POWERSHELL NOTES BY ROHIT PRAKASH {https://www.linkedin.com/in/rohit-prakash-204391252/}

POWERSHELL

Windows PowerShell is a **command-line shell** and **scripting language** designed especially for system administration. Its analogue in Linux is called as **Bash** Scripting.

POWERSHELL CMDLETS

Windows PowerShell commands, called **cmdlets**, let you manage the computers from the command line.

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

Cmdlet vs Command

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

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

Powershell - Files and Folder Operations

- CREATE FOLDERS AND FILES-   New-Item -Path 'path of the directory or File' -ItemType Directory or File
- COPY FOLDERS AND FILES-   Copy-Item ‘what to copy’ ‘where to copy’
- DELETING FOLDERS AND FILES-   Remove-Item 'what to be deleted'
- MOVING FOLDERS AND FILES-   Move-Item ‘From path’ ‘to path’
- RETREIVING ITEM- Get-content ‘path of file of which content of a file as an array is to retrieved’
- CHECK FOLDER/FILE EXISTENCE- Test-path ‘path of folder and file whose existence need to check’
- READ FILES- Get-content ‘path of file whose content want to know’
- CLEAR CONTENT- Clear-content ‘path of file whose content want to erase’
- ADD CONTENT- Add-content ‘path of file content need to be added’

Powershell - Date and Time Operations

- GET SYSTEM DATE- Get-Date for date and time and Get-Date -DisplayHint Date for only date
- SET SYSTEM DATE- set-date -Date (Get-Date).AddDays(1){ to add one more day to current date.} and set-date -Date (Get-Date).AddDays(-1) { to substract one day from current date}
- GET SYSTEM TIME- Get-Date -DisplayHint Time
- SET SYSTEM TIME- First input what to be done ($timeToAdd = New-TimeSpan -Minutes -60) and then adjust (set-date -adjust $timeToAdd)

Powershell - Advanced Cmdlets

Sr.No.	Cmdlet Type & Description
1	**Get-Unique Cmdlet** Example : $list | sort | get-unique
2	**Group-Object Cmdlet** Example : Get-ChildItem -File | Group-Object Extension
3	**Measure-Object Cmdlet** Example : get-content D:\temp\test\test.txt | measure-object -character -line -word
4	**Compare-Object Cmdlet** Example : Compare-Object -ReferenceObject $(Get-Content D:\temp\test\test.txt) -DifferenceObject $(Get-Content D:\temp\test\test1.txt)
5	**Format-List Cmdlet** Example : Format-List -InputObject $filename
6	**Format-Wide Cmdlet** Example : Format-Wide -InputObject $filename
7	**Where-Object Cmdlet** Example : Get-Service | Where-Object {$_.Status -eq "Stopped"}
8	**Get-ChildItem Cmdlet** Example : example above same for group object
9	**ForEach-Object Cmdlet** Example : 1000,2000,3000 | ForEach-Object -Process {$_/1000}
10	**Start-Sleep Cmdlet** Example : Start-Sleep -s 15
11	**Read-Host Cmdlet** Example : $choice = Read-Host "Please put your choice"

12	**Select-Object Cmdlet** Example : Get-Process | Select-Object -Property ProcessName, Id, WS -Last 5
13	**Sort-Object Cmdlet** Example : Get-Process | Sort-Object -Property ProcessName, Id, WS -Last
14	**Write-Warning Cmdlet** Example : Write-Warning "Test Warning"
15	**Write-Host Cmdlet** Example : Write-Host (2,4,6,8,10,12) -Separator ", -> " -ForegroundColor DarkGreen -BackgroundColor White
16	**Invoke-Item Cmdlet** Example : Invoke-Item "D:\Temp\test.txt"
17	**Invoke-Expression Cmdlet** Example : > $Command = 'Get-Process' > $Command Get-Process > Invoke-Expression $Command
18	**Measure-Command Cmdlet** Example : Measure-Command { Get-EventLog "Windows PowerShell" }
19	**Invoke-History Cmdlet** Example :Invoke-History Measure-Command { Get-EventLog "Windows PowerShell" }
20	**Add-History Cmdlet** Example : > get-history Id CommandLine -- -----------

	13 clear-history 14 get-history 15 dir 16 dir 17 dir 18 dir
21	**Get-History Cmdlet** Example : Example above
22	**Get-Culture Cmdlet** Example : > get-culture LCID Name DisplayName ---- ---- ----------- 1033 en-US English (United States)

Powershell - Scripting

Windows PowerShell Scripting is a fully developed scripting language and has a rich expression parser/

Features

Cmdlets − Cmdlets perform common system administration tasks, for example managing the registry, services, processes, event logs, and using Windows Management Instrumentation (WMI).

Task oriented − PowerShell scripting language is task based and provide supports for existing scripts and command-line tools.

Consistent design − As cmdlets and system data stores use common syntax and have common naming conventions, data sharing is easy. The output from one cmdlet can be pipelined to another cmdlet without any manipulation.

Simple to Use − Simplified, command-based navigation lets users navigate the registry and other data stores similar to the file system navigation.

Object based − PowerShell possesses powerful object manipulation capabilities. Objects can be sent to other tools or databases directly.

Extensible interface. − PowerShell is customizable as independent software vendors and enterprise developers can build custom tools and utilities using PowerShell to administer their software.

Variables

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

Creating variable

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

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

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

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

Using variable

Type the following command in PowerShell ISE Console.

```
 $location
```

Output

You can see following output in PowerShell console.

```
Path

----

D:\test
```

Getting information of variable

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

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

Output

You can see following output in PowerShell console.

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


Name         MemberType   Definition
```

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

Powershell - Special Variables

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

Operator	Description
$$	Represents the last token in the last line received by the session.
$?	Represents the execution status of the last operation. It contains TRUE if the last operation succeeded and FALSE if it failed.
$^	Represents the first token in the last line received by the session.
$_	Same as $PSItem. Contains the current object in the pipeline object. You can use this variable in commands that perform an action on every object or on selected objects in a pipeline.
$ARGS	Represents an array of the undeclared parameters and/or parameter values that are passed to a function, script, or script block.
$CONSOLEFILENAME	Represents the path of the console file (.psc1) that was most recently used in the session.
$ERROR	Represents an array of error objects that represent the most recent errors.
$EVENT	Represents a PSEventArgs object that represents the event that is being processed.
$EVENTARGS	Represents an object that represents the first event argument that derives from EventArgs of the event that is being processed.
$EVENTSUBSCRIBER	Represents a PSEventSubscriber object that represents the event subscriber of the event that is being processed.
$EXECUTIONCONTEXT	Represents an EngineIntrinsics object that represents the execution context of the PowerShell host.

$FALSE	Represents FALSE. You can use this variable to represent FALSE in commands and scripts instead of using the string "false".
$FOREACH	Represents the enumerator (not the resulting values) of a ForEach loop. You can use the properties and methods of enumerators on the value of the $ForEach variable.
$HOME	Represents the full path of the user's home directory.
$HOST	Represents an object that represents the current host application for PowerShell.
$INPUT	Represents an enumerator that enumerates all input that is passed to a function.
$LASTEXITCODE	Represents the exit code of the last Windows-based program that was run.
$MATCHES	The $Matches variable works with the -match and -notmatch operators.
$MYINVOCATION	$MyInvocation is populated only for scripts, function, and script blocks. PSScriptRoot and PSCommandPath properties of the $MyInvocation automatic variable contain information about the invoker or calling script, not the current script.
$NESTEDPROMPTLEVEL	Represents the current prompt level.
$NULL	$null is an automatic variable that contains a NULL or empty value. You can use this variable to represent an absent or undefined value in commands and scripts.
$PID	Represents the process identifier (PID) of the process that is hosting the current PowerShell session.
$PROFILE	Represents the full path of the PowerShell profile for the current user and the current host application.

$PSCMDLET	Represents an object that represents the cmdlet or advanced function that is being run.
$PSCOMMANDPATH	Represents the full path and file name of the script that is being run.
$PSCULTURE	Represents the name of the culture currently in use in the operating system.
$PSDEBUGCONTEXT	While debugging, this variable contains information about the debugging environment. Otherwise, it contains a NULL value.
$PSHOME	Represents the full path of the installation directory for PowerShell.
$PSITEM	Same as $_. Contains the current object in the pipeline object.
$PSSCRIPTROOT	Represents the directory from which a script is being run.
$PSSENDERINFO	Represents information about the user who started the PSSession, including the user identity and the time zone of the originating computer.
$PSUICULTURE	Represents the name of the user interface (UI) culture that is currently in use in the operating system.
$PSVERSIONTABLE	Represents a read-only hash table that displays details about the version of PowerShell that is running in the current session.
$SENDER	Represents the object that generated this event.
$SHELLID	Represents the identifier of the current shell.
$STACKTRACE	Represents a stack trace for the most recent

	error.
$THIS	In a script block that defines a script property or script method, the $This variable refers to the object that is being extended.
$TRUE	Represents TRUE. You can use this variable to represent TRUE in commands and scripts.

Powershell - Operators

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

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

COMMON USED EXAMPLES OF ABOVE OPERATORS

Operator	Description	Example
#NAME?	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
eq (equals)	Compares two values to be equal or not.	A -eq B will give false
ne (not equals)	Compares two values to be not equal.	A -ne B will give true
gt (greater than)	Compares first value to be greater than second one.	B -gt A will give true
ge (greater than or equals to)	Compares first value to be greater than or equals to second one.	B -ge A will give true
lt (less than)	Compares first value to be less than second one.	B -lt A will give false
le (less than or equals to)	Compares first value to be less than or equals to second one.	B -le A will give false
#ERROR!	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C

#ERROR!	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
AND (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A -AND B) is false
OR (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A -OR B) is true
NOT (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	-NOT(A -AND B) is true
> (Redirectional Opeator)	Redirectional operator. Assigns output to be printed into the redirected file/output device.	dir > test.log will print the directory listing in test.log file

Powershell - Looping

A **loop** statement allows us to execute a statement or group of statements multiple times

1. 'for' Loop

The for loop runs a block of code a specific number of times, typically used when you know the number of iterations in advance.

Example: Create Multiple Azure Resource Groups

```
for ($i = 1; $i -le 5; $i++) {
   $resourceGroupName = "MyResourceGroup$i"
   New-AzResourceGroup -Name $resourceGroupName -Location "EastUS"
}
```

This script creates five resource groups named MyResourceGroup1, MyResourceGroup2, …, MyResourceGroup5 in the "EastUS" region.

2. ‘foreach’ Loop

The foreach loop iterates over each item in a collection (like an array or list).

Example: Start Multiple Azure VMs

```
$vms = @("VM1", "VM2", "VM3")


foreach ($vm in $vms) {
   Start-AzVM -ResourceGroupName "MyResourceGroup" -Name $vm
}
```

This script starts the VMs named VM1, VM2, and VM3 in the resource group MyResourceGroup.

3. ‘while’ Loop

The while loop runs as long as a specified condition is true.

Example: Wait Until an Azure VM is Running

```
$vmName = "MyVM"
$resourceGroupName = "MyResourceGroup"
$vm = Get-AzVM -ResourceGroupName $resourceGroupName -Name $vmName


while ($vm.Statuses[1].Code -ne "PowerState/running") {
   Write-Output "Waiting for VM to start..."
   Start-Sleep -Seconds 10
   $vm = Get-AzVM -ResourceGroupName $resourceGroupName -Name $vmName
}

Write-Output "VM is running!"
```

This script checks the status of a VM and waits until it is running, checking every 10 seconds.

4. ‘do-while’ Loop

The do-while loop runs a block of code at least once and then continues as long as a specified condition is true.

Example: Continuously Check Azure Storage Account Availability

```
$storageAccountName = "mystorageaccount"

$resourceGroupName = "MyResourceGroup"


do {

  $storageAccount = Get-AzStorageAccount -ResourceGroupName $resourceGroupName -Name $storageAccountName

  if ($storageAccount) {

    Write-Output "Storage account exists."

  } else {

    Write-Output "Storage account does not exist. Retrying in 10 seconds..."

    Start-Sleep -Seconds 10

  }

} while (-not $storageAccount)
```

This script checks if a storage account exists and retries every 10 seconds until it finds the storage account.

Combining Loops with Azure Cmdlets

You can combine these loops with various Azure cmdlets to perform complex automation tasks. Here are a few more examples:

Example: List All VMs in All Resource Groups

```
$resourceGroups = Get-AzResourceGroup
```

```
foreach ($rg in $resourceGroups) {
    $vms = Get-AzVM -ResourceGroupName $rg.ResourceGroupName
    foreach ($vm in $vms) {
        Write-Output "VM Name: $($vm.Name) in Resource Group: $($rg.ResourceGroupName)"
    }
}
```

This script lists all VMs in all resource groups in the Azure subscription.

Example: Scale Down All VMs in a Specific Resource Group

```
$resourceGroupName = "MyResourceGroup"
$vms = Get-AzVM -ResourceGroupName $resourceGroupName


foreach ($vm in $vms) {
    Stop-AzVM -ResourceGroupName $resourceGroupName -Name $vm.Name -Force
    Write-Output "Stopped VM: $($vm.Name)"
}
```

This script stops all VMs in the specified resource group.

Powershell - Special Variables

PowerShell has a number of special variables that are automatically populated by the shell. These variables provide access to important system and environment information, control behavior, and manage input/output. Here are some of the most commonly used special variables in PowerShell:

1. $_ and $PSItem

- **Description**: Represents the current object in the pipeline.
- **Example**: Get-Process | Where-Object { $_.CPU -gt 100 }

2. $?

- **Description**: Contains the execution status of the last command. It is $true if the last command succeeded and $false if it failed.
- **Example**:

```
Get-Process notepad
```

```
if ($?) {
   Write-Output "Notepad is running."
} else {
   Write-Output "Notepad is not running."
}
```

3. $LASTEXITCODE

- **Description**: Contains the exit code of the last native application that was run.
- **Example**:

```
ping -n 1 localhost
Write-Output "Exit code: $LASTEXITCODE"
```

4. $PSCmdlet

- **Description**: Provides access to the cmdlet object within the context of an advanced function or cmdlet.
- **Example**:

```
function Get-CallerPreference {
   [CmdletBinding()]
   param ()
   $PSCmdlet.MyInvocation.PSCommandPath
}
```

5. $Error

- **Description**: Contains an array of error objects representing the most recent errors that have occurred.
- **Example**:

```
Get-Process fakeprocess
Write-Output $Error[0]
```

6. $PSCommandPath

- **Description**: Contains the full path and file name of the script that is being run.
- **Example**:Write-Output "This script is running from: $PSCommandPath"

7. $PSScriptRoot

- **Description**: Contains the directory from which the script is being run.
- **Example**:Write-Output "This script is located in: $PSScriptRoot"

8. $MyInvocation

- **Description**: Provides information about the current command, script, or script block.
- **Example**: Write-Output "You are running: $($MyInvocation.MyCommand)"

9. $Home

- **Description**: Contains the full path of the user's home directory.
- **Example**: Write-Output "Your home directory is: $Home"

10. $Profile

- **Description**: Contains the full path to the user's PowerShell profile script.
- **Example:** Write-Output "Your PowerShell profile script is located at: $Profile"

11. $Host

- **Description**: Contains an object that represents the current host application for PowerShell.
- **Example**: Write-Output "PowerShell host: $($Host.Name)"

12. $PID

- **Description**: Contains the process identifier (PID) of the current PowerShell session.
- **Example**: Write-Output "Current PowerShell PID: $PID"

13. $Args

- **Description**: Contains an array of the arguments that were passed to a script, function, or script block.
- **Example**:

```
function Show-Args {

  param($first, $second)

  Write-Output "First: $first, Second: $second"

  Write-Output "All Args: $Args"

}

Show-Args -first "one" -second "two" "three" "four"
```

14. $PSVersionTable

- **Description**: Contains a read-only hash table that displays details about the version of PowerShell that is currently running.

- **Example**: Write-Output $PSVersionTable

Powershell - Conditions

In PowerShell, conditional statements allow you to control the flow of your script based on certain conditions. The primary conditional statements are if, elseif, else, and switch. These can be particularly useful when managing Azure resources to execute different actions based on specific conditions.

1. ‘if’ Statement

The if statement executes a block of code if a specified condition evaluates to $true.

Example: Check if an Azure Resource Group Exists

```
$resourceGroupName = "MyResourceGroup"


if (Get-AzResourceGroup -Name $resourceGroupName -ErrorAction SilentlyContinue) {

   Write-Output "Resource group exists."

} else {

   Write-Output "Resource group does not exist."

}
```

2. ‘elseif and else’ Statements

The elseif statement provides an alternative condition if the previous if condition is $false. The else statement runs a block of code if none of the preceding conditions are $true.

Example: Create a Resource Group if it Doesn’t Exist

```
$resourceGroupName = "MyResourceGroup"

$location = "EastUS"


if (Get-AzResourceGroup -Name $resourceGroupName -ErrorAction SilentlyContinue) {

   Write-Output "Resource group already exists."

} elseif ($location -eq "EastUS") {

   Write-Output "Creating resource group in EastUS."

   New-AzResourceGroup -Name $resourceGroupName -Location $location
} else {

    Write-Output "Resource group does not exist and location is not EastUS."

}
```

3. ‘switch’ Statement

The switch statement evaluates an expression against a set of conditions and executes the corresponding block of code.

Example: Start or Stop an Azure VM Based on User Input

```
$vmName = "MyVM"

$resourceGroupName = "MyResourceGroup"

$action = Read-Host "Enter action (start/stop)"

switch ($action) {

    "start" {

        Start-AzVM -ResourceGroupName $resourceGroupName -Name $vmName

        Write-Output "VM started."

    }

    "stop" {

        Stop-AzVM -ResourceGroupName $resourceGroupName -Name $vmName

        Write-Output "VM stopped."

    }

    default {

        Write-Output "Invalid action. Please enter 'start' or 'stop'."

    }

}
```

Combining Conditions with Azure Cmdlets

You can combine these conditional statements with Azure cmdlets to automate various tasks based on conditions. Here are a few more practical examples:

Example: Check VM Status and Take Action

```
$vmName = "MyVM"

$resourceGroupName = "MyResourceGroup"

$vm = Get-AzVM -ResourceGroupName $resourceGroupName -Name $vmName -Status

if ($vm.Statuses[1].Code -eq "PowerState/running") {

   Write-Output "VM is already running."

} else {

   Write-Output "VM is not running. Starting VM..."

   Start-AzVM -ResourceGroupName $resourceGroupName -Name $vmName

}
```

Example: Scale Up/Down Azure SQL Database Based on Usage

```
$resourceGroupName = "MyResourceGroup"

$sqlServerName = "MySqlServer"

$databaseName = "MyDatabase"

$usageThreshold = 80

$currentUsage = (Get-AzSqlDatabaseUsage -ResourceGroupName $resourceGroupName -
ServerName $sqlServerName -DatabaseName $databaseName).CurrentValue


if ($currentUsage -gt $usageThreshold) {

   Write-Output "Usage is above threshold. Scaling up..."

   Set-AzSqlDatabase -ResourceGroupName $resourceGroupName -ServerName
$sqlServerName -DatabaseName $databaseName -RequestedServiceObjectiveName "S3"

} else {

   Write-Output "Usage is below threshold. Scaling down..."

   Set-AzSqlDatabase -ResourceGroupName $resourceGroupName -ServerName
$sqlServerName -DatabaseName $databaseName -RequestedServiceObjectiveName "S1"

}
```

COMMON POWERSHELL SCRIPTS EXAMPLES

1. **Creating a Resource Group**

```
# Variables
$resourceGroupName = "MyResourceGroup"
$location = "EastUS"


# Create a Resource Group
New-AzResourceGroup -Name $resourceGroupName -Location $location
```

2. **Creating a Virtual Machine**

```
# Variables
$resourceGroupName = "MyResourceGroup"
$location = "EastUS"
$vmName = "MyVM"
$vmSize = "Standard_DS1_v2"
$adminUsername = "azureuser"
$adminPassword = ConvertTo-SecureString "P@ssw0rd!" -AsPlainText -Force


# Create a Virtual Machine
$vmConfig = New-AzVMConfig -VMName $vmName -VMSize $vmSize | `
   Set-AzVMOperatingSystem -Windows -ComputerName $vmName -Credential (New-Object PSCredential ($adminUsername, $adminPassword)) | `
   Set-AzVMSourceImage -PublisherName "MicrosoftWindowsServer" -Offer "WindowsServer" -Skus "2019-Datacenter" -Version "latest" | `
   Add-AzVMNetworkInterface -Id (New-AzNetworkInterface -Name "$vmName-NIC" -ResourceGroupName $resourceGroupName -Location $location -SubnetId (New-AzVirtualNetworkSubnetConfig -Name "default" -AddressPrefix "10.0.0.0/24").Id -SecurityGroupId (New-AzNetworkSecurityGroup -ResourceGroupName $resourceGroupName -Location $location -Name "$vmName-NSG").Id).Id


New-AzVM -ResourceGroupName $resourceGroupName -Location $location -VM $vmConfig
```

3. **Starting and Stopping a Virtual Machine**

```
# Variables

$resourceGroupName = "MyResourceGroup"

$vmName = "MyVM"


# Start a VM

Start-AzVM -ResourceGroupName $resourceGroupName -Name $vmName


# Stop a VM

Stop-AzVM -ResourceGroupName $resourceGroupName -Name $vmName -Force
```

4. **Creating an Azure Storage Account**

```
# Variables

$resourceGroupName = "MyResourceGroup"

$storageAccountName = "mystorageaccount"

$location = "EastUS"


# Create a Storage Account

New-AzStorageAccount -ResourceGroupName $resourceGroupName -Name
$storageAccountName -SkuName "Standard_LRS" -Location $location
```

5. **Uploading a File to Azure Blob Storage**

```
# Variables

$resourceGroupName = "MyResourceGroup"

$storageAccountName = "mystorageaccount"

$containerName = "mycontainer"

$filePath = "C:\path\to\file.txt"

$blobName = "file.txt"
```

```
# Get the Storage Account Key

$storageAccountKey = (Get-AzStorageAccountKey -ResourceGroupName $resourceGroupName -
Name $storageAccountName).Value[0]


# Create a Storage Context

$context = New-AzStorageContext -StorageAccountName $storageAccountName -
StorageAccountKey $storageAccountKey


# Create a Blob Container

New-AzStorageContainer -Name $containerName -Context $context


# Upload the File

Set-AzStorageBlobContent -File $filePath -Container $containerName -Blob $blobName -Context
$context
```

6. **Listing All VMs in a Subscription**

```
# List all VMs

Get-AzVM | Select-Object ResourceGroupName, Name, Location, VmSize, ProvisioningState
```

7. **Resizing a Virtual Machine**

```
# Variables

$resourceGroupName = "MyResourceGroup"

$vmName = "MyVM"

$newVmSize = "Standard_DS2_v2"


# Resize the VM

$vm = Get-AzVM -ResourceGroupName $resourceGroupName -Name $vmName

$vm.HardwareProfile.VmSize = $newVmSize

Update-AzVM -ResourceGroupName $resourceGroupName -VM $vm
```

8. **Creating an Azure SQL Database**

```
# Variables

$resourceGroupName = "MyResourceGroup"

$serverName = "myserver"

$location = "EastUS"

$adminUsername = "sqladmin"

$adminPassword = ConvertTo-SecureString "P@ssw0rd!" -AsPlainText -Force

$databaseName = "mydatabase"


# Create a SQL Server

$server = New-AzSqlServer -ResourceGroupName $resourceGroupName -ServerName $serverName -Location $location -SqlAdministratorCredentials (New-Object PSCredential ($adminUsername, $adminPassword))


# Create a SQL Database

New-AzSqlDatabase -ResourceGroupName $resourceGroupName -ServerName $serverName -DatabaseName $databaseName -RequestedServiceObjectiveName "S0"
```

9. **Deleting a Resource Group**

```
# Variables

$resourceGroupName = "MyResourceGroup"


# Delete the Resource Group

Remove-AzResourceGroup -Name $resourceGroupName -Force
```

10. **Assigning a Role to a User**

```
# Variables

$resourceGroupName = "MyResourceGroup"

$roleName = "Contributor"

$userPrincipalName = "user@domain.com"
```

```
# Assign the Role

New-AzRoleAssignment -ResourceGroupName $resourceGroupName -RoleDefinitionName
$roleName -UserPrin
```

THANKS