

MANUAL

POWERSHELL

1. ¿Qué es PowerShell?	3
2. ¿Cómo ejecutar PowerShell?	3
3. Políticas de seguridad en la ejecución de scripts.....	5
4. Cómo escribir y ejecutar un script.....	6
4.1. Escribir un script en un editor de texto y ejecutarlo en la consola.....	6
4.2. Escribir y ejecutar un script en PowerShell ISE	6
5. Mi primer script	7
6. Estructura de los cmdlets	8
7. Comandos básicos	9
8. Alias	10
9. Ayuda sobre cmdlets.....	11
10. Comentarios	11
11. Variables	12
11.1. Variables predefinidas	12
11.2. Variables definidas por el usuario	14
11.3. Reglas para nombrar variables y asignarles valor	15
11.4. Tipo de asignación de datos a una variable.....	15
12. Comandos Write-Host y Read-Host.....	15
12.1. Write-Host	15
12.2. Read-Host	16
13. Comillas.....	17
14. Operadores.....	17
14.1. Operadores de cadenas de caracteres	17
14.2. Operadores aritméticos	18
14.3. Operadores de comparación	19
14.4. Operadores lógicos.....	19
15. Redirección de salida.....	20
16. Tuberías.....	20
17. Estructuras de control de flujo.....	21
17.1. if.....	21
17.2. switch	22
17.3. while.....	23
17.4. do while	24
17.5. do until	24
17.6. for.....	24

17.7. foreach	25
18. Funciones	25
19. Manipulación de objetos	26
20. Documentación oficial de PowerShell	27

1. ¿Qué es PowerShell?

En 2006 Microsoft lanzó su nueva **interfaz de consola** denominada **PowerShell** con un nuevo grupo de órdenes y la capacidad de crear scripts usando una sintaxis más moderna. Se trata de un intérprete de comandos orientado a objetos. Los comandos incluidos en PowerShell reciben el nombre de cmdlets.

La programación de scripts con PowerShell es muy útil para resolver tareas repetitivas, típicas de los administradores de sistemas, como configuraciones específicas, tareas de automatización, creación de usuarios, etc. que desde un entorno gráfico se tardaría más tiempo en realizar.

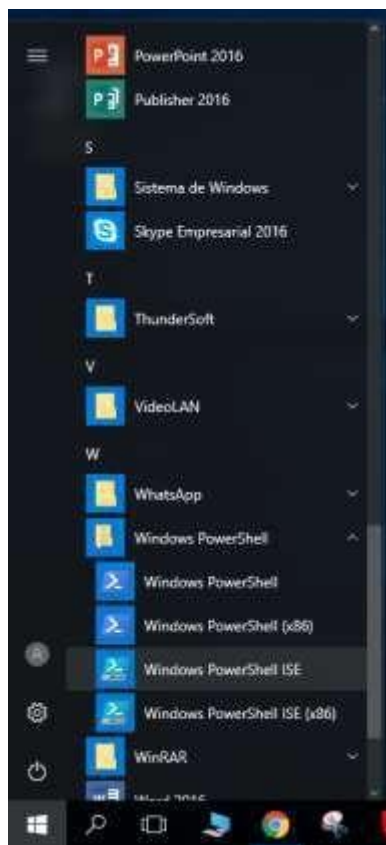
Actualmente PowerShell viene incluido en todos los sistemas operativos de Microsoft. Además, en 2016 Microsoft publicó el código de PowerShell en GitHub para que pueda portarse a otros sistemas y actualmente se trabaja en el soporte para GNU/Linux y macOS.

2. ¿Cómo ejecutar PowerShell?

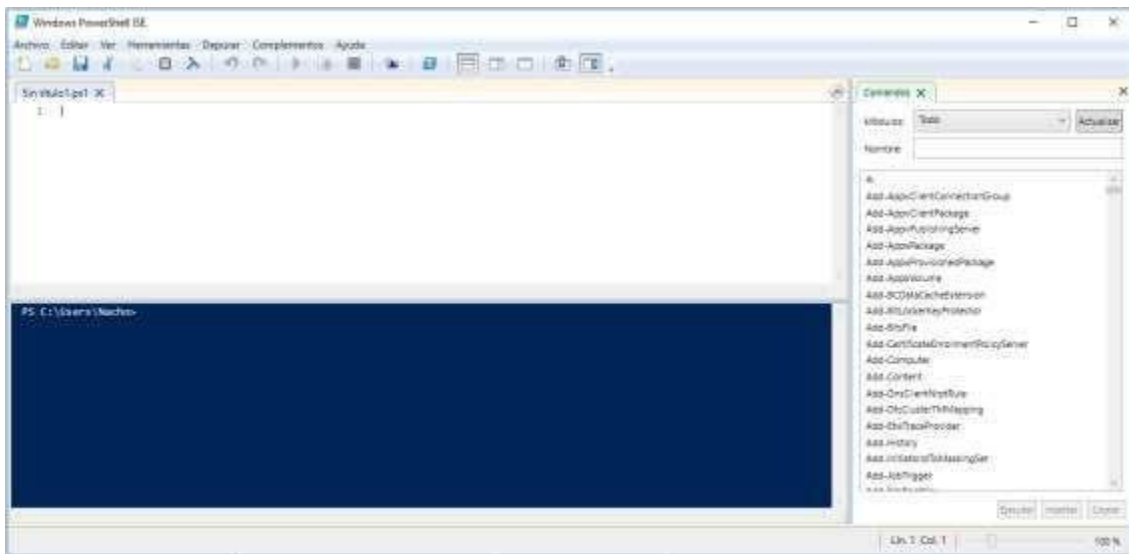
Hay dos opciones para ejecutar *PowerShell*:

1. **Windows PowerShell ISE:** Esta opción ofrece un entorno integrado de scripting para PowerShell.

Para acceder: *Botón Inicio > Carpeta Windows PowerShell > Windows PowerShell ISE*

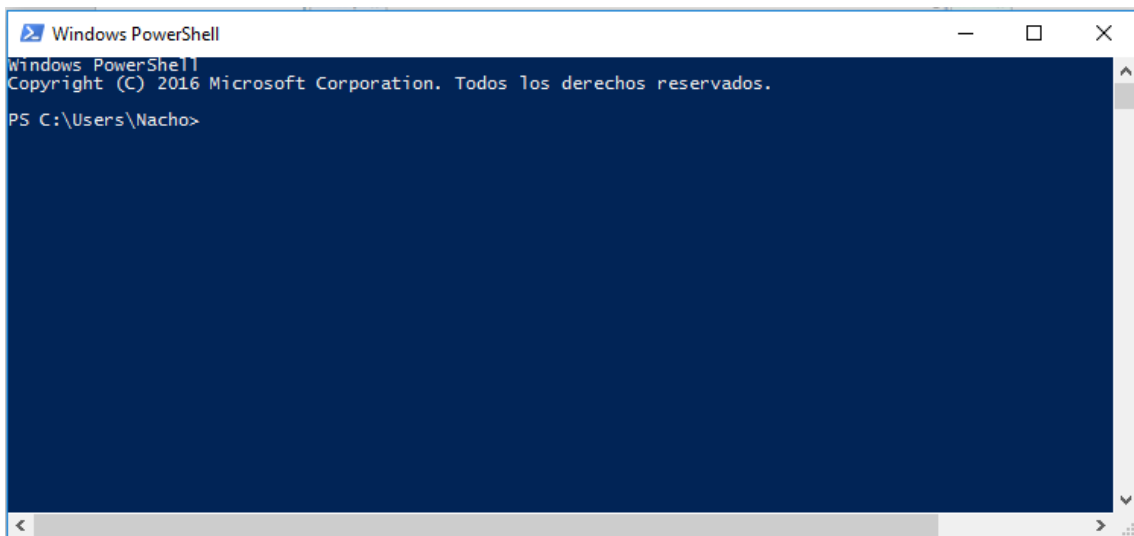


Tras pulsar se abre el entorno de desarrollo y ya podemos comenzar utilizar PowerShell:



2. **Windows PowerShell:** Esta opción ofrece una consola de texto más básica, que se parece a la línea de comandos tradicional.

Para acceder: *Botón Inicio > Carpeta Windows PowerShell > Windows PowerShell*



NOTA: Para abrir PowerShell con permisos de Administrador, se debe pulsar con el botón derecho en el icono de PowerShell (o PowerShell ISE) y elegir *Ejecutar como Administrador*.

3. Políticas de seguridad en la ejecución de scripts

De forma predeterminada, el sistema no permite la ejecución de ningún tipo de *scripts*. Si un usuario quiere ejecutar *scripts*, debe ajustar esta política predeterminada del sistema. Existen cuatro niveles de seguridad disponibles en cuanto a ejecución de scripts:

- *Restricted*: no permite la ejecución de *scripts*. Solo puede utilizarse PowerShell en modo interactivo. Esta es la opción predeterminada.
- *AllSigned*: es la opción más restrictiva. Todos los scripts deben estar autenticados antes de poder ejecutarse.
- *RemoteSigned*: solo deben estar autenticados los scripts que procedan de una ubicación remota. Por ejemplo, los que hayan sido descargados.
- *Unrestricted*: se ejecutará cualquier script sin importar su origen. Se trata de la opción menos recomendada.

Para saber la configuración actual en las políticas de seguridad en la ejecución de scripts, hay que ejecutar el *cmdlet* `Get-ExecutionPolicy`. Al ejecutar este comando nos devolverá uno de los cuatro valores antes descritos (*Restricted*, *AllSigned*, *RemoteSigned* o *Unrestricted*).

Para establecer la política de ejecución de *scripts* es necesario ejecutar el *cmdlet*:

`Set-ExecutionPolicy` seguido del nombre de política elegido.

Ejemplo: Establece el nivel de seguridad en *RemoteSigned*

```
Set-ExecutionPolicy RemoteSigned
```

NOTA: Para ejecutar nuestros propios scripts basta con establecer la política de seguridad en *RemoteSigned*

NOTA: Para ejecutar `Set-ExecutionPolicy` se requieren privilegios administrativos. Es decir, hay que ejecutar *PowerShell* como administrador.

4. Cómo escribir y ejecutar un script

A la hora de escribir y ejecutar un script en PowerShell tenemos dos opciones:

- Utilizar un editor de texto plano (Ej: notepad) y ejecutarlo en la consola
- Utilizar PowerShell ISE para escribir y ejecutar el script.

4.1. Escribir un script en un editor de texto y ejecutarlo en la consola

Los pasos necesarios para escribir y ejecutar un script en PowerShell son:

(1) Usar cualquier editor como **notepad** para escribir el script.

Sintaxis:

```
notepad nombre-script.ps1
```

(2) Ejecuta el script de la siguiente forma:

Sintaxis:

```
.\nombre-script.ps1
```

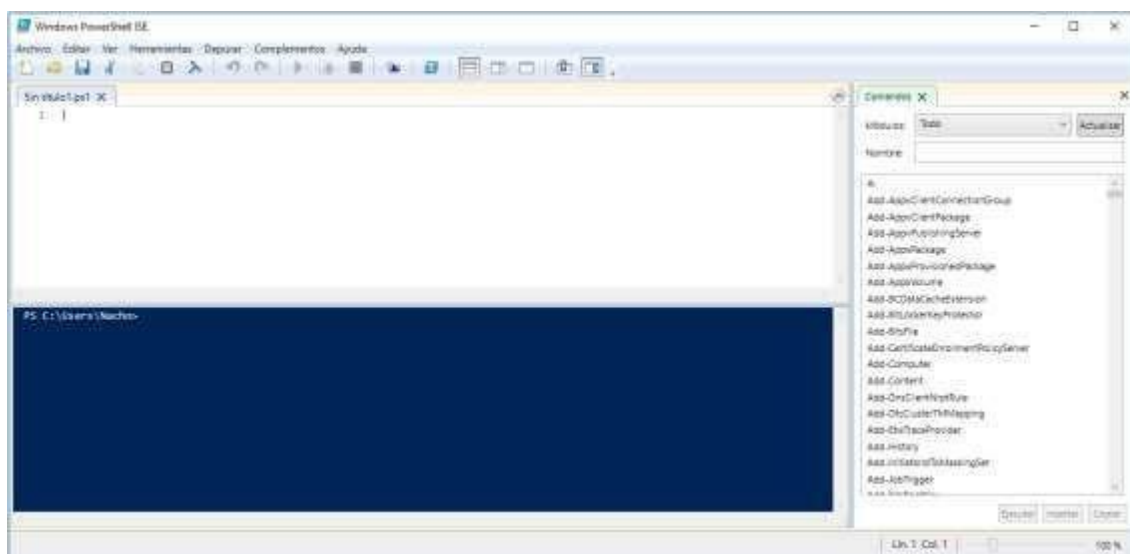
NOTA: Para que el archivo de texto sea tratado como un script de PowerShell es necesario que tenga la extensión .ps1


4.2. Escribir y ejecutar un script en PowerShell ISE

Los pasos necesarios para escribir y ejecutar un script en PowerShell son:

(1) Hacer clic sobre el botón *Nuevo* de la barra de herramientas 

Esta acción mostrará en la parte superior del entorno integrado el editor donde podremos escribir el script.



(2) Para ejecutar el script hay que pulsar el botón *Ejecutar script*  y el resultado de la ejecución se mostrará en la parte inferior del entorno integrado (consola)

5. Mi primer script

Ya estamos listo para escribir nuestro primer script de PowerShell (`holamundo.ps1`) que imprimirá en pantalla el mensaje “Hola mundo!”.

Escribe el siguiente script en el editor integrado de PowerShell ISE:

```
# Mi primer script

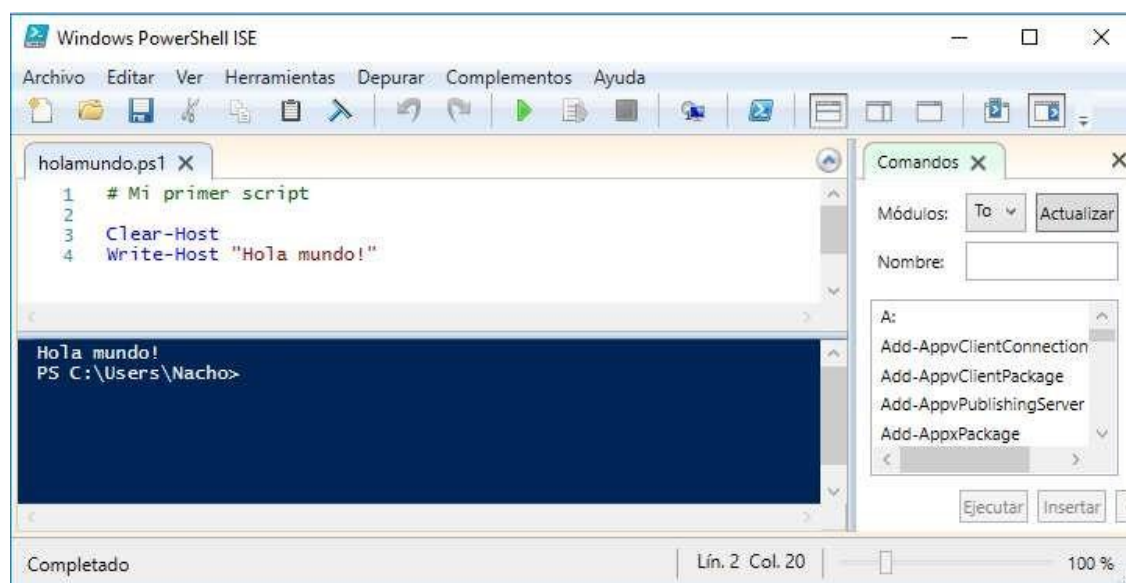
Clear-Host
Write-Host "Hola mundo!"
```

Explicación del código del script línea a línea:

Comando	Significado
# Mi primer script	# seguido de cualquier texto se considera un comentario y no será interpretado. Los comentarios sirven para proporcionar más información sobre el script, explicar partes del código, etc.
Clear-Host	Limpia la pantalla
Write-Host "Hola mundo!"	Imprime el mensaje “Hola mundo” en pantalla. Para imprimir un texto en pantalla se utiliza el comando Write-Host

Finalmente ejecuta el script:

```
$ .\holamundo.ps1
```



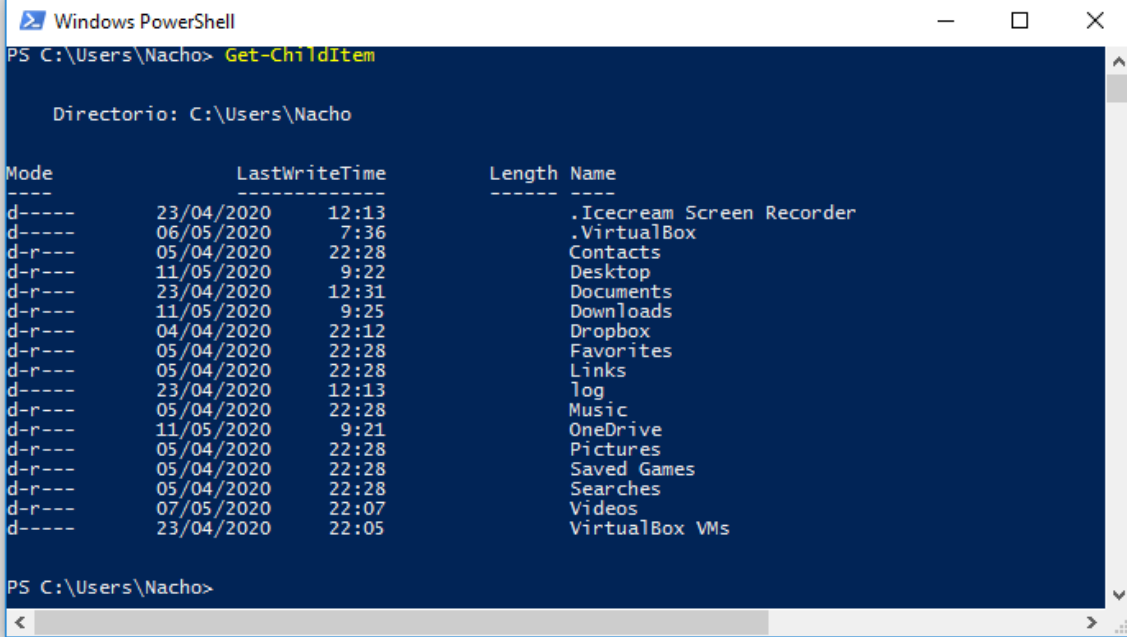
6. Estructura de los cmdlets

Los comandos PowerShell son llamados **cmdlets** y están estructurados de la siguiente manera:

Verbo-Nombre

Ejemplo: Mostrar el nombre de los archivos y las carpetas que hay en el directorio actual

Get-ChildItem



```
Windows PowerShell
PS C:\Users\Nacho> Get-ChildItem

Directorio: C:\Users\Nacho

Mode                LastWriteTime         Length Name
----                -
d-----          23/04/2020    12:13             .Icecream Screen Recorder
d-----          06/05/2020     7:36             .VirtualBox
d-r---          05/04/2020   22:28             Contacts
d-r---          11/05/2020     9:22             Desktop
d-r---          23/04/2020   12:31             Documents
d-r---          11/05/2020     9:25             Downloads
d-r---          04/04/2020   22:12             Dropbox
d-r---          05/04/2020   22:28             Favorites
d-r---          05/04/2020   22:28             Links
d-----          23/04/2020    12:13             log
d-r---          05/04/2020   22:28             Music
d-r---          11/05/2020     9:21             OneDrive
d-r---          05/04/2020   22:28             Pictures
d-r---          05/04/2020   22:28             Saved Games
d-r---          05/04/2020   22:28             Searches
d-r---          07/05/2020   22:07             Videos
d-----          23/04/2020   22:05             VirtualBox VMs

PS C:\Users\Nacho>
```

NOTA: PowerShell no es case sensitive (salvo en el caso de PowerShell para Mac OS o Linux).

7. Comandos básicos

Comando	Descripción
Get-Command	Devuelve todos los comandos (alias, function, cmdlet, ..) de PowerShell.
Get-Member	Devuelve las propiedades y métodos de un objeto así como su tipo.
Get-ChildItem	Muestra el contenido de una carpeta
Get-Location	Devuelve el directorio actual
Set-Location	Cambia de carpeta
New-Item	Crea un archivo/carpeta.
Remove-Item	Suprime un archivo/carpeta.
Move-Item	Mueve un archivo/carpeta.
Rename-Item	Renombra un archivo/carpeta
Copy-Item	Copia un archivo/carpeta.
Clear-Item	Borra el contenido de un archivo
Get-Content	Envía el contenido de un archivo a la salida
Set-Content	Establece el contenido de un archivo
Get-Service	Visualiza los servicios del sistema indicando su nombre y su estado

Ejemplo: Devuelve la ruta de acceso de su ubicación de directorio actual

```
Get-Location
```

Ejemplo: Establece la ubicación actual a C:\ (cambiar de carpeta)

```
Set-Location -Path C:\
```

Ejemplo: Crea una carpeta llamada NuevaCarpeta dentro de C:\temp

```
New-Item -Path 'C:\temp\NuevaCarpeta' -ItemType Directory
```

Ejemplo: Crea un archivo miArchivo.txt en la ruta C:\temp\NuevaCarpeta

```
New-Item -Path 'C:\temp\New Folder\miArchivo.txt' -ItemType File
```

Ejemplo: Elimina la carpeta C:\temp\NuevaCarpeta

```
Remove-Item -Path C:\temp\NuevaCarpeta
```

Ejemplo: Copia el archivo `miArchivo.txt` ubicado en `C:\origen` a `C:\destino` con el nombre `copia.txt`

```
Copy-Item -Path C:\origen\miArchivo.txt -Destination C:\destino\copia.txt
```

Ejemplo: Muestra todos los cmdlets, funciones y alias de PowerShell que están instalados en el equipo.

```
Get-Command
```

Ejemplo: Muestra todos los cmdlets disponibles en el equipo

```
Get-Command -Type cmdlet
```

Ejemplo: Muestra todos los alias disponibles en el equipo

```
Get-Command -Type alias
```

Ejemplo: Muestra todas las funciones disponibles en el equipo

```
Get-Command -Type function
```

8. Alias

PowerShell para facilitar la transición a los usuarios provenientes de entornos Unix y CMD tiene multitud de alias. Los alias son mecanismos que PowerShell proporciona para permitir a los usuarios hacer referencia a los comandos con nombres alternativos.

A continuación, se muestran a modo de ejemplo los alias para algunos cmdlets:

Cmdlet PowerShell	Alias
Get-ChildItem	dir, ls, qci
Set-Location	cd, chdir, sl
New-Item	md, ni, mkdir
Remove-Item	del, erase, rd, ri, rm, rmdir
Move-Item	mi, move, mv
Rename-Item	ren, rni
Copy-Item	copy, cp, cpi

9. Ayuda sobre cmdlets

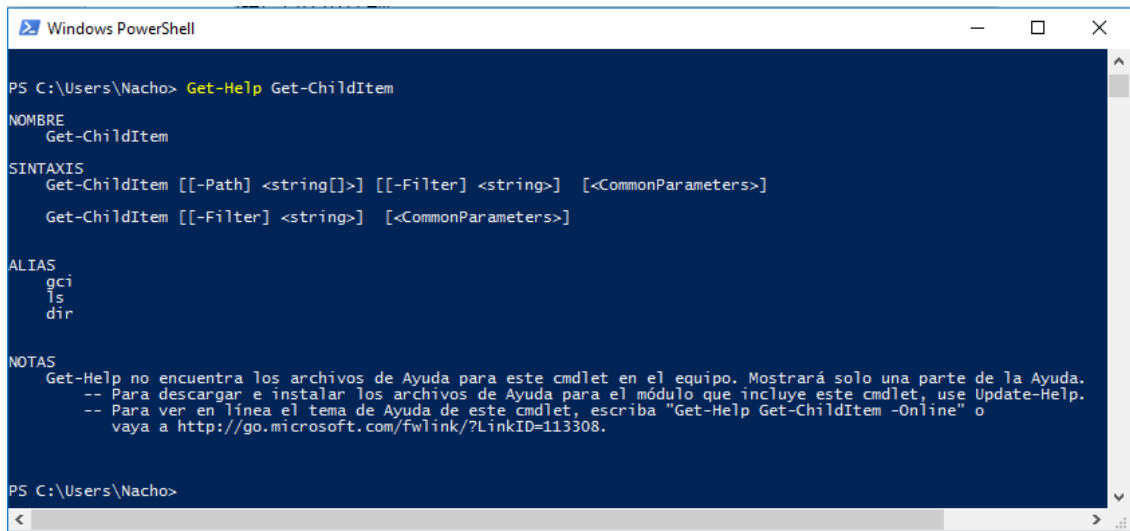
Para obtener ayuda acerca de un comando se utiliza el comando `Get-Help`.

Sintaxis:

```
Get-Help <comando>
```

Ejemplo: Obtener ayuda sobre el comando `Get-ChildItem`

```
Get-Help Get-ChildItem
```



`Get-Help` comando devuelve la ayuda estándar de un comando, es decir la mínima posible.

PowerShell ofrece tres niveles de detalle en la ayuda:

- Ayuda estándar: (`Get-Help comando`)
- Ayuda detallada: (`Get-Help comando -Detailed`)
- Ayuda completa: (`Get-Help comando -Full`)

10. Comentarios

En PowerShell hay que distinguir dos tipos de comentarios:

- Comentarios de una sola línea
- Comentarios de varias líneas

Ejemplo: Comentario de una sola línea

```
#Comentario de una línea
```

Ejemplo: Comentario de varias líneas

```
<#Comentario que ocupa
varias líneas#>
```

11. Variables

Hay dos tipos de variables:

- **Variables predefinidas por PowerShell:** creadas y mantenidas por PowerShell.
- **Variables definidas por el usuario:** creadas y mantenidas por el usuario.

11.1. Variables predefinidas

Algunas de las variables predefinidas más importantes son:

Variable	Descripción
<code>\$\$</code>	Variable que contiene el último valor del comando escrito en la consola.
<code>\$?</code>	Variable que contiene true si la última operación ha tenido éxito o false en el caso contrario.
<code>\$_</code>	Variable que contiene el objeto actual transmitido mediante una pipe .
<code>\${^}</code>	Nombre del script
<code>\$Args</code>	Variable que contiene el array de argumentos pasados a una función o a un script. Para conocer el número de argumentos pasados utilizar <code>\$Args.count</code> . Para acceder al valor de cada argumento utilizar <code>\$Args[0]</code> , <code>\$Args[1]</code> ...
<code>\$ConsoleFileName</code>	Variable que contiene la ruta de acceso al archivo de consola (.psc1) que ha sido utilizado en la última sesión.
<code>\$Error</code>	Variable de tipo array que contiene todos los errores encontrados desde el inicio de la sesión PowerShell actual (consulte el capítulo Gestión de errores y depuración).
<code>\$Event</code>	Variable que contiene el evento actual tratado en el bloque de script de un comando de grabación de evento, típicamente como Register-ObjectEvent.
<code>\$EventArgs</code>	Variable que contiene los argumentos en relación con \$Event. Como esta última, \$EventArgs se utiliza únicamente en el bloque de scripts de un comando de grabación de evento.
<code>\$False</code>	Variable que contiene el valor false. Esta variable es una constante.
<code>\$Foreach</code>	Variable que hace referencia al iterador de un bucle Foreach.

\$Home	Variable que contiene la ruta (path) de la carpeta home del usuario.
\$Host	Variable que contiene informaciones del host (la consola PowerShell).
\$Input	Contiene un enumerador que permite enumerar los datos de entrada que se pasan a una función. \$Input solo puede utilizarse dentro de una función (bloque Process) o en un bloque de script.
\$LastExitCode	Variable que contiene el código de salida de la última ejecución de un fichero ejecutable Win32.
\$Matches	Variable con los operadores que trabajan en las expresiones regulares -Match y -NotMatch. \$Matches es un array que contiene el o los resultados de la comparación.
\$MyInvocation	Contiene informaciones sobre el script, la función o el bloque de script actual, tales como su nombre, sus parámetros, su ruta en el disco, etc.
\$NULL	Variable vacía o nula.
\$PID	Variable que contiene un número que identifica el proceso PowerShell en curso.
\$Profile	Variable que contiene las rutas (path) de los diferentes perfiles de Windows PowerShell.
\$PsCommandPath	Variable que contiene la ruta completa del archivo o script en ejecución.
\$PsHome	Variable que contiene la ruta (path) donde está instalado PowerShell.
\$PSVersionTable	Variable que contiene un array de solo lectura que muestra los detalles relativos a la versión de Windows PowerShell y su entorno.
\$Pwd	Variable que indica la ruta completa de la carpeta activa.
\$ShellID	Variable que indica el identificador del shell.
\$This	En el interior de un bloque de script, cuando se define una propiedad o método suplementarios (por ejemplo, con Add-Member), \$this hace referencia al objeto en curso.
\$True	Variable que contiene el valor true.

Nuevas variables automáticas incorporadas por PowerShell 6:

Variable	Descripción	Valor por defecto
\$ISCoreCLR	Valor booleano. Su valor es verdadero si PowerShell se basa en .NET Core.	True
\$IsLinux	Valor booleano. Su valor es verdadero si el OS que ejecuta PowerShell funciona bajo Linux.	Depende del OS en ejecución
\$IsMacOS	Valor booleano. Su valor es verdadero si el OS que ejecuta PowerShell funciona bajo Mac OS.	Depende del OS en ejecución
\$IsWindows	Valor booleano. Su valor es verdadero si el OS que ejecuta PowerShell funciona bajo Windows.	Depende del OS en ejecución

11.2. Variables definidas por el usuario

Para definir variables y asignar sus valores se utiliza la siguiente sintaxis:

```
$nombreVariable = valor de la variable
```

Ejemplo: Variable llamada `nombre` y que almacena un valor de tipo cadena

```
$nombre = "Ignacio"
```

Ejemplo: Variable llamada `numero` y que almacena un valor de tipo entero

```
$numero = 25
```

Acceso al valor de una variable

Para acceder al contenido de una variable, basta con escribir simplemente el nombre de la variable en la consola o en un script.

Ejemplo: Accede (imprime) el valor de la variable `$nombre`

```
$nombre
```

```
Ignacio
```

Tipo de una variable

Para determinar el tipo de una variable hay que utilizar el método `GetType()` junto con la propiedad `Name`.

Ejemplo: Muestra el tipo de la variable `$nombre`

```
$nombre.GetType().Name
```

```
String
```

11.3. Reglas para nombrar variables y asignarles valor

- A. El primer carácter debe ser siempre un símbolo de dólar (\$)
- B. Después, podemos utilizar cualquier combinación de letras, números o símbolos.
- C. También pueden utilizar espacios en blanco (aunque no se recomienda). En este caso, el nombre debe ir entre llaves ({}).
- D. Se pueden o no poner espacios a ambos lados del signo igual (=) cuando asigne un valor a la variable. Las siguientes asignaciones son todas correctas:

```
$var=10  
$var =10  
$var= 10  
$var = 10
```

11.4. Tipo de asignación de datos a una variable

En *PowerShell* existen dos formas de establecer la naturaleza del dato que va a almacenar una variable:

- **Implícita:** El tipo de dato que puede almacenar la variable se establece en función del valor asignado. Es la que hemos visto con los ejemplos anteriormente.

```
$numero = 25
```

- **Explícita:** El programador establece el tipo de dato que podrá contener la variable. De este modo, cuando asignemos un valor que no coincide con el tipo especificado, PowerShell intentará adaptarlo a la variable si es posible.

```
[int]$precio = 10
```

12. Comandos Write-Host y Read-Host

12.1. Write-Host

El comando `Write-Host` muestra en pantalla la información que se le pasa.

Sintaxis:

```
Write-Host [opciones] información a mostrar
```

Ejemplo: Muestra en pantalla el mensaje "Hola ¿Qué tal?"

```
Write-Host "Hola ¿Qué tal?"  
Hola ¿Qué tal?
```

Ejemplo: Muestra en pantalla el valor de la variable `$numero`

```
Write-Host $numero  
25
```


NOTA: Para mostrar información se puede omitir el nombre del cmdlet `Write-Host` de modo que, si en una línea del script se escribe únicamente el nombre de una variable, se mostrará igualmente su valor también.

Ejemplo: Muestra en pantalla el valor de la variable `$numero` pero sin utilizar el comando `Write-Host`

```
$numero  
25
```

12.2. Read-Host

Se usa para leer datos del usuario desde teclado y almacenarlos en una variable.

Sintaxis:

```
$variable = Read-Host
```

Ejemplo: El siguiente script primero pregunta al usuario el nombre y queda a la espera de que lo introduzca por teclado. Cuando el usuario introduce el nombre por teclado y pulsa la tecla intro, se almacena en la variable `$varnombre` y posteriormente muestre un mensaje saludando

```
#Script para leer tu nombre desde teclado  
  
Write-Host -NoNewline "Introduzca su nombre: "  
$varnombre = Read-Host  
Write-Host ";Hola $varnombre!"
```

Al ejecutarlo:

```
$ .\EjemploRead-Host.ps1  
Introduzca su nombre: Javier  
;Hola Javier!
```

NOTA: Se puede utilizar el comando `Read-Host` para indicar en la misma línea el texto que se mostrará por pantalla para solicitar al usuario la información.

Ejemplo:

```
$varnombre = Read-Host "Introduzca su nombre: "
```

```
#Es equivalente a:  
Write-Host -NoNewline "Introduzca su nombre: "  
$varnombre = Read-Host
```

13. Comillas

Hay tres tipos de comillas

Comillas	Nombre	Significado
"	Comillas dobles	Dentro de las comillas dobles las cadenas pueden evaluar variables y caracteres especiales.
'	Comillas simples	Dentro de las comillas simples todos los caracteres son interpretados literalmente. Es decir, ninguno de los caracteres especiales conserva su significado dentro de las comillas simples.
`	Comilla invertida	Se utilizan para escapar y evitar la interpretación de caracteres especiales. Por ejemplo, para usar una comilla doble dentro de una cadena, debe escaparse usando el carácter (`).

14. Operadores

14.1. Operadores de cadenas de caracteres

Operador	Significado
+	Concatena cadenas de texto

Ejemplo:

```
$nombre = "Javier"
$apellidos = "Gómez Rodríguez"

$nombrecompleto = $nombre + $apellidos    #Concatenación
Write-Host $nombrecompleto
```

Salida:

```
JavierGómez Rodríguez
```

Ejemplo:

```
$nombre = "Javier"
$apellidos = "Gómez Rodríguez"

$nombrecompleto = $nombre + " " + $apellidos    #Concatenación
Write-Host $nombrecompleto
```

Salida:

```
Javier Gómez Rodríguez
```

Ejemplo:

```
$nombre = "Javier"
$apellidos = "Gómez Rodríguez"

$nombrecompleto = "$nombre $apellidos"      #Concatenación
Write-Host $nombrecompleto
```

Salida:

```
Javier Gómez Rodríguez
```

14.2. Operadores aritméticos

Se utilizan para realizar operaciones aritméticas

Sintaxis:

```
operando1 operador operando2
```

Operador	Significado
+	Suma
-	Resta
*	Multipliación
/	División
%	Resto (módulo)

Ejemplos:

```
$resultado = 1 + 3      # $resultado = 4
$resultado = 2 - 1      # $resultado = 1
$resultado = 10 * 3     # $resultado = 30
$resultado = 10 / 2     # $resultado = 5
$resultado = 20 % 3     # $resultado = 2
```

Otros operadores aritméticos son:

Operador	Significado	Ejemplo	Equivalencia
+=	Incrementa el valor de la variable	\$x += 2	\$x = \$x + 2
-=	Decrementa el valor de la variable	\$x -= 2	\$x = \$x - 2
*=	Multiplica el valor de la variable	\$x *= 2	\$x = \$x * 2
/=	Divide el valor de la variable	\$x /= 2	\$x = \$x / 2
++	Incrementa en 1 el valor de la variable	\$x++	\$x = \$x + 1
--	Decrementa en 1 el valor de la variable	\$x--	\$x = \$x - 1

14.3. Operadores de comparación

Los operadores de comparación devolverán \$true o \$false en función del resultado de la comparación

Operador de comparación	Significado	Ejemplo (Devuelve \$true)
-eq	Igualdad	2 -eq 2
-ieq	Igualdad Con valores de tipo cadena de texto NO tiene en cuenta las diferencias entre mayúsculas y minúsculas	"Hola" -ieq "HOLA"
-ceq	Igualdad Con valores de tipo cadena de texto SI tiene en cuenta las diferencias entre mayúsculas y minúsculas	"Hola" -ceq "Hola"
-ne	Distinto	2 -ne 4
-lt	Menor que	2 -lt 4
-le	Menor o igual que	2 -le 4
-gt	Mayor que	4 -gt 2
-ge	Mayor o igual que	4 -ge 2

Otros operadores de comparación especiales son:

Operador de comparación	Significado	Ejemplo (Devuelve \$true)
-like	Es como	"Alberto" -like "Alb*"
-notlike	No es como	"Alberto" -notlike "Alc*"
-contains	Contiene	1,3,5,7,9 -contains 5
-notcontains	No contiene	1,3,5,7,9 -notcontains 2

14.4. Operadores lógicos

Operador lógico	Significado	Ejemplo (Devuelve \$true)
-and	Devuelve \$true si las dos expresiones devuelven como valor \$true	(5 -ge 3) -and (5 -le 5)
-or	Devuelve \$true si al menos una de las dos expresiones devuelve como valor \$true	(5 -gt 3) -or (5 -lt 5)
-xor	Devuelve \$true si y solo si una de las dos expresiones devuelve como valor \$true	(5 -gt 3) -xor (8 -le 5)
-not !	Devuelve \$true si la expresión sobre la que actúa devuelve el valor \$false	-not (5 -lt 3) !(5 -lt 3)

15. Redirección de salida

La salida estándar de los comandos se proporciona por defecto por pantalla, pero es posible enviar la salida a un archivo.

Símbolo redirección	Sintaxis	Significado
>	comando > archivo	Redirige la salida estándar a archivo. (Si archivo existe se sobrescribe y si no se crea)
>>	comando >> archivo	Redirige la salida estándar a archivo. (Si archivo existe se añaden los datos al final del archivo.)
2>	comando 2> archivo	Redirige el error estándar a archivo
*>	comando *> archivo	Redirige todos los flujos de salida (tanto la salida estándar como el error estándar) a archivo

Ejemplos: Redirecciona la salida del comando `Get-ChildItem` y la vuelca en `archivo.txt`

```
Get-ChildItem > archivo.txt
```

Sin embargo, cada vez que ejecutemos ese comando el contenido de `archivo.txt` será reemplazado por la salida del comando `Get-ChildItem`. Si queremos agregar la salida del comando al final del archivo, en lugar de reemplazarla, entonces ejecutamos:

```
Get-ChildItem >> archivo.txt
```

16. Tuberías

Una tubería (pipe) permite redirigir la salida estándar de un comando para que se convierta en la entrada estándar de otro comando por medio del operador `|`

Sintaxis:

```
comando1 | comando2
```

Ejemplo: Cuenta el número de cmdlet que tiene PowerShell

```
Get-Command -CommandType cmdlet | Measure-Object
```

```
PS C:\Users\Nacho> Get-Command -CommandType cmdlet | Measure-Object

Count      : 825
Average    :
Sum        :
Maximum    :
Minimum    :
Property   :
```

17. Estructuras de control de flujo

17.1. if

La instrucción `if` elige entre alternativas:

```
if (condición)
{
    #bloque de instrucciones
}
```

Ejemplo:

```
$var = Read-Host "Introduzca un carácter"

if ($var -eq 'A')
{
    "El carácter introducido por el usuario es una 'A'"
}
```

Si queremos distinguir entre dos alternativas, se utiliza la estructura `if-else`:

```
if (condición)
{
    # Bloque de instrucciones 1
}
else
{
    # Bloque de instrucciones 2
}
```

Ejemplo:

```
[int]$var1 = Read-Host 'Introduzca un número: '
[int]$var2 = Read-Host 'Introduzca un número: '

if ($var1 -ge $var2)
{
    "$var1 es más grande o igual que $var2"
}
else
{
    "$var1 es más pequeño que $var2"
}
```

Por último, la construcción `if-elseif-else`, puede considerar todas las alternativas que se desee:

```
if (condición)
{
    # Bloque de instrucciones 1
}
elseif (condición)
{
    # Bloque de instrucciones 2
}
else
{
    # Bloque de instrucciones 3
}
```

17.2. switch

La instrucción `switch` es una buena alternativa a la instrucción `if-elseif-else` múltiple. Permite evaluar los diferentes valores que puede tomar una expresión y realizar diferentes acciones en cada caso.

```
switch (expresión)
{
    <Valor_1> { bloque de instrucciones 1; Break }
    <Valor_2> { bloque de instrucciones 2; Break }
    <Valor_3> { bloque de instrucciones 3; Break }
    default { bloque de instrucciones 4}
}
```

donde `expresión` se compara con los diferentes valores hasta que se encuentra una coincidencia. El valor `default` es opcional. Su bloque de instrucciones solo se ejecuta cuando la expresión no corresponde con ningún valor.

Ejemplo:

```
[int]$numero = Read-Host 'Introduzca un número entre 1 y 5: '

switch($numero)
{
    1 { 'Ha introducido el número 1 '; Break }
    2 { 'Ha introducido el número 2 '; Break }
    3 { 'Ha introducido el número 3 '; Break }
    4 { 'Ha introducido el número 4 '; Break }
    5 { 'Ha introducido el número 5 '; Break }
    default { "; Número introducido incorrecto !"}
}
```

Utilizar switch con expresiones

Cuando se utiliza switch con expresiones, el valor comprobado se asigna a la variable automática \$_

Ejemplo:

```
[int]$numero = Read-Host 'Introduzca un número entero positivo'

switch($numero)
{
    {$_ -lt 10} { 'Número estrictamente inferior a 10 '; Break}
    {$_ -ge 10 -and $_ -le 20} { 'Número entre 10 y 20' ; Break}
    Default { 'Número superior a 20' }
}
```

Utilizar switch con expresiones regulares

Se debe especificar el parámetro -regex

Ejemplo:

```
$cadena = Read-Host 'Introduzca una cadena'

switch -regex ($cadena)
{
    '^[aeiouy]' { 'La cadena empieza por una vocal' ; break}
    '^[^aeiouy]' { 'La cadena no empieza por una vocal' ; break}
}
```

17.3. while

El bucle while repite una serie de comandos mientras una condición sea cierta.

```
while (condición)
{
    #bloque de instrucciones
}
```

Ejemplo:

```
$i=0
while ($i -lt 3)
{
    Write-Host $i
    $i++
}
```

Al ejecutarlo obtenemos:

```
0
1
2
```


17.4. do while

El bucle `do while` es similar al bucle `while`, ya que se repite mientras que la condición sea cierta. En este caso, la diferencia está en que la condición se evalúa al final y por tanto el bloque de instrucciones se ejecuta al menos una vez:

```
do
{
    #bloque de instrucciones
}
while (condición)
```

17.5. do until

El bucle `do until` se ejecuta hasta que la condición sea cierta. Se evalúa la condición al final del bloque de instrucciones.

```
do
{
    #bloque de instrucciones
}
until (condición)
```

17.6. for

El bucle `for` permite ejecutar un número determinado de veces un bloque de instrucciones.

```
for (inicial ;condición ;incremento)
{
    #bloque de instrucciones
}
```

Ejemplo: Imprime los números del 1 al 10

```
for ($i=1; $i -le 10; $i++)
{
    Write-Host $i
}
```

17.7. foreach

El bucle `foreach` itera por los valores de una lista:

```
foreach (elemento in lista)
{
    #bloque de instrucciones
}
```

Ejemplo:

```
foreach ($nombre in "Alberto", "Carlos", "Juan")
{
    Write-Host "Se llama $nombre"
}
```

Al ejecutarlo se obtiene:

```
Se llama Alberto
Se llama Carlos
Se llama Juan
```

18. Funciones

Se pueden utilizar funciones para agrupar varios comandos o sentencias. Para definir una función:

```
Function <nombre de función> {
    param (<lista de parámetros>)
    # bloque de instrucciones
}
```

Ejemplo:

```
Function Get-Resta {
    param([int]$Numero1, [int]$Numero2)

    $Resultado = $Numero1 - $Numero2
    Write-host $Resultado
}
```

Para llamar a una función únicamente es necesario indicar el nombre de la función e indicarle los parámetros. Tenemos varias formas de indicarle los parámetros:

Ejemplos:

```
Get-Suma -Numero1 12 -Numero2 8 #Salida: 4
```

```
Get-Suma -Numero2 8 -Numero1 12 #Salida: 4
```

```
Get-Suma -Numero2 12 -Numero1 8 #Salida: -4
```

```
Get-Suma 12 8 #Salida: 4
```

19. Manipulación de objetos

PowerShell es orientado a objetos y por tanto, cada variable que se define es en realidad un objeto y tendrá asociados una serie de métodos y propiedades que podremos utilizar para realizar ciertas acciones sin tener que escribir ningún código.

Ejemplo:

```
$saludo = ";Hola mundo!"
```

Si queremos saber la posición en la que aparece la primera letra 'o' se puede utilizar al método IndexOf que nos permite buscar una determinada cadena dentro del objeto en el que se utilice:

```
Write-Host $saludo.IndexOf('o') #Devuelve: 2
```

Si queremos convertir en mayúsculas el contenido de la variable anterior:

```
Write-Host $saludo.ToUpper() #Devuelve: ";HOLA MUNDO!"
```

Para conocer la lista completa de las acciones que podemos realizar con la variable podemos ejecutar:

```
$saludo | Get-Member
```

Esto nos devuelve una lista de los nombres de los métodos y las propiedades asociadas a la variable:

```
PS C:\Users\Nacho> $saludo | Get-Member

TypeName: System.String

Name      MemberType Definition
-----
Clone      Method      System.Object Clone(), System.Object ICloneable.Clone()
CompareTo  Method      int CompareTo(System.Object value), int CompareTo(string value)
Contains   Method      bool Contains(string value)
CopyTo     Method      void CopyTo(int sourceIndex, char[] destination, int destinationIndex)
EndsWith   Method      bool EndsWith(string value), bool EndsWith(string value, bool ignoreCase)
Equals     Method      bool Equals(System.Object obj), bool Equals(string value)
GetEnumerator Method      System.CharEnumerator GetEnumerator(), System.Collections.Generic.IEnumerable<char> GetEnumerator()
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
GetTypeCode Method      System.TypeCode GetTypeCode(), System.TypeCode IConvertible.GetTypeCode()
IndexOf    Method      int IndexOf(char value), int IndexOf(char value, int startIndex), int IndexOfAny(char[] anyOf), int IndexOfAny(char[] anyOf, int startIndex)
Insert     Method      string Insert(int startIndex, string value)
IsNormalized Method      bool IsNormalized(), bool IsNormalized(System.Text.NormalizationOptions options)
LastIndexOf Method      int LastIndexOf(char value), int LastIndexOf(char value, int startIndex), int LastIndexOfAny(char[] anyOf), int LastIndexOfAny(char[] anyOf, int startIndex)
Normalize  Method      string Normalize(), string Normalize(System.Text.NormalizationOptions options)
PadLeft    Method      string PadLeft(int totalWidth), string PadLeft(int totalWidth, char padChar)
PadRight   Method      string PadRight(int totalWidth), string PadRight(int totalWidth, char padChar)
Remove     Method      string Remove(int startIndex, int count), string RemoveAt(int index)
Replace    Method      string Replace(char oldChar, char newChar), string Replace(string oldString, string newString)
Split      Method      string[] Split(Params char[] separator), string[] Split(Params string[] separator)
StartsWith Method      bool StartsWith(string value), bool StartsWith(string value, bool ignoreCase)
Substring  Method      string Substring(int startIndex), string Substring(int startIndex, int length)
ToBoolean  Method      bool IConvertible.ToBoolean(System.IFormatProvider provider)
ToByte     Method      byte IConvertible.ToByte(System.IFormatProvider provider)
ToChar     Method      char IConvertible.ToChar(System.IFormatProvider provider)
```

ToCharArray	Method	char[] ToCharArray(), char[] ToCharArray(int startInde
ToDateTime	Method	datetime IConvertible.ToDateTime(System.IFormatProvide
ToDecimal	Method	decimal IConvertible.ToDecimal(System.IFormatProvider
ToDouble	Method	double IConvertible.ToDouble(System.IFormatProvider pr
ToInt16	Method	int16 IConvertible.ToInt16(System.IFormatProvider provi
ToInt32	Method	int IConvertible.ToInt32(System.IFormatProvider provid
ToInt64	Method	long IConvertible.ToInt64(System.IFormatProvider provi
ToLower	Method	string ToLower(), string ToLower(cultureinfo culture)
ToLowerInvariant	Method	string ToLowerInvariant()
ToSByte	Method	sbyte IConvertible.ToSByte(System.IFormatProvider prov
ToSingle	Method	float IConvertible.ToSingle(System.IFormatProvider pro
ToString	Method	string ToString(), string ToString(System.IFormatProvi
ToType	Method	System.Object IConvertible.ToType(type conversionType,
ToUInt16	Method	uint16 IConvertible.ToUInt16(System.IFormatProvider pr
ToUInt32	Method	uint32 IConvertible.ToUInt32(System.IFormatProvider pr
ToUInt64	Method	uint64 IConvertible.ToUInt64(System.IFormatProvider pr
ToUpper	Method	string ToUpper(), string ToUpper(cultureinfo culture)
ToUpperInvariant	Method	string ToUpperInvariant()
Trim	Method	string Trim(Params char[] trimChars), string Trim()
TrimEnd	Method	string TrimEnd(Params char[] trimChars)
TrimStart	Method	string TrimStart(Params char[] trimChars)
Chars	ParameterizedProperty	char Chars(int index) {get;}
Length	Property	int Length {get;}

20. Documentación oficial de PowerShell

Para consultar la documentación oficial de PowerShell consulta el sitio de Microsoft:

<https://docs.microsoft.com/es-es/powershell/>

