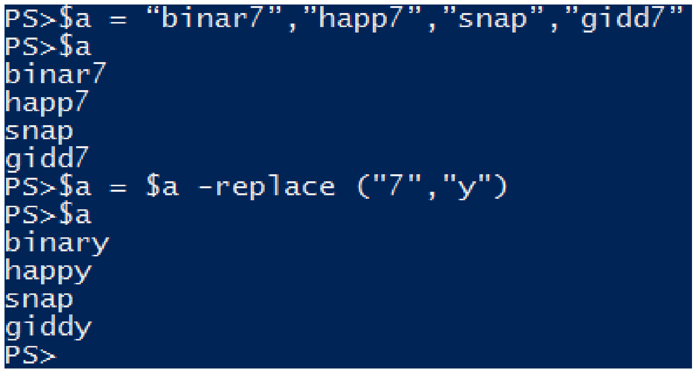
What the [square] brackets after the variable say is that we want to call the first entry on our list. However, splitting to create arrays or just to format text is not the only manipulation we can perform on text. We can also replace text. Say we wanted to replace text. We can simply append the replace switch at the end, and if formatted correctly, we can replace text.
For example, say we are typing quickly and have a happy of typing 7 instead of y when defining our variables. We can replace characters instead of having to retype all of our text.

Ex: *$a = "binar7","happ7","snap","gidd7"*

Let us re-define our variable while replacing 7 with y.

Ex: *$a = $a –replace ("7","y")*

Now when we call *$a*, we get our originally intended list. We can also output this array to a text file. Remember that > means overwrite a file and >> means append to the end of a file.



```
PS>$a = "binar7","happ7","snap","gidd7"
PS>$a
binar7
happ7
snap
gidd7
PS>$a = $a -replace ("7","y")
PS>$a
binary
happy
snap
giddy
PS>
```

Ex: *$a > C:\users_list.txt*

Or say we have a text file and want to use it as an array that can be called by a variable. Let us take the previously outputted file and assign it to the variable using the *Get-Content* cmdlet.

Ex: *$b = Get-Content C:\users_list.txt*

Now we can call contents of a text file based on line number that we want to enumerate from. For example, say we have a text file that has usernames all on separate lines.

Ex: *"John Doe","John Smith","Frank Zappa","Bob Saget" > C:\test_users.txt*
Ex: *(Get-Content C:\test_users.txt)[2]*

In the next section, you will see how we can use a for loop to count.

Using a basic for loop

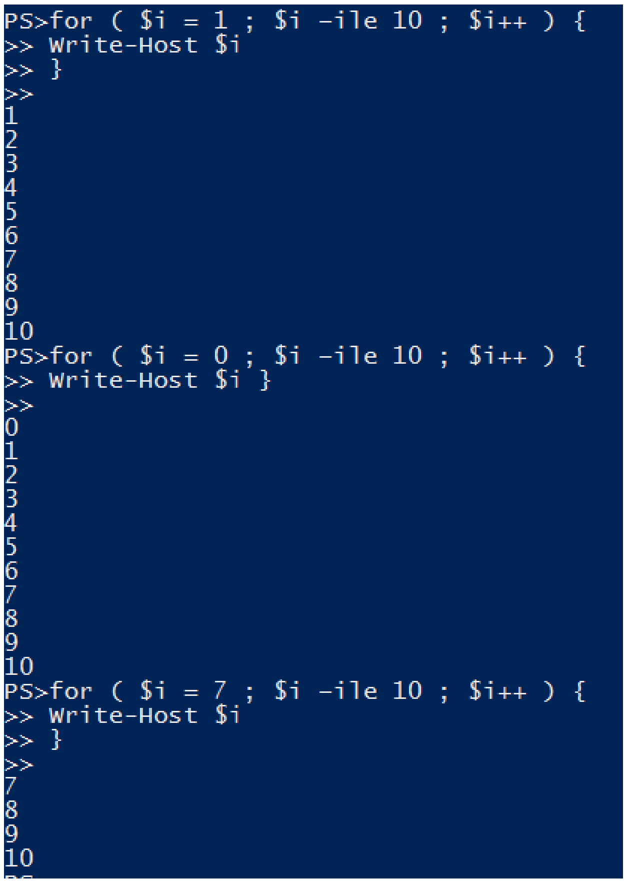
A for loop can be used for performing many tasks. This makes repetitive tasks much simpler. This is done by counting. We can count by any increment we want. This can be especially handy when creating multiple users that follow a similar naming convention.

Let's take a look at a basic for loop. The following loop will count to 10, starting at 1.

[Note: if you copy and paste this into your PowerShell window, you will have to press enter on a blank line to proceed, just like with a tick `]

Ex:

for ($i = 1 ; $i –ile 10 ; $i++) {
Write-Host $i
}



```
PS>for ( $i = 1 ; $i -ile 10 ; $i++ ) {
>> Write-Host $i
>> }
>>
1
2
3
4
5
6
7
8
9
10
PS>for ( $i = 0 ; $i -ile 10 ; $i++ ) {
>> Write-Host $i }
>>
0
1
2
3
4
5
6
7
8
9
10
PS>for ( $i = 7 ; $i -ile 10 ; $i++ ) {
>> Write-Host $i
>> }
>>
7
8
9
10
```

If we wanted to change the starting number, we can change the initial definition of *$i*. Say we wanted to start at 7:

Ex:

for ($i = 7 ; $i –ile 10 ; $i++) {
Write-Host $i
}

It is also important to note that everything within the curly brackets is executed for each increment. This means that if we have 2 lines that say *Write-Host $i*, we will have our number printed on our screen twice for each execution of the for loop.

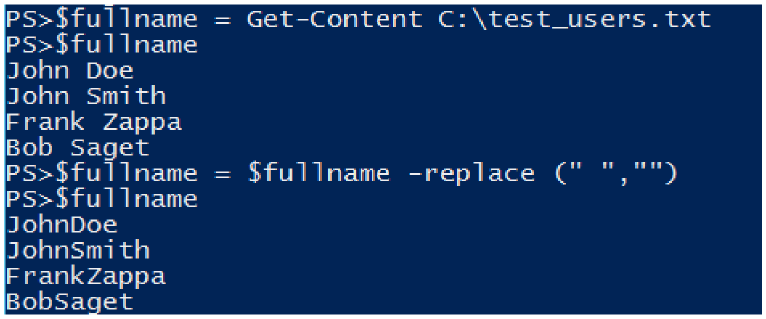
Ex:

for ($i = 7 ; $i –ile 10 ; $i++) {
Write-Host $i
Write-Host $i
}

As well, with for loops, we can increment by whatever number we choose. For example, if we wanted to start at 3 and increment by 3 to the number 27, our syntax would look like:

Ex:

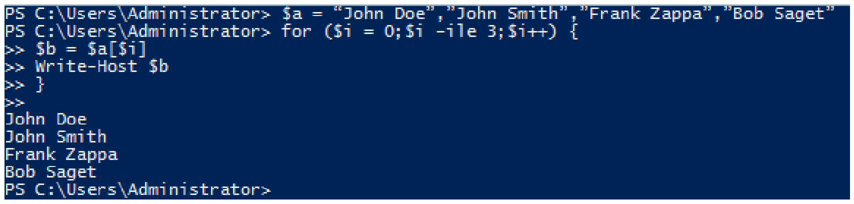
for ($i = 3 ; $i –ile 27 ; $i+= 3) {
Write-Host $i
}



```
PS>$fullname = Get-Content C:\test_users.txt
PS>$fullname
John Doe
John Smith
Frank Zappa
Bob Saget
PS>$fullname = $fullname -replace (" ","")
PS>$fullname
JohnDoe
JohnSmith
FrankZappa
BobSaget
```

Consider alternative uses for this. If we define an array of strings and want to call them, one at a time, we can do this by first defining the array and then using a for loop. Note the placement of the variable that is counting (in this case *$i*). This variable will count wherever it is located. Here, we use it to cycle through items in our array with *$a[$i]*

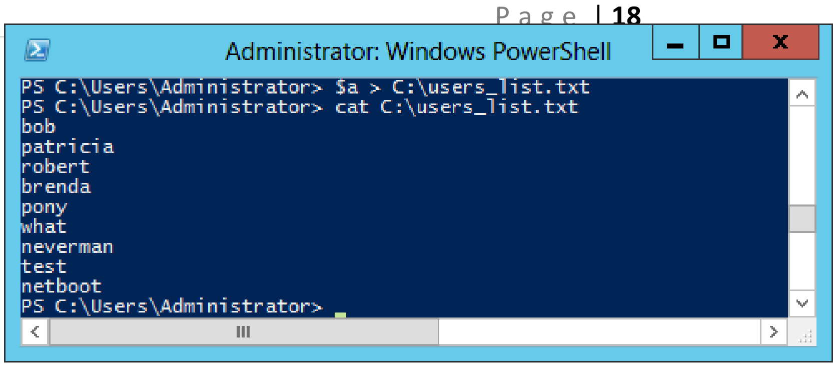
Ex: *$a = “John Doe”,”John Smith”,”Frank Zappa”,”Bob Saget”*
for ($i = 0;$i –ile 3;$i++) {
$b = $a[$i]
Write-Host $b
}



```
PS C:\Users\Administrator> $a = “John Doe”,”John Smith”,”Frank Zappa”,”Bob Saget”
PS C:\Users\Administrator> for ($i = 0;$i -ile 3;$i++) {
>> $b = $a[$i]
>> Write-Host $b
>> }
>>
John Doe
John Smith
Frank Zappa
Bob Saget
PS C:\Users\Administrator>
```

We can also find out the contents of a text file by concatenating the contents on the screen. We can either use the full cmdlets name or the alias.

Ex: *Get-Content C:\users_list.txt*
or
Ex: *cat C:\users_list.txt*



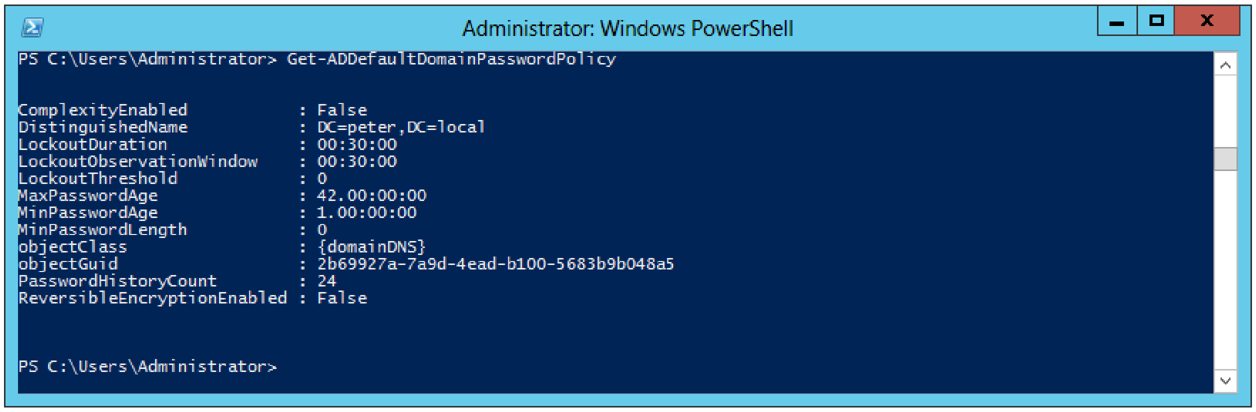
```
PS C:\Users\Administrator> $a > C:\users_list.txt
PS C:\Users\Administrator> cat C:\users_list.txt
bob
patricia
robert
brenda
pony
what
neverman
test
netboot
PS C:\Users\Administrator>
```

Understanding the ActiveDirectory module

Managing the Default Domain Password Policy

Once we get comfortable working with verbs and nouns, our potential for advanced management of Active Directory becomes apparent. For example, look at the noun ADDefaultDomainPasswordPolicy. We can get or set our password policy for AD. [Note: the –*Credential (Get-Credential)* argument is optional and only needs to be run if you are not logged into a privileged user.]

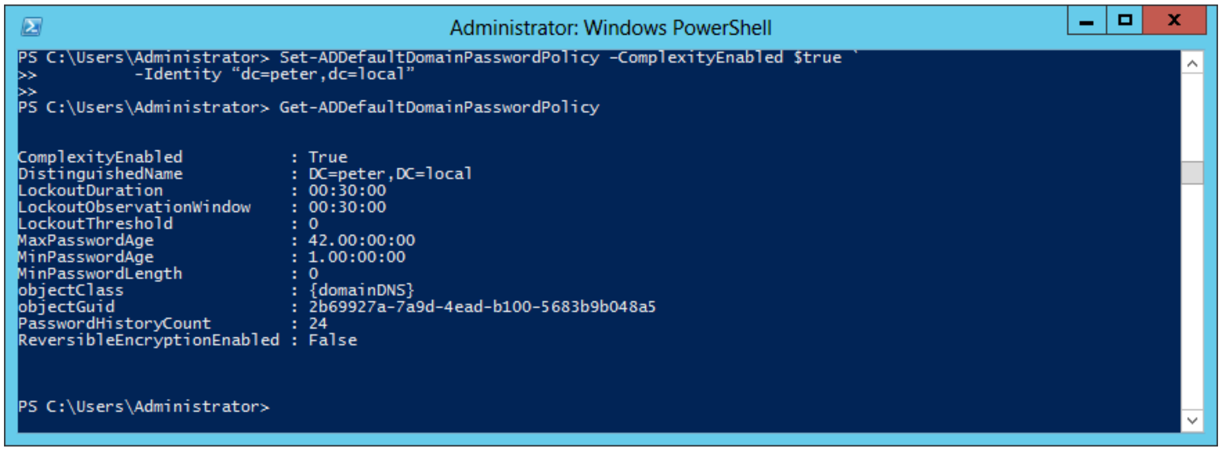
Ex: *Get-ADDefaultDomainPasswordPolicy -Credential (Get-Credential)*



```
PS C:\Users\Administrator> Get-ADDefaultDomainPasswordPolicy


ComplexityEnabled           : False
DistinguishedName           : DC=peter,DC=local
LockoutDuration             : 00:30:00
LockoutObservationWindow    : 00:30:00
LockoutThreshold            : 0
MaxPasswordAge              : 42.00:00:00
MinPasswordAge              : 1.00:00:00
MinPasswordLength           : 0
objectClass                 : {domainDNS}
objectGuid                  : 2b69927a-7a9d-4ead-b100-5683b9b048a5
PasswordHistoryCount        : 24
ReversibleEncryptionEnabled : False


PS C:\Users\Administrator>
```

We can see what the current requirements are for the domain. Now, if we want to change these settings, we can do this easily with the *Set-ADDefaultDomainPasswordPolicy* cmdlet. There is only 1 requirement for this and that is specifying the domain the change is occurring for (identity).

Ex: *Set-ADDefaultDomainPasswordPolicy –ComplexityEnabled $true `*
-Identity “dc=peter,dc=local”



```
PS C:\Users\Administrator> Set-ADDefaultDomainPasswordPolicy -ComplexityEnabled $true `
>>          -Identity “dc=peter,dc=local”
>>
PS C:\Users\Administrator> Get-ADDefaultDomainPasswordPolicy


ComplexityEnabled           : True
DistinguishedName           : DC=peter,DC=local
LockoutDuration             : 00:30:00
LockoutObservationWindow    : 00:30:00
LockoutThreshold            : 0
MaxPasswordAge              : 42.00:00:00
MinPasswordAge              : 1.00:00:00
MinPasswordLength           : 0
objectClass                 : {domainDNS}
objectGuid                  : 2b69927a-7a9d-4ead-b100-5683b9b048a5
PasswordHistoryCount        : 24
ReversibleEncryptionEnabled : False


PS C:\Users\Administrator>
```

To find the syntax to change anything listed, check out:

Ex: *Set-ADDefaultDomainPasswordPolicy -?*

In order to understand how to format the Identity portion of an Active Directory command, it is important to understand the components of AD. For example, dc stands for Domain Component. Since we are changing the password policy for peter.local, we use “dc=peter,dc=local” as our LDAP string. If we were changing the password policy for a subdomain like sub.peter.local, we would use “dc=sub,dc=peter,dc=local” as our LDAP formatted string.

Creating new users

One consideration to take into account when setting up users is the need to meet the password policy defined for the user. If these requirements are not met, PS will return an error. As well, if we do not pass a securestring to the cmdlet, it will also return an error. Creating users can be very complex. You can also Delegate control, specify encryption types, and even associate X509 certificates with a user.

Ex: *New-ADUser -?*
or
Ex: *Get-Help New-ADUser –detailed | more*

For example, if we want to create a new user and pass a password that meets the password policy, we can store a securestring like this:

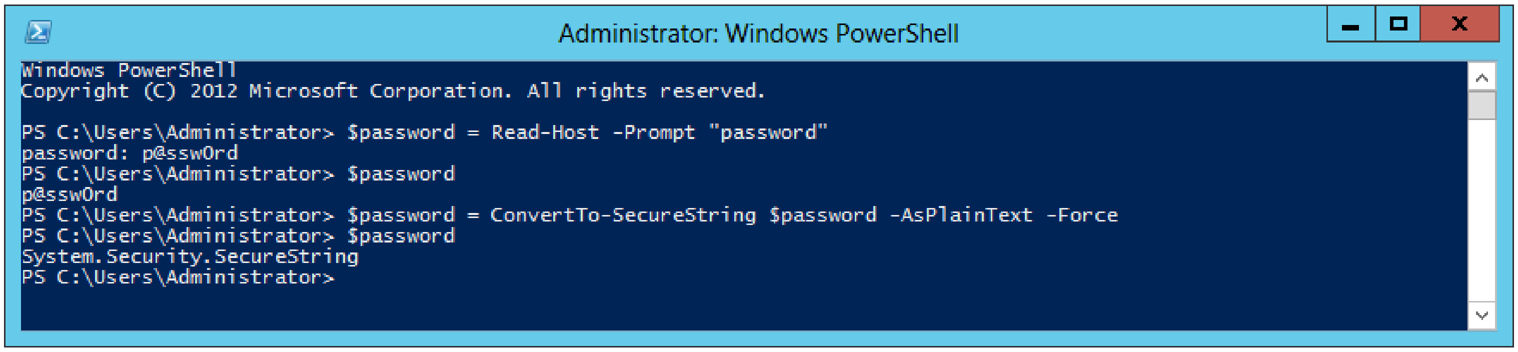
Ex: *$testpass = Read-Host –Prompt “password” –AsSecureString*

Notice how if we try to call the variable (enter just the variable name into the console window), it shows up as a System.Security.SecureString object. Now when we want to create a user with those credentials, we should be able to just pass the *$testpass* variable to our cmdlet for successful account creation.

Ex: *New-ADUser -Name TestUser -AccountPassword $testpass -Enabled 1*

Note how we must specify that we want the user account to be enabled with a Boolean value (a Boolean value is just a *1* or a *0*). We can either use a *1* to indicate true or we can also use *$true* to indicate true as well. We could also use *0* or *$false*. Also notice how we cannot use the *$cred* variable because it is not the right type of object. We can however store a System.Security.SecureString object by taking a string and converting it to a SecureString.

Ex: *$password = Read-Host –Prompt “password”*
$password = ConvertTo-SecureString $password –AsPlainText –Force

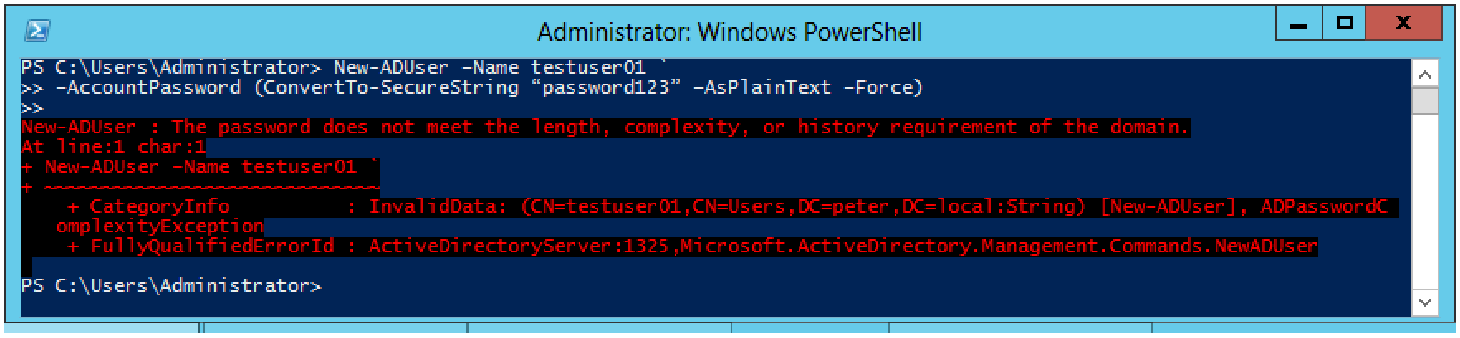


```
Windows PowerShell
Copyright (C) 2012 Microsoft Corporation. All rights reserved.

PS C:\Users\Administrator> $password = Read-Host -Prompt "password"
password: p@ssw0rd
PS C:\Users\Administrator> $password
p@ssw0rd
PS C:\Users\Administrator> $password = ConvertTo-SecureString $password -AsPlainText -Force
PS C:\Users\Administrator> $password
System.Security.SecureString
PS C:\Users\Administrator>
```

This is the same as 2 steps before but this demonstrates how we can have a string stored in a variable, and then convert it to the data type [SecureString]. Notice how when we try to view it when it is a string, we can but once we convert the data type to [System.Security.SecureString], there is protections against viewing the contents of the variable. This is valuable because we can also encapsulate cmdlets within our commands. Say we want to create users with a default password. We can do this on 1 line instead of doing it across multiple lines.

Ex: *New-ADUser –Name testuser01 `*
–AccountPassword (ConvertTo-SecureString "password123" –AsPlainText –Force)

Note how the part that uses the cmdlets *ConvertTo-SecureString* is enclosed in parenthesis. This indicates that it is a separate part and that the output should be used in its place. Remember, the password must meet the complexity requirements of the domain. If it does not, PowerShell will tell you.

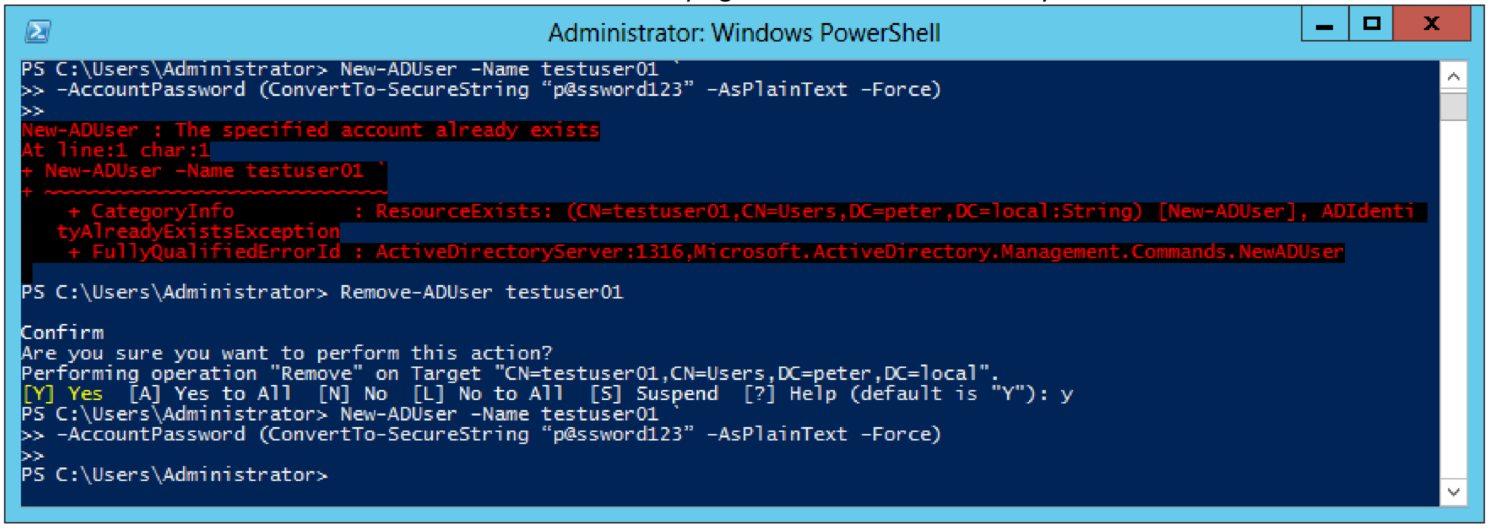


```
PS C:\Users\Administrator> New-ADUser -Name testuser01 `
>> -AccountPassword (ConvertTo-SecureString "password123" -AsPlainText -Force)
>>
New-ADUser : The password does not meet the length, complexity, or history requirement of the domain.
At line:1 char:1
+ New-ADUser -Name testuser01 `
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidData: (CN=testuser01,CN=Users,DC=peter,DC=local:String) [New-ADUser], ADPasswordC
   omplexityException
    + FullyQualifiedErrorId : ActiveDirectoryServer:1325,Microsoft.ActiveDirectory.Management.Commands.NewADUser

PS C:\Users\Administrator>
```

If we use a password that is complex:

Ex: *New-ADUser –Name testuser01 `*
–AccountPassword (ConvertTo-SecureString "p@ssword123" –AsPlainText –Force)

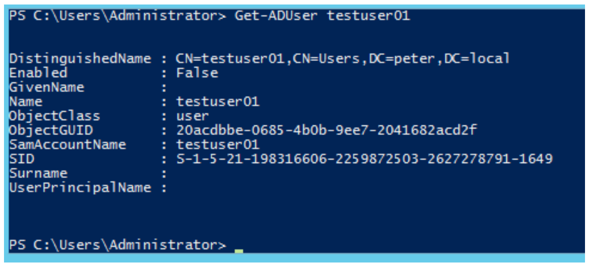
And we make sure that the username we are trying to create does not already exist:



```
PS C:\Users\Administrator> New-ADUser -Name testuser01 `
>> -AccountPassword (ConvertTo-SecureString "p@ssword123" -AsPlainText -Force)
>>
New-ADUser : The specified account already exists
At line:1 char:1
+ New-ADUser -Name testuser01 `
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ResourceExists: (CN=testuser01,CN=Users,DC=peter,DC=local:String) [New-ADUser], ADIdenti
   tyAlreadyExistsException
    + FullyQualifiedErrorId : ActiveDirectoryServer:1316,Microsoft.ActiveDirectory.Management.Commands.NewADUser

PS C:\Users\Administrator> Remove-ADUser testuser01

Confirm
Are you sure you want to perform this action?
Performing operation "Remove" on Target "CN=testuser01,CN=Users,DC=peter,DC=local".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
PS C:\Users\Administrator> New-ADUser -Name testuser01 `
>> -AccountPassword (ConvertTo-SecureString "p@ssword123" -AsPlainText -Force)
>>
PS C:\Users\Administrator>
```

Removing users is easier than adding them. We can use *Get-ADUser* to get user information or *Remove-ADUser* to clean up old users.

Ex: *Get-ADUser testuser01*

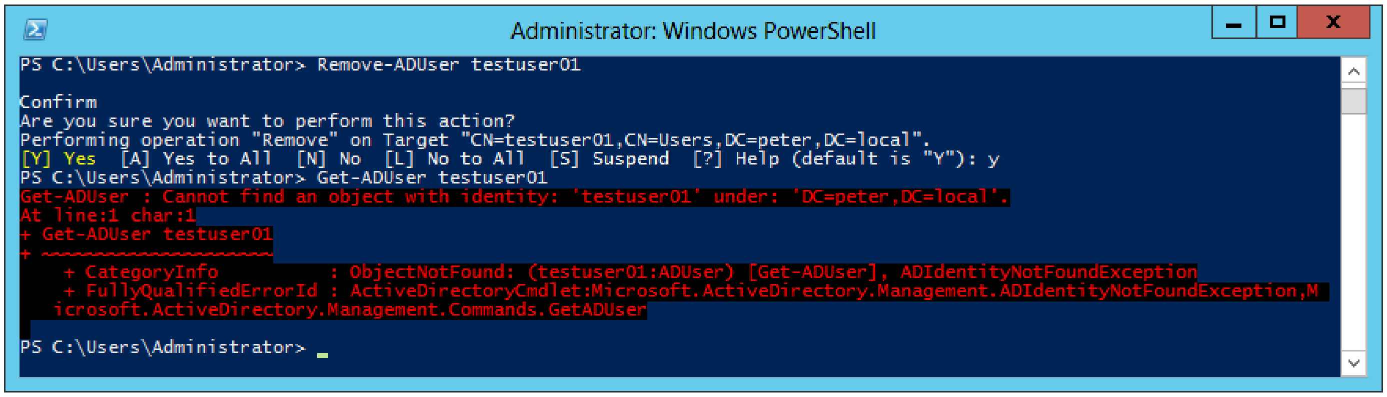


```
PS C:\Users\Administrator> Get-ADUser testuser01


DistinguishedName : CN=testuser01,CN=Users,DC=peter,DC=local
Enabled           : False
GivenName         :
Name              : testuser01
ObjectClass       : user
ObjectGUID        : 20acdbbe-0685-4b0b-9ee7-2041682acd2f
SamAccountName    : testuser01
SID               : S-1-5-21-198316606-2259872503-2627278791-1649
Surname           :
UserPrincipalName :



PS C:\Users\Administrator>
```

Remember, user accounts must explicitly be enabled with the *–Enabled $true* or *–Enabled 1* argument passed to the *New-ADUser* cmdlet. To remove the user we just created:

Ex: *Remove-ADUser testuser01*

After you are prompted to confirm deleting the user, we can no longer *Get-ADUser testuser01*



```
PS C:\Users\Administrator> Remove-ADUser testuser01

Confirm
Are you sure you want to perform this action?
Performing operation "Remove" on Target "CN=testuser01,CN=Users,DC=peter,DC=local".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
PS C:\Users\Administrator> Get-ADUser testuser01
Get-ADUser : Cannot find an object with identity: 'testuser01' under: 'DC=peter,DC=local'.
At line:1 char:1
+ Get-ADUser testuser01
+ ~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (testuser01:ADUser) [Get-ADUser], ADIdentityNotFoundException
    + FullyQualifiedErrorId : ActiveDirectoryCmdlet:Microsoft.ActiveDirectory.Management.ADIdentityNotFoundException,M
   icrosoft.ActiveDirectory.Management.Commands.GetADUser

PS C:\Users\Administrator> _
```

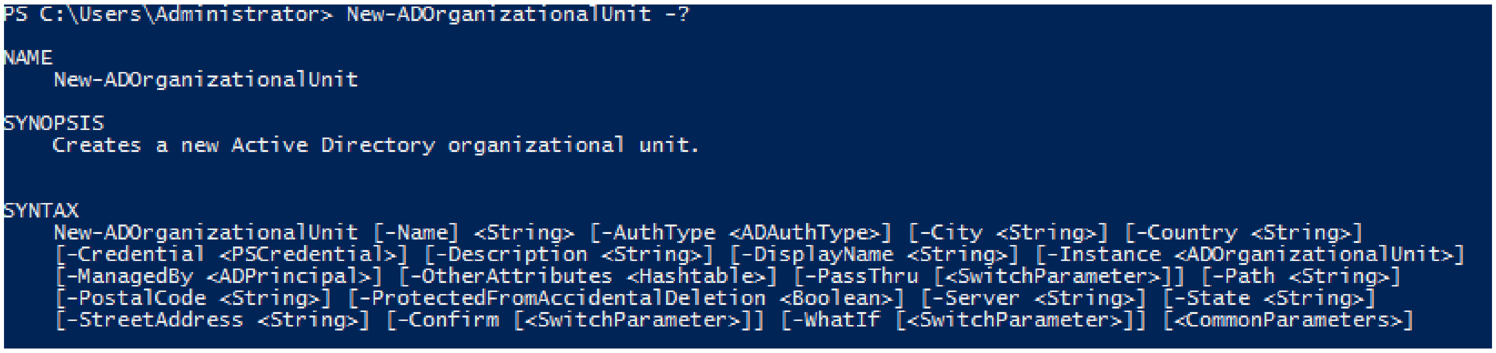
Creating New Organizational Units

As should be apparent now, many of these commands are pretty obvious. Just using the *Get-Command* cmdlet to find all the active directory information can be extremely PowerShell and insightful.

Ex: *Get-Command –module activedirectory*

[Remember, you can also specify a verb at the same time so you know what you can get, set, etc.]
Let's look at the Syntax for the cmdlet *New-ADOrganizationalUnit*

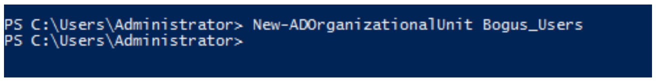
Ex: *New-ADOrganizationalUnit -?*



```
PS C:\Users\Administrator> New-ADOrganizationalUnit -?

NAME
    New-ADOrganizationalUnit

SYNOPSIS
    Creates a new Active Directory organizational unit.


SYNTAX
    New-ADOrganizationalUnit [-Name] <String> [-AuthType <ADAuthType>] [-City <String>] [-Country <String>]
    [-Credential <PSCredential>] [-Description <String>] [-DisplayName <String>] [-Instance <ADOrganizationalUnit>]
    [-ManagedBy <ADPrincipal>] [-OtherAttributes <Hashtable>] [-PassThru [<SwitchParameter>]] [-Path <String>]
    [-PostalCode <String>] [-ProtectedFromAccidentalDeletion <Boolean>] [-Server <String>] [-State <String>]
    [-StreetAddress <String>] [-Confirm [<SwitchParameter>]] [-WhatIf [<SwitchParameter>]] [<CommonParameters>]
```

The only required argument is the *–Name*. This means we can create a new OU at the top-level of the domain with the following:

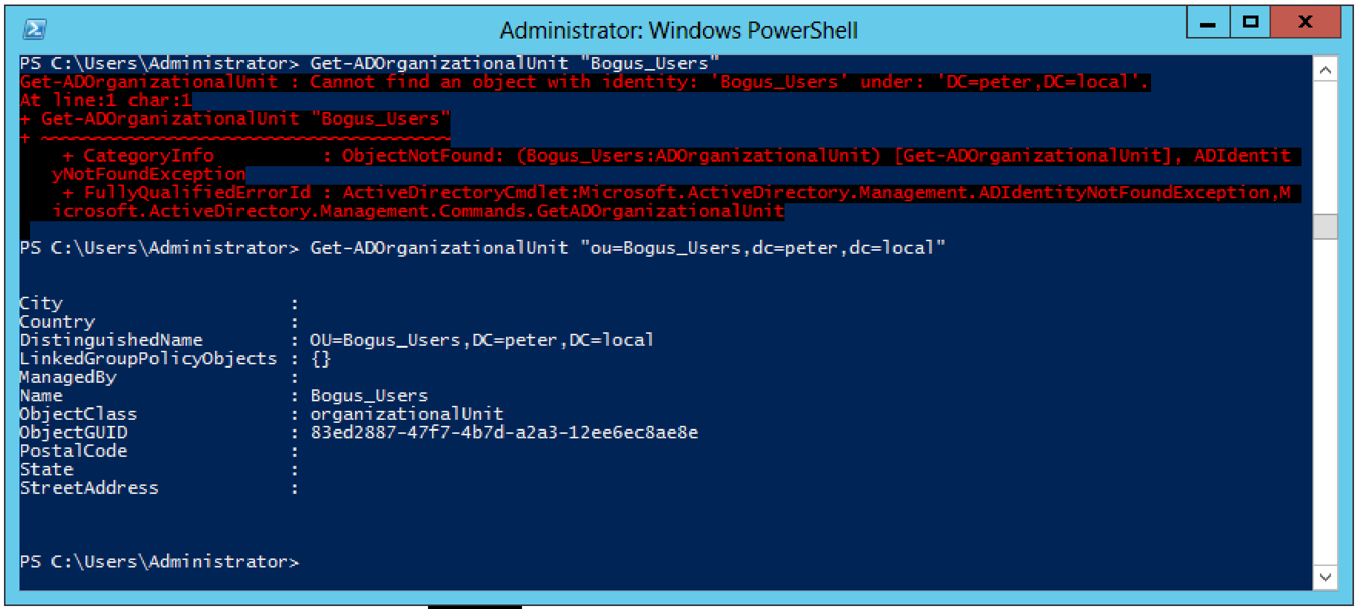
Ex: *New-ADOrganizationalUnit Bogus_Users*



```
PS C:\Users\Administrator> New-ADOrganizationalUnit Bogus_Users
PS C:\Users\Administrator>
```

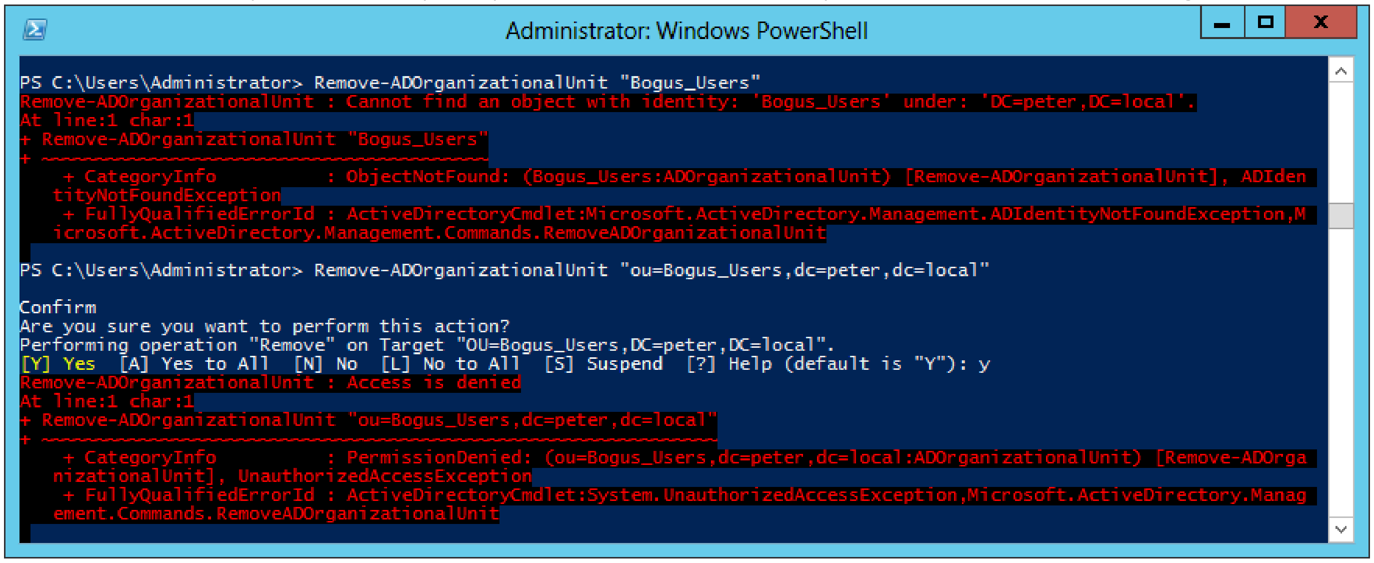
When we get to *Get-ADOrganizationalUnit* and remove *Remove-ADOrganizationalUnit*, we must be more aware of how the cmdlet functions. We cannot just use *Remove-ADOrganizationalUnit Bogus_Users* because it will not find the object. We must be more specific and say where the object is:

Ex: *Get-ADOrganizationalUnit "ou=Bogus_Users,dc=peter,dc=local"*



```
PS C:\Users\Administrator> Get-ADOrganizationalUnit "Bogus_Users"
Get-ADOrganizationalUnit : Cannot find an object with identity: 'Bogus_Users' under: 'DC=peter,DC=local'.
At line:1 char:1
+ Get-ADOrganizationalUnit "Bogus_Users"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (Bogus_Users:ADOrganizationalUnit) [Get-ADOrganizationalUnit], ADIdentit
   yNotFoundException
    + FullyQualifiedErrorId : ActiveDirectoryCmdlet:Microsoft.ActiveDirectory.Management.ADIdentityNotFoundException,M
   icrosoft.ActiveDirectory.Management.Commands.GetADOrganizationalUnit

PS C:\Users\Administrator> Get-ADOrganizationalUnit "ou=Bogus_Users,dc=peter,dc=local"


City                     :
Country                  :
DistinguishedName        : OU=Bogus_Users,DC=peter,DC=local
LinkedGroupPolicyObjects : {}
ManagedBy                :
Name                     : Bogus_Users
ObjectClass              : organizationalUnit
ObjectGUID               : 83ed2887-47f7-4b7d-a2a3-12ee6ec8ae8e
PostalCode               :
State                    :
StreetAddress            :


PS C:\Users\Administrator>
```

It is very important to read the red text that PowerShell outputs as it will always tell you the problem. Sometimes the problem is very complex but sometimes it is easy to resolve. Note the following errors:

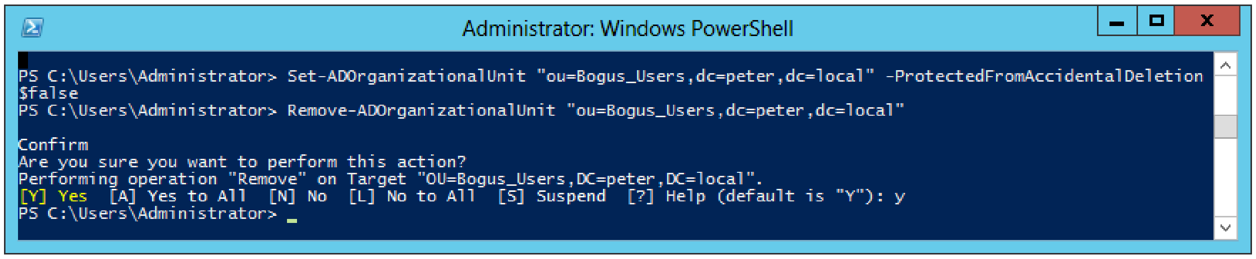


```
PS C:\Users\Administrator> Remove-ADOrganizationalUnit "Bogus_Users"
Remove-ADOrganizationalUnit : Cannot find an object with identity: 'Bogus_Users' under: 'DC=peter,DC=local'.
At line:1 char:1
+ Remove-ADOrganizationalUnit "Bogus_Users"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (Bogus_Users:ADOrganizationalUnit) [Remove-ADOrganizationalUnit], ADIden
   tityNotFoundException
    + FullyQualifiedErrorId : ActiveDirectoryCmdlet:Microsoft.ActiveDirectory.Management.ADIdentityNotFoundException,M
   icrosoft.ActiveDirectory.Management.Commands.RemoveADOrganizationalUnit

PS C:\Users\Administrator> Remove-ADOrganizationalUnit "ou=Bogus_Users,dc=peter,dc=local"

Confirm
Are you sure you want to perform this action?
Performing operation "Remove" on Target "OU=Bogus_Users,DC=peter,DC=local".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
Remove-ADOrganizationalUnit : Access is denied
At line:1 char:1
+ Remove-ADOrganizationalUnit "ou=Bogus_Users,dc=peter,dc=local"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : PermissionDenied: (ou=Bogus_Users,dc=peter,dc=local:ADOrganizationalUnit) [Remove-ADOrga
   nizationalUnit], UnauthorizedAccessException
    + FullyQualifiedErrorId : ActiveDirectoryCmdlet:System.UnauthorizedAccessException,Microsoft.ActiveDirectory.Manag
   ement.Commands.RemoveADOrganizationalUnit
```

First, AD cannot find the OU we are trying to remove. Ok, replace it with an LDAP string to specify the exact object. Then we find the Object but access is denied! Well, when creating an OU, it is protected from accidental deletion by the *–ProtectedFromAccidentalDeletion $true* argument being the default.

So how can we fix this? We can edit the properties of an object (like a user or an OU) with the Set verb.

Ex: *Set-ADOrganizationalUnit "ou=Bogus_Users,dc=peter,dc=local" `*
-ProtectedFromAccidentalDeletion $false

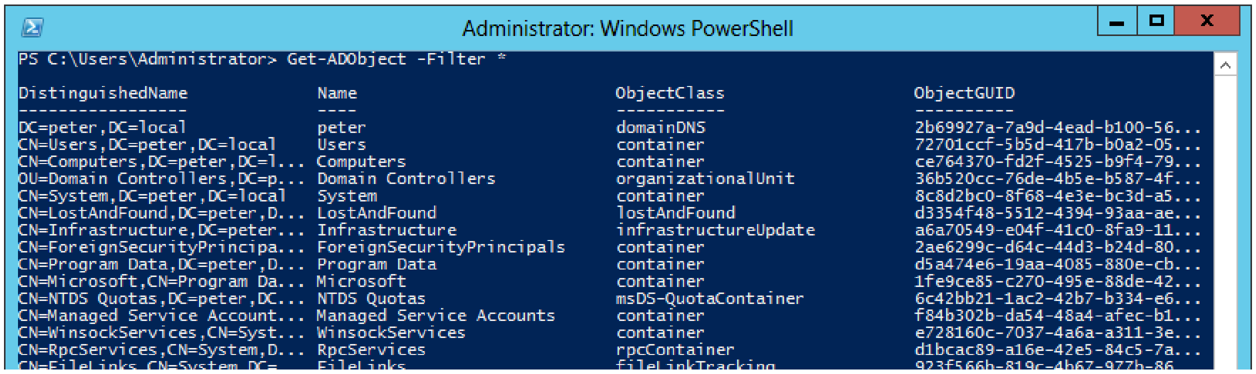


```
PS C:\Users\Administrator> Set-ADOrganizationalUnit "ou=Bogus_Users,dc=peter,dc=local" -ProtectedFromAccidentalDeletion
$false
PS C:\Users\Administrator> Remove-ADOrganizationalUnit "ou=Bogus_Users,dc=peter,dc=local"

Confirm
Are you sure you want to perform this action?
Performing operation "Remove" on Target "OU=Bogus_Users,DC=peter,DC=local".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
PS C:\Users\Administrator> _
```

Getting Information about AD objects

Since everything in Active Directory is an object, it is beneficial to know how to find an object.

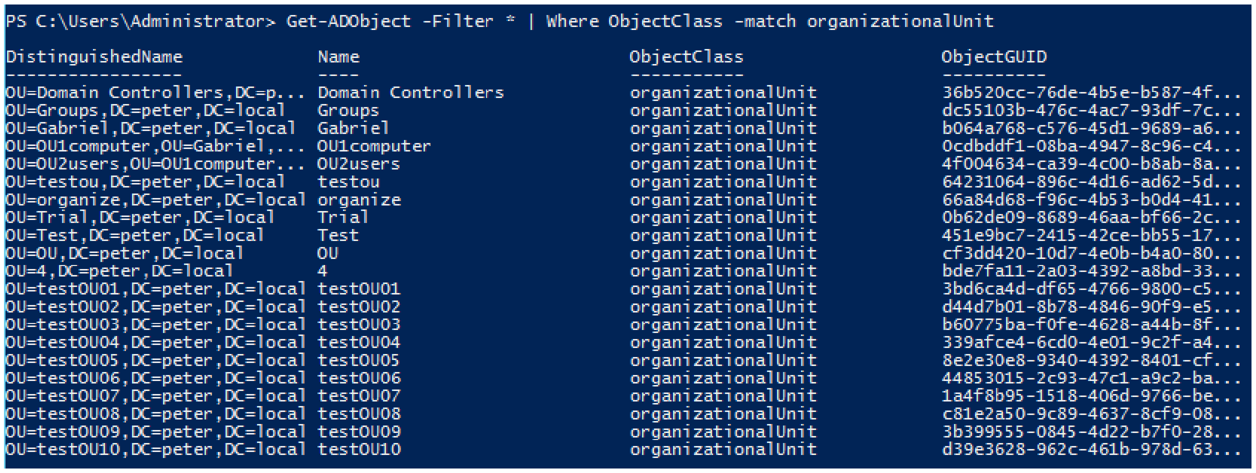
Ex: *Get-ADObject –Filter **



```
Administrator: Windows PowerShell
PS C:\Users\Administrator> Get-ADObject -Filter *

DistinguishedName             Name                          ObjectClass                   ObjectGUID
-----------------             ----                          -----------                   ----------
DC=peter,DC=local             peter                         domainDNS                     2b69927a-7a9d-4ead-b100-56...
CN=Users,DC=peter,DC=local    Users                         container                     72701ccf-5b5d-417b-b0a2-05...
CN=Computers,DC=peter,DC=l... Computers                     container                     ce764370-fd2f-4525-b9f4-79...
OU=Domain Controllers,DC=p... Domain Controllers            organizationalUnit            36b520cc-76de-4b5e-b587-4f...
CN=System,DC=peter,DC=local   System                        container                     8c8d2bc0-8f68-4e3e-bc3d-a5...
CN=LostAndFound,DC=peter,D... LostAndFound                  lostAndFound                  d3354f48-5512-4394-93aa-ae...
CN=Infrastructure,DC=peter... Infrastructure                infrastructureUpdate          a6a70549-e04f-41c0-8fa9-11...
CN=ForeignSecurityPrincipa... ForeignSecurityPrincipals     container                     2ae6299c-d64c-44d3-b24d-80...
CN=Program Data,DC=peter,D... Program Data                  container                     d5a474e6-19aa-4085-880e-cb...
CN=Microsoft,CN=Program Da... Microsoft                     container                     1fe9ce85-c270-495e-88de-42...
CN=NTDS Quotas,DC=peter,DC... NTDS Quotas                   msDS-QuotaContainer           6c42bb21-1ac2-42b7-b334-e6...
CN=Managed Service Account... Managed Service Accounts      container                     f84b302b-da54-48a4-afec-b1...
CN=WinsockServices,CN=Syst... WinsockServices               container                     e728160c-7037-4a6a-a311-3e...
CN=RpcServices,CN=System,D... RpcServices                   rpcContainer                  d1bcac89-a16e-42e5-84c5-7a...
CN=FileLinks,CN=System,DC=    FileLinks                     fileLinkTracking              923f566b-819c-4b67-977b-86
```

The command indicates that we want to *Get-ADObject* and filter by a wildcard (so, get everything). This is where using our filtering techniques discussed earlier become very helpful.

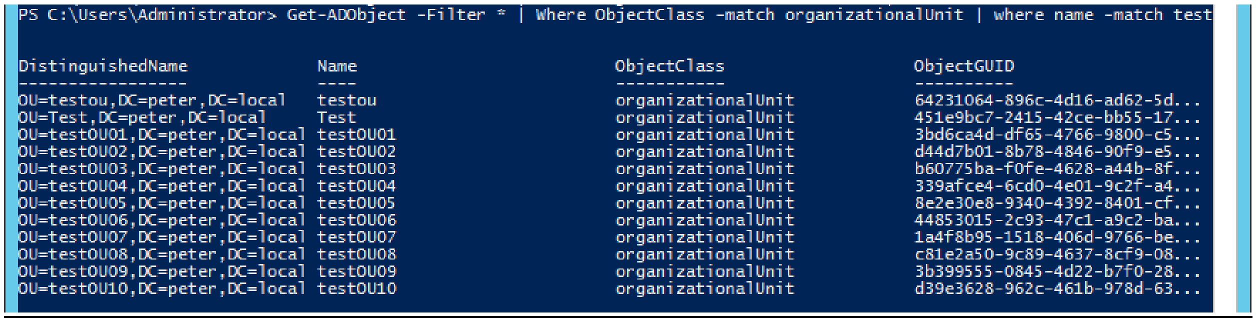
Ex: *Get-ADObject -Filter * | Where ObjectClass -match organizationalUnit*



```
PS C:\Users\Administrator> Get-ADObject -Filter * | Where ObjectClass -match organizationalUnit

DistinguishedName             Name                          ObjectClass                   ObjectGUID
-----------------             ----                          -----------                   ----------
OU=Domain Controllers,DC=p... Domain Controllers            organizationalUnit            36b520cc-76de-4b5e-b587-4f...
OU=Groups,DC=peter,DC=local   Groups                        organizationalUnit            dc55103b-476c-4ac7-93df-7c...
OU=Gabriel,DC=peter,DC=local  Gabriel                       organizationalUnit            b064a768-c576-45d1-9689-a6...
OU=OU1computer,OU=Gabriel,... OU1computer                   organizationalUnit            0cdbddf1-08ba-4947-8c96-c4...
OU=OU2users,OU=OU1computer... OU2users                      organizationalUnit            4f004634-ca39-4c00-b8ab-8a...
OU=testou,DC=peter,DC=local   testou                        organizationalUnit            64231064-896c-4d16-ad62-5d...
OU=organize,DC=peter,DC=local organize                      organizationalUnit            66a84d68-f96c-4b53-b0d4-41...
OU=Trial,DC=peter,DC=local    Trial                         organizationalUnit            0b62de09-8689-46aa-bf66-2c...
OU=Test,DC=peter,DC=local     Test                          organizationalUnit            451e9bc7-2415-42ce-bb55-17...
OU=OU,DC=peter,DC=local       OU                            organizationalUnit            cf3dd420-10d7-4e0b-b4a0-80...
OU=4,DC=peter,DC=local        4                             organizationalUnit            bde7fa11-2a03-4392-a8bd-33...
OU=testOU01,DC=peter,DC=local testOU01                      organizationalUnit            3bd6ca4d-df65-4766-9800-c5...
OU=testOU02,DC=peter,DC=local testOU02                      organizationalUnit            d44d7b01-8b78-4846-90f9-e5...
OU=testOU03,DC=peter,DC=local testOU03                      organizationalUnit            b60775ba-f0fe-4628-a44b-8f...
OU=testOU04,DC=peter,DC=local testOU04                      organizationalUnit            339afce4-6cd0-4e01-9c2f-a4...
OU=testOU05,DC=peter,DC=local testOU05                      organizationalUnit            8e2e30e8-9340-4392-8401-cf...
OU=testOU06,DC=peter,DC=local testOU06                      organizationalUnit            44853015-2c93-47c1-a9c2-ba...
OU=testOU07,DC=peter,DC=local testOU07                      organizationalUnit            1a4f8b95-1518-406d-9766-be...
OU=testOU08,DC=peter,DC=local testOU08                      organizationalUnit            c81e2a50-9c89-4637-8cf9-08...
OU=testOU09,DC=peter,DC=local testOU09                      organizationalUnit            3b399555-0845-4d22-b7f0-28...
OU=testOU10,DC=peter,DC=local testOU10                      organizationalUnit            d39e3628-962c-461b-978d-63...
```

We can also filter our results multiple times:

Ex: *Get-ADObject –Filter * `*
| Where ObjectClass –match organizationUnit `
| Where Name –match test



```
PS C:\Users\Administrator> Get-ADObject -Filter * | Where ObjectClass -match organizationalUnit | where name -match test


DistinguishedName             Name                          ObjectClass                   ObjectGUID
-----------------             ----                          -----------                   ----------
OU=testou,DC=peter,DC=local   testou                        organizationalUnit            64231064-896c-4d16-ad62-5d...
OU=Test,DC=peter,DC=local     Test                          organizationalUnit            451e9bc7-2415-42ce-bb55-17...
OU=testOU01,DC=peter,DC=local testOU01                      organizationalUnit            3bd6ca4d-df65-4766-9800-c5...
OU=testOU02,DC=peter,DC=local testOU02                      organizationalUnit            d44d7b01-8b78-4846-90f9-e5...
OU=testOU03,DC=peter,DC=local testOU03                      organizationalUnit            b60775ba-f0fe-4628-a44b-8f...
OU=testOU04,DC=peter,DC=local testOU04                      organizationalUnit            339afce4-6cd0-4e01-9c2f-a4...
OU=testOU05,DC=peter,DC=local testOU05                      organizationalUnit            8e2e30e8-9340-4392-8401-cf...
OU=testOU06,DC=peter,DC=local testOU06                      organizationalUnit            44853015-2c93-47c1-a9c2-ba...
OU=testOU07,DC=peter,DC=local testOU07                      organizationalUnit            1a4f8b95-1518-406d-9766-be...
OU=testOU08,DC=peter,DC=local testOU08                      organizationalUnit            c81e2a50-9c89-4637-8cf9-08...
OU=testOU09,DC=peter,DC=local testOU09                      organizationalUnit            3b399555-0845-4d22-b7f0-28...
OU=testOU10,DC=peter,DC=local testOU10                      organizationalUnit            d39e3628-962c-461b-978d-63...
```

This is barely scratching the surface of PowerShell. The best way to learn is to explore. Don't be intimidated by it, just play around, mess things up, and keep your images handy in case you have to reinstall.

<u>For a TechNet blog on how to import user information from a CSV [Comma Separated Values file]:</u>
http://blogs.technet.com/b/heyscriptingguy/archive/2011/12/22/use-powershell-to-read-a-csv-file-and-create-active-directory-user-accounts.aspx