

PowerShell Basics

PS Versions

OS	PS Version	PS Compatible Versions	Build Version	WSMan Stack Version	PS Remoting Protocol Version
Win 7	2.0	1.0, 2.0	6.1.7601.17514	2.0	2.1
Win 8	4.0	1.0, 2.0, 3.0, 4.0	6.3.9600.18773	3.0	2.2
Win 10	5.1	1.0, 2.0, 3.0, 4.0 ...	10.0.15063.674	3.0	2.3

PS comparison operators

Operator	Purpose
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-lt	Less than
-le	Less than or equal to
-ge	Greater than or equal to
-like	Wildcard string comparison

PS Commands (General) and Description

gpmc.msc	Open Group Policy
dsa.msc	Open Active Directory
\$PSVersionTable	To check PS version

1. Understanding the PowerShell platform

```
# Get-Command - Retrieves a list of all system commands
Get-Command

# Can expand by searching for just a verb or noun
Get-Command -verb "get"
Get-Command -noun "service"

# Get-Help can be used to explain a command
Get-Help Get-Command
Get-Help Get-Command -examples
Get-Help Get-Command -detailed
Get-Help Get-Command -full
Get-Help Get-Command -Online # PS 3

# Pipelining - combine CmdLets for power
Get-ChildItem | Where-Object { $_.Length -gt 10kb }

Get-ChildItem | Where-Object { $_.Length -gt 10kb } | Sort-Object Length

# Can break commands up among several lines
# (note pipe must be last char on line)
Get-ChildItem |
  Where-Object { $_.Length -gt 10kb } |
  Sort-Object Length

# To specify columns in the output and get nice formatting, use Format-Table
Get-ChildItem |
  Where-Object { $_.Length -gt 10kb } |
  Sort-Object Length |
  Format-Table -Property Name, Length -AutoSize

# You can also use the Select-Object to retrieve certain properties from an object
Get-ChildItem | Select-Object Name, Length

# If you have an especially long command without pipes, you can also use
# a line continuation character of the reverse single quote ` (typically
# located to the left of the number 1 on your keyboard)
# Note that just as with the |, the ` must be the very last character
# on the line. No spaces or comments are allowed after it

Get-ChildItem -Path C:\PS `
  -File "*.ps1" `
  -Verbose
```

```
# Can combine line continuation and pipes
Get-ChildItem -Path C:\PS
-File "*.ps1"
-Verbose
Format-Table -Property Name, Length -AutoSize

# Out-GridView
#-----
#region Out-GridView

# with no params, just displays the results in the output panel
Get-ChildItem | Out-GridView

# Use -passthru to pipe the results to the next item
# (without -PassThru nothing gets displayed)
Get-ChildItem | Out-GridView -PassThru

# Can add useful titles to the display
Get-ChildItem | Out-GridView -PassThru -Title "Hello world"

# You can send the output of the GridView to a variable
$ov = ""
Get-ChildItem | Out-GridView -PassThru -OutVariable ov

Clear-Host
$ov # Show the result

# Instead use Select-Object
Get-ChildItem |
Select-Object -Property Name, Length |
Out-GridView -PassThru
```

Examples:-

Get-Service

The Get-Service cmdlet gets objects that represent the services on a local computer or on a remote computer, including running and stopped services.

Syntax

```
Get-Service [[-Name] <string[]>] [-ComputerName <string[]>] [-DependentServices] [-RequiredServices] [-Include <string[]>] [-Exclude <string[]>] [<CommonParameters>]
```

```
Get-Service -DisplayName <string[]> [-ComputerName <string[]>] [-DependentServices] [-RequiredServices] [-Include <string[]>] [-Exclude <string[]>] [<CommonParameters>]
```

```
Get-Service [-ComputerName <string[]>] [-DependentServices] [-RequiredServices] [-Include <string[]>] [-Exclude <string[]>] [-InputObject <ServiceController[]>] [<CommonParameters>]
```

Parameter	Description
-DisplayName	Specifies, as a string array, the display names of services to be retrieved.
-ComputerName	Gets the services running on the specified computers. The default is the local computer.
-Name	Specifies the service names of services to be retrieved.

Examples:-

1a. Get all services on a local computer.

```
PS C:\> Get-Service
```

```
PS C:\> Get-Service Spooler [For getting a particular service (Printer spooler service) status]
```

1b. Get all services on a remote computer.

```
PS C:\> Get-Service -Computername PC-1
```

```
PS C:\> Get-Service Spooler -Computername PC-1 [For getting a particular service (Printer spooler service) status]
```

1c. Get all services on multiple computers.

```
PS C:\> Get-Service -Computername PC-1, PC-2, PC-3
```

1d. Get results as a table.

```
PS C:\> Get-Service Spooler -Computername PC-1, PC-2, PC-3 | Select Machinename,Name,Status
```

MachineName	Name	Status
PC-01	Spooler	Stopped
PC-02	Spooler	Running
PC-03	Spooler	Running

"|" is a pipeline operator, which formats the results as a table.

References:-

Subject	Command	Description
Get services that begin with a search string	PS C:\> Get-Service "wmi"	This command retrieves services with service names that begin with WMI
Display services that include a search string	PS C:\> Get-Service -Displayname "**network"	This command displays services with a display name that includes the word network
Get services that begin with a search string and an exclusion	PS C:\> Get-Service -Name "win*" -Exclude "WinRM"	These commands get only the services with service names that begin with win, except for the WinRM service.
Display services that are currently active	PS C:\> Get-Service Where-Object {\$_.Status -eq "Running"}	This command displays only the services that are currently active
Sort services by property value	PS C:\> Get-Service "s*" Sort-Object status	This command shows that when you sort services in ascending order by the value of their Status property

2. Variables in PowerShell

All variables start with a \$. Show a simple assignment
`$hi = "Hello world"`

Print the value
`$hi`

This is a shortcut to Write-Host
`Write-Host $hi`

```
PS C:\PS\Beginning PowerShell Scripting for Developers> $hi.ToUpper()
HELLO WORLD
PS C:\PS\Beginning PowerShell Scripting for Developers> $hi.ToLower()
hello world
```

Variables are objects. Show the type
`$hi.GetType()`

Display all the members of this variable (object)
`$hi | Get-Member`

Use some of those members
`$hi.ToUpper()`
`$hi.ToLower()`
`$hi.Length`

```
43 # Types are mutable
44 Clear-Host
45 $hi = 5
46 $hi.GetType()
47
```

IsPublic	IsSerial	Name	BaseType
True	True	Int32	System.ValueType

Accessing methods on objects
`"PowerShell Rocks".ToUpper()`
`"PowerShell Rocks".Contains("PowerShell")`

For nonstrings you need to wrap in () so PS will evaluate as an object
`(33).GetType()`

Comparisons
`$var = 33`

`$var -gt 30`
`$var -lt 30`
`$var -eq 33`

```
# List is:
# -eq      Equals
# -ne      Not equal to
# -lt      Less Than
# -gt      Greater then
# -le      Less than or equal to
# -ge      Greater then or equal to

# -in      See if value in an array
# -notin   See if a value is missing from an array
# -Like    Like wildcard pattern matching
# -NotLike Not Like
# -Match   Matches based on regular expressions
```

```
# Calculations are like any other language
$var = 3 * 11 # Also uses +, -, and /
$var
```

```
33
```

```
# Supports post unary operators ++ and --
$var++
$var
```

```
34
```

```
Clear-Host
$var = 33
$post = $var++ X
$post
$var
```

```
Clear-Host
$var = 33
$post = ++$var ✓
$post
$var
```

```
# Whatever is on the right is converted to the data type on the left
# Can lead to some odd conversions
42 -eq "042" # True because the string on the right is converted to an int
"042" -eq 42 # False because int on the right is converted to a string
```

```
$_ # Current Object
Set-Location "c:\ps\01 - intro"
Get-Childitem | Where-Object {$_.Name -like "*.ps1"}
```

3. Strings, Arrays & Hash tables

```
# String Quoting
```

```
Clear-Host
"This is a string"
'This is a string too!'
```

```
# Mixed quoted
```

```
'I just wanted to say "Hello World", OK?'
"I can't believe how cool Powershell is!"
```

```
# newline `n
```

```
"Power`nShell"
```

```
PS C:\PS> "Power`nShell"
```

```
Power
Shell
```

```
# Here Strings - for large blocks of text -----
```

```
Clear-Host
$heretext = @"
Some text here
Some more here
    a bit more

a blank line above
"@
```

```
# Without here strings
$sql = 'SELECT col1' `
      + '      , col2' `
      + '      , col3' `
      + ' FROM someTable ' `
      + ' WHERE col1 = ''a value'' '
```

```
# With here strings
$sql = @'
SELECT col1
      , col2
      , col3
FROM someTable
WHERE col1 = 'a value'
'@
```

- ✓ # Use these variables in a string
"There are \$items items are in the folder \$loc."
- ✓ # To actually display the variable, escape it with a backtick
"There are ` \$items items are in the folder ` \$loc."
- ✗ # String interpolation only works with double quotes
'There are \$items items are in the folder \$loc.'

```
113 # String Interpolation works with here strings
114 $hereinterpolation = @"
115 Items`tFolder
116 -----`t-----
117 $items`t`t$loc
118
119 "@
120
121 $hereinterpolation
122
```

```
$hereinterpolation
```

```
Items    Folder
-----
100      C:\PS
```

```
141 # String Formatting - C# like syntax is supported
142 #   In C you'd use:
143 [string]::Format("There are {0} items.", $items)
144
145 # Powershell shortcut
146 "There are {0} items." -f $items
147
148 "There are {0} items in the location {1}." -f $items, $loc
149
150 "There are {0} items in the location {1}. Wow, {0} is a lot of items!" -f $items, $loc
```

```
PS C:\PS> [string]::Format("There are {0} items.", $items)
```

```
There are 100 items.
```

```
PS C:\PS> "There are {0} items." -f $items
```

```
There are 100 items.
```



```

145 # Powershell shortcut
146 "There are {0} items." -f $items
147
148 "There are {0} items in the location {1}." -f $items, $loc
149
150 "There are {0} items in the location {1}. Wow, {0} is a lot of items!" -f $items, $loc
151

```

```

PS C:\PS> "There are {0} items in the location {1}." -f $items, $loc
There are 100 items in the location C:\PS.

PS C:\PS> "There are {0} items in the location {1}. Wow, {0} is a lot of items!" -f $items, $loc
There are 100 items in the location C:\PS. Wow, 100 is a lot of items!

```

```

152 # Predefined formats
153 # N - Number
154 "N0 {0:N0} formatted" -f 12345678.119
155 "N1 {0:N1} formatted" -f 12345678.119
156 "N2 {0:N2} formatted" -f 12345678.119
157 "N2 {0:N9} formatted" -f 12345678.119
158 "N0 {0:N0} formatted" -f 123.119
159 "N0 {0,8:N0} formatted" -f 123.119
160

```

```

N0 12,345,678 formatted
N1 12,345,678.1 formatted
N2 12,345,678.12 formatted
N2 12,345,678.119000000 formatted
N0 123 formatted
N0      123 formatted

```

P - Percentage

```

"P0 {0:P0} formatted" -f 0.1234 → # P0 12 % formatted
"P2 {0:P2} formatted" -f 0.1234 → # P2 12.34 % formatted

```

```

195 # Custom date formatting. Note MM is Month, mm is minute
196 "Today is {0:M/d/yyyy}. Be well." -f $(Get-Date)
197 "Today is {0,10:MM/dd/yyyy}. Be well." -f $(Get-Date)
198 "Today is {0,10:yyyyMMdd}. Be well." -f $(Get-Date)
199 "Today is {0,10:MM/dd/yyyy hh:mm:ss}. Be well." -f $(Get-Date)

```

```

Today is 9/13/2015. Be well.
Today is 09/13/2015. Be well.
Today is 20150913. Be well.
Today is 09/13/2015 12:44:34. Be well.

```

Simple array

```

Clear-Host
$array = "Arcane", "Code"
$array

```

```

PS C:\PS> $array

```

```

Arcane
Code

```

```

$array[0]

```

```

PS C:\PS> $array[0]

```

```

Arcane

```

```

$array[1]

```

```

PS C:\PS> $array[1]

```

```

Code

```

Formal Array Creation Syntax

```
$array = @("Power", "Shell")
$array
```

```
PS C:\PS> $array = @("Power", "Shell")
$array

Power
Shell
```

```
PS C:\PS> $numbers = 1, 42, 256
```

```
PS C:\PS> $numbers -contains 42
```

```
True
```

```
PS C:\PS> $numbers -notcontains 99
```

```
True
```

Creating a Hash Table

- ➔ @{Key=Value;Key2=Value;Key3=Value}
- ➔ Create an empty hash table
 - -PS C:\> \$hash = @{}
- ➔ Some cmdlets will create hash tables
- ➔ You can modify, add, and delete items in a hash table

Enumerating a Hash Table

Write the hash table to the pipeline	PS C:\> \$hash=@{A=123;B="foo";C=3.14} PS C:\> \$hash
Reference values by key as a property	PS C:\> \$hash.b foo
Reference items by Item() property	PS C:\> \$hash.item("c") 3.14
Assign a new value to a key	PS C:\> \$hash.a=678

Example: one

```
$e = @{"Name"="Jeff";Title="MVP";Computer=$env:COMPUTERNAME}
$e
```

```

Name                                     Value
----                                     -
Title                                    MVP
Name                                     Jeff
Computer                                CHI-WIN8-01

```

```
#enumerating key
```

```
$e.Keys
```

```

Title
Name
Computer

```

```
$e.computer
```

```

PS C:\> $e.computer
CHI-WIN8-01

```

```
#creating an empty hash
```

```
$f=@{}
```

```
#adding to it
```

```

$f.Add("Name","Jeff")
$f.Add("Company","Globomantics")
$f.Add("Office","Evanston")
$f

```

```

Name                                     Value
----                                     -
Company                                Globomantics
Office                                Evanston
Name                                    Jeff

```

```
#changing an item
```

```

$f.Office
$f.Office = "chicago"
$f

```

```

Name                                     Value
----                                     -
Company                                Globomantics
Office                                Chicago
Name                                    Jeff

```

```
#removing an item
```

```

$f.Remove("name")
$f

```

```

Name                                     Value
----                                     -
Company                                Globomantics
Office                                Chicago

```

```
#group-object can create a hash table
```

```

$source = get-eventlog system -newest 100 |
group $source -AsHashTable
$source

```

```

Name                                     Value
----                                     -
Microsoft-Windows-Kernel-Pr... {System.Diagnostics.EventLogEntry}
Microsoft-Windows-WindowsUp... {System.Diagnostics.EventLogEntry, Sys
Service Control Manager        {System.Diagnostics.EventLogEntry, Sys
Microsoft-Windows-Kernel-Boot  {System.Diagnostics.EventLogEntry, Sys
storflt                          {System.Diagnostics.EventLogEntry}
Microsoft-Windows-DNS-Client    {System.Diagnostics.EventLogEntry, Sys
Microsoft-Windows-DHCPv6-Cl     {System.Diagnostics.EventLogEntry, Sys

```

```
#get a specific entry
```

```
$source.EventLog
```

```

Index Time          EntryType Source InstanceID Message
-----
3409 Apr 09 12:00 Information EventLog 2147489661 The sy
3321 Apr 09 11:20 Information EventLog 2147489661 The sy
3320 Apr 09 11:20 Information EventLog 2147489653 The Ev
3319 Apr 09 11:20 Information EventLog 2147489657 Micros
3315 Apr 06 17:58 Information EventLog 2147489654 The Ev

```

```
#handle names with spaces
```

```
$source.'Service Control Manager'
```


Index	Time	EntryType	Source	InstanceID	Message
3410	Apr 09 12:51	Information	Service Control M...	1073748864	The st
3368	Apr 09 11:30	Information	Service Control M...	1073748864	The st
3348	Apr 09 11:20	Information	Service Control M...	3221232498	The fo

```
$hash = @{
    Name="Jeff"
    Company="Globomantics"
    Office="Chicago"
    Computer=$env:computername
    OS = (get-ciminstance win32_operatingsystem -Property Caption).Caption
}
```

\$hash

Name	Value
Computer	CHI-WIN8-01
Name	Jeff
Office	Chicago
Company	Globomantics
OS	Microsoft Windows 8 Enterprise

#hashtables as object properties

```
$os = Get-CimInstance win32_operatingsystem
$cs = Get-CimInstance win32_computersystem
```

```
$properties = [ordered]@{
    Computername = $os.CSName
    MemoryMB = $cs.TotalPhysicalMemory/1mb -As [int]
    LastBoot = $os.LastBootUpTime
}
```

\$properties

Name	Value
Computername	CHI-WIN8-01
MemoryMB	1104
LastBoot	4/9/2013 11:20:31 AM

Example: two

Hash tables

#-----

```
$hash = @{
    "Key" = "Value";
    "PowerShell" = "PowerShell.com";
    "Arcane Code" = "arcanecode.com"
}
```

\$hash # Display all values

\$hash["PowerShell"] # Get a single value from the key

PowerShell PowerShell.com

\$hash."Arcane Code" # Get single value using object syntax

Arcane Code arcanecode.com

You can use variables as keys

\$mykey = "PowerShell"

\$hash.\$mykey # Using variable as a property

PowerShell.com

\$hash.\$(\$mykey) # Evaluating as an expression

PowerShell.com

\$hash.\$("Power" + "Shell")

PowerShell.com

4. Cmdlets

4.1 Get-ChildItem

Commonly used command in FileSystem

- Aliases: dir, ls, gci

Retrieves child objects in a hierarchy

- Active Directory Organizational Unit
- Certificate store
- Registry key

Use filtering options when possible

- -Include, -Exclude, -Filter
- These parameters vary in use by provider

4.2 Where-Object

Used for filtering objects in the pipeline

- Aliases: where, ?

Traditional syntax uses a script block and \$_

- PS C:\> get-service m* | where {\$_.status -eq 'running'}

New syntax can be simpler

- PS C:\> get-service m* | where status -eq 'running'

New syntax won't work for complex filtering

4.3 Select-Object

```
PS C:\> get-process | select
id,name,workingset,path -first 5
#unique
get-process -ComputerName chi-dc01 | select Name -unique
```

Example:-

```
Get-eventlog system -newest 100 -ComputerName CHI-FP01 | Group Source
```

Count	Name	Group
3	EventLog	{System.Diagnostics.EventLogEntry, System.Diagnos...
79	Service Control Manager	{System.Diagnostics.EventLogEntry, System.Diagnos...
5	Microsoft-windows-wind...	{System.Diagnostics.EventLogEntry, System.Diagnos...
2	Microsoft-windows-Grou...	{System.Diagnostics.EventLogEntry, System.Diagnos...
6	Microsoft-windows-Time...	{System.Diagnostics.EventLogEntry, System.Diagnos...
1	Microsoft-windows-Filt...	{System.Diagnostics.EventLogEntry}
1	TermService	{System.Diagnostics.EventLogEntry}
1	Microsoft-windows-Iphl...	{System.Diagnostics.EventLogEntry}
1	Microsoft-windows-DNS-...	{System.Diagnostics.EventLogEntry}
1	Virtual Disk Service	{System.Diagnostics.EventLogEntry}

4.4 Sort-Object

```
PS C:\> get-process | sort WorkingSet
-descending | select -first 10
```

Example:-

```
get-process -computername chi-dc01 | Sort WS
get-process -ComputerName chi-dc01 | Sort WS -Descending
get-process -ComputerName chi-dc01 |
Sort WS -Descending | Select -First 5

get-prqcess -ComputerName chi-dc01 |
Sort WS -Descending | Select -last 5
```

4.5 Get-Content

```
PS C:\> get-content computers.txt | foreach
{get-service wuauserv -comp $_}
```

Example:-one

```
#legacy syntax
get-service | where {$_.status -eq 'Stopped'}
```

```
#new syntax
get-service | where status -eq 'Stopped'
```

```
dir \\chi-fp01\Executive | where Length -ge 100kb
```

Example:-two

```
dir \\chi-fp01\Executive\*.docx |
where { ($_.LastWritetime -gt (Get-Date).AddDays(-90)) `
-AND ($_.length -gt 15KB)}
```

Example:-three (Select files in a directory)

```
#properties
dir \\chi-fp01\public -file | Select Name,Length,LastWriteTime
```

4.6 Get-Service

```
get-service winmgmt -computername chi-dc03 | Select DependentServices
```

```
DependentServices
-----
{wscsvc, vmms, smstsmgr, SharedAccess...}
```

#so expand it

```
get-service winmgmt -computername chi-dc03 |
Select -expandproperty DependentServices
```

Status	Name	DisplayName
Running	UALSVC	User Access Logging Service
Stopped	SharedAccess	Internet Connection Sharing (ICS)
Stopped	NcaSvc	Network Connectivity Assistant
Running	iphlpvc	IP Helper

5. Working with Objects

```
#-----#
# Demo 0 -- Object Oriented Terminology
#-----#
<#
class = blueprints
properties = describe
methods (functions) = actions
object = house
instantiating a new object = building a new house
each object is an instance of the class = each house is a copy of the blueprint
```

5.1 Creating Objects from HashTables

For this first demonstration, we are going to use a function to help us create our new object. The most common way of creating object is to start by defining its properties within a hash table.

```
#-----#
# Demo 1 -- Create a new object
# This is the most common method of creating objects
#-----#
function Create-Object ($Schema, $Table, $Comment)
{
    # Build a hash table with the properties
    $properties = [ordered]@{ Schema = $Schema
    Table = $Table
    Comment = $Comment
    }
```

PSObject is just an empty shell that you can use as a starting point when creating your own objects. As part of creating the new object to our empty shell, we are going to add the properties that we defined in our properties hash table. So it's going to take New-Object, it's going to generate a new object based on the PSObject class, which is an empty class, and then to that it's going to then add those properties. It is then going to return that into the \$object variable.

```
# Start by creating an object of type PSObject
$object = New-Object -TypeName PSObject -Property $properties

# Return the newly created object
return $object
}
```

```
$myObject = Create-Object -Schema "MySchema" -Table "MyTable" -Comment "MyComment"
$myObject
```

```
Schema    Table    Comment
-----
MySchema  MyTable  MyComment
```

Schema, Table, and Comment, and it has the values. If I want to reference a specific value, I could do that, for example here in a string, where I say My Schema =, and then I am referencing the schema property of my object.

```
# Display in text. Note because it is an object need to wrap in $( ) to access a property
"My Schema = $($myObject.Schema)"
```

```
$myObject.Schema = "New Schema"
$myObject.Comment = "New Comment"
$myObject
```

```
Schema    Table    Comment
-----
New Schema MyTable  New Comment
```

6. Program Flow

6.1 If/else

```
21 # if/else
22 $var = 2
23 if ($var -eq 1) # Be sure to use -eq instead of =
24 {
25     Clear-Host
26     "If branch"
27 }
28 else
29 {
30     Clear-Host
31     "else branch"
32 }
33
```

else branch
PS C:\PS>

6.2 If elseif else

```
# if elseif else
$var = 2
if ($var -eq 1)
{
    Clear-Host
    "If -eq 1 branch"
}
elseif ($var -eq 2)
{
    Clear-Host
    "ElseIf -eq 2 branch"
}
else
{
    Clear-Host
    "else branch"
}

ElseIf -eq 2 branch
```


6.3 Switch statement for multiple conditions

```
# Switch statement for multiple conditions
```

```
Clear-Host
$var = 42 # Also test with 43 and 49
switch ($var)
{
    41 {"Forty One"}
    42 {"Forty Two"}
    43 {"Forty Three"}
    default {"default"}
}
```

```
Forty Two
```

```
# Will match all lines that match
```

```
Clear-Host
$var = 42
switch ($var)
{
    42 {"Forty Two"}
    "42" {"Forty Two String"}
    default {"default"}
}
# Note type coercion will cause both 42 lines to have a match
```

```
Forty Two
Forty Two String
```

```
# To stop processing once a block is found use break
```

```
Clear-Host
$var = 42
switch ($var)
{
    42 {"Forty Two"; break}
    "42" {"Forty Two String"; break}
    default {"default"}
}
# Note, if you want to put multiple commands on a single
```

```
Forty Two
```

```
# Switch works with collections, looping and executing for each match
```

```
Clear-Host
switch (3,1,2,42)
{
    1 {"One"}
    2 {"Two"}
    3 {"Three"}
    default {"The default answer"}
}
```

```
Three
One
Two
The default answer
```

```
# String compares are case insensitive by default
```

```
Clear-Host
switch ("PowerShell")
{
    "powershell" {"lowercase"}
    "POWERSHELL" {"uppercase"}
    "PowerShell" {"mixedcase"}
}
```

```
Lowercase
uppercase
mixedcase
```

```
Clear-Host
switch -casesensitive ("PowerShell")
{
    "powershell" {"lowercase"}
    "POWERSHELL" {"uppercase"}
    "PowerShell" {"mixedcase"}
}
```

mixedcase

6.4 Looping

```
#-----
# Looping
#-----
#region Looping

# while
Clear-Host
$i = 1
while ($i -le 5)
{
    "`$i = $i"
    $i = $i + 1
}
```

```
$i = 1
$i = 2
$i = 3
$i = 4
$i = 5
```

```
# Do
Clear-Host
$i = 1
do
{
    "`$i = $i"
    $i++
} while($i -le 5)
```

```
$i = 1
$i = 2
$i = 3
$i = 4
$i = 5
```

```
# Use until to make the check more positive
Clear-Host
$i = 1
do
{
    "`$i = $i"
    $i++
} until($i -gt 5)
```

```
$i = 1
$i = 2
$i = 3
$i = 4
$i = 5
```

```
# When used in a nested loop, break exits to the outer loop
Clear-Host
foreach ($outside in 1..3)
{
    "`$outside=$outside"
    foreach ($inside in 4..6)
    {
        "`$inside = $inside"
        break
    }
}
```

```
$outside=1
    $inside = 4
$outside=2
    $inside = 4
$outside=3
    $inside = 4
PS C:\PS>
```

```
# Use loop labels to break to a certain loop
```

```
Clear-Host
:outsideloop foreach ($outside in 1..3)
{
    "`$outside=$outside"
    foreach ($inside in 4..6)
    {
        "`$inside = $inside"
        break outsideloop
    }
}
```

```
$outside=1
    $inside = 4
```

```
# Using continue inside an inner loop
```

```
Clear-Host
foreach ($outside in 1..3)
{
    "`$outside=$outside"
    foreach ($inside in 4..6)
    {
        "`$inside = $inside"
        continue
        "this will never execute as continue goes back to start of inner for loop"
        # note, because we continue to the inside loop, the above line
        # will never run but it will go thru all iterations of the inner loop
    }
}
```

```
    $inside = 5
    $inside = 6
$outside=2
    $inside = 4
    $inside = 5
    $inside = 6
$outside=3
    $inside = 4
    $inside = 5
    $inside = 6
```

6.5 Script Blocks

```
#-----
# Script Blocks
#-----
#region Script Blocks

# A basic script block is code inside {}
# The for (as well as other loops) execute a script block
for ($f = 0; $f -le 5; $f++)
{
    "`$f = $f"
}
```

```
# A script block can exist on its own
# (note, to put multiple commands on a single line use the ; )
{Clear-Host; "Powershell is cool."}

# Executing only shows the contents of the block, doesn't execute it
PS C:\PS> {Clear-Host; "Powershell is cool."}

Clear-Host; "Powershell is cool."
```

```
# To actually run it, use an ampersand & in front
&{Clear-Host; "Powershell is cool."}

Powershell is cool.
```

```
# You can store script blocks inside a variable
$cool = {Clear-Host; "Powershell is cool."}

$cool # Just entering the variable though only shows the contents, doesn't run it

Clear-Host; "Powershell is cool."
```

```
$cool # To actually run it, use the & character

Powershell is cool.
```

```
# Since scripts can be put in a variable, you can do interesting things
Clear-Host
$cool = {"Powershell is cool."; " So is ArcaneCode"}
for ($i=0;$i -lt 3; $i++)
{
    &$cool;
}

Powershell is cool.
So is ArcaneCode
Powershell is cool.
So is ArcaneCode
Powershell is cool.
So is ArcaneCode
```

7. Reusing Code with Functions and Modules

7.1 Functions

```
#-----
# Functions
#-----
#region Functions

$hw = {
    Clear-Host
    "Hello world"
}

& $hw

Hello world
```

This, however, has many limitations. For example, what happens if somebody comes along and reuses your **hw** variable? To solve that, we can actually use something called a function to hold our reusable code.

```
# Functions are basically script blocks with names.
function Write-HelloWorld()
{
    Clear-Host
    "Hello World"
}
```

It does not actually do anything.

To actually execute the function, I simply use the name of the function, Write-HelloWorld, and I will run this. I can run this as many times as I want to.

```
# Running the above simply places the function in memory for us to use
# To use it, call it like you would a cmdlet
Write-HelloWorld

Hello World
```

```
# When writing functions, use an approved verb
# Get a list of approved verbs
Get-Verb
```


- 7.2 Error Handling
- 7.3 Adding Help
- 7.4 Modules
- 7.5 Profile

Remoting

1. Basics

First, you will need to enable remoting on the computer you want to control. On the remote computer, enter the command below. (-Force will run without prompts)

Enable-PSRemoting -Force

```
26 Enable-PSRemoting -Force
27
```

```
PS C:\PS\Beginning PowerShell Scripting for Developers\demo> Enable-PSRemoting -Force

WinRM is already set up to receive requests on this computer.
WinRM is already set up for remote management on this computer.
```

```
# If you are NOT running on a domain, for example doing this on a home
# network, you will need to do a few other things.
# On both the remote computer and the local computer, run:
```

```
Set-Item wsman:\localhost\client\trustedhosts *
```

```
# Instead of an *, you could specify the IP Addresses of the machines.
```

```
# You will then need to restart the windows remote management service
# on both computers.
```

```
Restart-Service WinRM I
```

```
# On the local computer you are using, you can test by using Test-WSMan
# followed by the name of the remote computer.
```

```
Test-WSMan ACSrv Remote PC Hostname
```

```
wsmid      : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

```
# Now execute a command on the remote system
```

```
Invoke-Command -ComputerName ACSrv `
    -ScriptBlock { Get-ChildItem C:\ } `
    -Credential ArcaneCode
```

Mode	LastWriteTime	Length	Name	PSComputerName
----	-----	-----	----	-----
d----	10/15/2014 12:33 PM		Dev	ACSrv
d----	8/22/2013 10:52 AM		PerfLogs	ACSrv
d-r--	10/15/2014 1:00 PM		Program Files	ACSrv
d----	10/15/2014 1:00 PM		Program Files (x86)	ACSrv
d----	10/15/2014 12:24 PM		PS	ACSrv
d----	10/15/2014 12:19 PM		Sysinternals	ACSrv
d-r--	10/13/2014 4:26 PM		Users	ACSrv

You can also open up a PowerShell window which will execute
on the remote computer

```
Enter-PSSession -ComputerName ACSrv -Credential ArcaneCode
```

Hostname

Username

Navigate to root directory

```
[ACSrv]: PS C:\Users\ArcaneCode\Documents> set-location c:\
[ACSrv]: PS C:\> gci|
```

or

```
PS C:\WINDOWS\system32>
[ACSrv]: PS C:\Users\ArcaneCode\Documents> cd \
[ACSrv]: PS C:\> gci|
```

```
#-----
# Fix for network connection profile public
#-----
Get-NetConnectionProfile
```

```
Set-NetConnectionProfile `
-InterfaceAlias 'vEthernet (HWired)' `
-NetworkCategory Public
```

2. One to One Remoting

- Enable **PSSession** on the remote computer

Configure WinRM and enable PSRemoting on the remote computer

```
PS C:\Windows\system32> Enable-PSRemoting -Force
```

WinRM has been updated to receive requests.
WinRM service type changed successfully.
WinRM service started.

WinRM has been updated for remote management.

Created a WinRM listener on HTTP://* to accept WS-Man requests to any IP on this machine.
WinRM firewall exception enabled.

Now we can create an **interactive session** with a remote computer.

PS C:\Users\user> **Enter-PSSession -ComputerName PC-01**

```
[PC-01]: PS C:\Users\user\Documents>
[PC-01]: PS C:\Users\user\Documents>
[PC-01]: PS C:\Users\user\Documents>
[PC-01]: PS C:\Users\user\Documents> EXIT
PS C:\Users\user>
```

Example 1:-

- To enable RDP with PSSession:

1. Launch PowerShell as Administrator.
2. Create a PS Session with the desired target computer.
3. Type the following command once possession is established:

Set-ItemProperty -Path "HKLM:\System\CurrentControlSet\Control\Terminal Server" -Name "fDenyTSConnections" -Value 0

NOTE: Enabling RDP through PowerShell will not configure the Windows Firewall with the appropriate ports to allow RDP connections.

Type the following:

Enable-NetFirewallRule -DisplayGroup "Remote Desktop"

- To disable RDP

1. Launch PowerShell as Administrator.
2. Create a PS Session with the desired target computer.
3. Type the following command once possession is established:

Set-ItemProperty -Path "HKLM:\System\CurrentControlSet\Control\Terminal Server" -Name "fDenyTSConnections" -Value 1

Example 1:-

- To enable RDP with PowerShell (without PSSession)

1. Launch PowerShell as Administrator.
2. Type the following command and create a script block and use the Invoke-Command cmdlet:

Invoke-Command -Computername "server1", "Server2" -ScriptBlock {Set-ItemProperty -Path "HKLM:\System\CurrentControlSet\Control\Terminal Server" -Name "fDenyTSConnections" -Value 0 }

NOTE: Enabling RDP through PowerShell will not configure the Windows Firewall with the appropriate ports to allow RDP connections.

Type the following:

Invoke-Command -Computername "server1", "Server2" -ScriptBlock {Enable-NetFirewallRule -DisplayGroup "Remote Desktop"}

- To disable RDP

1. Launch PowerShell as Administrator.
2. Type the following command:

Invoke-Command -Computername "server1", "Server2" -ScriptBlock {Set-ItemProperty -Path "HKLM:\System\CurrentControlSet\Control\Terminal Server" -Name "fDenyTSConnections" -Value 1}

Example 2:-

- One to Many Remoting

PS C:\> \$computers = get-content c:\work\computers.txt | New-PSSession -credential avtech\administrator

✓ To check the status of Windows update service

PS C:\> Invoke-Command -scriptblock { get-service wuauserv } -session \$computers

✓ To execute Powershell scripts that reside locally

PS C:\> Invoke-Command -filepath c:\scripts\Weekly.ps1 -session \$computers

Example 3:-

```

PowerShell Remoting Fundamentals
Administrator: Windows PowerShell

PS C:\> $computers = "chi-core01","chi-core02","chi-web02","chi-fp02","chi-hvr2"
PS C:\> icm { get-ciminstance win32_operatingsystem } -ComputerName $computers | select PSComputername,Caption,InstallDate | format-list

PSComputerName : chi-web02
Caption        : Microsoft Windows Server 2012 R2 Standard
InstallDate    : 1/27/2015 3:34:41 PM

PSComputerName : chi-hvr2
Caption        : Microsoft Hyper-V Server 2012 R2
InstallDate    : 10/21/2013 5:18:48 PM

PSComputerName : chi-fp02
Caption        : Microsoft Windows Server 2012 R2 Standard
InstallDate    : 11/12/2013 4:33:51 PM

PSComputerName : chi-core01
Caption        : Microsoft Windows Server 2012 R2 Datacenter
InstallDate    : 10/22/2013 1:29:28 PM

PSComputerName : chi-core02
Caption        : Microsoft Windows Server 2012 R2 Datacenter
InstallDate    : 6/5/2015 8:57:16 AM

PS C:\> icm { tzutil /g } -ComputerName $computers
Central Standard Time
Eastern Standard Time
Eastern Standard Time
Central Standard Time
Eastern Standard Time

PS C:\> icm { [pscustomobject]@{"TimeZone"= (tzutil /g)} } -ComputerName $computers

TimeZone                PSComputerName                RunspaceId
-----
Central Standard Time   chi-core02                     bb7b8426-5d52-48d3-9245-23ab9caea26a
Central Standard Time   chi-core01                     99b205dd-aedd-4d2b-b4f2-eeb06e330433
Eastern Standard Time   chi-hvr2                       6c9d8448-1ef2-42f8-9c4c-4b82b6a2b5f8
Eastern Standard Time   chi-web02                     1b3cf038-6fec-4137-b249-37c3bf5d15be
Eastern Standard Time   chi-fp02                      40dfd038-db9a-47af-9661-4f399e9d45e7

PS C:\> icm { [pscustomobject]@{"TimeZone"= (tzutil /g)} } -ComputerName $computers | Select * -exclude runspaceid

TimeZone                PSComputerName
-----
Central Standard Time   chi-core02
Eastern Standard Time   chi-hvr2
Central Standard Time   chi-core01
Eastern Standard Time   chi-fp02
Eastern Standard Time   chi-web02
  
```



```

PS C:\> $dcs = New-PSSession -ComputerName chi-dc01,chi-dc02,chi-dc04 -Credential globomantics\administrator
PS C:\> $dcs

```

Id	Name	ComputerName	State	ConfigurationName	Availability
2	Session2	chi-dc02	Opened	Microsoft.PowerShell	Available
1	Session1	chi-dc01	Opened	Microsoft.PowerShell	Available
3	Session3	chi-dc04	Opened	Microsoft.PowerShell	Available

```

PS C:\> Get-PSSession

```

Id	Name	ComputerName	State	ConfigurationName	Availability
2	Session2	chi-dc02	Opened	Microsoft.PowerShell	Available
1	Session1	chi-dc01	Opened	Microsoft.PowerShell	Available
3	Session3	chi-dc04	Opened	Microsoft.PowerShell	Available

```

PS C:\> icm {get-service adws,dns,kdc } -session $dcs | Sort Status

```

Status	Name	DisplayName	PSComputerName
Running	adws	Active Directory Web Services	chi-dc04
Running	kdc	Kerberos Key Distribution Center	chi-dc01
Running	kdc	Kerberos Key Distribution Center	chi-dc04
Running	dns	DNS Server	chi-dc04
Running	dns	DNS Server	chi-dc01
Running	adws	Active Directory Web Services	chi-dc02
Running	adws	Active Directory Web Services	chi-dc01
Stopped	kdc	Kerberos Key Distribution Center	chi-dc02
Stopped	dns	DNS Server	chi-dc02

```

PS C:\> icm {get-service adws,dns,kdc } -session $dcs | where {$_.status -eq 'stopped'}

```

Status	Name	DisplayName	PSComputerName
Stopped	dns	DNS Server	chi-dc02
Stopped	kdc	Kerberos Key Distribution Center	chi-dc02

```

PS C:\> icm {get-service adws,dns,kdc | Where {$_.status -ne "running" }} -session $dcs

```

Status	Name	DisplayName	PSComputerName
Stopped	dns	DNS Server	chi-dc02
Stopped	kdc	Kerberos Key Distribution Center	chi-dc02

```

PS C:\> icm {get-service adws,dns,kdc | Where {$_.status -ne "running" } | start-service -PassThru } -session $dcs

```

Status	Name	DisplayName	PSComputerName
Running	dns	DNS Server	chi-dc02
Running	kdc	Kerberos Key Distribution Center	chi-dc02

Next...

1. PowerShell Background Jobs
2. PowerShell Scheduled Jobs

Scheduled Jobs

Cmdlets for creating the job definition in the PSScheduledJob module

Scheduled job definition is stored to disk

C:\Users\<YOU>\AppData\Local\Microsoft\Windows\PowerShell\ScheduledJobs\<JOB NAME>

Results also written to disk



```
PS C:\> help about_scheduled_jobs
```

Examples

1. if elseif else

```
Clear-Host
```

```
If (10 -gt 11)
{
  Write-Host "Yes"
}
elseif (11 -lt 10)
{
  Write-Host "This time, yes"
}
elseif (20 -gt 40)
{
  Write-Host "Third time was a charm"
}
else
{
  Write-Host "You're really terrible at math, aren't you?"
}
```

Result:

```
You're really terrible at math, aren't you?
```

Filtering and Sorting

```
Get-Service c* | where {$_.status -eq "stopped"}
```

```
Get-Service c* | where {$_.status -eq "running"}
```

```
Get-Process | where {$_.ProcessName -eq "chrome"}
```

```
Get-Process | where {$_.ProcessName -eq "chrome"} | Sort-Object VM
```

Creating an Array

PowerShell will treat any comma separated list as an array

```
PS C:\> $arr = 4,6,8,10,12
```

PowerShell cmdlets typically write an array of objects to the pipeline

```
PS C:\> $services = get-service s*
```

Create an array starting with one element

```
PS C:\> $arr = ,1
```

Create an empty array

```
PS C:\> $arr=@()
```

Test if something is an array

```
PS C:\> $arr -is [array]
```

The variable used for the array is an object in itself

```
PS C:\> $arr.count
```

Reference:-

TN <http://blogs.technet.com/b/heyscriptingguy/>

<http://arcanecode.com/>

1. PowerShell Background Jobs
2. PowerShell Scheduled Jobs