

PowerShell Cheat Sheet

> hackr.io



PowerShell Basics

PowerShell is a cross-platform and a commonly used task automation solution combining the power of the command-line shell, a scripting language, and a configuration management framework. You can run PowerShell on Windows, Linux, and macOS operating systems. Unlike other available shells that only accept and return data, PowerShell accepts and returns .NET objects.

This shell comes with the following features.

- Robust **command-line history**
- Extensive **features**, like tab completion and command prediction
- **Command and parameter aliases**
- Pipeline for **changing commands**
- **In-console help system**, like Unix man pages

Considering the features of any scripting language, you can use PowerShell to automate the system management process and task. It lets you build, test, and deploy solutions in CI/CD environments.

PowerShell is built on the .NET Common Language Runtime (CLR), which means all the inputs and outputs are .NET objects. You do not have to parse text output to retrieve information from it.

The PowerShell scripting language comes with the following features.

- **Functions, classes, objects, and modules**
- **Easy formatting system** for providing clear output for enhanced readability
- **Built-in support** for different data formats, such as CSV, JSON, and XML
- **Type system** for creating dynamic types

Now, let's get into some useful PowerShell commands.

PowerShell Commands Cheat Sheet

cmdlets

Cmdlets are PowerShell's internal commands. These cmdlets will return one or more objects to the pipeline where at the end of that pipeline, we mention some properties of the objects in the following table to see their values displayed on the screen.

Command	Description
Get-Help	This command allows you to get support with PowerShell.
Get-PSdrive	This command offers you a list of available PSDrives, such as c, env, hklm, hkcu, alias, etc.
Get-ChildItem	In any registry, children are the subkeys of the current key. To get the required details, you can use the following command.
Get-ChildItem -recurse	Run this command to list all the children recursively of the current PSdrive, folder, or registry key.

Get-ChildItem -recursive	Use this command To include the hidden folders (directories).
(Get-ChildItem).name or Get-ChildItem -name	Run any of these commands to get the list file and directory names in the current folder.
(Get-ChildItem).count	Use this command to get the number of entries in the collection of objects returned by the Get-Children.

PSdrives

PSdrives are the collection of entities grouped together so they can be accessed as a filesystem drive. The “PSprovider” does this grouping.

By default, a PS session can access several PSdrives including c:, env:, alias:, and HKLM:, where c: refers to the usual Windows c-drive; env: is the space of Windows environmental variables; alias: is the collection of cmdlet aliases; and HKLM is a hive in the registry.

Any PS session will enter the user’s home folder. If you want to switch from a PS session to another PSdrive and retrieve the information from that drive, consider the following commands:

Commands	Description
Switching to env-	The prompt character will change to the “ENV:\>”. Set-Location env by running the following command: Set-Location env-
Env:\> Get-Childitem	This command will get you all the environment variables.

Env:\> Get-Childitem userprofile	Use this command to get the environment variables of “userprofile.”
Env:\> Set-Location alias:	Run the following command to change the prompt character to “Alias.”
Alias:\> Get-Childitem	Run this command to get all the children of all aliases.
Alias:\> Set-Location C:\	Use this command to get the “C:/>” prompt again, back to the default drive.
C:\Users\user_name>\$alias:ls	Run this command to find what alias “ls” stands for.

Pipelines

Cmdlets uses the pipelines to pass the objects but not the character streams like Unix. The pipeline character is | (ASCII 124), followed by a command that handles the output passed through the pipeline. The pipeline consists of the following three stages.

*Get-ChildItem *.txt | Where-Object length -lt 1000 | Sort-Object length*

The following table highlights some of the basic pipeline commands:

Command	Description
(Get-Item /Users/paashibansal/Desktop).lastwritetime.year	Easily sets the value of the ‘lastwritetime.year’ property to the present date and time without affecting the file’s content.
(Get-ChildItem data.txt.rtf -name).name # -> null	Provides an empty result
"data.txt.rtf" Rename-Item -NewName "data_new.txt.rtf"	Changes the old file names and file extensions to the new ones
Get-ChildItem data.txt Rename-Item -new {\$_.name}	A trivial renaming command that invokes an automatic variable

Get-ChildItem data.txt.rtf -name Rename-Item -new {\$_.name}	If the piped object \$_ doesn't have the member property (name), you will receive an error, as parameter \$_.name is null
Get-ChildItem Select-Object basename Sort-Object *	Displays the list of the names of all the files that are present in the current folder sorted in alphabetical order.
Move-Item *.txt subdirectory	Moves all the files to the folder subdirectory
Get-ChildItem *.txt Move-Item ..\	Gives the error message that Move-Item lacks input

Alias

Cmdlets come with several aliases. The following table highlights a few of aliases, along with their descriptions:

Command	Description
Add-Content	Appends value to a file
Get-Content	Finds file content in an array
Set-Location	Changes folder, key, or PS drive
Clear-Host	Clears console
Remove-Item	Delete files
Get-ChildItem -Path .\	Lists Folder, Key, or PSDrive Children
Write-Output	Sends the array to the console, pipeline, or redirect it to the file
Foreach-Object	Traverses each object in a pipeline
Format-Table	Formats the table with selected properties for each object in each column
Format-List	Formats the process properties by name
Get-Alias	Provides Cmdlet Alias
Get-Command	Provides you with commands from the current session only
Get-Member	Retrieves all the object members

Get-ItemProperty .data.txt Format-List	Provides the specified item's properties
Get-ItemPropertyValue -Path '.data.txt' -Name LastWriteTime	Gives the current value for a specified property while using the name parameter
Get-Variable m*	Finds session variable names and sessions
New-Item -Path .\ -Name "testfile1.txt" -ItemType "file" -Value "This is a text string."	Creates a new file, directory, symbolic link, registry key, or registry entry
Get-Process	Gives the entire list of all the running processes
Get-Location	Provides the current directory's or registry key's location
Rename-Item -Path "old_name" -NewName "new_name"	Renames the old item name with the new name
Remove-Item .testfile1.txt	Removes the specified directory, files, or registry keys
Remove-Variable	Removes the specified variable
Start-Sleep	Suspends an activity for a specified period of time

Operators

- **Arithmetic Operators**

Operator	Description	Example
+	Adds integers; concatenates strings, arrays, and hash tables.	6 + 2 "file" + "name" @(1, "one") + @(2.0, "two") @{"one" = 1} + @{"two" = 2}
+	Makes a number out of an object	123
-	Subtracts one value from another	6 - 2
-	Calculates the opposite number	- -6
		(Get-Date).AddDays(-1)
*	Multiply numbers or copy strings	6 * 2

	and arrays for a specified number of times	
		@("!") * 4
		!" * 3
/	Divides two values	6 / 2
%	Modulus - returns the remainder of a division operation	7 % 2
-band	Bitwise AND	5 -band 3
-bnot	Bitwise NOT	-bnot 5
-bor	Bitwise OR	5 -bor 0x03
-bxor	Bitwise XOR	5 -bxor 3
-shl	Shifts bits to the left	102 -shl 2
-shr	Shifts bits to the right	102 -shr 2

Operator Precedence

Precedence	Operator	Description
1	()	Parentheses
2	-	For a negative number or unary operator
3	*, /, %	For multiplication and division
4	+,-	For addition and subtraction
5	-band, -bnot, -bor, -bxor, -shr, and -shl	For bitwise operations

● Assignment Operators

Operator	Description
=	Sets a variable's value to the specified value

+=	Increases a variable's value by the specified value or appends the specified value to the existing value
-=	Decreases a variable's value by a specified value
*=	Multiplies a variable's value by a specified value, or appends the specified value to the existing value
/=	Divides a variable's value by a specified value
%=	Divides the variable's value by a specified value and then assigns the remainder (modulus) to the variable.
#ERROR !	Increases a variable's value, assignable property, or array element by 1.
--	Decreases the variable's value, assignable property, or array element by 1.

- **Comparison Operators**

Type	Operator	Comparison test
Equality	-eq	equals
	-ne	not equals
	-gt	greater than
	-ge	greater than or equal
	-lt	less than
	-le	less than or equal
Matching	-like	string matches wildcard pattern
	-notlike	string doesn't match wildcard pattern
	-match	string matches regex pattern
	-notmatch	string doesn't match regex pattern
Replacement	-replace	replaces strings matching a regex pattern
Containment	-contains	collection contains a value
	-notcontains	collection doesn't contain a value

	-in	value is in a collection
	-notin	value is not in a collection
Type	-is	both objects are the same type
	-isnot	objects are not the same type

- **Logical Operators**

Operator	Description	Example
-and	Logical AND. TRUE when both statements are true.	(1 -eq 1) -and (1 -eq 2) FALSE
-or	Logical OR. TRUE when either of the statements is TRUE.	(1 -eq 1) -or (1 -eq 2) TRUE
-xor	Logical EXCLUSIVE OR. TRUE when only one statement is TRUE.	(1 -eq 1) -xor (2 -eq 2) FALSE
-not	Logical not. Negates the statement that follows.	-not (1 -eq 1) FALSE
!	Same as -not	!(1 -eq 1) FALSE

- **Redirection Operator**

Operator	Description	Syntax
>	Send specified stream to a file	n>
>>	Append specified stream to a file	n>>
>&1	Redirects the specified stream to the Success stream	n>&1

- **Type Operators**

Operator	Description	Example
----------	-------------	---------

-isNot	Returns TRUE when the input not an instance of the specified.NET type.	(get-date) -isNot [DateTime] FALSE
-as	Converts the input to the specified .NET type.	"5/7/07" -as [DateTime] Monday, May 7, 2007 00:00:00

Some Other Operators

Operator	Description
() Grouping Operator	Allows you to override operator precedence in expressions
&() Subexpression Operator	Gives you the result of one or more statements
@() Array Subexpression Operator	Returns the results of one or more statements in the form of arrays.
& Background Operator	The pipeline before & is executed by this command in a Powershell job.
[] Cast Operator	Converts objects to the specific type.

Regular Expressions

A regular expression is a pattern that is used to match text that includes literal characters, operators, and other constructs. PowerShell regular expressions are case-insensitive by default.

Method	Case Sensitivity
Select-String	use -CaseSensitive switch
switch statement	use the -casesensitive option
operators	prefix with 'c' (-cmatch, -csplit, or -creplace)

- **Character Literals**

A regular expression can be a literal character or a string.

```
PS /Users/paashibansal> # This statement returns true because book contains the string "oo"
PS /Users/paashibansal> 'book' -match 'oo'
True
PS /Users/paashibansal> █
```

- **Character Groups**

These allow you to match any number of characters one time, while `[^character group]` only matches characters NOT in the group.

```
PS /Users/paashibansal> # This expression returns true if the pattern matches big, bog, or bug.
PS /Users/paashibansal> 'big' -match 'b[iou]g'
True
PS /Users/paashibansal> █
```

- **Character Range**

A pattern can also be a range of characters. The characters can be alphabetic `[A-Z]`, numeric `[0-9]`, or even ASCII-based `[-~]` (all printable characters).

```
PS /Users/paashibansal> # This expression returns true if the pattern matches any 2 digit number.
PS /Users/paashibansal> 42 -match '[0-9][0-9]'
True
PS /Users/paashibansal> █
```

- **Numbers**

The `\d` character class will match any decimal digit. Conversely, `\D` will match any non-decimal digit.

```
PS /Users/paashibansal> # This expression returns true if it matches a server name.
PS /Users/paashibansal> # (Server-01 - Server-99).
PS /Users/paashibansal> 'Server-01' -match 'Server-\d\d'
True
PS /Users/paashibansal> █
```

- **Word Character**

The `\w` character class will match any word character `[a-zA-Z_0-9]`. To match any non-word character, use `\W`.

```

true
PS /Users/paashibansal> # This expression returns true.
PS /Users/paashibansal> # The pattern matches the first word character 'B'.
PS /Users/paashibansal> 'Book' -match '\w'
True
PS /Users/paashibansal>

```

- **Wildcard**

The period (.) is a wildcard character in regular expressions. It will match any character except a newline (\n).

```

PS /Users/paashibansal> # This expression returns true.
PS /Users/paashibansal> # The pattern matches any 4 characters except the newline.
PS /Users/paashibansal> 'a1\ ' -match '....'
True
PS /Users/paashibansal>

```

- **Whitespace**

Whitespace is matched using the \s character class. Any non-whitespace character is matched using \S. Literal space characters ' ' can also be used.

```

true
PS /Users/paashibansal> # This expression returns true.
PS /Users/paashibansal> # The pattern uses both methods to match the space.
PS /Users/paashibansal> ' - ' -match '\s- '
True
PS /Users/paashibansal>

```

- **Escaping Characters**

The backslash (\) is used to escape characters so the regular expression engine doesn't parse them.

The following characters are reserved: [()\.^\$|?*+{}].

```

true
PS /Users/paashibansal> # This returns true and matches numbers with at least 2 digits of precision.
PS /Users/paashibansal> # The decimal point is escaped using the backslash.
PS /Users/paashibansal> '3.141' -match '3\.\d{2,}'
True
PS /Users/paashibansal>

```

- **Substitution in Regular Expression.**

The regular expressions with the -replace operator allows you to dynamically replace text using captured text.

<input> -replace <original>, <substitute>

```
PS /Users/praashibansal> 'John D. Smith' -replace '(\w+) (\w+)\. (\w+)', '$1.$2.$3@contoso.com'
John.D.Smith@contoso.com
PS /Users/praashibansal>
```

Flow Control

- **ForEach-Object**

ForEach-Object is a cmdlet that allows you to iterate through items in a pipeline, such as with PowerShell one-liners. ForEach-Object will stream the objects through the pipeline.

Although the Module parameter of Get-Command accepts multiple values that are strings, it will only accept them via pipeline input using the property name, or parameter input.

If you want to pipe two strings by value to Get-Command for use with the Module parameter, use the ForEach-Object cmdlet:

```
$ComputerName = 'DC01', 'WEB01'
foreach ($Computer in $ComputerName) {
    Get-ADComputer -Identity $Computer
}
```

- **For**

A “for” loop iterates while a specified condition is true.

For example:

```
for ($i = 1; $i -lt 5; $i++) {
    Write-Output "Sleeping for $i seconds"
    Start-Sleep -Seconds $i
}
```

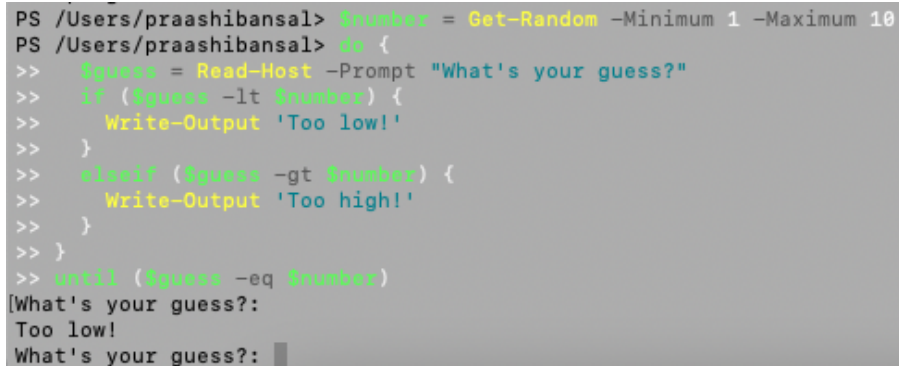
```
PS /Users/praashibansal> for ($i = 1; $i -lt 5; $i++) {
>> Write-Output "Sleeping for $i seconds"
>> Start-Sleep -Seconds $i
>> }
Sleeping for 1 seconds
Sleeping for 2 seconds
Sleeping for 3 seconds
Sleeping for 4 seconds
PS /Users/praashibansal>
```

- **Do**

There are two different “do” loops in PowerShell. Do *Until* runs while the specified condition is false.

Example 1:

```
$number = Get-Random -Minimum 1 -Maximum 10
do {
    $guess = Read-Host -Prompt "What's your guess?"
    if ($guess -lt $number) {
        Write-Output 'Too low!'
    }
    elseif ($guess -gt $number) {
        Write-Output 'Too high!'
    }
}
until ($guess -eq $number)
```



```
PS /Users/paashibansal> $number = Get-Random -Minimum 1 -Maximum 10
PS /Users/paashibansal> do {
>>     $guess = Read-Host -Prompt "What's your guess?"
>>     if ($guess -lt $number) {
>>         Write-Output 'Too low!'
>>     }
>>     elseif ($guess -gt $number) {
>>         Write-Output 'Too high!'
>>     }
>> }
>> until ($guess -eq $number)
What's your guess?:
Too low!
What's your guess?:
```

Example 2:

```
$number = Get-Random -Minimum 1 -Maximum 10
do {
    $guess = Read-Host -Prompt "What's your guess?"
    if ($guess -lt $number) {
        Write-Output 'Too low!'
    } elseif ($guess -gt $number) {
        Write-Output 'Too high!'
    }
}
while ($guess -ne $number)
```

```

PS /Users/paashibansal> $number = Get-Random -Minimum 1 -Maximum 10
PS /Users/paashibansal> do {
>> $guess = Read-Host -Prompt "What's your guess?"
>> if ($guess -lt $number) {
>>     Write-Output 'Too low!'
>> } elseif ($guess -gt $number) {
>>     Write-Output 'Too high!'
>> }
>> }
>> while ($guess -ne $number)
What's your guess?:
Too low!
What's your guess?: 4
Too low!
What's your guess?:

```

- **While**

Similar to the Do While loop, a While loop runs as long as the specified condition is true. The difference however, is that a While loop evaluates the condition at the top of the loop before any code is run. So, it doesn't run if the condition evaluates to false.

For example:

```

$date = Get-Date -Date 'November 22'
while ($date.DayOfWeek -ne 'Thursday') {
    $date = $date.AddDays(1)
}
Write-Output $date

```

```

PS /Users/paashibansal> $date = Get-Date -Date 'November 22'
PS /Users/paashibansal> while ($date.DayOfWeek -ne 'Thursday') {
>>     $date = $date.AddDays(1)
>> }
[PS /Users/paashibansal> Write-Output $date

Thursday, 24 November 2022 00:00:00
PS /Users/paashibansal>

```

Variables

PowerShell allows you to store all types of values. For example, it can store command results and command expression elements like names, paths, and settings. Here are some of PowerShell's different variables.

- **User-created variables:** These are created and maintained by the user. The variables you create at the PowerShell command line will only exist until the PowerShell window is open. When you close the PowerShell window, these variables are deleted. If you want to save a variable, you need to add it to your PowerShell profile. You can create variables and declare them with three

different scopes: global, script, or local.

- **Automatic variables:** These variables store the state of PowerShell and are created by PowerShell. Only PowerShell can change their values as required to maintain accuracy. Users can't change these variables' value. For example, the \$PSHOME variable will store the path to the PowerShell installation directory.
- **Preference variables:** These variables store user preferences for PowerShell and are created by PowerShell. These variables are populated with default values and can be changed by the users. For example, the \$MaximumHistoryCount variable specifies the maximum number of entries in the session history.

To create a new variable, you need to use an assignment statement and assign a value to the variable. There is no need to declare the variable before using it. The default value of all variables is \$null.

For example-

```
$MyVariable = 1, 2, 3
```

```
$MyVariable
```

```
PS /Users/praashibansal> $MyVariable = 1, 2, 3
PS /Users/praashibansal> $MyVariable
1
2
3
PS /Users/praashibansal>
```

Function

- **Naming Your Function**

Use a Pascal case name with an approved verb and a singular noun to name a function. You can get a list of approved verbs by running Get-Verb:

Get-Verb | Sort-Object -Property Verb


```
PS /Users/praashibansal> Get-Verb | Sort-Object -Property Verb
```

Verb	AliasPrefix	Group	Description
Add	a	Common	Adds a resource to a container, or attaches an item to another item
Approve	ap	Lifecycle	Confirms or agrees to the status of a resource or process
Assert	as	Lifecycle	Affirms the state of a resource
Backup	ba	Data	Stores data by replicating it
Block	bl	Security	Restricts access to a resource
Build	bd	Lifecycle	Creates an artifact (usually a binary or document) out of some set of input files (usually source code or declarative documents)
Checkpoint	ch	Data	Creates a snapshot of the current state of the data or of its configuration
Clear	cl	Common	Removes all the resources from a container but does not delete the container
Close	cs	Common	Changes the state of a resource to make it inaccessible, unavailable, or unusable
Compare	cr	Data	Evaluates the data from one resource against the data from another resource
Complete	cmp	Lifecycle	Concludes an operation
Compress	cm	Data	Compacts the data of a resource
Confirm	cn	Lifecycle	Acknowledges, verifies, or validates the state of a resource or process
Connect	cc	Communications	Creates a link between a source and a destination
Convert	cv	Data	Changes the data from one representation to another when the cmdlet supports bidirectional conversion or when the cmdlet supports conversion
ConvertFrom	cf	Data	Converts one primary type of input (the cmdlet noun indicates the input) to one or more supported output types
ConvertTo	ct	Data	Converts from one or more types of input to a primary output type (the cmdlet noun indicates the output type)
Copy	cp	Common	Copies a resource to another name or to another container
Debug	db	Diagnostic	Examines a resource to diagnose operational problems
Deny	dn	Lifecycle	Refuses, objects, blocks, or opposes the state of a resource or process
Deploy	dp	Lifecycle	Sends an application, website, or solution to a remote target(s) in such a way that a consumer of that solution can access it after deployment
Disable	d	Lifecycle	Configures a resource to an unavailable or inactive state
Disconnect	dc	Communications	Breaks the link between a source and a destination
Dismount	dm	Data	Detaches a named entity from a location
Edit	ed	Data	Modifies existing data by adding or removing content

- **Creating a Simple Function**

Use a function keyword followed by the function name to create a simple function. Then, use an open and closing curly brace. The function will execute code contained within those curly braces.

For example:

```
function Get-Version {
    $PSVersionTable.PSVersion
}
```

```
PS /Users/praashibansal> function Get-Version {
>>     $PSVersionTable.PSVersion
>> }
PS /Users/praashibansal> Get-Version
```

Major	Minor	Patch	PreReleaseLabel	BuildLabel
7	0	10		

```
PS /Users/praashibansal> █
```

Working with Modules

A module is a package containing PowerShell members, such as cmdlets, providers, functions, workflows, variables, and aliases. You can implement package members in a PowerShell script, a compiled DLL, or both. PowerShell automatically imports the modules the first time you run any command in an installed module. You can use the commands in a module without setting up or profile configuration.

- **How to Use a Module**

To use any module, you need to first install them. Then, find the command that comes with the module and use them.

- **Installing a Module**

If you get a module as a folder, install it before you use it on the PowerShell command line. Some modules are preinstalled. You can use the following command to create a Modules directory for the current user:

New-Item -Type Directory -Path \$HOME\Documents\PowerShell\Modules

```
PS /Users/praashibansal> New-Item -Type Directory -Path $HOME\Documents\PowerShell\Modules

Directory: /Users/praashibansal/Documents/PowerShell

Mode                LastWriteTime         Length Name
----                -
d-----           05/17/2022    19:49             Modules

PS /Users/praashibansal>
```

Copy the entire module folder into the Modules directory. You can use any method to copy the folder, including Windows Explorer, Cmd.exe, and PowerShell.

- **Finding the Installed Modules**

Run the following to find the modules installed in a default module location (not imported).

Get-Module -ListAvailable

```
PS /Users/praashibansal> Get-Module -ListAvailable

Directory: /usr/local/microsoft/powershell/7/Modules

ModuleType Version      PreRelease Name                               PSEdition ExportedCommands
-----
Manifest 1.2.5             Microsoft.PowerShell.Archive         Desk      {Compress-Archive, Expand-Archive}
Manifest 7.0.0.0           Microsoft.PowerShell.Host            Core      {Start-Transcript, Stop-Transcript}
Manifest 7.0.0.0           Microsoft.PowerShell.Management      Core      {Add-Content, Clear-Content, Clear-ItemProperty, Join-Path..}
Manifest 7.0.0.0           Microsoft.PowerShell.Security        Core      {Get-Credential, Get-ExecutionPolicy, Set-ExecutionPolicy, ConvertFrom-SecureString..}
Manifest 7.0.0.0           Microsoft.PowerShell.Utility         Core      {Export-Alias, Get-Alias, Import-Alias, New-Alias..}
Script 1.4.7             PackageManagement                    Desk      {Find-Package, Get-Package, Get-PackageProvider, Get-PackageSource..}
Script 2.2.5             PowerShellGet                         Desk      {Find-Command, Find-DscResource, Find-Module, Find-RoleCapability..}
Script 2.0.5             PSDesiredStateConfiguration          Core      {Configuration, New-DscChecksum, Get-DscResource, Invoke-DscResource}
Script 2.0.4             PSReadLine                           Desk      {Get-PSReadLineKeyHandler, Set-PSReadLineKeyHandler, Remove-PSReadLineKeyHandler, Get-PSReadLineOption..}
Binary 2.0.3             ThreadJob                             Desk      Start-ThreadJob

PS /Users/praashibansal>
```

- **Finding Commands in a Module**

Run the following command to find a module's commands:

Get-Command -Module <module-name>

Get-Command -Module Microsoft.PowerShell.Archive

```
PS /Users/praashibansal> Get-Command -Module Microsoft.PowerShell.Archive

CommandType Name                Version      Source
-----
Function    Compress-Archive    1.2.5        Microsoft.PowerShell.Archive
Function    Expand-Archive      1.2.5        Microsoft.PowerShell.Archive

PS /Users/praashibansal>
```

- **Importing a Module**

Run the following command with the proper module name:

Import-Module <module-name>

- **Removing a Module Name**

You can run the following command with the proper module name:

Remove-Module <module-name>

- **View Default Module Locations**

Use the following command to view default module locations:

\$Env:PSModulePath

- **Add a Default Module Location**

You can use the following command format:

\$Env:PSModulePath = \$Env:PSModulePath + ";<path>"

- **Add a Default Module Location on Linux or MacOS**

Use the following to execute the same command as above, only with Linux or macOS:

\$Env:PSModulePath += ";<path>"

Hash Tables

A hash table is a complex data structure to store data in the form of key-value pairs. We also refer to a hash table as a dictionary or associative array. To understand a hash table, consider a series of IP addresses and the respective computer names. A hash stores this data in the form of key-value pairs, where IP addresses refer to keys and computer names to their corresponding values.

The hash table syntax is as follows:

@{ <name> = <value>; [<name> = <value>] ...}

An ordered dictionary's syntax is as follows:

[ordered]@{ <name> = <value>; [<name> = <value>] ...}

- **Creating Hash Tables**

If you want to create a hash table, follow these steps:

- Start the hash table with an at sign (@) and enclose it in curly braces ({}).
- A hash table should contain at least one key-value pair, and hence, enter the data after creating a hash table.
- Separate key from its value using an equal sign (=).
- Separate the key/value pairs in a hash table with a semicolon (;).
- Enclose the space between the keys in quotation marks. Values must be valid PowerShell expressions. Also, enclose strings in quotation marks, even if there are no spaces between them.
- Save a hash table as a variable to manage it efficiently.
- When assigning an ordered hash table to a variable, place the [ordered] attribute before the @ symbol. If you place it before the variable name, the command fails.

For example:

```
$hash = @{ }
```

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

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

```
PS /Users/praashibansal> $hash = @{ Number = 1; Shape = "Square"; Color = "Blue" }  
PS /Users/praashibansal> [hashtable]$hash = [ordered]@{  
>>    Number = 1; Shape = "Square"; Color = "Blue"  
PS /Users/praashibansal> $hash  
  
Name                Value  
----                -  
Shape                Square  
Color                Blue  
Number               1  
  
PS /Users/praashibansal> 
```

- **Adding and Removing Keys and Values**

To add keys and values to a hash table, use the following command format:

\$hash["<key>"] = "<value>"

For example, you can add a "Time" key with a value of "Now" to the hash table with the following statement format:

```
$hash["Time"] = "Now"
```

Or

```
$hash.Add("Time", "Now")
```

Or, you can remove the key with this statement format:

```
$hash.Remove("Time")
```

Asynchronous Event Handling

These cmdlets allow you to register and unregister event subscriptions and list the existing subscriptions. You can also list pending events and handle or remove them as desired.

PowerShell eventing cmdlets

Eventing Cmdlet name	Description
Register-ObjectEvent	This cmdlet registers an event subscription for events generated by .NET objects
Register-WmiEvent	Registers an event subscription for events generated by WMI objects
Register-EngineEvent	Registers an event subscription for events generated by PowerShell itself
Get-EventSubscriber	Gets a list of the registered event subscriptions in the session
Unregister-Event	Removes one or more of the registered event subscriptions
Wait-Event	Waits for an event to occur. This cmdlet can wait for a specific event or any event. It also allows a timeout to be specified, limiting how long it will wait for the event. The default is to wait forever.
Get-Event	Gets pending unhandled events from the event queue
Remove-Event	Removes a pending event from the event queue

New-Event	This cmdlet is called in a script to allow the script to add its own events to the event queue
-----------	--