

Practical  365

PowerShell for Beginners

BY ALEX RODRICK

Introduction

Sometimes my friends and colleagues used to ask me from where they can start learning PowerShell to which I never had a good answer. Even if I could come up with some points, they always found it difficult to search for the content which they can understand easily. Another challenge they faced was the order in which they should read the topics.

Keeping myself in place of a beginner, I have written this eBook in a way which could be easily understood. I have also added some practice questions that can help in understanding the actual concept behind the various topics, the answers for which can be found at the end.

I hope this guide can help all levels up-level this skill. After following this eBook I hope you will not consider yourself a beginner!

I have also mentioned some tips from my personal experience. I hope you have found them helpful. If

I like to thank my mentor Vikas Sukhija who not only helped me understanding PowerShell but also promoted me to learn it.

About the Author

Alex is an expert in various Messaging technologies like Microsoft Exchange, O365, IronPort, Symantec Messaging Gateway. He is passionate about PowerShell scripting and like to automate day to day activities to improve efficiency and overall productivity.



Contents

Introduction.....	1
About the Author	2
1. How to Use This eBook.....	4
2. Variable.....	6
3. Data Types in PowerShell.....	7
4. Arrays.....	8
5. Hashtable	9
6. Wildcards	10
7. Pipeline or pipe “ ”	11
8. Format List, Format Table, Select/Select-Object	12
9. Read-Host and Write-Host.....	16
10. If/Then.....	17
11. Where/Where-Object or short form ‘?’	29
12. Loops.....	21
12.1. ForEach() or short form ‘%’ loop.....	21
12.2. For Loop.....	24
12.3. While Loop.....	25
12.4. Do While Loop.....	26
12.5. Do Until Loop	27
13. Switch.....	29
14. Functions	32
15. Connecting to O365.....	37
16. Tips:	43
17. Answers to Practice questions:.....	48

1. How to Use This eBook

This eBook contains all the essential information that is required for a beginner to start writing their own scripts. To fully utilize the eBook, please go through each chapter one by one. The topics have been explained using an example so that one can easily understand. A small practice question is available after the topic so that you can try out what you have learnt and if any assistance is required, the answers can be found at the end of the eBook.

What is PowerShell?

In the modern world, everyone is striving towards automation. For an Administrator, performing same repetitive tasks can become monotonous which in turn not only reduces the efficiency but can also leads to errors.

PowerShell is an interactive command line tool through which you can automate such mundane tasks. You can execute programs known as 'script (saved as .ps1 file)' which contains various cmdlets for the respective task.

What is a cmdlet?

A cmdlet is simply a command through which you can perform an action. The two most helpful cmdlets that everyone should be aware of are:

- Get-Command
- Get-Help

Using the cmdlet 'Get-Command', you can find all the available cmdlets even if you do not know the exact cmdlet. For example, you want to restart a service from PowerShell, but you do not know the cmdlet. Although you can assume that it may contain the word 'service'.

In the below screenshot, you found all the available cmdlets that contain the word 'service' and therefore, found the cmdlet to restart a service.

```
PS C:\WINDOWS\system32> Get-Command *Service*

CommandType      Name                                Version      Source
-----
Function          Get-NetFirewallServiceFilter       2.0.0.0      NetSecurity
Function          Set-NetFirewallServiceFilter       2.0.0.0      NetSecurity
Cmdlet            Get-Service                        3.1.0.0      Microsoft.PowerShell.Management
Cmdlet            New-Service                        3.1.0.0      Microsoft.PowerShell.Management
Cmdlet            New-WebServiceProxy                3.1.0.0      Microsoft.PowerShell.Management
Cmdlet            Restart-Service                    3.1.0.0      Microsoft.PowerShell.Management
Cmdlet            Resume-Service                     3.1.0.0      Microsoft.PowerShell.Management
Cmdlet            Set-Service                        3.1.0.0      Microsoft.PowerShell.Management
Cmdlet            Start-Service                      3.1.0.0      Microsoft.PowerShell.Management
Cmdlet            Stop-Service                       3.1.0.0      Microsoft.PowerShell.Management
Cmdlet            Suspend-Service                   3.1.0.0      Microsoft.PowerShell.Management
Application        iaStorAfsService.exe              17.8.0.0     C:\WINDOWS\system32\iaStorAfsService.exe
Application        SecurityHealthService.exe          4.18.19.0    C:\WINDOWS\system32\SecurityHealthServ...
Application        SensorDataService.exe              10.0.19.0    C:\WINDOWS\system32\SensorDataService.exe
Application        services.exe                       10.0.19.0    C:\WINDOWS\system32\services.exe
Application        services.msc                       0.0.0.0      C:\WINDOWS\system32\services.msc
Application        TieringEngineService.exe           10.0.19.0    C:\WINDOWS\system32\TieringEngineServi...
Application        Windows.WARP.JITService.exe        0.0.0.0      C:\WINDOWS\system32\Windows.WARP.JITSe...
```

But how can you make use of this cmdlet? You can find more information about it using the cmdlet 'Get-Help'.

This provides the basic information about the cmdlet and what all parameters can be used in it.

```
PS C:\WINDOWS\system32> Get-Help Restart-Service

NAME
    Restart-Service

SYNTAX
    Restart-Service [-InputObject] <ServiceController[]> [-Force] [-PassThru] [-Include <string[]>] [-Exclude <string[]>] [-WhatIf] [-Confirm] [<CommonParameters>]

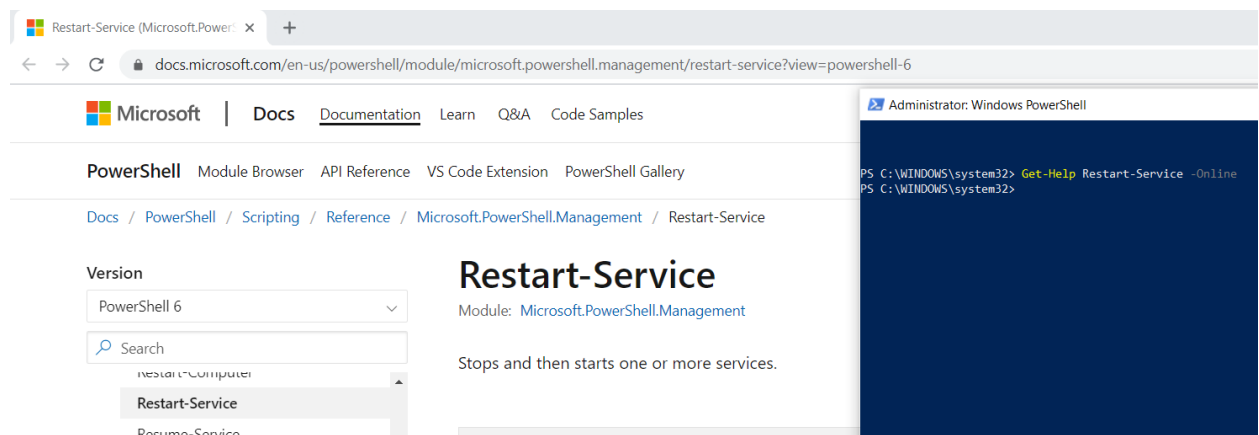
    Restart-Service [-Name] <string[]> [-Force] [-PassThru] [-Include <string[]>] [-Exclude <string[]>] [-WhatIf] [-Confirm] [<CommonParameters>]

    Restart-Service -DisplayName <string[]> [-Force] [-PassThru] [-Include <string[]>] [-Exclude <string[]>] [-WhatIf] [-Confirm] [<CommonParameters>]

ALIASES
    None

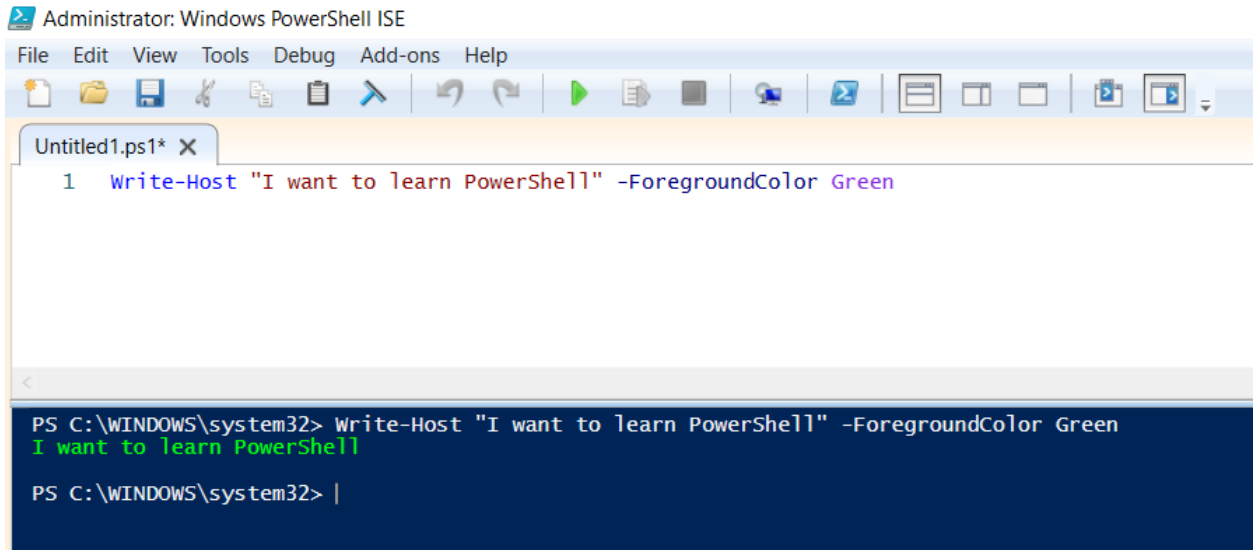
REMARKS
    Get-Help cannot find the Help files for this cmdlet on this computer. It is displaying only partial help.
    -- To download and install Help files for the module that includes this cmdlet, use Update-Help.
    -- To view the Help topic for this cmdlet online, type: "Get-Help Restart-Service -Online" or go to https://go.microsoft.com/fwlink/?LinkID=113385.
```

If you want to find the MS article about a cmdlet, just add the parameter '-Online' at the end and it will open the MS page in the browser.



PowerShell vs PowerShell ISE

Now you know how you can execute a program or a cmdlet in PowerShell but where should you write a program. You can do it in PowerShell ISE.



2. Variable

Any character that has a symbol '\$' in front of it becomes a variable. A variable comes in play when you need to store a value so that you can make use of it later in the script. eg You have 2 values '10' and '4' and you must perform 3 functions on it, like addition, subtraction, multiplication. So, there are 2 ways to perform this task

(a) $10+4$, $10-4$, $10*4$

(b) $\$a = 10$

$\$b = 4$

$\$a + \b

$\$a - \b

$\$a * \b

```
PS C:\Users\acer> 10+4
14
PS C:\Users\acer> 10-4
6
PS C:\Users\acer> 10*4
40
PS C:\Users\acer>
PS C:\Users\acer> $a = 10
PS C:\Users\acer> $b = 4
PS C:\Users\acer> $a+$b
14
PS C:\Users\acer> $a-$b
6
PS C:\Users\acer> $a*$b
40
PS C:\Users\acer>
```

3. Data Types in PowerShell

There are different data types in PowerShell that include integers, floating point values, strings, Booleans, and datetime values which are like what you use in our daily life. The GetType method returns the current data type of the given variable.

```
PS C:\> $v = 1 + 1
PS C:\> $v
2
PS C:\> $v.GetType()

IsPublic IsSerial Name
-----
True     True     Int32

PS C:\>
PS C:\> $v = '1' + '1'
PS C:\> $v
11
PS C:\> $v.GetType()

IsPublic IsSerial Name
-----
True     True     String
```

Variables may be converted from one type to another by explicit conversion.

```
PS C:\> $v = 1.9
PS C:\>
PS C:\> [int32]$v
2
PS C:\> [float]$v
1.9
PS C:\> [string]$v
1.9
PS C:\> [boolean]$v
True
PS C:\> [datetime]$v
09 January 2020 00:00:00
```

- Integer: It is of the type 'whole number'. Any decimal value is rounded off.
- Floating Point: The value is of decimal type.
- String: As the name suggests, it is used for storing letters and characters.
- Boolean: This value can be either True or False. If it is existing, then it is True otherwise it'll be False.
- DateTime: As the name suggests, it is of the type of date and time.

4. Arrays

Arrays are most often used to contain a list of values, such as a list of usernames or cities.

PowerShell arrays can be defined by wrapping a list of items in parentheses and prefacing with the '@' symbol. Eg

\$nameArray = @("John","Joe","Mary")

Items within an array can be accessed using their numerical index, beginning with 0, within square brackets like so: **\$nameArray[0]**. Eg

\$nameArray[0] will be the first value in the array, ie "John"

\$nameArray[1] will be the second value in the array, ie "Joe"

```
PS C:\Users\acer> $nameArray = @("John","Joe","Mary")
PS C:\Users\acer> $nameArray[0]
John
PS C:\Users\acer> $nameArray[1]
Joe
PS C:\Users\acer> $nameArray[2]
Mary
PS C:\Users\acer>
```

In simple terms, an array is like a column in excel containing similar type of data.

Name
John
Joe
Mary

Practice 1: Create arrays 1 and 2 as below. The 3rd array should show the sum of each corresponding value in array 1 and 2.

Array1	Array2	Array3
1	4	5
2	5	7
3	6	9

Note: Make use of the concept of concatenation. Our focus should be to understand how indexing happens in an array and how you can make use of it to achieve the above task.

In case of queries, please check the answer section.

5. Hashtable

A more advanced form of array, known as a hashtable, is assigned with squiggly brackets prefaced by the '@' sign. While arrays are typically (but not always) used to contain similar data, hashtables are better suited for related (rather than similar) data. Individual items within a hashtable are named rather than assigned a numerical index, eg

\$user=@{FirstName="John"; LastName="Smith"; MiddleInitial="J"; Age=40}

Items within a hashtable are easily accessed using the variable and the key name. eg

\$user.LastName will return the value "Smith".

```
PS C:\Users\acer> $user=@{FirstName="John"; LastName="Smith"; MiddleInitial="J"; Age=40}
PS C:\Users\acer> $user.firstname
John
PS C:\Users\acer> $user.lastname
Smith
PS C:\Users\acer> $user.middleinitial
J
PS C:\Users\acer> $user.age
40
PS C:\Users\acer>
```

To easily understand hashtables, you can relate it to the following example.

As per the below example, you have a table in which the first column contains as "FirstName", the second column contains the "LastName", the third column contains the "MiddleInitial" and the fourth column contains the "Age".

FirstName	LastName	MiddleInitial	Age
John	Smith	D	40
Joe	Parker	L	32
Gary	Smith	N	25

Now if you store the above table in a hashtable "\$user", then in order to call the first column you would have to run the command "\$user.FirstName" and it will list out all the first names in column 1.

```
PS C:\Users\acer> $user=@{FirstName="John","Joe","Garry"; LastName="Smith","Parker","Smith"; MiddleInitial="D","L","N"; Age=40,32,25}
PS C:\Users\acer> $user

Name      Value
----
Age       {40, 32, 25}
FirstName {John, Joe, Garry}
MiddleInitial {D, L, N}
LastName   {Smith, Parker, Smith}

PS C:\Users\acer> $user.Age
40
32
25
PS C:\Users\acer> $user.FirstName
John
Joe
Garry
PS C:\Users\acer> $user.LastName
Smith
Parker
Smith
PS C:\Users\acer>
```

Practice 2: Create 2 hashtables as below and the third hashtable should be calculated as (DaysWorked * SalaryPerDay).

Hashtable 1	
Name	DaysWorked
John	12
Joe	20
Mary	18

Hashtable 2	
Name	SalaryPerDay
John	100
Joe	120
Mary	150

Hashtable 3	
Name	Salary
John	1200
Joe	2400
Mary	2700

In case of queries, please check the answer section.

6. Wildcards

Wildcards are something which you can use to match partial values instead of exact values.

The '*' wildcard will match zero or more characters.

The '?' wildcard will match a single character.

The screenshot shows a PowerShell console window on the left and a text editor window titled 'Untitled1.ps1*' on the right.

PowerShell Console Output:

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

Status  Name      DisplayName
-----
Stopped ALG        Application Layer Gateway Service

PS C:\> Get-Service App*

Status  Name      DisplayName
-----
Stopped AppIDSvc  Application Identity
Running Appinfo  Application Information
Stopped AppMgmt  Application Management
Stopped AppReadiness  App Readiness
Stopped AppVClient  Microsoft App-V Client
Stopped AppXSvc  AppX Deployment Service (AppXSVC)

PS C:\> Get-Service A*

Status  Name      DisplayName
-----
Stopped AarSvc_8bccb  Agent Activation Runtime_8bccb
Running AdobeARMService  Adobe Acrobat Update Service
Stopped AJRouter  AllJoyn Router Service
Stopped ALG  Application Layer Gateway Service
Stopped AppIDSvc  Application Identity
Running Appinfo  Application Information
Stopped AppMgmt  Application Management
Stopped AppReadiness  App Readiness
Stopped AppVClient  Microsoft App-V Client
Stopped AppXSvc  AppX Deployment Service (AppXSVC)
Stopped AssignedAccessM...  AssignedAccessManager Service
Running AudioEndpointBu...  Windows Audio Endpoint Builder
Running Audiosrv  Windows Audio
Stopped autotimesvc  Cellular Time
Stopped AxInstSV  ActiveX Installer (AxInstSV)
```

Text Editor Content (Untitled1.ps1*):

```
1 Get-Service ALG
2
3 Get-Service App*
4
5 Get-Service A*|
```

In the first case, you are checking for the exact service with the name "ALG". In the second case, you want to find out the services whose name starts with "App", therefore you added wildcard "*" at the

end of it. In this way PowerShell will look for services that starts with “App” and can have anything after that. Similarly, in the third case you are looking for all the services whose name starts with “A”.

[m-n] Match a range of characters from ‘m to n’, so [f-m]ake will match fake/jake/make

[abc] Match a set of characters a,b,c.., so [fm]ake will match fake/make

7. Pipeline or pipe “|”

Each pipeline operator sends the results of the preceding command to the next command. The output of the first command can be sent for processing as input to the second command.

Example can be seen in the next point.

8. Format List, Format Table, Select/Select-Object

Sometimes when you execute a command, all the information may not show up directly. To retrieve all the information, you can make use of 'format-list' or 'fl' in short.

```
PS C:\> Get-Date

13 May 2020 17:05:44

PS C:\> Get-Date | FL

DisplayHint : DateTime
Date       : 13/05/2020 00:00:00
Day        : 13
DayOfWeek  : Wednesday
DayOfYear  : 134
Hour       : 17
Kind       : Local
Millisecond : 774
Minute     : 5
Month      : 5
Second     : 56
Ticks      : 637249863567749090
TimeOfDay  : 17:05:56.7749090
Year       : 2020
DateTime   : 13 May 2020 17:05:56

PS C:\> Get-Date | Format-List

DisplayHint : DateTime
Date       : 13/05/2020 00:00:00
Day        : 13
DayOfWeek  : Wednesday
DayOfYear  : 134
Hour       : 17
Kind       : Local
Millisecond : 830
Minute     : 5
Month      : 5
Second     : 58
Ticks      : 637249863588305093
TimeOfDay  : 17:05:58.8305093
Year       : 2020
DateTime   : 13 May 2020 17:05:58
```

In the example, you can see that when you run 'Get-Date' command, all information does not show up. To get all the details, you first executed the command 'Get-Date' followed by the pipeline operator '|' which takes the output of the preceding command, followed by FL which 'lists' all the information.

When you do a 'Format-List', you get the default view which shows a certain set of attributes. If you would like to view all the attributes, then add the wildcard '*' after 'Format-List' to view all of them. Example,

```
PS C:\WINDOWS\system32> Get-Service BITS | Format-List

Name           : BITS
DisplayName    : Background Intelligent Transfer Service
Status        : Stopped
DependentServices : {}
ServicesDependedOn : {RpcSs}
CanPauseAndContinue : False
CanShutdown   : False
CanStop       : False
ServiceType   : Win32ShareProcess

PS C:\WINDOWS\system32> Get-Service BITS | Format-List *

Name           : BITS
RequiredServices : {RpcSs}
CanPauseAndContinue : False
CanShutdown   : False
CanStop       : False
DisplayName    : Background Intelligent Transfer Service
DependentServices : {}
MachineName    : .
ServiceName    : BITS
ServicesDependedOn : {RpcSs}
ServiceHandle  : SafeServiceHandle
Status        : Stopped
ServiceType   : Win32ShareProcess
StartType     : Manual
Site          :
Container     :
```

Now, there could be a case where there are lots of attributes, but you only need to view the attributes that contains a word, example 'Name'. Then instead of manually searching through all the attributes, we can quickly do so in the following way.

```
PS C:\WINDOWS\system32> Get-Service BITS | Format-List *name*

Name           : BITS
DisplayName    : Background Intelligent Transfer Service
MachineName    : .
ServiceName    : BITS
```

Now, if you want the information to show up in the format of a table, you can make use of 'Format-Table' or 'FT'.

```
PS C:\> Get-Date | Format-Table
DisplayHint Date          Day DayOfWeek DayOfYear Hour Kind Millisecond Minute Month
-----
DateTime 13/05/2020 00:00:00 13 Wednesday      134 17 Local          17    26    5

PS C:\> Get-Date | FT
DisplayHint Date          Day DayOfWeek DayOfYear Hour Kind Millisecond Minute Month
-----
DateTime 13/05/2020 00:00:00 13 Wednesday      134 17 Local          115   26    5
```

Additional parameters that can be used with Format-Table

- 'AutoSize' or 'Auto': Sometimes when there is a lot of information to be displayed, the output is truncated. If you specify '-AutoSize' or '-Auto' after 'Format-Table', the information is properly displayed by increasing the width of the column automatically. Example, while searching for the service 'BITS', the display name is truncated as it is quite long.

```
PS C:\WINDOWS\system32> Get-Service BITS
Status Name          DisplayName
-----
Stopped BITS          Background Intelligent Transfer Ser...

PS C:\WINDOWS\system32> Get-Service BITS | Format-Table
Status Name          DisplayName
-----
Stopped BITS          Background Intelligent Transfer Ser...
```

Now, if you want to view the display name correctly, you can do it as below.

```
PS C:\WINDOWS\system32> Get-Service BITS | Format-Table -AutoSize
Status Name DisplayName
-----
Stopped BITS Background Intelligent Transfer Service

PS C:\WINDOWS\system32> Get-Service BITS | Format-Table -Auto
Status Name DisplayName
-----
Stopped BITS Background Intelligent Transfer Service
```

- 'Sort-Object' or 'Sort': Data becomes quite easy to view when it is sorted. To sort data, you can make use of this parameter. Example, if you want to sort data based on an attribute, you can do it as below.

```
PS C:\WINDOWS\system32> Get-Service a*
```

Status	Name	DisplayName
Stopped	AarSvc_3283f	AarSvc_3283f
Running	AdobeARMService	Adobe Acrobat Update Service
Running	AESMSvc	Intel® SGX AESM
Stopped	AJRouter	AllJoyn Router Service
Stopped	ALG	Application Layer Gateway Service
Stopped	AntiVirMailService	Avira Mail Protection
Running	AntivirProtecte...	Avira Protected Service
Running	AntivirSchedule...	Avira Scheduler
Running	AntivirService	Avira Real-Time Protection
Stopped	AntiVirWebService	Avira Web Protection
Stopped	AppIDSvc	Application Identity
Running	Appinfo	Application Information
Stopped	AppReadiness	App Readiness
Running	AppXSvc	AppX Deployment Service (AppXSVC)
Running	AudioEndpointBu...	Windows Audio Endpoint Builder
Running	Audiosrv	Windows Audio
Stopped	autotimesvc	Cellular Time
Running	Avira.ServiceHost	Avira Service Host
Running	AviraOptimizerHost	Avira Optimizer Host
Running	AviraPhantomVPN	Avira Phantom VPN
Running	AviraSecurity	Avira Security
Running	AviraUpdaterSer...	Avira Updater Service
Stopped	AxInstSV	ActiveX Installer (AxInstSV)

```
PS C:\WINDOWS\system32> Get-Service a* | Sort-Object Status
```

Status	Name	DisplayName
Stopped	AppIDSvc	Application Identity
Stopped	AntiVirWebService	Avira Web Protection
Stopped	AxInstSV	ActiveX Installer (AxInstSV)
Stopped	autotimesvc	Cellular Time
Stopped	AppReadiness	App Readiness
Stopped	AntiVirMailService	Avira Mail Protection
Stopped	ALG	Application Layer Gateway Service
Stopped	AarSvc_3283f	AarSvc_3283f
Stopped	AJRouter	AllJoyn Router Service
Running	AviraOptimizerHost	Avira Optimizer Host
Running	Appinfo	Application Information
Running	Avira.ServiceHost	Avira Service Host
Running	AviraSecurity	Avira Security
Running	AviraUpdaterSer...	Avira Updater Service
Running	AviraPhantomVPN	Avira Phantom VPN
Running	Audiosrv	Windows Audio
Running	AntivirService	Avira Real-Time Protection
Running	AntivirSchedule...	Avira Scheduler
Running	AntivirProtecte...	Avira Protected Service
Running	AESMSvc	Intel® SGX AESM
Running	AudioEndpointBu...	Windows Audio Endpoint Builder
Running	AppXSvc	AppX Deployment Service (AppXSVC)
Running	AdobeARMService	Adobe Acrobat Update Service

Using FT and FL, you can only display the data/information, but it cannot be used to capture the information provided by the command. To use the information provided by a command, you need to make use of 'Select or Select-Object'

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

Status Name          DisplayName
-----
Running EventLog         Windows Event Log

PS C:\> Get-Service EventLog | fl

Name                : EventLog
DisplayName          : Windows Event Log
Status              : Running
DependentServices   : {Webserv, NcdAutoSetup, AppVClient, netprofm...}
ServicesDependedOn  : {}
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
ServiceType         : Win32OwnProcess, Win32ShareProcess

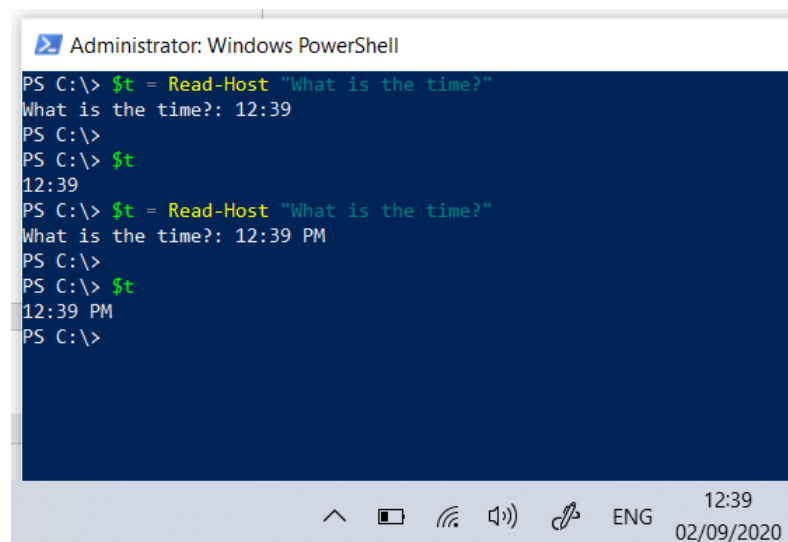
PS C:\> $service = Get-Service EventLog | Select Name,Status
PS C:\>
PS C:\> $service

Name      Status
-----
EventLog  Running
```

In this example, you can see that if you want to capture only specific details about a command, you can make use of 'Select'. The 'Get-Service' command provides a lot of info or parameters and if you want to capture only specific parameters then you can mention them with 'Select'.

9. Read-Host and Write-Host

If you are writing a script that requires an input from the user then you can make use of the cmdlet 'Read-Host'. Using this cmdlet, you can ask a question, for which the user can input any answer. This answer can then be saved in a variable which can be further used in the script.



```
Administrator: Windows PowerShell

PS C:\> $t = Read-Host "What is the time?"
What is the time?: 12:39
PS C:\>
PS C:\> $t
12:39
PS C:\> $t = Read-Host "What is the time?"
What is the time?: 12:39 PM
PS C:\>
PS C:\> $t
12:39 PM
PS C:\>
```

The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. It displays a series of commands and their outputs. The first command is `$t = Read-Host "What is the time?"`, which prompts the user to enter a time. The user enters '12:39'. The second command is `$t`, which outputs '12:39'. The third command is `$t = Read-Host "What is the time?"`, which prompts the user to enter a time. The user enters '12:39 PM'. The fourth command is `$t`, which outputs '12:39 PM'. The window also shows the system tray at the bottom with icons for network, volume, and power, and the date and time '12:39 02/09/2020'.

The Write-Host cmdlet can be used to display information in the console. Its handy while troubleshooting/monitoring a script to identify till which step the script has executed.

```
PS C:\> Write-Host "It is a good day" -ForegroundColor Green
It is a good day
PS C:\> Write-Host "It is a good day" -ForegroundColor Yellow
It is a good day
PS C:\> Write-Host "It is a good day"
It is a good day
PS C:\> Write-Host "It is a good day" -ForegroundColor Cyan
It is a good day
```

Practice 3: Ask a question “What is your name” and then display the answer provided in “Green” color.

In case of queries, please check the answer section.

10. If/Then

This is the simplest form of decision making in PowerShell. It basically works like this: Something is compared with something and depending on the comparison do this activity.

The comparison statement must have a logical response of either TRUE or FALSE. Think of it as a yes or no question. Example,

```
PS C:\Users\acer> $i = 10
PS C:\Users\acer> $j = 5
PS C:\Users\acer> if ($i -gt $j)
>> {
>> Write-Host "$i is greater than $j"
>> }
>> else
>> {
>> Write-Host "$j is greater than $i"
>> }
>>
10 is greater than 5
PS C:\Users\acer>
```

Similarly, if you have more than 2 conditions, you can involve ‘elseif’ as well. Example,

```
If (comparison 1)
{
    Then Action 1
}
Elseif (comparison 2)
{
    Then Action 2
}
Else
{
    Then Action 3
}
```

Comparison Operators

The following operators are all Case-Insensitive by default:

"-eq"	Equal
"-ne"	Not equal
"-ge"	Greater than or equal
"-gt"	Greater than
"-lt"	Less than
"-le"	Less than or equal
"-like"	Wildcard comparison
"-notlike"	Wildcard comparison
"-match"	Regular expression comparison
"-notmatch"	Regular expression comparison
"-replace"	Replace operator
"-contains"	Containment operator
"-notcontains"	Containment operator
"-shl"	Shift bits left (PowerShell 3.0)
"-shr"	Shift bits right – preserves sign for signed values.(PowerShell 3.0)
"-in"	Like –contains, but with the operands reversed.(PowerShell 3.0)
"-notin"	Like –notcontains, but with the operands reversed.(PowerShell 3.0)

To perform a Case-Sensitive comparison just prefix any of the above with "c"
eg -ceq for case-sensitive Equals or -creplace for case-sensitive replace.

Comparison of types:

"-is"	Is of a type
"-isnot"	Is not of a type
"-as"	As a type, no error if conversion fails

Logical Operators:

"-and"	Logical And
"-or"	Logical Or
"-not"	logical not
"!"	logical not

Example, let us find the status of the service 'Spooler' and give the output as 'Service is Good' if it is in the 'Running' state and as 'Service is Bad' if it is in Stopped state.

Before starting let us validate the current status first.

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

Status	Name	DisplayName
Running	Spooler	Print Spooler

```
1 $service = Get-Service Spooler | Select Status
2
3 if($service.Status -eq "Running")
4 {
5     Write-Host "Service is Good" -ForegroundColor Green
6 }
7 elseif($service.status -eq "Stopped")
8 {
9     Write-Host "Service is Bad" -ForegroundColor Red
10 }
11
```

```
PS C:\> $service = Get-Service Spooler | Select Status
if($service.Status -eq "Running")
{
    Write-Host "Service is Good" -ForegroundColor Green
}
elseif($service.status -eq "Stopped")
{
    Write-Host "Service is Bad" -ForegroundColor Red
}

Service is Good
PS C:\> |
```

Let us understand what is happening in the background.

- i. In line 1, the status of the service 'Spooler' is saved in the variable \$service. Please note that you are only saving 1 attribute 'Status' using 'Select' instead of saving all attributes.
- ii. In line 3, the 'if' condition is checking whether the status attribute in variable \$service is equal to value "Running" which you know is the same. Therefore, the execution goes to line 5 and the output is displayed as 'Service is Good'.

Let us check another service 'Fax' (current status 'Stopped') and display our output accordingly.

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

Status	Name	DisplayName
Stopped	Fax	Fax

```
1 $service = Get-Service Fax | Select Status
2
3 if($service.Status -eq "Running")
4 {
5     Write-Host "Service is Good" -ForegroundColor Green
6 }
7 elseif($service.status -eq "Stopped")
8 {
9     Write-Host "Service is Bad" -ForegroundColor Red
10 }
11
```

```
PS C:\> $service = Get-Service Fax | Select Status
if($service.Status -eq "Running")
{
    Write-Host "Service is Good" -ForegroundColor Green
}
elseif($service.status -eq "Stopped")
{
    Write-Host "Service is Bad" -ForegroundColor Red
}

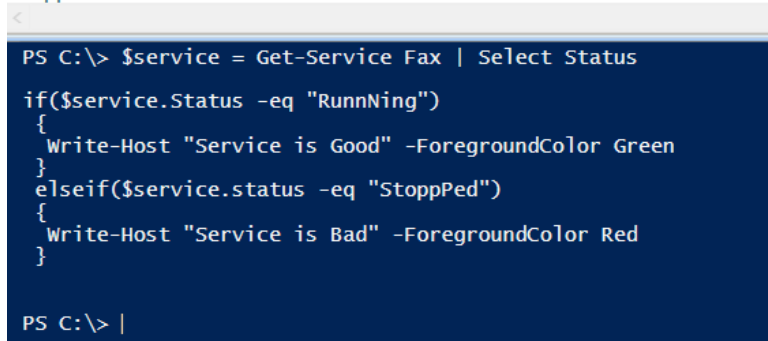
Service is Bad
PS C:\>
```

- i. Once the status is saved in the variable \$service, the 'if' condition is checked in line 3. You can see that the service is in stopped state, therefore the execution skips the action written in the 'if' part (line 4,5,6) and moves on to line 7 where it checks the 'elseif' condition.
- ii. Since this condition is being met, the execution moves to the action written in the 'elseif' part and therefore, the output is displayed as 'Service is Bad'.

Note: In a scenario where none of the conditions are met, no output will be displayed.

Example, there is a typo in the conditions due to which none of them are true. Therefore, no output is displayed.

```
1 $service = Get-Service Fax | Select Status
2
3 if($service.Status -eq "RunnNing")
4 {
5     Write-Host "Service is Good" -ForegroundColor Green
6 }
7 elseif($service.status -eq "StoppPed")
8 {
9     Write-Host "Service is Bad" -ForegroundColor Red
10 }
11
```

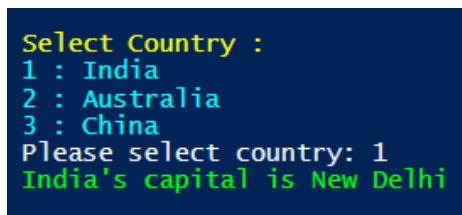


```
PS C:\> $service = Get-Service Fax | Select Status
if($service.Status -eq "RunnNing")
{
    Write-Host "Service is Good" -ForegroundColor Green
}
elseif($service.status -eq "StoppPed")
{
    Write-Host "Service is Bad" -ForegroundColor Red
}

PS C:\> |
```

Practice 4:

- i. Ask the user for 2 values and then compare them and then display the value which is bigger in the form as "The higher number is : X"
- ii. Display the menu as below and ask the user to select the country. Based on the choice entered, display the corresponding country's capital.



```
Select Country :
1 : India
2 : Australia
3 : China
Please select country: 1
India's capital is New Delhi
```

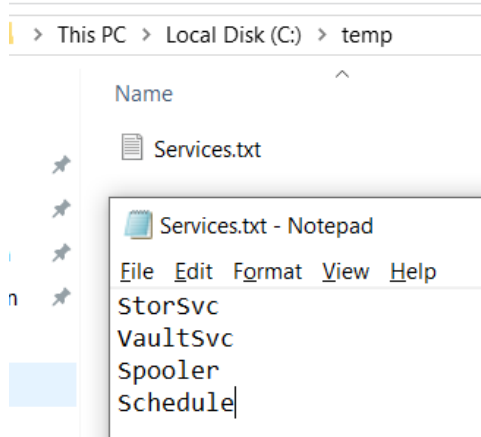
11. Loops

Loops are the most important feature that are used to automate/action bulk activities.

11.1. ForEach() loop

This is the most used loop in PowerShell and can be used most of the times to serve our purpose.

Example, you need to find the status of the few services that are saved in a text/CSV file.



```
$result = @()                                ## Empty table to store the
information                                  ## List of services whose status has
$services = Get-Content C:\Temp\Services.txt ## to be checked

foreach($s in $services)
{
    $data = $null
    $data = Get-Service $s | Select Name,Status
    $result += $data
}

$result
```

```
Checking status of service : StorSvc
Checking status of service : VaultSvc
Checking status of service : Spooler
Checking status of service : Schedule

Name      Status
----      -
StorSvc   Running
VaultSvc  Running
Spooler   Running
Schedule  Running
```

Let us understand how the loop is working in the background:

You first need to save the list of users or any data that must be worked upon in an array. In our case, you imported the data present in the txt file using the cmdlet 'Get-Content' and saved it in an array '\$services'.

Now, let us first understand the syntax of this loop.

```
foreach($s in $services)
{
    code
}
```

When you write the foreach statement, the first object is a temporary variable '\$s' followed by the operator 'in', followed by the array in which the data is stored '\$services'.

In our case you have 4 services saved in the array, so our loop will run 4 times because the purpose of this loop is to pick up the value present in the array one by one and perform the task that is mentioned in the code written between the curly braces.

So, in the background the following 4 steps would be happening:

- I. In the first iteration, \$s will pick the first value stored in \$services which is 'StorSvc' and check for its status.
- II. In the second iteration, \$s will pick the second value stored in \$services which is 'VaultSvc' and check for its status.
- III. In the third iteration, \$s will pick the third value stored in \$services which is 'Spooler' and check for its status.
- IV. In the fourth and last iteration, \$s will pick the first value stored in \$services which is 'Schedule' and check for its status.

Another way of writing the same foreach loop is as below:

```
$result = @()                                ## Empty table to store the
information                                   ## List of services whose status has
$services = Get-Content C:\Temp\Services.txt  ## to be checked

$services | foreach{
    write-Host "Checking status of service : "$_
    $data = $null
    $data = Get-Service $_ | Select Name,Status
    $result += $data
}

$result
```

In the above code, instead of using a temporary variable you are making use of a pipeline variable '\$_' which performs the same role as a temporary role. This pipeline variable is used in 'where' command as well and serves the same purpose.

Practice 6: Create the below CSV and save it locally on your computer.

Name	Age
John	8
Joe	12
Mary	7
Tom	15
Lily	16
Emily	9

Now import this CSV in PowerShell.

Our result should be a table which should contain the name of the student and mention whether they are in Junior School (Age between 4-10) or in Senior School (Age between 11-17).

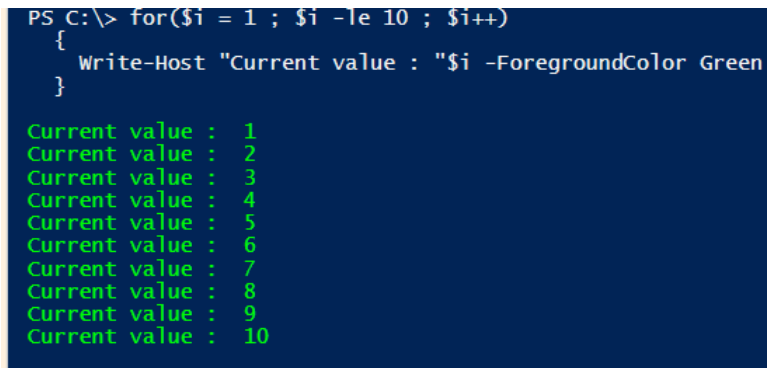
Name	School
John	Junior
Joe	Senior
Mary	Junior
Tom	Senior
Lily	Senior
Emily	Junior

Once the above table has been created, export it to a CSV on your local computer.

11.2. For Loop

This loop is used in case you require a counter, ie if you want to run the loop for 'n' number of times.

```
for($i = 1 ; $i -le 10 ; $i++)
{
    Write-Host "Current value : "$i -ForegroundColor Green
}
```



```
PS C:\> for($i = 1 ; $i -le 10 ; $i++)
{
    Write-Host "Current value : "$i -ForegroundColor Green
}

Current value : 1
Current value : 2
Current value : 3
Current value : 4
Current value : 5
Current value : 6
Current value : 7
Current value : 8
Current value : 9
Current value : 10
```

Now, let us see how this loop is working. The syntax includes 3 parts and each part is separated by a semi colon ';'.
for(1st part ; 2nd part ; 3rd part)

```
{
    Code
}
```

Part	As per example	Explanation
1	<code>\$i = 1</code>	It is the initiation in which you must define the value from which the loop should start.
2	<code>\$i -le 10</code>	It is the condition that is checked after 1st part is completed. If it is true, the code written inside the curly braces is executed.
3	<code>\$i++</code>	Once all the code has been executed, the execution will now come to this part which is generally incrementing/decrementing the value that was defined in the first part. In our case, you are incrementing <code>\$i</code> which will update the value to 2. After this step, the execution will go back to Part 2. Therefore, for the rest of the loop, the execution will move between parts 2 and 3 till the condition in Part 2 fails

Practice 7:

Suppose there are 20 students (Roll Number 1-20) and they must be randomly assigned a group out of 4 colors (Red, Green, Yellow, Blue). The output table should look like something like below but please note that the color assigned should be random to each student.

RollNumber	Group
1	Red
2	Yellow
3	Blue
4	Red
5	Green
6	Yellow

11.3. While Loop

This loop also works based on a condition and can be used in scenarios where a counter must be used or in a situation where a change of state must be monitored.

The while statement runs a statement list zero or more times based on the results of a conditional test. The syntax of this loop is pretty simple.

```
while(condition)
{
    Code
}
```

```
PS C:\> $i = 0
while($i -lt 3)
{
    Write-Host "Current Value is : " $i -ForegroundColor Yellow
    $i++
}
```

```
Current Value is : 0
Current Value is : 1
Current Value is : 2
```

```
PS C:\> while(Get-Service Fax | ?{$_.Status -eq "Stopped"})
{
    Write-Host "Stopped" -ForegroundColor Red
    Start-Sleep -Seconds 2
}

Stopped
Stopped
Stopped
Stopped
Stopped
```

In the above example, the 'fax' service is being monitored till the status remains 'Stopped'. Once the status changes, the loop will stop as well.

Practice 8:

Open 3 instances of 'Notepad' on your machine. Write a code that should display that 'Notepad is running' till it is open. Start closing notepad one by one and once all 3 are closed, the loop should end.

```
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
PS C:\WINDOWS\system32>
```

11.4. Do While Loop

A 'Do While' will execute at least once before validating the condition, even if the condition is True in the first iteration, the loop will still execute once.

This loop works based on the positive result of the condition.

The syntax of the loop is as below:

```
Do
{
    Code
}while(Condition)
```

```
PS C:\> $i = 0
do
{
    Write-Host "Current Value is : "$i -ForegroundColor Cyan
    $i++
}while($i -lt 3)

Current Value is : 0
Current Value is : 1
Current Value is : 2
```

In the above example, \$i starts with value 0 which is printed once then it comes to the 'while' condition. It checks whether \$i is less than 3 which is false, hence the execution goes back to the code inside the 'do' loop. The code is executed, and the condition is checked and so on till the condition becomes true and it stops.

Practice 9:

Open 3 instances of notepad. Write a code using 'do while' loop which should display "Notepad is running" but an interval of 1 second. Once all the instances have been closed, it should stop displaying that "Notepad is running". At the end, it should also show the number of times, the statement was displayed.

```
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
PS C:\WINDOWS\system32>
11
PS C:\WINDOWS\system32>
```

11.5. Do Until Loop

A 'Do Until' will execute at least once before validating the condition, even if the condition is True in the first iteration, the loop will still execute once.

The syntax of the loop is as below:

```
Do
{
    Code
}Until(Condition)
```

```
PS C:\> $i = 0
do
{
    Write-Host "Current Value is : "$i -ForegroundColor Yellow
    $i++
}until($i -ge 3)

Current Value is : 0
Current Value is : 1
Current Value is : 2
```

This loop works like 'Do While' with the only difference that it is based on the negative result of the condition.

Practice 10: Perform the same task as completed in 'Do While' loop but now using 'Do Until' loop.

Some other important statements that could be useful in a loop are as below:

- **Continue**

The continue statement immediately returns script flow to the top of the innermost While, Do, For, or ForEach loop. It does not execute the rest of the lines in that loop.

```
$processes = Get-Process
foreach($process in $processes)
{
    Write-Host "Process : "$process -ForegroundColor Yellow
    if($process.PM / 1024 / 1024 -le 100)
    {
        continue
    }
    Write-Host ('Process ' + $process.Name + ' is using more than 100 MB RAM.') -
    ForegroundColor Green
}
```

In the above example, all the processes running on your system is saved in \$processes. A foreach loop is executed on it which checks each process one by one.

The 'continue' statement is added in the 'if' condition where it is checking whether the process is utilizing more than 100MB RAM or not.

In case the process is not utilizing more than 100MB RAM, the 'if' condition will be 'True' and the 'continue' statement will take the script flow back to the top to pick up the next process.

In case the process is utilizing more than 100MB RAM, the 'if' condition will fail and the statement in green will be displayed.

Small snippet of the code:

```
PS C:\> $processes = Get-Process
PS C:\> foreach($process in $processes)
>> {
>>     Write-Host "Process : "$process -ForegroundColor Yellow
>>     if($process.PM / 1024 / 1024 -le 100)
>>     {
>>         continue
>>     }
>>     Write-Host ('Process ' + $process.Name + ' is using more than 100 MB RAM.') -ForegroundColor Green
>> }
Process : System.Diagnostics.Process (ApplicationFrameHost)
Process : System.Diagnostics.Process (armsvc)
Process : System.Diagnostics.Process (browserhost)
Process : System.Diagnostics.Process (chrome)
Process : System.Diagnostics.Process (chrome)
Process : System.Diagnostics.Process (chrome)
Process : System.Diagnostics.Process (chrome)
Process : System.Diagnostics.Process (chrome)
Process chrome is using more than 100 MB RAM.
Process : System.Diagnostics.Process (chrome)
Process chrome is using more than 100 MB RAM.
Process : System.Diagnostics.Process (chrome)
Process : System.Diagnostics.Process (chrome)
```

- **Break**

The break statement causes Windows PowerShell to immediately exit the innermost While, Do, For, or ForEach loop or Switch code block.

```
$processes = Get-Process
foreach($process in $processes)
{
    if($process.PM / 1024 / 1024 -gt 100)
    {
        Write-Host ('Process ' + $process.Name + ' is using more than 100 MB RAM.') -
        ForegroundColor Red
        break
    }
}
```

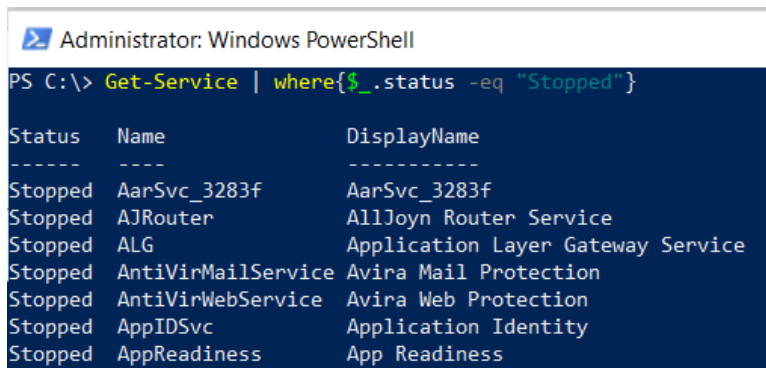
- **Exit**

Exit causes Windows PowerShell to exit a script or a Windows PowerShell instance.

```
if(<some fatal error>)
{
    Exit <optional return code>
}
```

12. Where/Where-Object

The 'where' command is used to look for information that is being passed/provided by the previous command. Example, using the command 'Get-Service' you can find the information about the services on our system. But if you only want to focus on services that are in 'Stopped' state then you can use the 'where' command.



The screenshot shows a Windows PowerShell window titled 'Administrator: Windows PowerShell'. The command entered is `PS C:\> Get-Service | where{$_.status -eq "Stopped"}`. The output is a table with three columns: Status, Name, and DisplayName. All services listed are in a 'Stopped' state.

Status	Name	DisplayName
Stopped	AarSvc_3283f	AarSvc_3283f
Stopped	AJRouter	AllJoyn Router Service
Stopped	ALG	Application Layer Gateway Service
Stopped	AntiVirMailService	Avira Mail Protection
Stopped	AntiVirWebService	Avira Web Protection
Stopped	AppIDSvc	Application Identity
Stopped	AppReadiness	App Readiness

Now you can also make use of other logical operators and comparison operators as discussed in the previous point. Example, using the logical operator 'and', comparison operator 'like' and wildcard '*', you were able to find the services whose name starts with 'A' and which are in 'Stopped' state.

```
Administrator: Windows PowerShell
PS C:\> Get-Service | where{$_.status -eq "Stopped" -and $_.name -like "A*"}

Status      Name                DisplayName
-----
Stopped     AarSvc_3283f        AarSvc_3283f
Stopped     AJRouter            AllJoyn Router Service
Stopped     ALG                 Application Layer Gateway Service
Stopped     AntiVirMailService  Avira Mail Protection
Stopped     AntiVirWebService   Avira Web Protection
Stopped     AppIDSvc            Application Identity
Stopped     AppReadiness        App Readiness
Stopped     autotimesvc         Cellular Time
Stopped     AxInstSV            ActiveX Installer (AxInstSV)

PS C:\>
```

There is another way to execute a 'where' command which is a bit faster as compared to the traditional way.

```
PS C:\> (Get-Service).where({$_ .name -like "A*"})

Status      Name                DisplayName
-----
Running     AdobeARMService     Adobe Acrobat Update Service
Stopped     AJRouter            AllJoyn Router Service
Stopped     ALG                 Application Layer Gateway Service
Stopped     AppIDSvc            Application Identity
Running     Appinfo             Application Information
Stopped     AppMgmt             Application Management
Stopped     AppReadiness        App Readiness
Stopped     AppVClient          Microsoft App-V Client
Stopped     AppXSvc             AppX Deployment Service (AppXSVC)
Stopped     AssignedAccessM...  AssignedAccessManager Service
Running     AudioEndpointBu...  Windows Audio Endpoint Builder
Running     Audiosrv            Windows Audio
Stopped     AxInstSV            ActiveX Installer (AxInstSV)
```

The short form for 'where' is '?' and can be used by simply replacing the word 'where' with '?'

Practice 5:

- i. Open two instances of notepad on your machine and find their Process ID. The output should only show the "Process Name" and "Id".
- ii. Create a folder "C:\Temp\Test" and copy paste any one sample CSV and two TXT file in it. First show the total files in the folder and then find only the csv file in it and display its size in KB and MB as shown in the screenshot.

Directory: C:\temp\test

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a----	5/31/2020 12:44 PM	51986	Notepad1.txt
-a----	5/31/2020 12:44 PM	675854	Notepad2.txt
-a----	5/29/2020 2:29 PM	1177861	SampleFile.csv

```

FileName : SampleFile.csv
Size in KB : 1150.2548828125
Size in MB : 1.12329578399658

```

13. Switch

The switch command is generally preferred when you must perform an action based on a list like in case of a menu. The same task can be performed using multiple 'if and else' but it becomes simpler when you use a 'switch' command. Example

```

Write-Host "Check Service Status" -ForegroundColor Green
Write-Host "1: Check status of Windows Audio service" -ForegroundColor Yellow
Write-Host "2: Check status of Print Spooler service" -ForegroundColor Yellow
Write-Host "3: Check status of Netlogon service" -ForegroundColor Yellow

$choice = Read-Host "Please enter your choice"

switch($choice)
{
    1
    {
        Get-Service Audiosrv
    }
    2
    {
        Get-Service spooler
    }
    3
    {
        Get-Service Netlogon
    }
}

```

```

Check Service Status
1: Check status of Windows Audio service
2: Check status of Print Spooler service
3: Check status of Netlogon service
Please enter your choice: 1

Status   Name      DisplayName
-----
Running  Audiosrv  Windows Audio

```

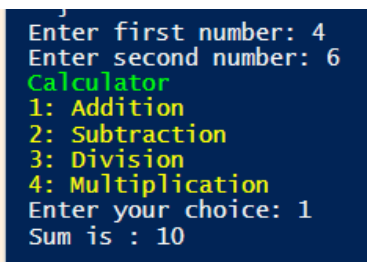
In the above example, you have a menu which gives us the options to check 3 types of services, this is displayed using 'Write-Host'. You are then asked for our choice which is stored in a

variable using 'Read-Host'. The 'switch' command uses the variable which will contain our choice or the value on which you must check upon. In our case, you have three options '1','2' & '3', therefore you have 3 conditions/cases in our 'switch'.

The syntax of a switch command is quite simple and is of the form:

```
switch(variable containing choice)
{
    Condition 1 or Choice 1
    { Action }
    Condition 2 or Choice 2
    { Action }
}
```

Practice 11: Ask user to first enter 2 numbers and then publish a menu to ask which action should be performed on them.



```
Enter first number: 4
Enter second number: 6
Calculator
1: Addition
2: Subtraction
3: Division
4: Multiplication
Enter your choice: 1
Sum is : 10
```

Make use of different data types to involve calculation for values that contain a decimal point.

14. Functions

You have learnt the basics of PowerShell and you should now be able to write most of the scripts if you have successfully understood the logic behind the practice questions.

Once you get a hang of things and you start writing scripts, you would come across a situation in which you are writing the same piece of code multiple times in the script. This makes the code redundant and increases the complexity of the script by adding unnecessary lines to the code. To simplify things, you can make use of *functions* which contains code in 'general' form, i.e. the code will be independent.

You have been using the cmdlet 'Get-Service' a lot in our examples, this cmdlet is also a type of an inbuilt function stored somewhere in the windows which does some magic to find the status of the service. The example may create some doubts that why are you complicating things by using functions, but our purpose here is to understand how you can create a function and use them wisely in our scripts.

Functions can be broadly be created in 2 types as below:

- I. Without Parameters
- II. With Parameters

Without Parameters: This is the simplest form of a function which when called will perform the set of commands you write inside it. Example,

The screenshot shows a PowerShell console window on the left and a text editor window titled 'Untitled1.ps1' on the right. In the console, the function 'hi' is defined with three 'Write-Host' commands: 'My name is John' (green), 'My age is 30' (yellow), and 'I like PowerShell' (cyan). The function is then called 'hi', and the output is displayed in the corresponding colors. The editor window shows the same function definition in a syntax-highlighted format.

Let us first understand the syntax of writing this type of function which is very simple and straightforward.

function "our choice of name"

```
{  
  1st line in code  
  2nd line in code  
  etc  
}
```

As a good practice, you should always write the functions at the beginning of our script. Now when you need to call the function, you just have to write the name of the function. In the above example, you wrote a function with name *'hi'* with 3 *'Write-Host'* commands in it. And to call it, you just wrote the name of the function to execute it.

With Parameters: This is a bit tricky, but you should be able to understand it easily if you understood the first type of function.

The screenshot shows a PowerShell console window on the left and a text editor window titled 'Untitled1.ps1' on the right. In the console, the function 'hi' is defined with two parameters: '[string]\$name' and '[int]\$age'. The function uses 'Write-Host' commands to output the name and age. The function is then called 'hi -name Joe -age 35', and the output is displayed. The editor window shows the same function definition in a syntax-highlighted format.

For example, you want the code to be generic so that it can be used for any name and age. You can make use of a function with parameters. The first way to write this type of function is as below

function "our choice of name"

```
{  
  param  
  (  
    [data type]variable1,  
    [data type]variable2
```

```

    )
    1st line in code using variable 1
    2nd line in code using variable 2
}

```

Now let us understand the syntax using the above example. You defined the name of the function as *'hi'*, you also know that you must ask the user to provide name which is of data type *'string'* and age which is of data type *'int'*.
Now that you know what data type of parameters you are going to use; you just need to mention a variable which will store the value. So, our function should now look like.

```

function hi
{
    param
    (
        [string]$name,
        [int]$age
    )
}

```

And to call the function, you will pass on the information as below.

hi -name Joe -Age 35

So, what happens when you execute the above command. You are calling the function *'hi'*, passing the name *'Joe'* which gets temporarily stored in the variable *'\$name'* and passing the age *'35'* which gets temporarily stored in the variable *'\$age'*.
Using these 2 variables, you printed the rest of the statements.

Practice 12:

- i) Using functions without parameter, find the count of total services that are in running or stopped state.

```

PS C:\> countstart
Total services in running state = 127

PS C:\> countstop
Total services in stopped state = 147

```

- ii) Using function with parameter, find the count of total services that are in running or stopped state.

```

PS C:\> countservice -status Running
Total services in running state = 129

PS C:\> countservice -status Stopped
Total services in stopped state = 145

```

Practice 13: Now that you have learnt all the basics, let us make use of all of them together.

- 1) Write a menu-based script which when executed will first show the below menu:

```
Area Calculator
Main Menu
Please select the option to perform the respective task
1: Area of Square
2: Area of Rectangle
3: Area of Circle
4: Area of Triangle
5: Exit
Enter your choice: |
```

- 2) If option 1 is selected, display a new menu and the previous one should disappear.

```
Area of Square
Enter the side of the square: 4
```

- 3) Display the result and show another menu that will give the options as below and perform the respective tasks.

```
Area of Square
Enter the side of the square: 4
Area of the square : 16
Please select the next option
1: Return to Main Menu
2: Exit
Enter your choice: |
```

- 4) If '1' is selected, it should take you back to the 'Main Menu'. If '2' is selected, it should exit from the script and if any other entry is made, it should display that the entry is incorrect and ask the user to re-enter the option.

Please select the next option

1: Return to Main Menu
2: Exit

Enter your choice: 5

Enter correct option

Please select the next option

1: Return to Main Menu
2: Exit

Enter your choice: |

Give it your best attempt before checking the answer.

15. Connecting to O365

Nowadays everything is moving towards the cloud and to perform our tasks, you must connect PowerShell to the online version of the application rather than using the on-premise one.

Before you start looking at the different ways, you must understand what a module is? A module is just like a PowerShell script which contains various functions. Once you import the module in our PowerShell session, you can make use of the functions that are present in that module.

Example, the module “Microsoft.PowerShell.Management” is installed on our machines which enables us to make use of cmdlets such as “Get-Service”, “Copy-Item”, “Get-EventLog”.

```
PS C:\Windows\system32> Get-Module Microsoft.PowerShell.Management
```

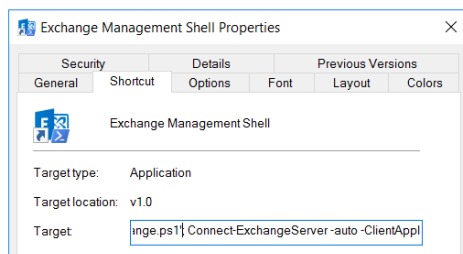
ModuleType	Version	Name	ExportedCommands
Manifest	3.1.0.0	Microsoft.PowerShell.Management	{Add-Computer, Add-Content, Checkpoint-Computer, C}

Let us now check how you can connect to a few online applications.

1) Exchange Online

- Before connecting to Exchange Online let us first understand how the Exchange Management Shell (EMS) works in an on-premise environment.
- EMS is nothing but a shortcut for PowerShell through which it connects to any available Exchange Server in your environment.
- If you open the properties of EMS shortcut, you can see that it uses an inbuilt function “Connect-ExchangeServer” to automatically connect to any Exchange server in the environment.

```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -noexit -command ". 'C:\Program Files\Microsoft\Exchange Server\V15\bin\RemoteExchange.ps1'; Connect-ExchangeServer -auto -ClientApplication:ManagementShell "
```



- Similarly, you can connect to a specific server via a normal PowerShell window using the cmdlet “New-PSSession”. The below 3 commands can be executed to perform this action:

\$Credential = Get-Credential

Since security is quite important, you need to have proper rights to authenticate and connect to the Exchange Server. So, you will save our administrator credentials in the variable ‘\$Credential’.

*\$Session = New-PSSession -ConfigurationName Microsoft.Exchange -ConnectionUri
http://ExchangeServerFQDN/PowerShell/ -Authentication Kerberos -Credential \$UserCredential*

Now, let us understand what is happening in the above command.

- I. As the name suggests, the 'New-PSSession' cmdlet creates a new PowerShell session(connection) to any local or remote computer.
- II. Since you want to connect to an Exchange Server, you have mentioned the "ConfigurationName" as "Microsoft.Exchange".
Example, you have an exchange server "ExchangeServer01" and a non-exchange server "Server01".
When you execute the above command and connect to "ExchangeServer01", it will work as Microsoft Exchange is installed on the server. Now, if you try to connect to server "Server01", it will fail because this server does not have exchange installed on it. Therefore, whichever configuration you use it should be present on the respective host that you are connecting to. Also, if you do not mention anything then it will connect to a normal PowerShell session.
- III. The parameter "ConnectionURI" is the remote host that you must connect to, which in our case is the FQDN of an Exchange Server.

Through the above command, you have created a PowerShell session to an exchange server. The next step would be to make use of this session for which you would need to execute the below command.

Import-PSSession \$Session -DisableNameChecking

Once you have completed our work, you should always remember to disconnect the remote PowerShell session by executing the below command.

Remove-PSSession \$Session

Now that you have understood how you can connect to an Exchange Server using New-PSSession, lets us connect to Exchange Online.

\$Credential = Get-Credential

*\$Session = New-PSSession -ConfigurationName Microsoft.Exchange -ConnectionUri
<https://outlook.office365.com/powershell-liveid/> -Credential \$UserCredential -Authentication Basic -
AllowRedirection*

Since you know what was happening in the similar command while connecting to an on-premise exchange server, you can relate what would be happening in case of the above command.

Import-PSSession \$Session -DisableNameChecking

And once you have completed our work, execute the command to disconnect the session.

Remove-PSSession \$Session

Note: The above steps make use of Basic Authentication which probably in most of the environments would have been blocked by now as it is approaching the end of life.

Therefore, you need to know mechanisms that supports more secure methods like 'Modern Authentication'.

EXO V2 Module: The Exchange Online PowerShell V2 module uses Modern Authentication. It not only contains better cmdlets than the previous version you discussed but only simplifies our work to connect to Exchange Online.

You can always search for the name of the module on Microsoft's site (<https://www.powershellgallery.com/>) but if you want you can do some digging from PowerShell as well.

```
PS C:\> find-module *exchangeonline*

Version      Name                Repository      Description
-----
1.0.1        ExchangeOnlineManagement PSGallery      This is a General Availability (GA) release of Exchange Online PowerShell V2 module....
2.0.3.5      ExchangeOnlineShell   PSGallery      Module for creation a session to manage Exchange Online Shell with or without Proxy Settings.Supports MFA and connection to Government cloud.
0.2.17       BitTitan.Runbooks.ExchangeOnline PSGallery      PowerShell module for common Exchange Online functions and resources used in BitTitan Runbooks.
0.2.20       BitTitan.Runbooks.ExchangeOnline... PSGallery      PowerShell module for common Exchange Online functions and resources used in BitTitan Runbooks.
```

You tried searching for a module with the string “*exchangeonline*” through the cmdlet “Find-Module” but found 2 similar instances “ExchangeOnlineManagement” and “ExchangeOnlineShell”.

To identify the right module, you can look for the author in these 2 cases. In our case, you want the one written by Microsoft.

```
PS C:\> find-module exchangeonline* | ft name,author

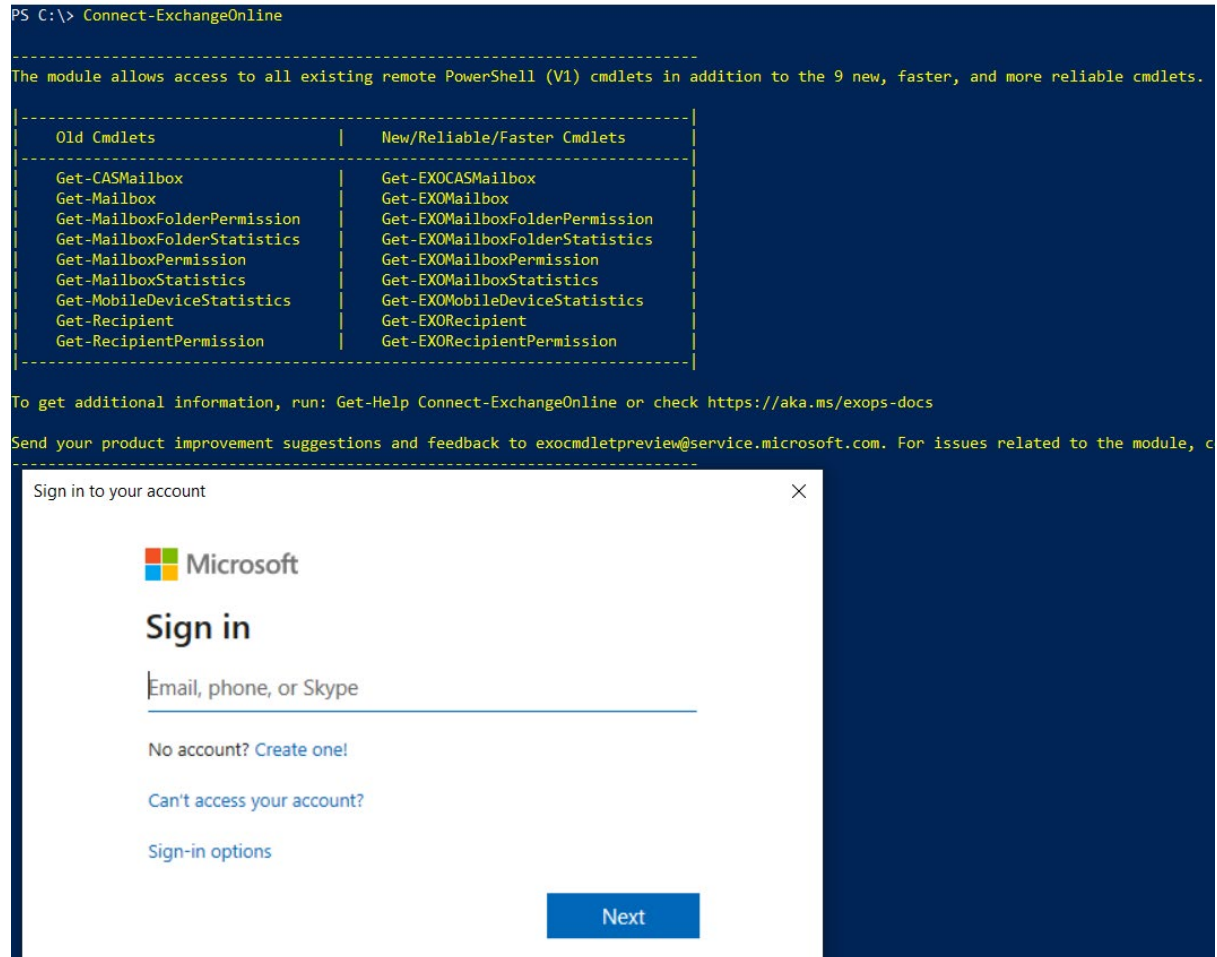
Name                Author
----
ExchangeOnlineManagement Microsoft Corporation
ExchangeOnlineShell  Andy Svintsitsky
```

To install the module, execute the below command.

```
PS C:\> Install-Module -Name ExchangeOnlineManagement

Untrusted repository
You are installing the modules from an untrusted repository. If you trust this repository, change it
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"): y
PS C:\>
PS C:\>
```

Once this has been completed, you must execute the cmdlet “Connect-ExchangeOnline” and you would then be prompted to provide our credentials to connect to EXO.



2) Azure Active Directory

To connect to Azure Active Directory, you must install a module to get the required cmdlets imported in our PowerShell window.

Install-Module -Name AzureAD

Once the module has been installed, execute the below command to connect to AzureAD.

Connect-AzureAD

Once you are connected, you can make use of cmdlets that have "AzureAD" in it. Example,

```
PS C:\> Get-Command "*AzureAD*"

CommandType      Name                                           Version  Source
-----
Alias             Get-AzureADApplicationProxyConnectorGroupMembers 2.0.2.4  AzureAD
Function          Get-AzureADDDL                                0.6      MScldLoginAssistant
Cmdlet            Add-AzureADApplicationOwner                   2.0.2.4  AzureAD
Cmdlet            Add-AzureADDeviceRegisteredOwner              2.0.2.4  AzureAD
Cmdlet            Add-AzureADDeviceRegisteredUser               2.0.2.4  AzureAD
Cmdlet            Add-AzureADDirectoryRoleMember                2.0.2.4  AzureAD
Cmdlet            Add-AzureADGroupMember                       2.0.2.4  AzureAD
Cmdlet            Add-AzureADGroupOwner                        2.0.2.4  AzureAD
Cmdlet            Add-AzureADMSLifecyclePolicyGroup             2.0.2.4  AzureAD
Cmdlet            Add-AzureADServicePrincipalOwner              2.0.2.4  AzureAD
Cmdlet            Confirm-AzureADDomain                        2.0.2.4  AzureAD
Cmdlet            Connect-AzureAD                              2.0.2.4  AzureAD
Cmdlet            Disconnect-AzureAD                           2.0.2.4  AzureAD
```

But there are some cmdlets or tasks that cannot be completed when you use the above module for which you require the module “MSOnline”

Install-Module MSOnline

Once the module has been installed, similarly connect to it as you did previously.

Connect-MsolService

Once you are connected, you can make use of cmdlets that have “Msol” in it. Example,

```
PS C:\> Get-Command *msol*

CommandType      Name                                           Version  Source
-----
Cmdlet            Add-MsolAdministrativeUnitMember              1.1.183.17 MSOnline
Cmdlet            Add-MsolForeignGroupToRole                    1.1.183.17 MSOnline
Cmdlet            Add-MsolGroupMember                           1.1.183.17 MSOnline
Cmdlet            Add-MsolRoleMember                           1.1.183.17 MSOnline
Cmdlet            Add-MsolScopedRoleMember                     1.1.183.17 MSOnline
Cmdlet            Confirm-MsolDomain                           1.1.183.17 MSOnline
Cmdlet            Confirm-MsolEmailVerifiedDomain               1.1.183.17 MSOnline
Cmdlet            Connect-MsolService                           1.1.183.17 MSOnline
Cmdlet            Convert-MsolDomainToFederated                 1.1.183.17 MSOnline
Cmdlet            Convert-MsolDomainToStandard                  1.1.183.17 MSOnline
Cmdlet            Convert-MsolFederatedUser                     1.1.183.17 MSOnline
Cmdlet            Disable-MsolDevice                           1.1.183.17 MSOnline
Cmdlet            Enable-MsolDevice                             1.1.183.17 MSOnline
```

3) SharePoint Online

In case of SharePoint Online, instead of installing the module via cmdlet “Install-Module”, you need to manually install the MSI for SharePoint Online Management Shell. The link for the current version available is:

<https://www.microsoft.com/en-us/download/details.aspx?id=35588>

Once the MSI has been installed, execute the below commands to connect to the Admin Portal. Replace <domainhost> with the name of your O365 domain accordingly.

```
$credential = Get-Credential
```

```
Import-Module Microsoft.Online.SharePoint.PowerShell -DisableNameChecking
```

```
Connect-SPOService -Url https://<domainhost>-admin.sharepoint.com -credential $credential
```

4) Skype for Business Online

Similarly, to connect to SharePoint Online, you can install the current version of the MSI from the below link:

<https://www.microsoft.com/en-us/download/details.aspx?id=39366>

Once the MSI has been installed, execute the below commands to connect to the Admin Portal.

```
Import-Module SkypeOnlineConnector
```

```
$sfboSession = New-CsOnlineSession -Credential $credential
```

```
Import-PSSession $sfboSession
```

16. Tips:

I am writing the below tips on my experience and what I have learnt from others. It may not be the best information, but it has helped me in developing my PowerShell skills.

- 1) '#' (hash) is a character that can be used to write remarks in a script. If you add a '#' in front of any line, it will be considered as a comment and it will not be compiled.
As a good practice, write a short summary in the beginning of the script about its purpose by using "#, <# #>". This way if someone else is reading the script, they can get a little overview before going through the entire script.
- 2) Always try to make your script as generic as possible. The static information like paths to a file or the path to which the data should be exported should be as dynamic as possible.
In this way, if the script must be moved or executed from a different place, then there should be minimum modification required.
- 3) After you make the script as generic as possible, there would always be some variables or constant values that would have to be used in the script. Example, you need to find the list of users present in an OU. To make the script generic, you can save the OU in a variable and then call the variable, when you are executing the command.
Also, such variables should be written in the beginning of the script with a short explanation so that the next person who reads the script can easily identify the purpose of the variable.

```
1 $OU = "OU=Sales,OU=Department,OU=LabUsers,DC=lab,DC=local","OU=Finance,OU=Department,OU=LabUsers,DC=lab,DC=local"
2
3 ## The OU variable contains the information about the OUs in which we must find the users
4
5 $result = @()                                ## This is an empty array which will store the user information found in the OUs
6 foreach($o in $OU)
7 {
8     $users = Get-ADUser -SearchBase $o -Filter *
9     $result += $users
10 }
11
12 $result
13
```

- 4) This is my personal preference, if I am executing a script, I always like to know the time taken by it to complete. There could be other ways to find the information, but I do it in the below way.

Start-Transcript

```
$date = Get-Date                                ## To be used to see the total run time of the script
```

```
<#
```

Code

```
#>
```

```
Write-Host "`nRun time to generate report was $(((get-date) - $date).totalseconds) seconds."
```

Stop-Transcript

- 5) Whenever you import a PS Session in a script, you should always remember to remove it at the end. In this way you close the session to the respective host and avoid causing any unwanted issues.

```
$UserCredential = Get-Credential
$Session = New-PSSession -ConfigurationName Microsoft.Exchange -ConnectionUri https://outlook.office365.com/powershell-liveid/ -Credential $UserCredential -Authentication Basic -AllowRedirection
Import-PSSession $Session -DisableNameChecking

<#
Code
#>

Remove-PSSession $Session
```

- 6) A good script or program should perform the steps accurately but should look good as well, therefore, there should be proper indentation while writing the code. Also, whenever you must write a code which involves curly braces '{}', always open and end it before writing the code inside it. This way the code will be properly indented, and you will not forget closing the loops. This is again my personal preference and you can do as per your convenience.

```
4
5     foreach($u in $users)
6     {
7     }
```

- 7) If you are using a temporary variable inside a loop, you should always set it as 'null' before using it. If you do not do this and if at a step the command fails, then the variable will contain the value from the last iteration. Example,

```
1
2 $result = @()
3 $users = Get-Content C:\Temp\Users.txt
4
5     foreach($u in $users)
6     {
7         $data = $null
8         $data = Get-ADuser $u
9         $result += $data
10    }
```

- 8) If you have a loop that will work on thousands of objects, then you should avoid using 'Write-Host' in it as it slows down the overall execution time.
- 9) It is always good to keep track of the execution of the script both when you are executing it manually or scheduling it. To do this, the simplest way is to make use of the cmdlets 'Start-Transcript' and 'Stop-Transcript'. By default, the transcript is saved in the 'Users' directory and if you want to save it at a specific location, you can define it while starting it.

```
1 Start-Transcript c:\Temp\Logs\logs.txt
2
3 <#
4     code
5     #>
6
7 Stop-Transcript
```

- 10) Sometimes, when you are working on a big script, it is easier to split it in multiple parts and then call them from the main script.

To call a second script from the first script, you can do it by writing it in the form

`.. \functions.ps1`

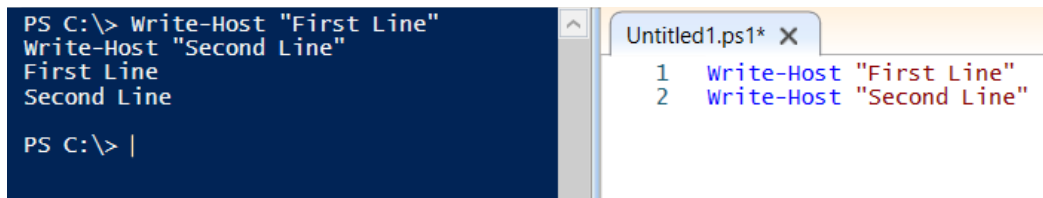
Make sure that both the scripts are kept in the same location as “.” looks for the file in the same location where the first one is present.

`.. \functions.ps1`

`## Importing script containing functions in the script`

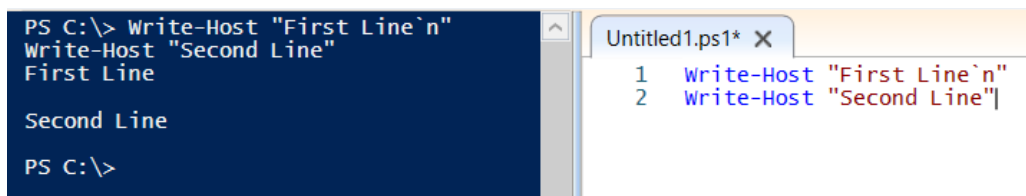
- 11) How to add a tab or a new line when you are displaying information in the PowerShell window.

Normal Case:

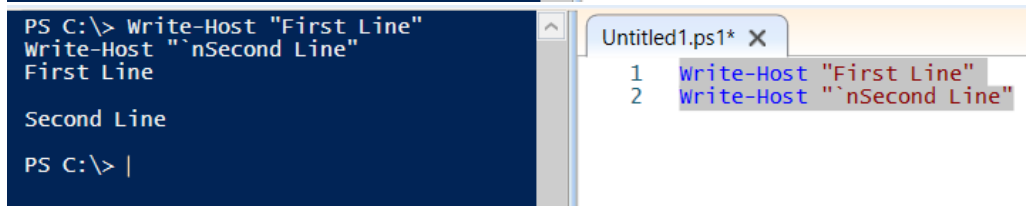


The screenshot shows a PowerShell console window on the left and a script editor window titled 'Untitled1.ps1*' on the right. The console displays the output of a script: 'First Line' followed by 'Second Line' on the next line. The script editor shows two lines of code: '1 Write-Host "First Line"' and '2 Write-Host "Second Line"'. The console output is wrapped to fit the window width.

Now, if you want an additional line between “First Line” and “Second Line”, you can either write ‘`n`’ at the end of “First Line” or at the beginning of “Second Line”

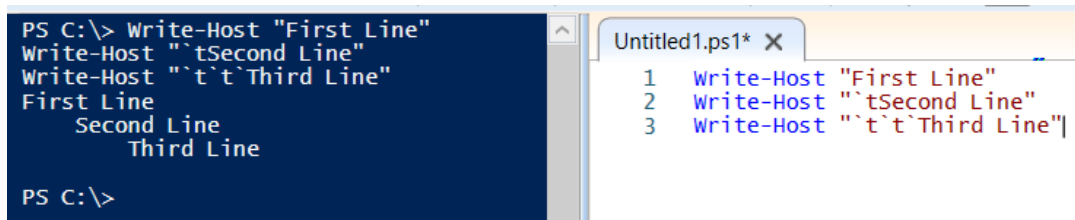


The screenshot shows a PowerShell console window on the left and a script editor window titled 'Untitled1.ps1*' on the right. The console displays the output of a script: 'First Line' followed by a blank line, then 'Second Line' on the next line. The script editor shows two lines of code: '1 Write-Host "First Line`n"' and '2 Write-Host "Second Line"|'. The console output shows a new line between the two lines of text.



The screenshot shows a PowerShell console window on the left and a script editor window titled 'Untitled1.ps1*' on the right. The console displays the output of a script: 'First Line' followed by a blank line, then 'Second Line' on the next line. The script editor shows two lines of code: '1 Write-Host "First Line"' and '2 Write-Host "`nSecond Line"'. The console output shows a new line between the two lines of text.

To introduce a tab in the string, you can use ‘`t`’.

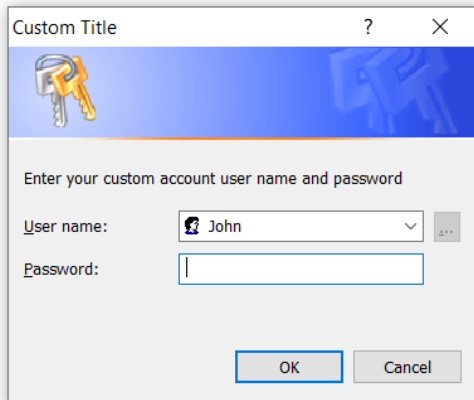


The screenshot shows a PowerShell console window on the left and a script editor window titled 'Untitled1.ps1*' on the right. The console displays the output of a script: 'First Line' followed by 'Second Line' with a tab, then 'Third Line' with two tabs. The script editor shows three lines of code: '1 Write-Host "First Line"', '2 Write-Host "`tSecond Line"', and '3 Write-Host "`t`tThird Line"|'. The console output shows the text indented with tabs.

- 12) Till PowerShellv5, you do not have the option to make changes to the title of the 'Get-Credential' popup box. An alternative to achieve this is via the below command.

```
$cred = $host.ui.PromptForCredential("Custom Title", "Enter your custom account user name and password", "John", "")
```

```
$host.ui.PromptForCredential("Custom Title", "Enter your custom account user name and password", "John", "")
```



In PowerShell 6.0, the option for 'Title' has been introduced and the above task can be performed using the command.

```
$cred = Get-Credential -Message "Enter your custom account user name and password" -Title "Custom Title" -UserName "John"
```

- 13) When you are working on an activity in which you must perform a removal action, PowerShell prompts us to confirm. If it is a bulk activity, it is not feasible to provide confirmation hundreds of time. In this case, you can add *"-confirm:\$false"* at the end of the command through which you do not have to confirm every time.

```
PS C:\> Remove-ADGroupMember -Identity TestDL01 -Members testuser1
Confirm
Are you sure you want to perform this action?
Performing the operation "Set" on target "CN=TestDL01,OU= LabGroups,DC= lab,DC=local".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): 
```

```
PS C:\>
PS C:\> Remove-ADGroupMember -Identity TestDL01 -Members testuser1 -Confirm:$false
PS C:\> 
```

- 14) Sometimes when you are checking an output of a command, you may not get all the values on the screen and the data is truncated.

```
PS C:\> Get-Module Microsoft.PowerShell.Management | fl ExportedCommands

ExportedCommands : {[Add-Computer, Add-Computer], [Add-Content, Add-Content], [Checkpoint-Computer, Checkpoint-Computer], [Clear-Content, Clear-Content]...}
```

In the above example, you can see that there are many commands present in the module “Microsoft.PowerShell.Management” but you do not see all those commands right away and a part of the information is displayed followed by 3 dots at the end.

To display all the information, you can first run the below command.

\$FormatEnumerationLimit = -1

```
PS C:\> Get-Module Microsoft.PowerShell.Management | fl ExportedCommands

ExportedCommands : {[Add-Computer, Add-Computer], [Add-Content, Add-Content], [Checkpoint-Computer, Checkpoint-Computer], [Clear-Content, Clear-Content]...}

PS C:\> $FormatEnumerationLimit = -1
PS C:\>
PS C:\> Get-Module Microsoft.PowerShell.Management | fl ExportedCommands

ExportedCommands : {[Add-Computer, Add-Computer], [Add-Content, Add-Content], [Checkpoint-Computer, Checkpoint-Computer], [Clear-Content, Clear-Content], [Clear-Eventlog, Clear-Eventlog], [Clear-Item, Clear-Item], [Clear-ItemProperty, Cl
[Get-Process, Get-Process], [Get-PSDrive, Get-PSDrive], [Get-PSProvider, Get-PSProvider], [Get-Service, Get-Service], [Get-TimeZone, Get-TimeZone], [Get-Transaction, Get-Transaction], [Get-WmiObject, Get-WmiObject], [I
[Rename-ItemProperty, Rename-ItemProperty], [Reset-ComputerMachinePassword, Reset-ComputerMachinePassword], [Resolve-Path, Resolve-Path], [Restart-Computer, Restart-Computer], [Restart-Service, Restart-Service], [Resto
Test-Path], [Undo-Transaction, Undo-Transaction], [Use-Transaction, Use-Transaction], [Wait-Process, Wait-Process], [Write-Eventlog, Write-Eventlog], [gcb, gcb], [gin, gin], [gtz, gtz], [scb, scb], [stz, stz]}
```

- 15) In the above point you were able to view the entire information in the output, what if you want to copy all of it. One way which you all are aware of is to select the content and then paste it but if a lot of text must be copied then it is a tedious task. The simplest way to copy large text is to add “ */ clip*” at the end of the command and all the information that gets displayed through that command gets copied on the clipboard.

```
PS C:\> Get-Module Microsoft.PowerShell.Management | fl ExportedCommands | clip
PS C:\>
```


17. Answers to Practice questions:

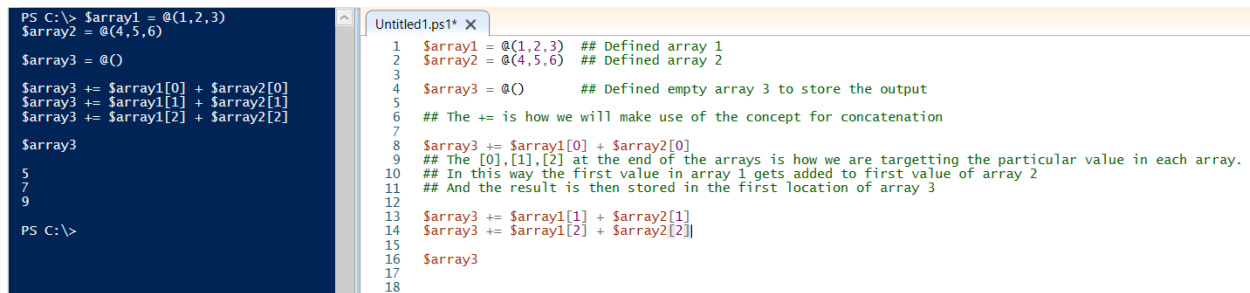
Practice 1:

```
$array1 = @(1,2,3)
$array2 = @(4,5,6)

$array3 = @()

$array3 += $array1[0] + $array2[0]
$array3 += $array1[1] + $array2[1]
$array3 += $array1[2] + $array2[2]

$array3
```



The screenshot shows a PowerShell console window on the left and a script editor window titled 'Untitled1.ps1' on the right. The console output shows the array \$array3 containing the values 5, 7, and 9. The script editor shows the code being executed, with comments explaining the steps: defining array 1, defining array 2, defining an empty array 3, and then concatenating the values from arrays 1 and 2 into array 3.

```
PS C:\> $array1 = @(1,2,3)
$array2 = @(4,5,6)

$array3 = @()

$array3 += $array1[0] + $array2[0]
$array3 += $array1[1] + $array2[1]
$array3 += $array1[2] + $array2[2]

$array3

5
7
9
PS C:\>
```

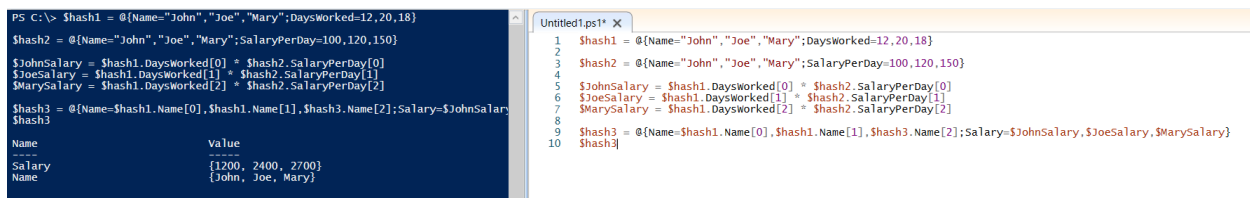
```
1 $array1 = @(1,2,3) ## Defined array 1
2 $array2 = @(4,5,6) ## Defined array 2
3
4 $array3 = @() ## Defined empty array 3 to store the output
5
6 ## The += is how we will make use of the concept for concatenation
7
8 $array3 += $array1[0] + $array2[0]
9 ## The [0],[1],[2] at the end of the arrays is how we are targeting the particular value in each array.
10 ## In this way the first value in array 1 gets added to first value of array 2
11 ## And the result is then stored in the first location of array 3
12
13 $array3 += $array1[1] + $array2[1]
14 $array3 += $array1[2] + $array2[2]
15
16 $array3
17
18
```

Practice 2:

```
$hash1 = @{'Name'='John','Joe','Mary';DaysWorked=12,20,18}
$hash2 = @{'Name'='John','Joe','Mary';SalaryPerDay=100,120,150}

$JohnSalary = $hash1.DaysWorked[0] * $hash2.SalaryPerDay[0]
$JoeSalary = $hash1.DaysWorked[1] * $hash2.SalaryPerDay[1]
$MarySalary = $hash1.DaysWorked[2] * $hash2.SalaryPerDay[2]

$hash3 =
@{'Name'=$hash1.Name[0],$hash1.Name[1],$hash3.Name[2];Salary=$JohnSalary,$JoeSalary,$MarySalary}
$hash3
```



The screenshot shows a PowerShell console window on the left and a script editor window titled 'Untitled1.ps1' on the right. The console output shows the hash table \$hash3 with the following structure: Name (John, Joe, Mary) and Salary (1200, 2400, 2700). The script editor shows the code being executed, with comments explaining the steps: defining hash1, defining hash2, calculating the salary for each person, and then creating the final hash3.

```
PS C:\> $hash1 = @{'Name'='John','Joe','Mary';DaysWorked=12,20,18}
$hash2 = @{'Name'='John','Joe','Mary';SalaryPerDay=100,120,150}

$JohnSalary = $hash1.DaysWorked[0] * $hash2.SalaryPerDay[0]
$JoeSalary = $hash1.DaysWorked[1] * $hash2.SalaryPerDay[1]
$MarySalary = $hash1.DaysWorked[2] * $hash2.SalaryPerDay[2]

$hash3 = @{'Name'=$hash1.Name[0],$hash1.Name[1],$hash3.Name[2];Salary=$JohnSalary,$JoeSalary,$MarySalary}
$hash3

Name Value
----
Salary {1200, 2400, 2700}
Name {John, Joe, Mary}
```

```
1 $hash1 = @{'Name'='John','Joe','Mary';DaysWorked=12,20,18}
2
3 $hash2 = @{'Name'='John','Joe','Mary';SalaryPerDay=100,120,150}
4
5 $JohnSalary = $hash1.DaysWorked[0] * $hash2.SalaryPerDay[0]
6 $JoeSalary = $hash1.DaysWorked[1] * $hash2.SalaryPerDay[1]
7 $MarySalary = $hash1.DaysWorked[2] * $hash2.SalaryPerDay[2]
8
9 $hash3 = @{'Name'=$hash1.Name[0],$hash1.Name[1],$hash3.Name[2];Salary=$JohnSalary,$JoeSalary,$MarySalary}
10 $hash3
```

Alternate way using [PSCustomObject]

```
$ht1 = [PSCustomObject]@{
    Name = "John","Joe","Mary"
    Daysworked = 12,20,18
}

$ht2 = [PSCustomObject]@{
    Name = "John","Joe","Mary"
    SalaryPerDay = 100,120,150
}

$ht3 = @()

$sal1 = $ht1.Daysworked[0] * $ht2.SalaryPerDay[0]
$sal2 = $ht1.Daysworked[1] * $ht2.SalaryPerDay[1]
$sal3 = $ht1.Daysworked[2] * $ht2.SalaryPerDay[2]

$ht = [PSCustomObject]@{
    Name = $ht1.Name[0]
    Salary = $sal1
}

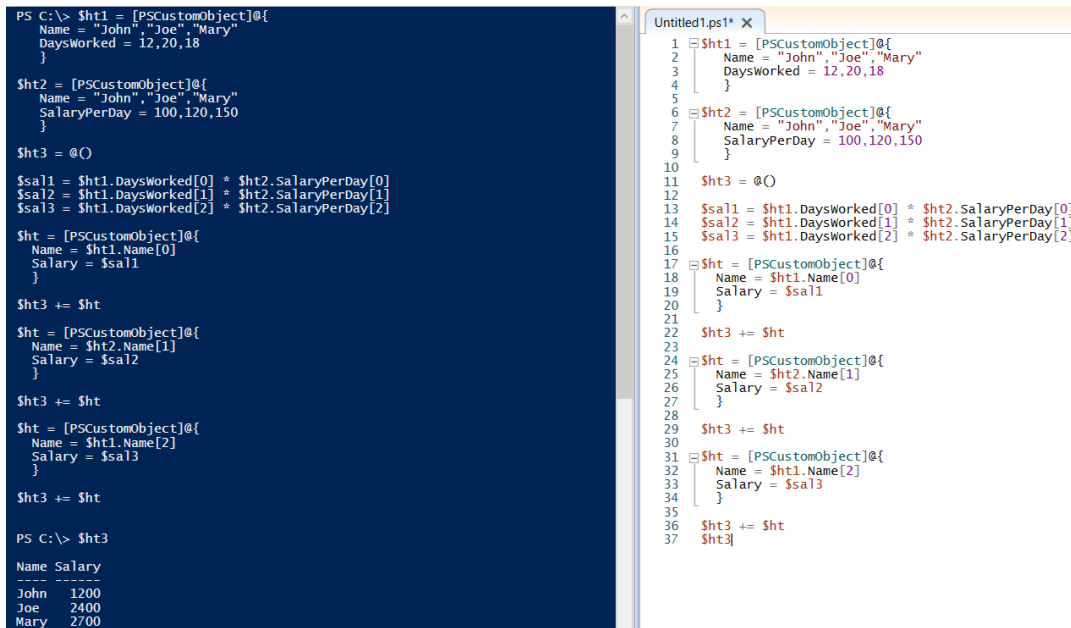
$ht3 += $ht

$ht = [PSCustomObject]@{
    Name = $ht2.Name[1]
    Salary = $sal2
}

$ht3 += $ht

$ht = [PSCustomObject]@{
    Name = $ht1.Name[2]
    Salary = $sal3
}

$ht3 += $ht
$ht3
```



The screenshot shows a PowerShell console window on the left and a text editor window on the right. The console window displays the execution of the script, including the creation of the objects, the calculation of salaries, and the final output table. The text editor window shows the script code being executed.

```
PS C:\> $ht1 = [PSCustomObject]@{
    Name = "John","Joe","Mary"
    Daysworked = 12,20,18
}

$ht2 = [PSCustomObject]@{
    Name = "John","Joe","Mary"
    SalaryPerDay = 100,120,150
}

$ht3 = @()

$sal1 = $ht1.Daysworked[0] * $ht2.SalaryPerDay[0]
$sal2 = $ht1.Daysworked[1] * $ht2.SalaryPerDay[1]
$sal3 = $ht1.Daysworked[2] * $ht2.SalaryPerDay[2]

$ht = [PSCustomObject]@{
    Name = $ht1.Name[0]
    Salary = $sal1
}

$ht3 += $ht

$ht = [PSCustomObject]@{
    Name = $ht2.Name[1]
    Salary = $sal2
}

$ht3 += $ht

$ht = [PSCustomObject]@{
    Name = $ht1.Name[2]
    Salary = $sal3
}

$ht3 += $ht

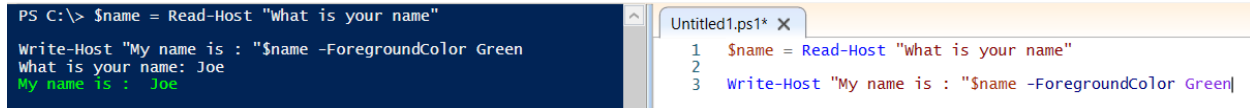
PS C:\> $ht3

Name Salary
----
John 1200
Joe 2400
Mary 2700
```

Practice 3:

```
$name = Read-Host "what is your name"
```

```
Write-Host "My name is : "$name -ForegroundColor Green
```



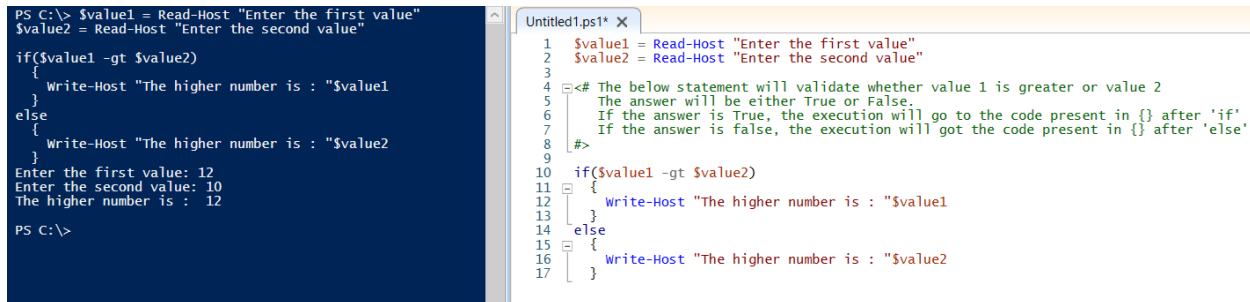
The screenshot shows a PowerShell console window on the left and a script editor window titled 'Untitled1.ps1' on the right. The console window displays the execution of the script: the prompt 'PS C:\>' is followed by '\$name = Read-Host "what is your name"', then 'Write-Host "My name is : "\$name -ForegroundColor Green', and finally the output 'My name is : Joe' in green text. The script editor shows the same two lines of code.

Practice 4:

First question:

```
$value1 = Read-Host "Enter the first value"  
$value2 = Read-Host "Enter the second value"
```

```
if($value1 -gt $value2)  
{  
    Write-Host "The higher number is : "$value1  
}  
else  
{  
    Write-Host "The higher number is : "$value2  
}
```



The screenshot shows a PowerShell console window on the left and a script editor window titled 'Untitled1.ps1' on the right. The console window displays the execution of the script: the prompt 'PS C:\>' is followed by '\$value1 = Read-Host "Enter the first value"', then '\$value2 = Read-Host "Enter the second value"', then 'if(\$value1 -gt \$value2)', then 'Write-Host "The higher number is : "\$value1', then 'else', then 'Write-Host "The higher number is : "\$value2', and finally the output 'The higher number is : 12' in green text. The script editor shows the same code, with line numbers 1 through 17. Line 4 has a comment: '<# The below statement will validate whether value 1 is greater or value 2. The answer will be either True or False. If the answer is True, the execution will go to the code present in {} after 'if'. If the answer is false, the execution will got the code present in {} after 'else'.'.

Second question:

```
Write-Host "Select Country" -ForegroundColor Yellow  
Write-Host "1 : India" -ForegroundColor Cyan  
Write-Host "2: Australia" -ForegroundColor Cyan  
Write-Host "3: China" -ForegroundColor Cyan
```

```
$choice = Read-Host "Please select country"
```

```
if($choice -eq "1")  
{  
    Write-Host "India's captital is New Delhi" -ForegroundColor Green  
}  
elseif($choice -eq "2")  
{  
    Write-Host "Australia's capital is Canberra" -ForegroundColor Green  
}  
elseif($choice -eq "3")  
{  
    Write-Host "China's capital is Beijing" -ForegroundColor Green  
}  
else  
{  
    Write-Host "Wrong choice" -ForegroundColor Red  
}
```

```

PS C:\> Write-Host "Select Country" -ForegroundColor Yellow
Write-Host "1 : India" -ForegroundColor Cyan
Write-Host "2: Australia" -ForegroundColor Cyan
Write-Host "3: China" -ForegroundColor Cyan

$choice = Read-Host "Please select country"

if($choice -eq "1")
{
    Write-Host "India's capital is New Delhi" -ForegroundColor Green
}
elseif($choice -eq "2")
{
    Write-Host "Australia's capital is Canberra" -ForegroundColor Green
}
elseif($choice -eq "3")
{
    Write-Host "China's capital is Beijing" -ForegroundColor Green
}
else
{
    Write-Host "Wrong choice" -ForegroundColor Red
}

Select Country
1 : India
2: Australia
3: China
Please select country: 2
Australia's capital is Canberra

PS C:\> |
  
```

```

1 Write-Host "Select Country" -ForegroundColor Yellow
2 Write-Host "1 : India" -ForegroundColor Cyan
3 Write-Host "2: Australia" -ForegroundColor Cyan
4 Write-Host "3: China" -ForegroundColor Cyan
5
6 $choice = Read-Host "Please select country"
7
8 if($choice -eq "1")
9 {
10     Write-Host "India's capital is New Delhi" -ForegroundColor Green
11 }
12 elseif($choice -eq "2")
13 {
14     Write-Host "Australia's capital is Canberra" -ForegroundColor Green
15 }
16 elseif($choice -eq "3")
17 {
18     Write-Host "China's capital is Beijing" -ForegroundColor Green
19 }
20 else
21 {
22     Write-Host "Wrong choice" -ForegroundColor Red
23 }
  
```

Practice 5:

First question:

`Get-Process | ?{$_.ProcessName -eq "notepad"} | select ProcessName,Id`

```

PS C:\> Get-Process | ?{$_.ProcessName -eq "notepad"} | Select ProcessName,Id

ProcessName Id
-----
notepad     7220
notepad     15288

PS C:\>
  
```

```

1 Get-Process | ?{$_.ProcessName -eq "notepad"} | Select ProcessName,Id
2
3 ## Alternative and direct way
4
5 Get-Process Notepad | Select ProcessName,Id
  
```

Second question:

`Get-ChildItem -Path "C:\temp\test"`

`$file = Get-ChildItem -Path "C:\temp\test" | where {$_.Name -like "*.csv"} | select Name,Length`
`$sizeinKB = $file.Length/1KB`
`$sizeinMB = $file.Length/1MB`

`Write-Host "`nFileName : "$file.Name`
`Write-Host "Size in KB : "$sizeinKB`
`Write-Host "Size in MB : "$sizeinMB`

```

PS C:\> Get-ChildItem -Path "C:\temp\test"

Directory: C:\temp\test

Mode                LastWriteTime         Length Name
----                -
-a----           5/31/2020 12:44 PM           51986 Notepad1.txt
-a----           5/31/2020 12:44 PM          675854 Notepad2.txt
-a----           5/29/2020  2:29 PM          1177861 SampleFile.csv

PS C:\> $file = Get-ChildItem -Path "C:\temp\test" | where {$_.Name -like "*.csv"} | select Name,Length
$sizeinKB = $file.Length/1KB
$sizeinMB = $file.Length/1MB

Write-Host "`nFileName : "$file.Name
Write-Host "Size in KB : "$sizeinKB
Write-Host "Size in MB : "$sizeinMB

FileName : SampleFile.csv
Size in KB : 1150.2548828125
Size in MB : 1.12329578399658
  
```

```

1 Get-ChildItem -Path "C:\temp\test"
2
3 $file = Get-ChildItem -Path "C:\temp\test" | where {$_.Name -like "*.csv"} | select Name,Length
4 $sizeinKB = $file.Length/1KB
5 $sizeinMB = $file.Length/1MB
6
7 Write-Host "`nFileName : "$file.Name
8 Write-Host "Size in KB : "$sizeinKB
9 Write-Host "Size in MB : "$sizeinMB
10
11 ## Alternative to find the csv file
12
13 Get-ChildItem -Path "C:\temp\test\*.csv" | select Name,Length
  
```

Practice 6:

`$csv = Import-Csv C:\temp\students.csv`

```

$result = @()
foreach($c in $csv)
{
    if([int]$c.Age -ge 4 -and [int]$c.Age -le 10)
    {
        $school = "Junior"
    }
    elseif([int]$c.Age -ge 11 -and [int]$c.Age -le 17)
    {
        $school = "Senior"
    }
    $temp = [PSCustomObject]@{
        Name = $c.Name
        School = $school
    }
    $result += $temp
}

$result

```

The screenshot shows a PowerShell console window on the left and a text editor window on the right. The console window displays the command to import a CSV file and the resulting output of the script. The text editor window shows the script code being executed.

PowerShell Console Output:

```

PS C:\> $csv = Import-Csv C:\temp\students.csv
$result = @()
foreach($c in $csv)
{
    if([int]$c.Age -ge 4 -and [int]$c.Age -le 10)
    {
        $school = "Junior"
    }
    elseif([int]$c.Age -ge 11 -and [int]$c.Age -le 17)
    {
        $school = "Senior"
    }
    $temp = [PSCustomObject]@{
        Name = $c.Name
        School = $school
    }
    $result += $temp
}

$result

```

Output:

Name	School
John	Junior
Joe	Senior
Mary	Junior
Tom	Senior
Lily	Senior
Emily	Junior

Text Editor Content:

```

1 $csv = Import-Csv C:\temp\students.csv
2
3 $result = @()
4
5 foreach($c in $csv)
6 {
7     if([int]$c.Age -ge 4 -and [int]$c.Age -le 10)
8     {
9         $school = "Junior"
10    }
11    elseif([int]$c.Age -ge 11 -and [int]$c.Age -le 17)
12    {
13        $school = "Senior"
14    }
15    $temp = [PSCustomObject]@{
16        Name = $c.Name
17        School = $school
18    }
19    $result += $temp
20 }
21
22 $result
23

```

Practice 7:

```

$groups = @("Red", "Green", "Yellow", "Blue")

$result = @()
for($i = 1 ; $i -le 20 ; $i++)
{
    $g = Get-Random $groups
    $temp = [PSCustomObject]@{
        RollNumber = $i
        Group = $g
    }
    $result += $temp
}

$result

```

```
PS C:\> $groups = @("Red","Green","Yellow","Blue")
```

```
$result = @()
for($i = 1 ; $i -le 20 ; $i++)
{
    $g = Get-Random $groups
    $temp = [PSCustomObject]@{
        RollNumber = $i
        Group = $g
    }
    $result += $temp
}
$result
```

RollNumber	Group
1	Yellow
2	Blue
3	Red
4	Red
5	Yellow
6	Blue
7	Green
8	Red
9	Red
10	Green
11	Green
12	Green
13	Blue
14	Red
15	Green
16	Green
17	Blue
18	Blue
19	Yellow
20	Green

Untitled1.ps1*

```
1 $groups = @("Red","Green","Yellow","Blue")
2
3 $result = @()
4 for($i = 1 ; $i -le 20 ; $i++)
5 {
6     $g = Get-Random $groups
7     $temp = [PSCustomObject]@{
8         RollNumber = $i
9         Group = $g
10     }
11     $result += $temp
12 }
13 $result
```

Practice 8:

```
while(Get-Process Notepad -ErrorAction SilentlyContinue)
{
    Write-Host "Notepad is running"
}
```

Practice 9:

```
$ct1 = 0
do
{
    $ch = $null
    $ch = Get-Process | ?{$_ .name -like "note*"}
    if($ch -ne $null)
    {
        $ct2 = 2
        Write-Host "Notepad is running"
        Start-Sleep -Seconds 1
        $ct1++
    }
} else
{
    $ct2 = 1
} while($ct2 -ne 1)
$ct1
```

```
PS C:\> $ct1 = 0
do
{
$ch = $null
$ch = Get-Process | ?{$_.name -like "note*"}
if($ch -ne $null)
{$ct2 = 2
Write-Host "Notepad is running"
Start-Sleep -Seconds 1
$ct1++
}
else
{$ct2 = 1}
}while($ct2 -ne 1)
$ct1
Notepad is running
Notepad is running
Notepad is running
Notepad is running
Notepad is running
5

PS C:\>
```

```
Untitled1.ps1* X
1 $ct1 = 0
2 do
3 {
4 $ch = $null
5 $ch = Get-Process | ?{$_.name -like "note*"}
6 if($ch -ne $null)
7 {$ct2 = 2
8 Write-Host "Notepad is running"
9 Start-Sleep -Seconds 1
10 $ct1++
11 }
12 else
13 {$ct2 = 1}
14 }while($ct2 -ne 1)
15 $ct1
```

Practice 10:

```
$ct1 = 0
do
{
$ch = $null
$ch = Get-Process | ?{$_.name -like "note*"}
if($ch -ne $null)
{$ct2 = 2
Write-Host "Notepad is running"
Start-Sleep -Seconds 1
$ct1++
}
else
{$ct2 = 1}
}while($ct2 -eq 1)
$ct1
```

```
PS C:\> $ct1 = 0
do
{
$ch = $null
$ch = Get-Process | ?{$_.name -like "note*"}
if($ch -ne $null)
{$ct2 = 2
Write-Host "Notepad is running"
Start-Sleep -Seconds 1
$ct1++
}
else
{$ct2 = 1}
}while($ct2 -eq 1)
$ct1
Notepad is running
Notepad is running
Notepad is running
3

PS C:\>
```

```
Untitled1.ps1* X
1 $ct1 = 0
2 do
3 {
4 $ch = $null
5 $ch = Get-Process | ?{$_.name -like "note*"}
6 if($ch -ne $null)
7 {$ct2 = 2
8 Write-Host "Notepad is running"
9 Start-Sleep -Seconds 1
10 $ct1++
11 }
12 else
13 {$ct2 = 1}
14 }until($ct2 -eq 1)
15 $ct1
```

Practice 11:

```
[int]$n1 = Read-Host "Enter first number"
[int]$n2 = Read-Host "Enter second number"
```

```

Write-Host "Calculator" -ForegroundColor Green
Write-Host "1: Addition" -ForegroundColor Yellow
Write-Host "2: Subtraction" -ForegroundColor Yellow
Write-Host "3: Division" -ForegroundColor Yellow
Write-Host "4: Multiplication" -ForegroundColor Yellow

```

```
$ch = Read-Host "Enter your choice"
```

```

switch($ch)
{
    1
    {
        $s = $n1+$n2
        Write-Host "Sum is : "$s
    }
    2
    {
        $s = $n1-$n2
        Write-Host "Difference is : "$s
    }
    3
    {
        $s = $n1/$n2
        Write-Host "Quotient is : "$s
    }
    4
    {
        $s = $n1*$n2
        Write-Host "Product is : "$s
    }
}

```

```

PS C:\> [int]$n1 = Read-Host "Enter first number"
[int]$n2 = Read-Host "Enter second number"

Write-Host "Calculator" -ForegroundColor Green
Write-Host "1: Addition" -ForegroundColor Yellow
Write-Host "2: Subtraction" -ForegroundColor Yellow
Write-Host "3: Division" -ForegroundColor Yellow
Write-Host "4: Multiplication" -ForegroundColor Yellow

$ch = Read-Host "Enter your choice"

switch($ch)
{
    1
    {
        $s = $n1+$n2
        Write-Host "Sum is : "$s
    }
    2
    {
        $s = $n1-$n2
        Write-Host "Difference is : "$s
    }
    3
    {
        $s = $n1/$n2
        Write-Host "Quotient is : "$s
    }
    4
    {
        $s = $n1*$n2
        Write-Host "Product is : "$s
    }
}

Enter first number: 18
Enter second number: 16
Calculator
1: Addition
2: Subtraction
3: Division
4: Multiplication
Enter your choice: 4
Product is : 288

```

```

Untitled1.ps1* X
1  [int]$n1 = Read-Host "Enter first number"
2  [int]$n2 = Read-Host "Enter second number"
3
4  Write-Host "Calculator" -ForegroundColor Green
5  Write-Host "1: Addition" -ForegroundColor Yellow
6  Write-Host "2: Subtraction" -ForegroundColor Yellow
7  Write-Host "3: Division" -ForegroundColor Yellow
8  Write-Host "4: Multiplication" -ForegroundColor Yellow
9
10 $ch = Read-Host "Enter your choice"
11
12 switch($ch)
13 {
14     1
15     {
16         $s = $n1+$n2
17         Write-Host "Sum is : "$s
18     }
19     2
20     {
21         $s = $n1-$n2
22         Write-Host "Difference is : "$s
23     }
24     3
25     {
26         $s = $n1/$n2
27         Write-Host "Quotient is : "$s
28     }
29     4
30     {
31         $s = $n1*$n2
32         Write-Host "Product is : "$s
33     }
34 }
35

```

Practice 12:

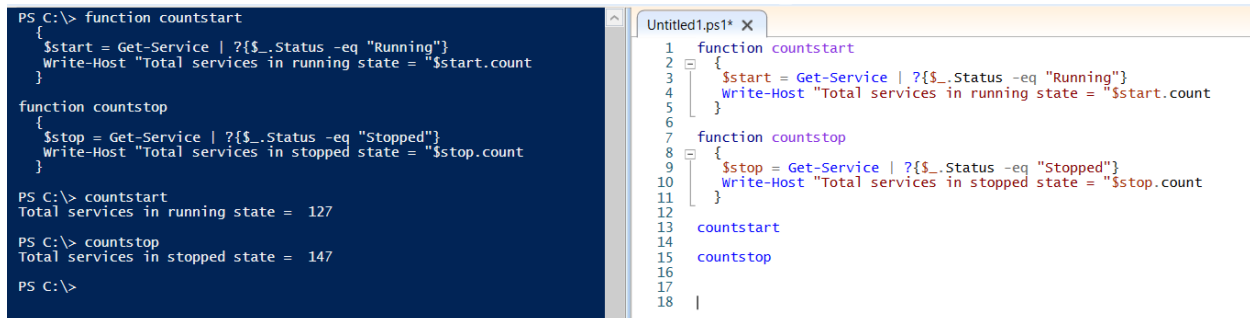
First question:


```
function countstart
{
    $start = Get-Service | ?{$_ .Status -eq "Running"}
    Write-Host "Total services in running state = "$start.count
}
```

```
function countstop
{
    $stop = Get-Service | ?{$_ .Status -eq "Stopped"}
    Write-Host "Total services in stopped state = "$stop.count
}
```

countstart

countstop



The screenshot shows a PowerShell console window on the left and a script editor window titled 'Untitled1.ps1' on the right. The console window displays the output of the functions: 'Total services in running state = 127' and 'Total services in stopped state = 147'. The script editor shows the code for the 'countstart' and 'countstop' functions, followed by calls to 'countstart' and 'countstop'.

```
PS C:\> function countstart
{
    $start = Get-Service | ?{$_ .Status -eq "Running"}
    Write-Host "Total services in running state = "$start.count
}

function countstop
{
    $stop = Get-Service | ?{$_ .Status -eq "Stopped"}
    Write-Host "Total services in stopped state = "$stop.count
}

PS C:\> countstart
Total services in running state = 127

PS C:\> countstop
Total services in stopped state = 147

PS C:\>
```

```
1 function countstart
2 {
3     $start = Get-Service | ?{$_ .Status -eq "Running"}
4     Write-Host "Total services in running state = "$start.count
5 }
6
7 function countstop
8 {
9     $stop = Get-Service | ?{$_ .Status -eq "Stopped"}
10    Write-Host "Total services in stopped state = "$stop.count
11 }
12
13 countstart
14
15 countstop
16
17
18 |
```

Second question:

```
function countservice
{
    param
    (
        [string]$status
    )
    if($status -eq "Running")
    {
        $start = Get-Service | ?{$_ .Status -eq "Running"}
        Write-Host "Total services in running state = "$start.count
    }
    elseif($status -eq "Stopped")
    {
        $stop = Get-Service | ?{$_ .Status -eq "Stopped"}
        Write-Host "Total services in stopped state = "$stop.count
    }
}
```

countservice -status Running

countservice -status Stopped

The screenshot shows a PowerShell console on the left and a text editor window titled 'Untitled1.ps1' on the right. The console displays the execution of the 'countservice' function with status 'Running' (resulting in 129) and 'Stopped' (resulting in 145). The text editor shows the source code of the 'countservice' function, which uses 'Get-Service' to count services in specific states.

```

PS C:\> function countservice
{
    param
    (
        [string]$status
    )
    if($status -eq "Running")
    {
        $start = Get-Service | ?{$_.Status -eq "Running"}
        Write-Host "Total services in running state = $($start.count)
    }
    elseif($status -eq "Stopped")
    {
        $stop = Get-Service | ?{$_.Status -eq "Stopped"}
        Write-Host "Total services in stopped state = $($stop.count)
    }
}

PS C:\> countservice -status Running
Total services in running state = 129

PS C:\> countservice -status Stopped
Total services in stopped state = 145
  
```

```

1 function countservice
2 {
3     param
4     (
5         [string]$status
6     )
7     if($status -eq "Running")
8     {
9         $start = Get-Service | ?{$_.Status -eq "Running"}
10        Write-Host "Total services in running state = $($start.count)
11    }
12    elseif($status -eq "Stopped")
13    {
14        $stop = Get-Service | ?{$_.Status -eq "Stopped"}
15        Write-Host "Total services in stopped state = $($stop.count)
16    }
17 }
18
19 countservice -status Running
20
21 countservice -status Stopped
  
```

Practice 13:

```

function mainmenu
{
    Write-Host "`t`t`tArea Calculator" -ForegroundColor Green
    Write-Host "`n`t`tMain Menu" -ForegroundColor Yellow
    Write-Host "`nPlease select the option to perform the respective task`n" -
ForegroundColor Yellow

    Write-Host "1: Area of Square" -ForegroundColor Green
    Write-Host "2: Area of Rectangle" -ForegroundColor Green
    Write-Host "3: Area of Circle" -ForegroundColor Green
    Write-Host "4: Area of Triangle" -ForegroundColor Green

    Write-Host "5: Exit`n" -ForegroundColor Green

    $choice = Read-Host "Enter your choice"
    return $choice
}

function checkmenu
{
    Write-Host "`n`nPlease select the next option" -ForegroundColor Yellow
    Write-Host "`n1: Return to Main Menu" -ForegroundColor Green
    Write-Host "`n2: Exit`n" -ForegroundColor Green

    $ch2 = Read-Host "Enter your choice"
    if($ch2 -eq "1")
    {
        continue
    }
    if($ch2 -eq "2")
    {
        exit
    }
    else
    {
        Write-Host "`nEnter correct option" -ForegroundColor Red
        checkmenu
    }
}

function square
{
    cls
    Write-Host "`t`t`tArea of Square`n" -ForegroundColor Green
    [int]$side = Read-Host "Enter the side of the square"
    $area = $side * $side
    Write-Host "`nArea of the square : "$area -ForegroundColor Green

    checkmenu
}
  
```

```

function rectangle
{
    cls
    Write-Host "`t`tArea of Rectangle`n" -ForegroundColor Green
    [int]$length = Read-Host "Enter length of the rectangle"
    [int]$breadth = Read-Host "Enter breadth of the rectangle"
    $area = $length * $breadth
    Write-Host "nArea of the rectangle : "$area -ForegroundColor Green

    checkmenu
}

function circle
{
    cls
    Write-Host "`t`tArea of Circle`n" -ForegroundColor Green
    [int]$radius = Read-Host "Enter the radius of the circle"
    $area = 3.14*$radius*$radius
    Write-Host "nArea of the circle : "$area -ForegroundColor Green

    checkmenu
}

function triangle
{
    cls
    Write-Host "`t`tArea of Triangle`n" -ForegroundColor Green
    [int]$height = Read-Host "Enter height of triangle"
    [int]$base = Read-Host "Enter base of triangle"
    $area = 0.5*$height*$base
    Write-Host "nArea of Triangle : "$area -ForegroundColor Green

    checkmenu
}
do
{
    cls
    $ch1 = mainmenu

    switch($ch1)
    {
        1
        {
            cls
            square
            checkmenu
        }
        2
        {
            cls
            rectangle
            checkmenu
        }
        3
        {
            cls
            circle
            checkmenu
        }
        4
        {
            cls
            triangle
            checkmenu
        }
    }
}while($ch1 -ne "5")

```