POWERSHELL - OVERVIEW

Windows PowerShell is a **command-line shell** and **scripting language** designed especially for system administration. It's analogue in Linux is called as Bash Scripting. Built on the .NET Framework, Windows PowerShell helps IT professionals to control and automate the administration of the Windows operating system and applications that run on Windows Server environment.

Windows PowerShell commands, called **cmdlets**, let you manage the computers from the command line. Windows PowerShell providers let you access data stores, such as the Registry and Certificate Store, as easily as you access the file system.

In addition, Windows PowerShell has a rich expression parser and a fully developed scripting language. So in simple words you can complete all the tasks that you do with GUI and much more.

PowerShell ISE

The Windows PowerShell **Integrated Scripting Environment** ISEISE is a host application for Windows PowerShell. In Windows PowerShell ISE, you can run commands and write, test, and debug scripts in a single Windows-based graphic user interface with multiline editing, tab completion, syntax coloring, selective execution, context-sensitive help, and support for right-to-left languages.

You can use menu items and keyboard shortcuts to perform many of the same tasks that you would perform in the Windows PowerShell console. For example, when you debug a script in the Windows PowerShell ISE, to set a line breakpoint in a script, right-click the line of code, and then click **Toggle Breakpoint**.

PowerShell Basic Commands

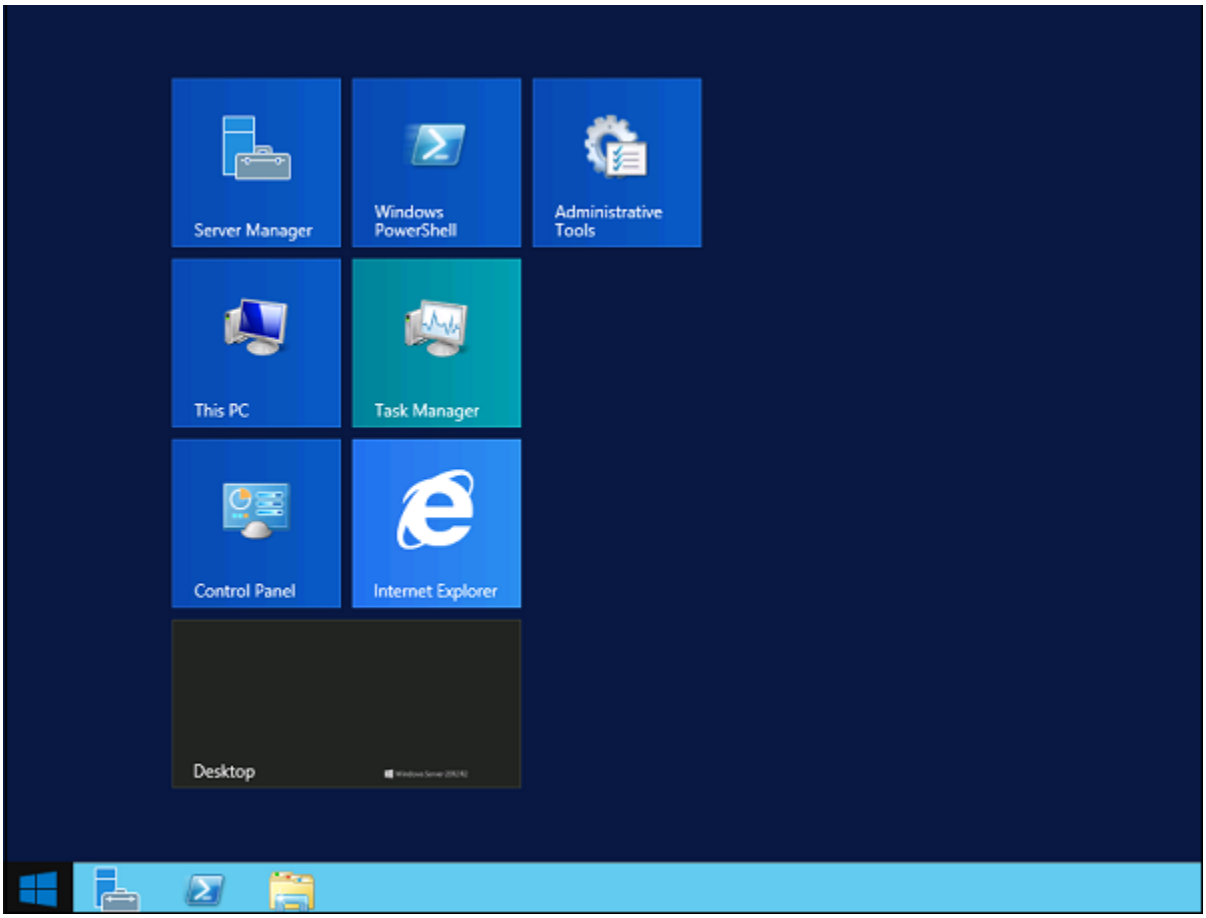
There are a lot of PowerShell commands and it is very difficult to put in all these commands in this tutorial, we will focus on some of the most important as well as basic commands of PowerShell.

The first step is to go to the Get-Help command which gives you an explanation about how to give a command and its parameter.

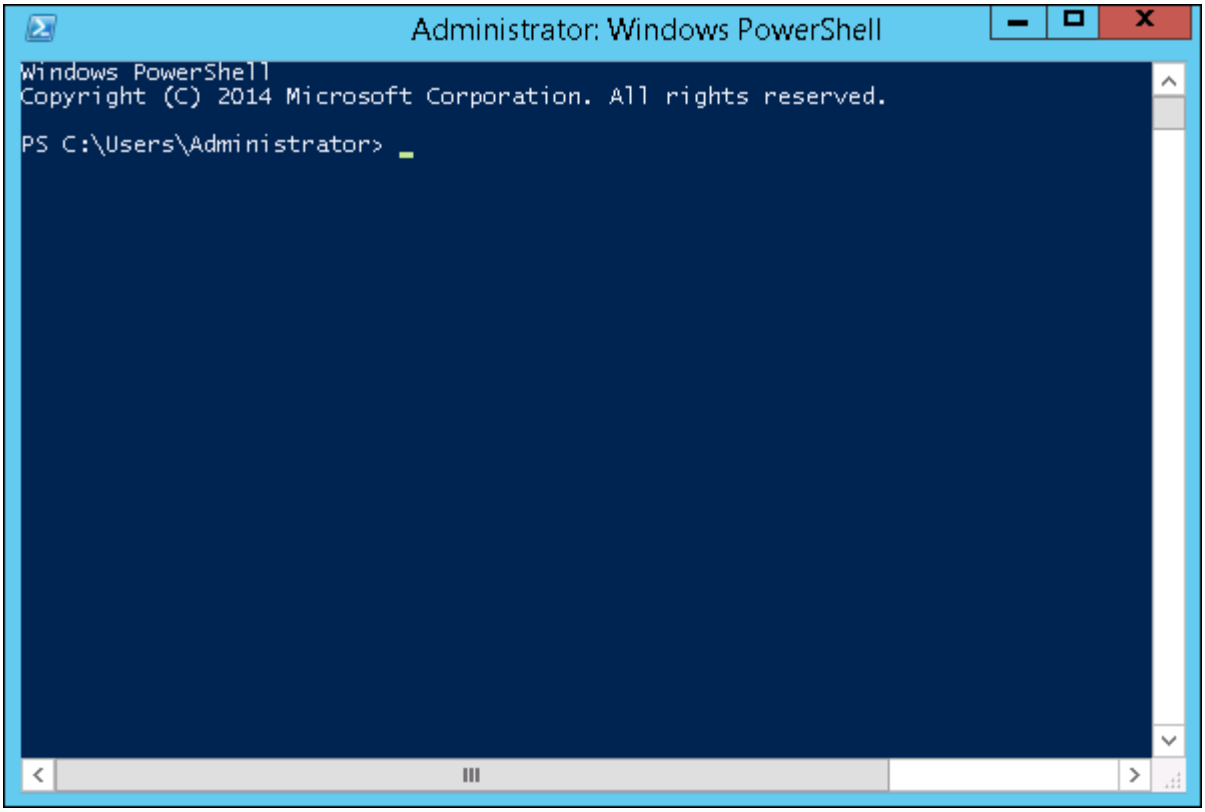
```
PS C:\Users\Administrator> Get-Help Add-AppxPackage

NAME
    Add-AppxPackage

SYNTAX
    Add-AppxPackage [-Path] <string> [-DependencyPath <string[]>] [-ForceApplicationS
    [-WhatIf] [-Confirm]  [<CommonParameters>]

    Add-AppxPackage [-Path] <string> -Register [-DependencyPath <string[]>] [-Disable
    [-ForceApplicationShutdown] [-InstallAllResources] [-WhatIf] [-Confirm]  [<Common

    Add-AppxPackage [-Path] <string> -Update [-DependencyPath <string[]>] [-ForceAppl
    [-InstallAllResources] [-WhatIf] [-Confirm]  [<CommonParameters>]

    Add-AppxPackage -MainPackage <string> [-Register] [-DependencyPackages <string[]>
    [-WhatIf] [-Confirm]  [<CommonParameters>]


ALIASES
    None
```

POWERSHELL - ENVIRONMENT SETUP

PowerShell Icon can be found in the task bar and in the start menu. Just by clicking on the icon, it will open.



To open it, just click on the icon and then the following screen will open and it means that PowerShell is ready for you to work on.

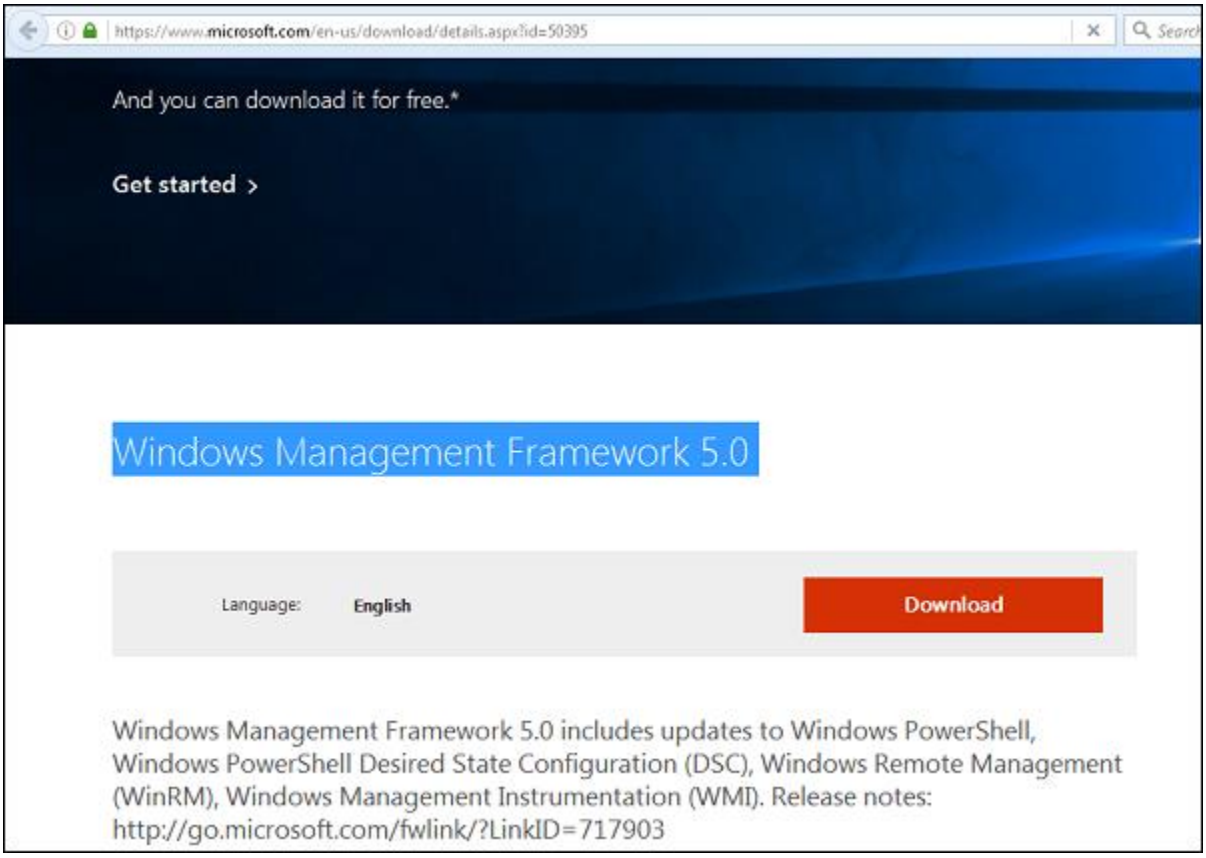


PowerShell Version

The latest version of PowerShell is 5.0 and to check what is installed in our server we type the following command – **:$PSVersionTable** as shown in the following screenshot and from the screen we also know that we have PSVersion 4.0

```
PS C:\Users\Administrator> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      4.0
WSManStackVersion              3.0
SerializationVersion           1.1.0.1
CLRVersion                     4.0.30319.34209
BuildVersion                   6.3.9600.17400
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0}
PSRemotingProtocolVersion      2.2


PS C:\Users\Administrator>
```

To update with the latest version where it has more Cmdlets we have to download **Windows Management Framework 5.0** from the following link – https://www.microsoft.com/en-us/download/details.aspx?id=50395 and install it.

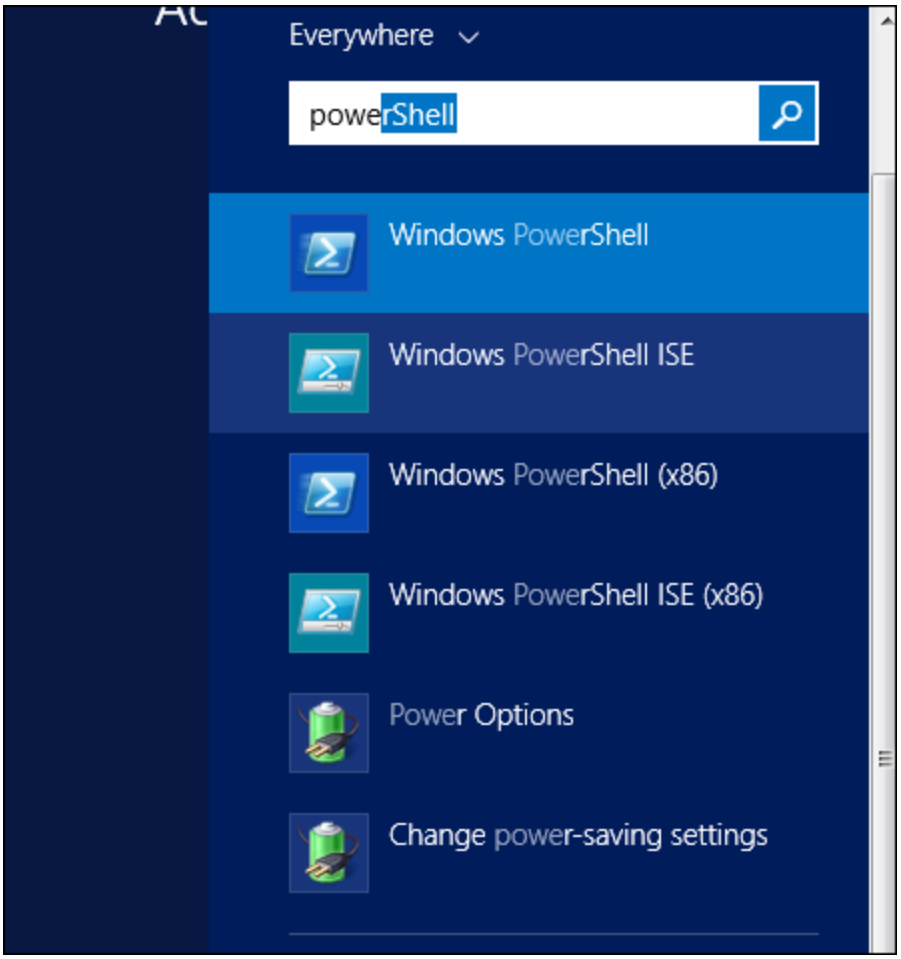


PowerShell ISE

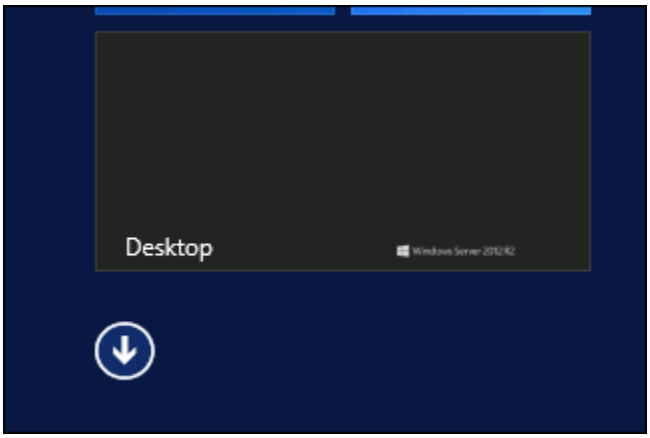
The Windows PowerShell **Integrated Scripting Environment** ISEISE is a host application for Windows PowerShell. In Windows PowerShell ISE, you can run commands and write, test, and debug scripts in a single Windows-based graphic user interface with multiline editing, tab completion, syntax coloring, selective execution, context-sensitive help, and support for right-to-left languages.

You can use menu items and keyboard shortcuts to perform many of the same tasks that you would perform in the Windows PowerShell console. For example, when you debug a script in the Windows PowerShell ISE, to set a line breakpoint in a script, right-click the line of code, and then click **Toggle Breakpoint**.

To open it you just go to Start - Search and then Type - PowerShell as shown in the following screenshot.

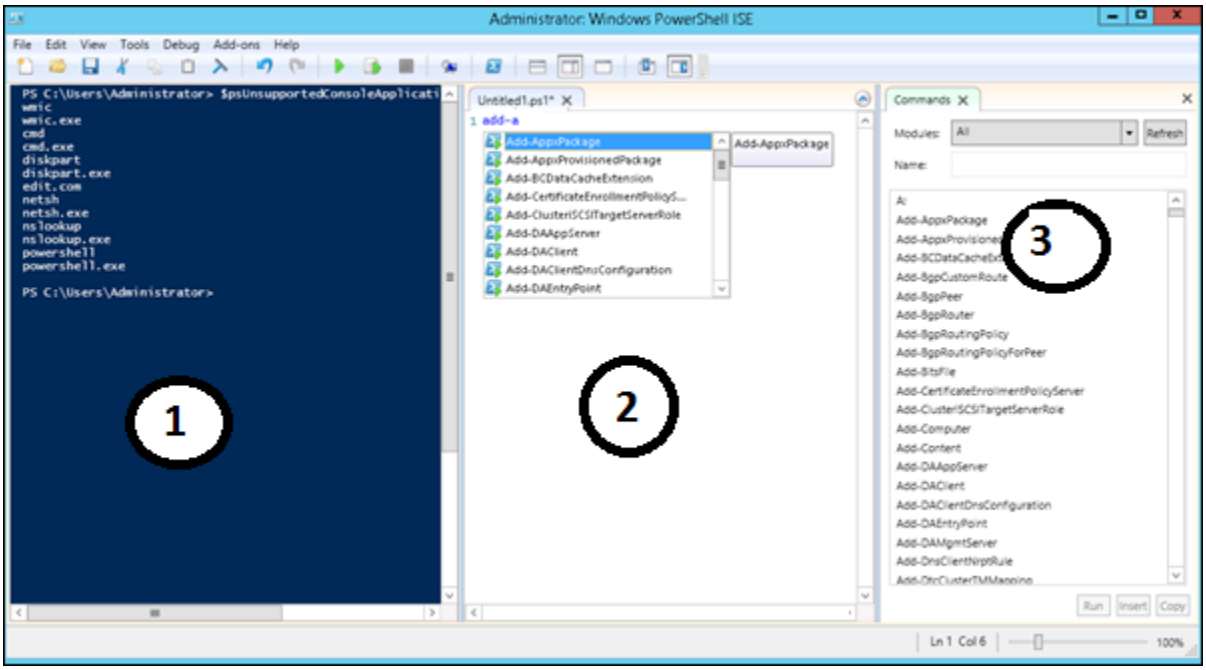


Then click on Windows PowerShell ISE. Or click on the downward Arrow as shown in the following screenshot.



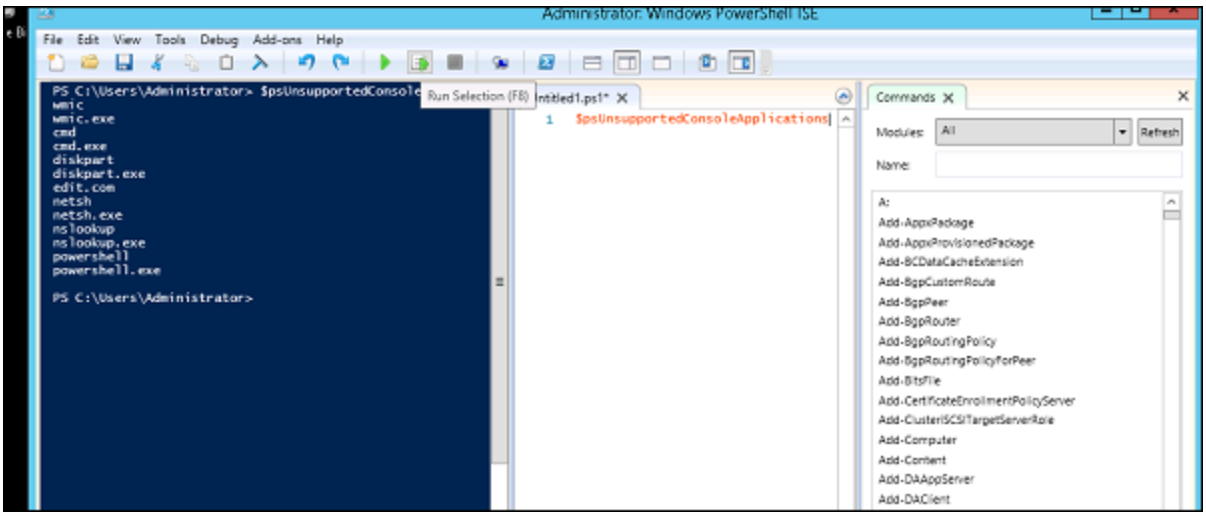
It will list all the applications installed on the server and then click on Windows PowerShell ISE.

The following table will be open −



It has three sections, which include - The **PowerShell Console** with number 1, then **Scripting File**number 2 and the third is the **Command Module** where you can find the module.

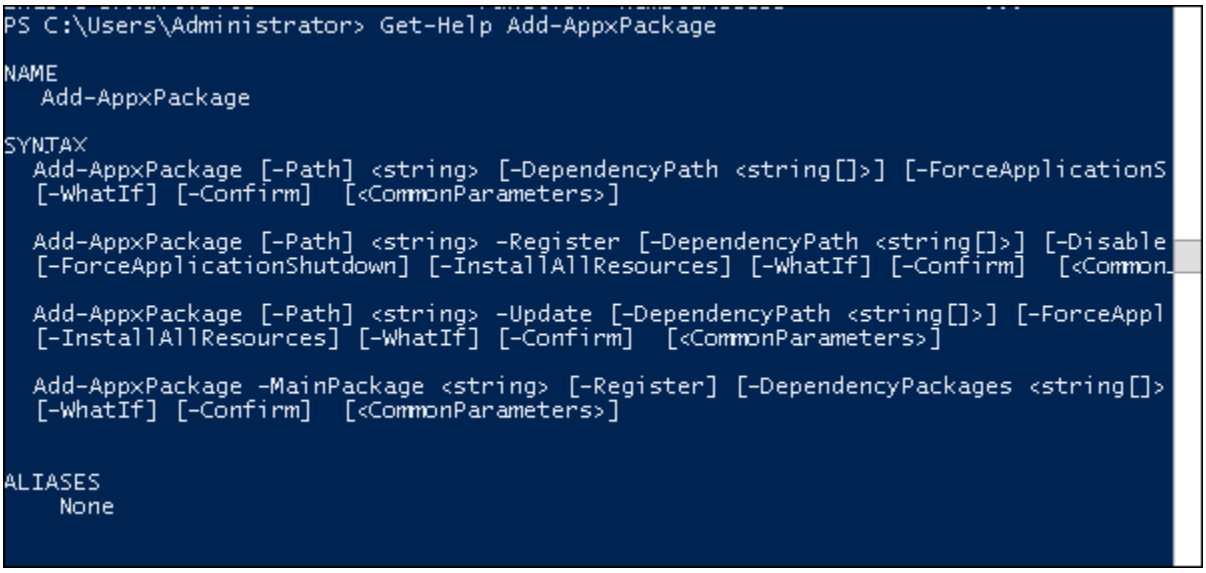
While creating the script you can run directly and see the result like the following example −



PowerShell Basic Commands

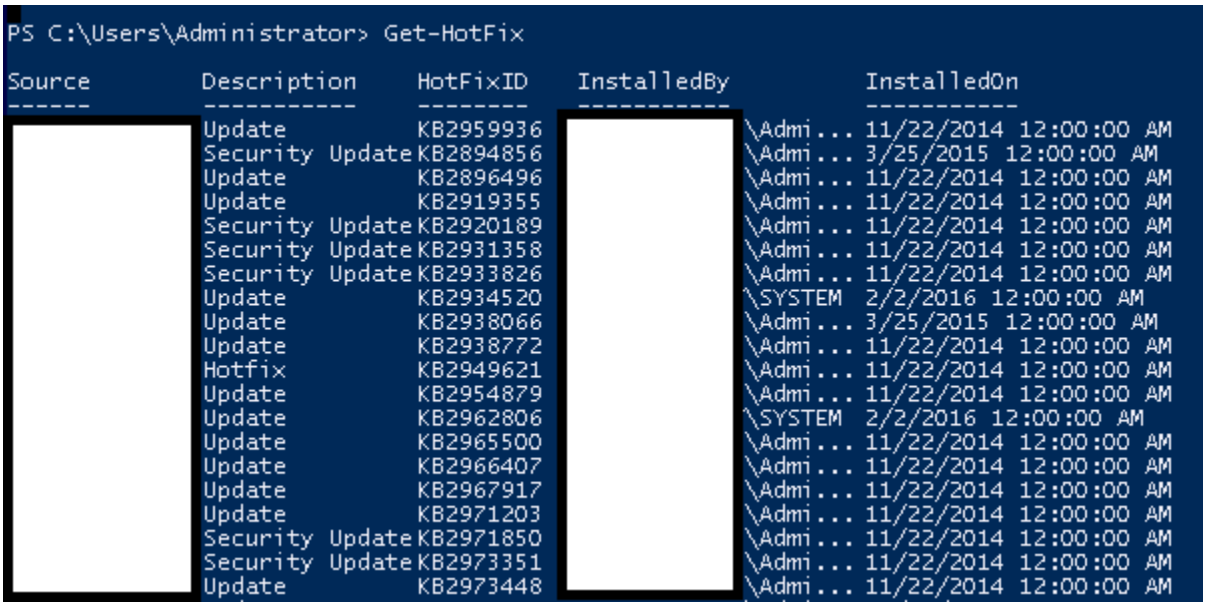
There are a lot of PowerShell commands and it is very difficult to put in all these commands in this tutorial, we will focus on some of the most important as well as basic commands of PowerShell.

The first step is to go to the Get-Help command which gives you an explanation about how to give a command and its parameter.



To get the list of Updates −

- Get-HotFix and to install a hot fix as follows
- Get-HotFix -id kb2741530



POWERSHELL - CMDLETS

A cmdlet or "Command let" is a lightweight command used in the Windows PowerShell environment. The Windows PowerShell runtime invokes these cmdlets at command prompt. You can create and invoke them programmatically through Windows PowerShell APIs.

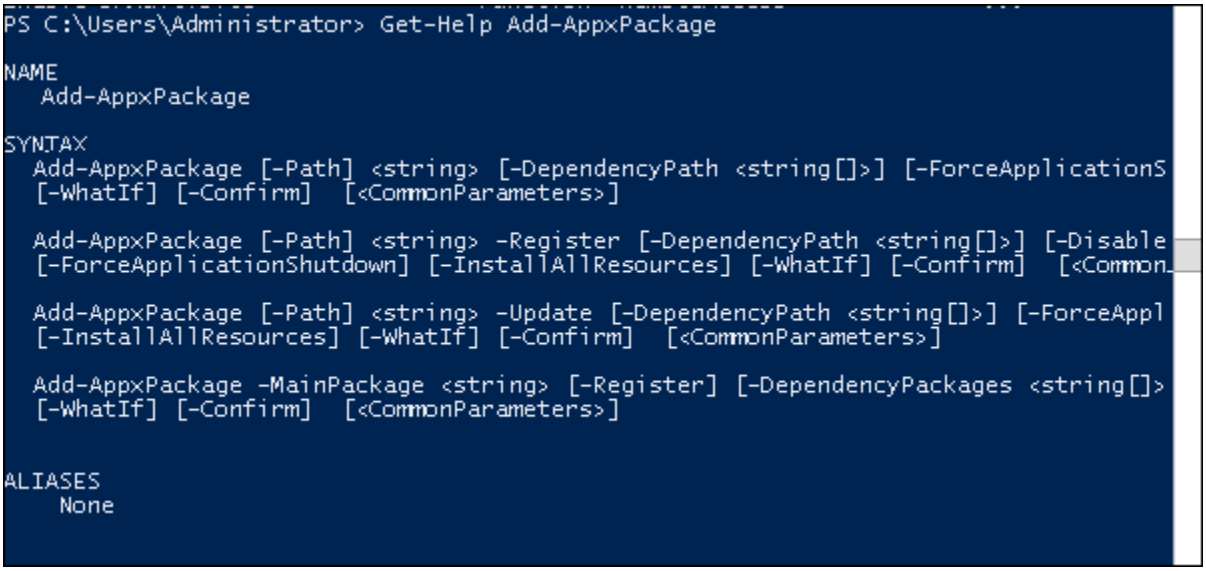
Cmdlet vs Command

Cmdlets are way different from commands in other command-shell environments in the following manners −

- Cmdlets are .NET Framework class objects; and not just stand-alone executables.
- Cmdlets can be easily constructed from as few as a dozen lines of code.
- Parsing, error presentation, and output formatting are not handled by cmdlets. It is done by the Windows PowerShell runtime.
- Cmdlets process works on objects not on text stream and objects can be passed as output for pipelining.
- Cmdlets are record-based as they process a single object at a time.

Getting Help

The first step is to go to the Get-Help command which gives you an explanation about how to give a command and its parameter.



POWERSHELL - FILES AND FOLDER OPERATIONS

Following are the examples of powershell scripts on Files and Folders.

Sr.No.	Operation & Description
1	Creating Folders `New-Item -Path 'D:\temp\Test Folder' -ItemType Directory` **New-Item** cmdlet is used to create a directory by passing the path using -Path as path of the directory and -ItemType as Directory.
2	Creating Files `New-Item -Path 'D:\temp\Test Folder\Test File.txt' -ItemType File`
3	Copying Folders `Copy-Item 'D:\temp\Test Folder' 'D:\temp\Test Folder1'`
4	Copying Files `Copy-Item 'D:\temp\Test Folder\Test File.txt' 'D:\temp\Test Folder1\Test File1.txt'`

5	Deleting Folders
	`Remove-Item 'D:\temp\Test Folder1'`
	`Remove-Item 'D:\temp\Test Folder' -Recurse`

6	Deleting Files

	Example Script to show how to delete filess using PowerShell scripts.
7	Moving Folders Example Script to show how to move folderss using PowerShell scripts.
8	Moving Files `Move-Item D:\temp\Test\Test.txt D:\temp\Test1`
9	Rename Folders `Rename-Item D:\temp\Test D:\temp\Test1`
10	Rename Files `Rename-Item D:\temp\Test\test.txt test1.txt`

11	Retrieving Item
	`Get-Content D:\temp\test\test.txt`
	`(Get-Content D:\temp\test\test.txt).length`

12	Check Folder Existence `Test-Path D:\temp\test`
13	Check File Existence `Test-Path D:\temp\test\test2.txt`

POWERSHELL - DATE AND TIME OPERATIONS

Following are the examples of powershell scripts on System Date and Time.

Sr.No.	Operation & Description

1	Get System Date Get-Date -displayhint date
2	Set System Date Example Script to show how to set system date using PowerShell scripts.
3	Get System Time Example Script to show how to get system time using PowerShell scripts.
4	Set System Time Example Script to show how to set system time using PowerShell scripts.

POWERSHELL - FILE I/O OPERATIONS

Following are the examples of powershell scripts of creating and reading different types of files.

Sr.No.	Operation & Description
1	Create Text File Example Script to show how to create a text file using PowerShell scripts.
2	Read Text File Example Script to show how to read a text file using PowerShell scripts.
3	Create XML File Example Script to show how to create a XML file using PowerShell scripts.

4	Read XML File Example Script to show how to read a XML file using PowerShell scripts.
5	Create CSV File Example Script to show how to create a CSV file using PowerShell scripts.
6	Read CSV File Example Script to show how to read a CSV file using PowerShell scripts.
7	Create HTML File Example Script to show how to create a HTML file using PowerShell scripts.
8	Read HTML File Example Script to show how to read a HTML file using PowerShell scripts.
9	Erasing file content Example Script to show how to erase file contents using PowerShell scripts.
10	Append Text Data Example Script to show how to append text to a file contents using PowerShell scripts.

POWERSHELL - ADVANCED CMDLETS

Cmdlets

A cmdlet or "Command let" is a lightweight command used in the Windows PowerShell environment. The Windows PowerShell runtime invokes these cmdlets at command prompt. You can create and invoke them programmatically through Windows PowerShell APIs. Following are advanced usage example of cmdlets.

Sr.No.	Cmdlet Type & Description

1

Get-Unique Cmdlet

```
$list = "one","two","two","three","four","five"

$list | sort | get-unique
```

```
PS D:\temp\sample\output> Get-ChildItem | Sort-Object Length


    Directory: D:\temp\sample\output


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         9/9/2018   8:19 AM              0 one.log
-a----         9/9/2018   8:18 AM              0 T2.log
-a----         9/9/2018   8:19 AM              0 three.log
-a----         9/9/2018   8:19 AM              0 two.log
-a----         9/8/2018  11:51 PM             14 t1.txt
d-----         9/9/2018   8:17 AM                Bin Folder
PS D:\temp\sample\output> Get-ChildItem | Sort-Object | Where-Object Name -
EQ one.log | Select-Object Name

Name
----
one.log
```

2

Group-Object Cmdlet

The Group-Object cmdlet puts objects into groups based on a property.

```
PS D:\temp\sample\output> Get-Service | Group-Object -Property StartType |
select -First 2

Count Name                      Group
----- ----                      -----
   85 Automatic                 {AdobeARMservice, AudioEndpointBuilder,
AudioSrv, avbackup...}
  108 Manual                    {AdobeFlashPlayerUpdateSvc, AeLookupSvc,
ALG, AppIDSvc...}
```

3

Measure-Object Cmdlet

The **Measure-Object** cmdlet calculates the property values of certain types of object. **Measure-Object** performs three types of measurements, depending on the parameters in the command.

The **Measure-Object** cmdlet performs calculations on the property values of objects. It can count objects and calculate the minimum, maximum, sum, and

	average of the numeric values. For text objects, it can count and calculate the number of lines, words, and characters.
	
	`get-content D:\temp\test\test.txt | measure-object -character -line -word`
	
	`Lines Words Characters Property`
	`----- ----- ---------- --------`
	
	`1 3 29`
	
	`Counts files and folder in the directory`
	
	`Get-ChildItem | Measure-Object`

4	Compare-Object Cmdlet Example program to showcase Compare-Object Cmdlet.

5	Format-List Cmdlet
	
	The **Format-List** cmdlet formats the output of a command as a list of properties in which each property is displayed on a separate line. You can use **Format-List** to format and display all or selected properties of an object as a list (format-list *).
	
	Because more space is available for each item in a list than in a table, PowerShell displays more properties of the object in the list, and the property values are less likely to be truncated.
	
	`PS D:\temp\sample\output> Get-Service | select -First 2`
	
	`Status Name DisplayName`
	`------ ---- -----------`
	`Running AdobeARMservice Adobe Acrobat Update Service`
	`Stopped AdobeFlashPlaye... Adobe Flash Player Update Service`
	
	`PS D:\temp\sample\output> Get-Service | select -First 2 | Format-List`
	
	`Name : AdobeARMservice`
	`DisplayName : Adobe Acrobat Update Service`
	`Status : Running`
	`DependentServices : {}`
	`ServicesDependedOn : {}`
	`CanPauseAndContinue : False`
	`CanShutdown : False`
	`CanStop : True`
	`ServiceType : Win32OwnProcess`

```
Name                  : AdobeFlashPlayerUpdateSvc
DisplayName           : Adobe Flash Player Update Service
Status                : Stopped
DependentServices     : {}
ServicesDependedOn    : {}
CanPauseAndContinue   : False
CanShutdown           : False
CanStop               : False
ServiceType           : Win32OwnProcess
```

```
PS C:\> Get-Process | Format-List -Property name, displayname
```

```
$A = Get-ChildItem D:\temp\test\*.txt

Format-List -InputObject $A

More example:

Get-Service | Format-List
```

6	Format-Wide Cmdlet Example program to showcase Format-Wide Cmdlet.
7	Where-Object Cmdlet **Where-Object** cmdlet can be used to select objects having particular property values from the collection of objects that are passed to it. Get stopped services.

```
Get-Service | Where-Object {$_.Status -eq "Stopped"}
```

Get processes based on process name.

```
Get-Process | Where-Object {$_.ProcessName -Match "^p.*"}
```

```
Alternative of above approach
PS D:\temp\sample\output> Get-Service | Where-Object -Property Status -EQ
"Stopped" | Sort-Object -Property Name |select -First 10

Status   Name               DisplayName
------   ----               -----------
Stopped  AdobeFlashPlaye... Adobe Flash Player Update Service
Stopped  AeLookupSvc        Application Experience
```

```
Stopped  ALG                 Application Layer Gateway Service
Stopped  AppIDSvc            Application Identity
Stopped  Appinfo             Application Information
Stopped  AppMgmt             Application Management
Stopped  aspnet_state        ASP.NET State Service
Stopped  AxInstSV            ActiveX Installer (AxInstSV)
Stopped  Browser             Computer Browser
Stopped  c2wts               Claims to Windows Token Service



PS D:\temp\sample\output> Get-Process | Where-Object ProcessName -Like
"*win*" | sort -Property ProcessName -Descending

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
   1375     110   224592     281804     770.89  12136   1 WINWORD
    115       9     3600       8500       0.75    780   1 winlogon
     81      10     1672       5000       0.22    820   0 wininit
```

8	Get-ChildItem Cmdlet `Get-ChildItem -Name`
9	ForEach-Object Cmdlet Example program to showcase ForEach-Object Cmdlet.
10	Start-Sleep Cmdlet Example program to showcase Start-Sleep Cmdlet.
11	Read-Host Cmdlet Example program to showcase Read-Host Cmdlet.
12	Select-Object Cmdlet `PS D:\temp\sample\output> Get-Process | sort -Property ProcessName | Select-Object -Property ProcessName -First 10`

13	Sort-Object Cmdlet
	`Get-Process | Sort-Object -Property WS | Select-Object -Last 5`
	`D:\temp\sample\output> Get-Process | Where-Object ProcessName -Like "*Id*"| sort CPU -Descending`

```
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    675      37    26100      41032       3.63    7580   1
ServiceHub.IdentityHost
    329      19     8096      15572       0.37    2852   0 MSOIDSVC
     72       6     1468       4388       0.00    2896   0 MSOIDSVCM
      0       0        0         24                  0   0 Idle
```

14 [Write-Warning Cmdlet]

Example program to showcase Write-Warning Cmdlet.

15 [Write-Host Cmdlet]

Example program to showcase Write-Host Cmdlet.

16 [Invoke-Item Cmdlet]

```
Open the file in notepad

Invoke-Item "D:\Temp\test.txt"

Open all the text file under Temp folder

Invoke-Item "D:\Temp\*.txt"
```

17 [Invoke-Expression Cmdlet]

Invoke-Expression cmdlet is used to perform a command or expression on local computer.

```
> $Command = 'Get-Process'

> Invoke-Expression $Command
```

18 [Measure-Command Cmdlet]

The `Measure-Command` cmdlet runs a script block or cmdlet internally, times the execution of the operation, and returns the execution time.

```
PS D:\temp\sample\output> Measure-Command {Get-Service | select -First 10}


Days              : 0
Hours             : 0
```

```
Minutes           : 0
Seconds           : 0
Milliseconds      : 5
Ticks             : 53473
TotalDays         : 6.18900462962963E-08
TotalHours        : 1.48536111111111E-06
TotalMinutes      : 8.91216666666667E-05
TotalSeconds      : 0.0053473
TotalMilliseconds : 5.3473
PS D:\temp\sample\output> Measure-Command {Get-Service | select -First 10} |
select -Property TotalMilliseconds

TotalMilliseconds
-----------------
           4.7609
```

```
Measure-Command { Get-EventLog "Windows PowerShell" }
```

19	Invoke-History Cmdlet Example program to showcase Invoke-History Cmdlet.
20	Add-History Cmdlet Example program to showcase Add-History Cmdlet.
21	Get-History Cmdlet Example program to showcase Get-History Cmdlet.
22	Get-Culture Cmdlet Example program to showcase Get-Culture Cmdlet.

POWERSHELL - SCRIPTING

Windows PowerShell is a **command-line shell** and **scripting language** designed especially for system administration. Its analogue in Linux is called as Bash Scripting. Built on the .NET Framework, Windows PowerShell helps IT professionals to control and automate the administration of the Windows operating system and applications that run on Windows Server environment.

Windows PowerShell commands, called **cmdlets**, let you manage the computers from the command line. Windows PowerShell providers let you access data stores, such as the Registry and Certificate Store, as easily as you access the file system.

In addition, Windows PowerShell has a rich expression parser and a fully developed scripting language. So in simple words you can complete all the tasks that you do with GUI and much more. Windows PowerShell Scripting is a fully developed scripting language and has a rich expression parser/

Features

- **Cmdlets** − Cmdlets perform common system administration tasks, for example managing the registry, services, processes, event logs, and using Windows Management Instrumentation WMIWMI.
- **Task oriented** − PowerShell scripting language is task based and provide supports for existing scripts and command-line tools.
- **Consistent design** − As cmdlets and system data stores use common syntax and have common naming conventions, data sharing is easy. The output from one cmdlet can be pipelined to another cmdlet without any manipulation.
- **Simple to Use** − Simplified, command-based navigation lets users navigate the registry and other data stores similar to the file system navigation.
- **Object based** − PowerShell possesses powerful object manipulation capabilities. Objects can be sent to other tools or databases directly.
- **Extensible interface.** − PowerShell is customizable as independent software vendors and enterprise developers can build custom tools and utilities using PowerShell to administer their software.

Variables

PowerShell variables are named objects. As PowerShell works with objects, these variables are used to work with objects.

Creating variable

Variable name should start with $ and can contain alphanumeric characters and underscore in their names. A variable can be created by typing a valid variable name.

Type the following command in PowerShell ISE Console. Assuming you are in D:\test folder.

```
$location = Get-Location
```

Here we've created a variable $location and assigned it the output of Get-Location cmdlet. It now contains the current location.

Using variable

Type the following command in PowerShell ISE Console.

```
$location
```

Output

You can see following output in PowerShell console.

```
Path
----
D:\test
```

Getting information of variable

Get-Member cmdlet can tell the type of variable being used. See the example below.

```
$location | Get-Member
```

other alternative

Get-service | select *

Output

You can see following output in PowerShell console.

```
   TypeName: System.Management.Automation.PathInfo

Name           MemberType   Definition
----           ----------   ----------
Equals         Method       bool Equals(System.Object obj)
GetHashCode    Method       int GetHashCode()
GetType        Method       type GetType()
ToString       Method       string ToString()
Drive          Property     System.Management.Automation.PSDriveInfo Drive
{get;}
Path           Property     System.String Path {get;}
Provider       Property     System.Management.Automation.ProviderInfo Provider
{get;}
ProviderPath   Property     System.String ProviderPath {get;}
```

POWERSHELL - SPECIAL VARIABLES

PowerShell Special variables store information about PowerShell. These are also called automatic variables. Following is the list of automatic variables −

Operator	
$$	Represents the last token in the last line received by the session.
$?	Represents the execution status of the last operation. It contains TRUE if the last
$^	Represents the first token in the last line received by the session.
$_	Same as $PSItem. Contains the current object in the pipeline object. You can use
$ARGS	Represents an array of the undeclared parameters and/or parameter values that ar
$CONSOLEFILENAME	Represents the path of the console file .psc1.psc1 that was most recently used in
$ERROR	Represents an array of error objects that represent the most recent errors.

$EVENT	Represents a PSEventArgs object that represents the event that is being processe
$EVENTARGS	Represents an object that represents the first event argument that derives from Ev
$EVENTSUBSCRIBER	Represents a PSEventSubscriber object that represents the event subscriber of th
$EXECUTIONCONTEXT	Represents an EngineIntrinsics object that represents the execution context of the
$FALSE	Represents FALSE. You can use this variable to represent FALSE in commands
$FOREACH	Represents the enumerator nottheresultingvaluesnottheresultingvalues of a ForE
$HOME	Represents the full path of the user's home directory.
$HOST	Represents an object that represents the current host application for PowerShell.
$INPUT	Represents an enumerator that enumerates all input that is passed to a function.
$LASTEXITCODE	Represents the exit code of the last Windows-based program that was run.
$MATCHES	The $Matches variable works with the -match and -notmatch operators.
$MYINVOCATION	MyInvocationispopulatedonlyforscripts,function,andscriptblocks.PSScriptRo automatic variable contain information about the invoker or calling script, not th
$NESTEDPROMPTLEVEL	Represents the current prompt level.
$NULL	$null is an automatic variable that contains a NULL or empty value. You can use
$PID	Represents the process identifier PIDPID of the process that is hosting the curren
$PROFILE	Represents the full path of the PowerShell profile for the current user and the cur
$PSCMDLET	Represents an object that represents the cmdlet or advanced function that is being
$PSCOMMANDPATH	Represents the full path and file name of the script that is being run.
$PSCULTURE	Represents the name of the culture currently in use in the operating system.
$PSDEBUGCONTEXT	While debugging, this variable contains information about the debugging environ
$PSHOME	Represents the full path of the installation directory for PowerShell.
$PSITEM	Same as $_. Contains the current object in the pipeline object.
$PSSCRIPTROOT	Represents the directory from which a script is being run.

$PSSENDERINFO	Represents information about the user who started the PSSession, including the u
$PSUICULTURE	Represents the name of the user interface UIUI culture that is currently in use in
$PSVERSIONTABLE	Represents a read-only hash table that displays details about the version of Powe
$SENDER	Represents the object that generated this event.
$SHELLID	Represents the identifier of the current shell.
$STACKTRACE	Represents a stack trace for the most recent error.
$THIS	In a script block that defines a script property or script method, the $This variabl
$TRUE	Represents TRUE. You can use this variable to represent TRUE in commands ar

POWERSHELL - OPERATORS

PowerShell provides a rich set of operators to manipulate variables. We can divide all the PowerShell operators into the following groups −

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Redirectional Operators
- Spilt and Join Operators
- Type Operators
- Unary Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators −

Assume integer variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
+ AdditionAddition	Adds values on either side of the operator.	A + B will give 30
- SubtractionSubtraction	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* MultiplicationMultiplication	Multiplies values on either side of the operator.	A * B will give 200
/ DivisionDivision	Divides left-hand operand by right-hand operand.	B / A will give 2
% ModulusModulus	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0

The Comparison Operators

Following are the assignment operators supported by PowerShell language −

Assume integer variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
eq equalsequals	Compares two values to be equal or not.	A -eq B will give false
ne notequalsnotequals	Compares two values to be not equal.	A -ne B will give

		true
gt greaterthangreaterthan	Compares first value to be greater than second one.	B -gt A will give true
ge greaterthanorequalstogreaterthanorequalsto	Compares first value to be greater than or equals to second one.	B -ge A will give true
lt lessthanlessthan	Compares first value to be less than second one.	B -lt A will give false
le lessthanorequalstolessthanorequalsto	Compares first value to be less than or equals to second one.	B -le A will give true

The Assignment Operators

Following are the assignment operators supported by PowerShell language −

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left	C += A is equivalent to C = C

	operand and assign the result to left operand.	+ A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A

The Logical Operators

The following table lists the logical operators −

Assume Boolean variables A holds true and variable B holds false, then −

Show Examples

Operator	Description	Example
AND logicalandlogicaland	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	A−ANDBA−ANDB is false
OR logicalorlogicalor	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	A−ORBA−ORB is true
NOT logicalnotlogicalnot	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	-NOTA−ANDBA−ANDBis true

Miscellaneous Operators

Following are various important operators supported by PowerShell language −

Show Examples

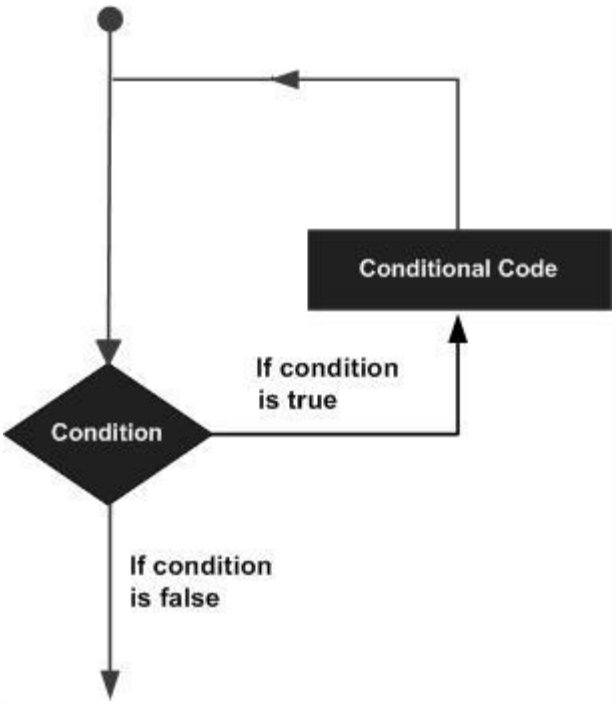
Operator	Description	Example
> RedirectionalOpeatorRedirectionalOpeator	Redirectional operator. Assigns output to be printed into the redirected file/output device.	dir > test.log will print the directory listing in test.log file

POWERSHELL - LOOPING

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



PowerShell programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

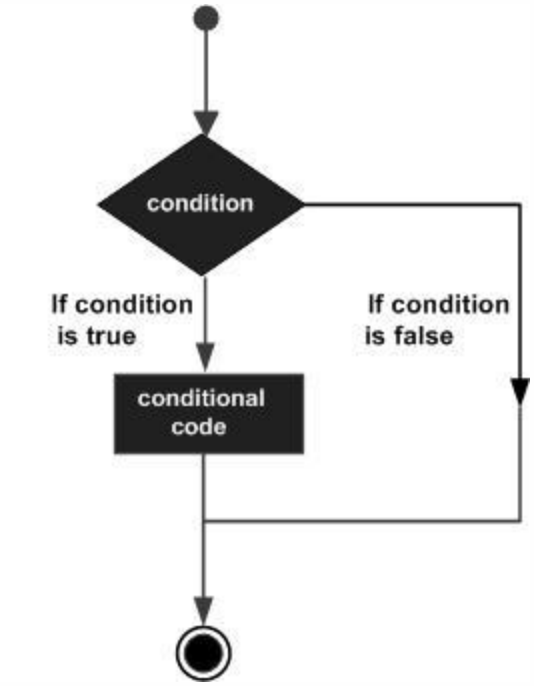
Sr.No.	Loop & Description
1	for loop Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
2	forEach loop Enhanced for loop. This is mainly used to traverse collection of elements including arrays.
3	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

4	do...while loop Like a while statement, except that it tests the condition at the end of the loop body.

POWERSHELL - CONDITIONS

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



PowerShell scripting language provides following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description

1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
3	nested if statement You can use one **if** or **elseif** statement inside another **if** or **elseif** statementss.
4	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.

POWERSHELL - ARRAY

PowerShell provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the any type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables or objects.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you can specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

Syntax

```
$A = 1, 2, 3, 4
or
$A = 1..4
```

Note − By default type of objects of array is System.Object. GetType method returns the type of the array. Type can be passed.

Example

The following code snippets are examples of this syntax −

```
[int32[]]$intA = 1500,2230,3350,4000


$A = 1, 2, 3, 4
$A.getType()
```

This will produce the following result −

Output

```
IsPublic    IsSerial    Name                                     BaseType
--------    --------    ----                                     --------
True        True        Object[]                                 System.Array
```

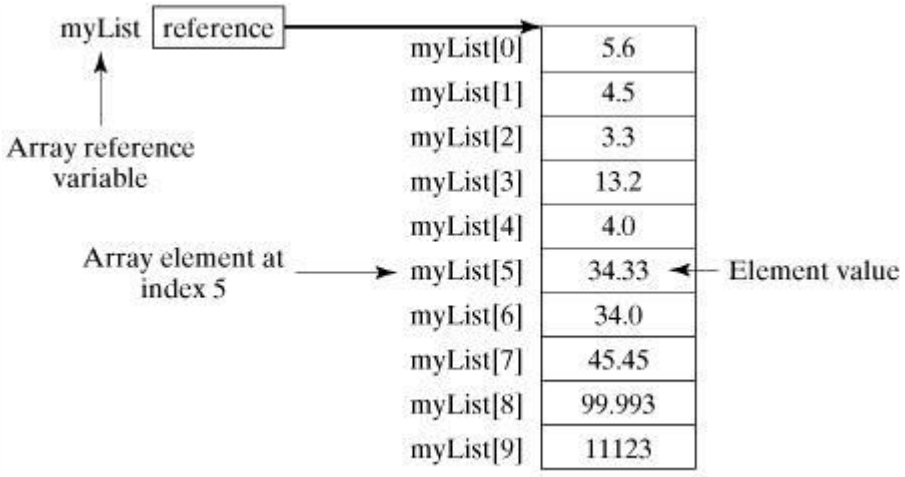
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList −

```
$myList = 5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays −

```
$myList = 5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123

write-host("Print all the array elements")
$myList

write-host("Get the length of array")
$myList.Length

write-host("Get Second element of array")
$myList[1]

write-host("Get partial array")
$subList = $myList[1..3]

write-host("print subList")
$subList
```

```
write-host("using for loop")
for ($i = 0; $i -le ($myList.length - 1); $i += 1) {
   $myList[$i]
}

write-host("using forEach Loop")
foreach ($element in $myList) {
   $element
}

write-host("using while Loop")
$i = 0
while($i -lt 4) {
   $myList[$i];
   $i++
}

write-host("Assign values")
$myList[1] = 10
$myList
```

This will produce the following result −

Output

```
Print all the array elements
5.6
4.5
3.3
13.2
4
34.33
34
45.45
99.993
11123
Get the length of array
10
Get Second element of array
4.5
```

```
Get partial array
print subList
4.5
3.3
13.2
using for loop
5.6
4.5
3.3
13.2
4
34.33
34
45.45
99.993
11123
using forEach Loop
5.6
4.5
3.3
13.2
4
34.33
34
45.45
99.993
11123
using while Loop
5.6
4.5
3.3
13.2
Assign values
5.6
10
3.3
13.2
4
34.33
34
45.45
99.993
11123
```

The Arrays Methods Examples

Here is a complete example showing operations on arrays using its methods

```
$myList = @(0..4)


write-host("Print array")
$myList
```

```
$myList = @(0..4)

write-host("Assign values")
$myList[1]  = 10
$myList
```

This will produce the following result −

Output

```
Clear array
Print array
0
1
2
3
4
Assign values
0
10
2
3
4
```

POWERSHELL - HASHTABLES

Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. Generally we used String or numbers as keys.

This tutorial introduces how to declare hashtable variables, create hashtables, and process hashtable using its methods.

Declaring hashtable Variables

To use an hashtable in a program, you must declare a variable to reference the hashtable. Here is the syntax for declaring an hashtable variable −

Syntax

```
$hash = @{ ID = 1; Shape = "Square"; Color = "Blue"}
or
$hash = @{}
```

Note − Ordered dictionaries can be created using similar syntax. Ordered dictionaries maintain the order in which entries are added whereas hashtables do not.

Example

The following code snippets are examples of this syntax −

```
$hash = [ordered]@{ ID = 1; Shape = "Square"; Color = "Blue"}
```

Print the hashtable.

```
$hash
```

Output

```
Name                           Value
----                           -----
ID                             1
Color                          Blue
Shape                          Square
```

The hashtable values are accessed through the **keys**.

```
> $hash["ID"]
 1
```

Processing Hashtable

Dot notation can be used to access hashtables keys or values.

```
> $hash.keys
ID
Color
Shape


> $hash.values
1
Blue
Square
```

Example

Here is a complete example showing how to create, initialize, and process hashtable −

```
$hash = @{ ID = 1; Shape = "Square"; Color = "Blue"}

write-host("Print all hashtable keys")
$hash.keys

write-host("Print all hashtable values")
$hash.values

write-host("Get ID")
$hash["ID"]

write-host("Get Shape")
$hash.Number

write-host("print Size")
$hash.Count

write-host("Add key-value")
$hash["Updated"] = "Now"

write-host("Add key-value")
$hash.Add("Created","Now")

write-host("print Size")
$hash.Count

write-host("Remove key-value")
$hash.Remove("Updated")

write-host("print Size")
$hash.Count
```

```
write-host("sort by key")
$hash.GetEnumerator() | Sort-Object -Property key
```

This will produce the following result −

Output

```
Print all hashtable keys
ID
Color
Shape
Print all hashtable values
1
Blue
Square
Get ID
1
Get Shape
print Size
3
Add key-value
Add key-value
print Size
5
Remove key-value
print Size
4
sort by key

Name                           Value
----                           -----
Color                          Blue
Created                        Now
ID                             1
Shape
Square
```

POWERSHELL - REGULAR EXPRESSION

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.

Here is the table listing down all the regular expression metacharacter syntax available in PowerShell −

Subexpression	Matches
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character except newline. Using **m** option allows it to match the newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
\A	Beginning of the entire string.
\z	End of the entire string.

\Z	End of the entire string except allowable final line terminator.
re*	Matches 0 or more occurrences of the preceding expression.
re+	Matches 1 or more of the previous thing.
re?	Matches 0 or 1 occurrence of the preceding expression.
re{ n}	Matches exactly n number of occurrences of the preceding expression.
re{ n,}	Matches n or more occurrences of the preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of the preceding expression.
a| b	Matches either a or b.
Rere	Groups regular expressions and remembers the matched text.

?:re?:re	Groups regular expressions without remembering the matched text.
?>re?>re	Matches the independent pattern without backtracking.
\w	Matches the word characters.
\W	Matches the nonword characters.
\s	Matches the whitespace. Equivalent to [\t\n\r\f].
\S	Matches the nonwhitespace.
\d	Matches the digits. Equivalent to [0-9].
\D	Matches the nondigits.
\A	Matches the beginning of the string.
\Z	Matches the end of the string. If a newline exists, it matches just before newline.
\z	Matches the end of the string.
\G	Matches the point where the last match finished.
\n	Back-reference to capture group number "n".
\b	Matches the word boundaries when outside the brackets. Matches the backspace 0x080x08 when inside the brackets.
\B	Matches the nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\Q	Escape quotequote all characters up to \E.
\E	Ends quoting begun with \Q.

Here is a complete examples showing how to use regex in PowerShell;

Sr.No.	**Match & Description**

1	Match Characters Example of supported regular expression characters.
2	Match Character Classes Example of supported character classes.
3	Match Quantifiers Example of supported quantifiers.

POWERSHELL - BACKTICK

Backtick ‘‘ operator is also called word-wrap operator. It allows a command to be written in multiple lines. It can be used for new line ‘n‘n or tab ‘t‘t in sentences as well. See the examples below −

Example 1

```
Get-Service * | Sort-Object ServiceType `
| Format-Table Name, ServiceType, Status -AutoSize
```

It will become

```
Get-Service * | Sort-Object ServiceType | Format-Table Name, ServiceType,
Status -AutoSize
```

Verify the output as

```
Name                                               ServiceType  Status
----                                               -----------  ------
MSSQLServerADHelper100                           Win32OwnProcess Stopped
ntrtscan                                         Win32OwnProcess Running
...
```

Example 2

Use of new line and tab.

```
> Write-host "Title Subtitle"
```

```
Title Subtitle


> Write-host "Title `nSubtitle"
Title
Subtitle


> Write-host "Title `tSubtitle"
Title   Subtitle
```

POWERSHELL - BRACKETS

Powershell supports three types of brackets.

- **Parenthesis brackets. −**
- **Braces brackets. − {}**
- **Square brackets. − []**

Parenthesis brackets

This type of brackets is used to

- pass arguments
- enclose multiple set of instructions
- resolve ambiguity
- create array

Example

```
> $array = @("item1", "item2", "item3")


> foreach ($element in $array) { $element }
item1
item2
item3
```

Braces brackets

This type of brackets is used to

- enclose statements
- block commands

Example

```
$x = 10

if($x -le 20){
   write-host("This is if statement")
}
```

This will produce the following result −

Output

```
This is if statement.
```

Square brackets

This type of brackets is used to

- access to array
- access to hashtables
- filter using regular expression

Example

```
> $array = @("item1", "item2", "item3")

> for($i = 0; $i -lt $array.length; $i++){ $array[$i] }
item1
item2
item3

>Get-Process [r-s]*
 Handles    NPM(K)      PM(K)     WS(K)    VM(M)    CPU(s)      Id    ProcessName
-------    ------      -----     -----    -----    ------      --    -----------
```

```
    320        72     27300     33764    227     3.95    4028
SCNotification

   2298        77     57792     48712    308             2884
SearchIndexer

   ...
```

POWERSHELL - ALIAS

PowerShell alias is another name for the cmdlet or for any command element.

Creating Alias

Use **New-Alias** cmdlet to create a alias. In the below example, we've created an alias help for Get-Help cmdlet.

```
New-Alias -Name help -Value Get-Help
```

Now invoke the alias.

```
help Get-WmiObject -Detailed
```

You will see the following output.

```
NAME
   Get-WmiObject

SYNOPSIS
   Gets instances of Windows Management Instrumentation (WMI) classes or
information about the available classes.

SYNTAX
   Get-WmiObject [
...
```

Getting Alias

Use **get-alias** cmdlet to get all the alias present in current session of powershell.

```
Get-Alias
```

You will see the following output.

```
CommandType     Name                       Definition
```

```
-----------     ----                    ----------
Alias           %                       ForEach-Object
Alias           ?                       Where-Object
Alias           ac                      Add-Content
Alias           asnp                    Add-PSSnapIn
...
```