

Sistemas Informáticos

PowerShell



Índice

1. Objetivos	3
2. Introducción a PowerShell	3
2.1. ¿Qué es PowerShell?	3
2.2. Compatibilidad con CMD	3
2.3. Versiones de PowerShell	3
2.4. Actualización a PowerShell 3.0	4
2.5. Cmdlets básicos	6
2.6. Variables y Constantes	13
2.7. Consultas a WMI	15
3. Scripts con PowerShell	17
3.1. Creación de Scripts	17
3.2. Entorno de Scripts Integrado	18
3.3. Estructuras Condicionales	21
3.4. Bucles	22
3.5. Argumentos y parámetros	25
3.6. Funciones	27
4. Compartición de Recursos	28
4.1. Eliminación de Carpetas Compartidas	31
5. Gestión de Archivos	32
5.1. Guardar Datos en un Fichero	32
5.2. Recuperar Datos de un Fichero de Texto	32
5.3. Recuperar Datos de un Fichero de CSV	32

1. Objetivos

- Conocer los cmdlets básicos.
- Crear variables en PowerShell.
- Utilizar el Entorno de Scripts Integrado (ISE).
- Crear y ejecutar scripts.
- Implementar estructuras de control.
- Administrar recursos compartidos mediante PowerShell.

2. Introducción a PowerShell

2.1. ¿Qué es PowerShell?

PowerShell es un framework de Microsoft que se compone de dos partes:

1. Un intérprete de comandos.
2. Un lenguaje para la escritura de scripts.

Como principal diferencia con otros lenguajes de scripting destacaría que al estar basado en .NET, permite la orientación a objetos. Por tanto, al trabajar con objetos, los cuales vienen caracterizados por propiedades y métodos, el trabajo con PowerShell es razonablemente cómodo de escribir siendo al mismo tiempo muy potente al acceder de una manera estructurada a los elementos que componen el objeto.

2.2. Compatibilidad con CMD

Los comandos en PowerShell en realidad se llaman **cmdlets** (leído 'commandlets'). El término cmdlet proviene de *Command Applet*.

La versión 3.0 de PowerShell -implementada tanto en Windows 8 como en Windows Server 2012- recoge una cantidad vastísima de estos cmdlets. Sin embargo, en PowerShell siguen funcionando los comandos que utilizábamos con el clásico CMD, aunque en realidad en algunos casos son **alias** (es decir etiquetas que apuntan a cmdlets) y en otros son **funciones**, pero que en ambos casos ofrecen las mismas salidas que los comandos a los que estamos habituados. Por ejemplo, para obtener un listado de los elementos que tenemos en un directorio utilizábamos `dir` en CMD. Este comando también funciona en PowerShell, aunque en realidad apunta al cmdlet `Get-ChildItem`. Además, algunos comandos Unix como `ls` también están recogidos como alias en PowerShell. Si queremos saber a qué cmdlet apunta un alias escribiremos: `Get-alias comando_CMD`.

2.3. Versiones de PowerShell

En la actualidad existen tres versiones de PowerShell:

- 1.0: Apareció en 2006 incluyéndose posteriormente como una característica opcional en Windows Server 2008.
- 2.0: Integrada en Windows Server 2008 R2 y Windows 7 mejoró aspectos como la ejecución remota de los scripts, la aparición del entorno de desarrollo ISE (figura 2-1).
- 3.0: Integrada en Windows Server 2012 y Windows 8, sus mejoras más destacables respecto a las anteriores son la facilidad en la escritura de scripts mediante la inclusión de la funcionalidad *Intellisense* la cual muestra información sobre los cmdlets que estamos escribiendo de una manera automatizada, la persistencia de las conexiones con máquinas remotas aún en caso de fallos en la red, y el Script Explorer que nos permite acceder a un gran número de scripts de ejemplo.
- 4.0: Integrada en Windows Server 2012 R2 y Windows 8.1.
- 5.0: Integrada en Windows 10.
- 5.1: Integrada en Windows Server 2016 y 2019.

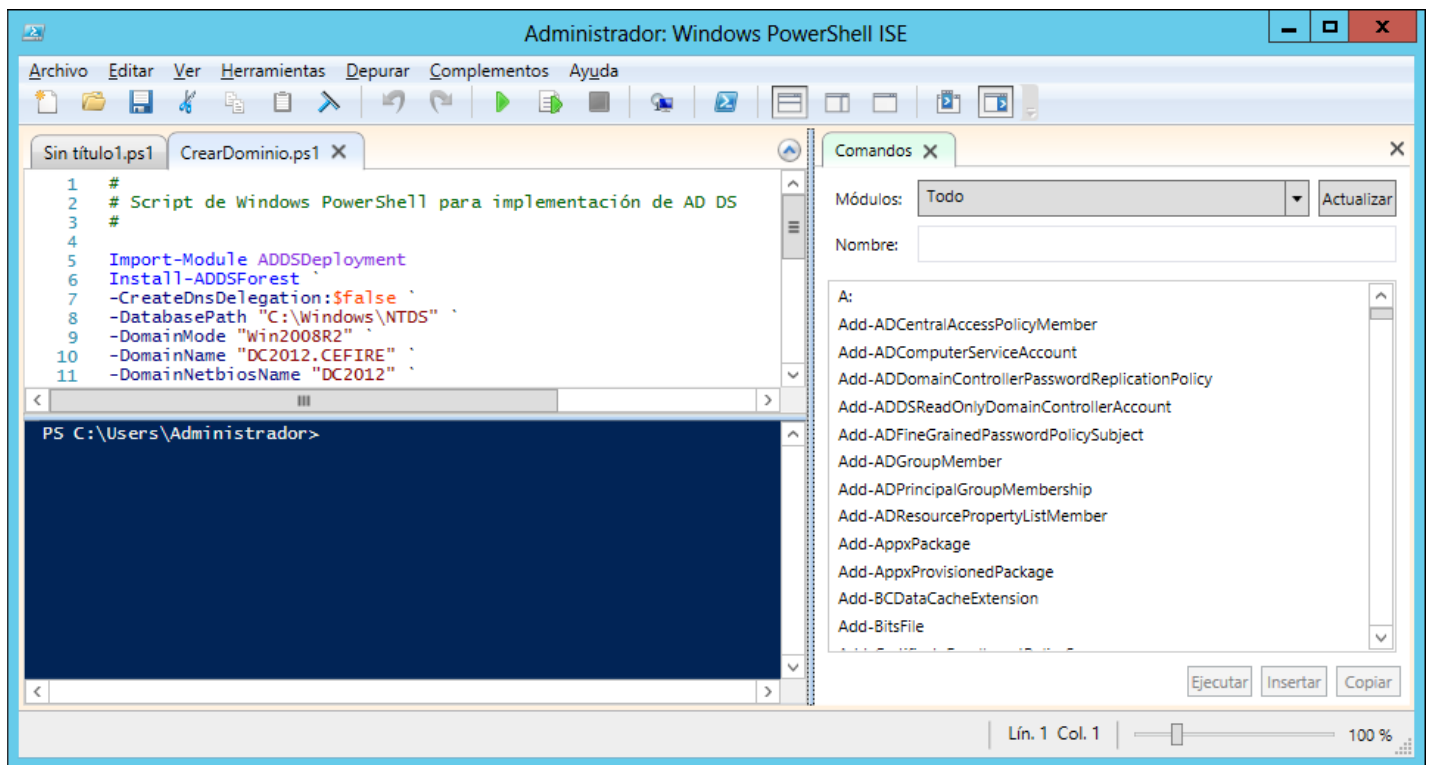


Figura 2-1. Entorno de desarrollo ISE.

Si queremos conocer la versión de PowerShell que tenemos instalada en nuestro sistema, abriremos PowerShell (figura 2-2).



Figura 2-2. Acceso directo a PowerShell.

A continuación escribiremos la siguiente instrucción:

```
> Get-Host | Select-Object Version
```

Y el sistema nos devolverá la versión instalada. El cmdlet `Get-Host` obtiene (*get*) información del entorno de ejecución. Como solo nos interesa la versión, la cual es una de las propiedades del objeto devuelto, bastará con utilizar a través de una tubería el cmdlet `Select-Object` para especificar la propiedad del objeto con la que queremos quedarnos. Puede parecer una manera algo farragosa de trabajar, pero en los siguientes apartados iremos viendo con mayor detalle cómo operar con PowerShell explotando la potencia que nos va a proporcionar la orientación a objetos.

2.4. Actualización a PowerShell 3.0

Para actualizar a PowerShell 3.0 en sistemas operativos que utilicen una versión anterior, ejecutaremos el fichero que nos permitirá instalar .NET Framework 4.0 y que podéis [descargar aquí](#) (el equipo servidor donde lo queramos instalar necesitará tener conexión a Internet).

Una vez que haya concluido el proceso de instalación de .NET Framework 4.0, necesitaremos descargar e instalar el Windows Management Framework 3.0 (concretamente el fichero Windows6.1-KB2506143-x64.msu, revisad las instrucciones de instalación), el cual contiene PowerShell 3.0.

Tras hacer doble clic en el fichero Windows6.1-KB2506143-x64.msu, que acabamos de descargar, y aceptar los términos de licencia, comenzará el proceso de instalación (figura 2-3).



Figura 2-3. Instalación de Windows Management Framework 3.0.

Después de reiniciar el sistema, veremos que ya tenemos instalada la versión 3.0 (figura 2-4).

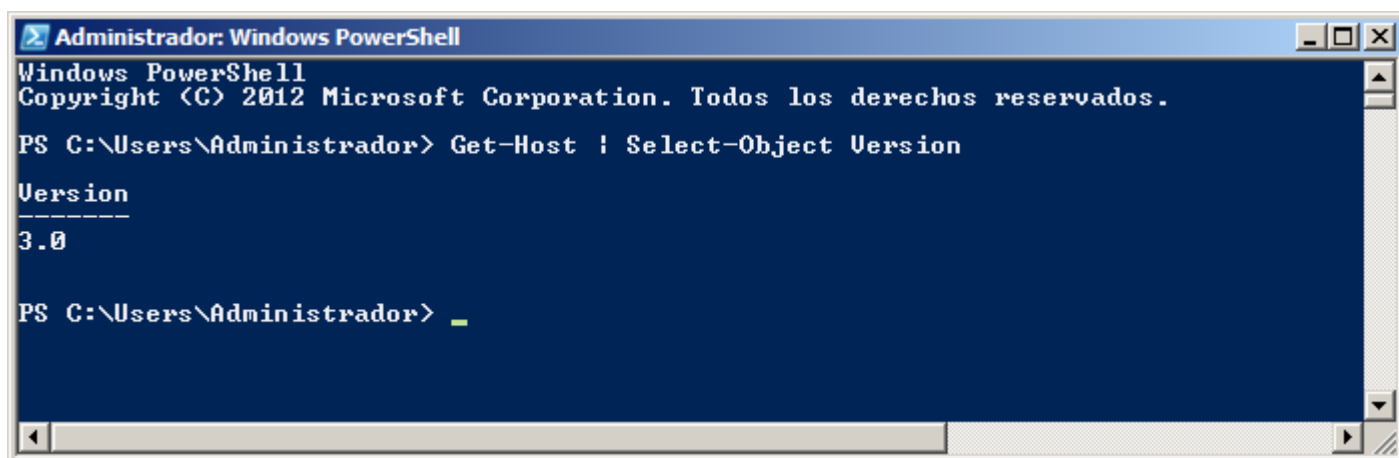


Figura 2-4. Versión 3.0 de PowerShell.

En los siguientes puntos iremos desgranando las funcionalidades básicas de PowerShell que nos van a permitir administrar nuestro sistema de una manera más potente.

2.5. Cmdlets básicos

2.5.1. Estructura de los cmdlets

Como norma general, los cmdlets están formados por un verbo (describe la acción a realizar) y un nombre (indica el objeto sobre el que se aplica la acción) unidos mediante un guión, como por ejemplo los que hemos visto anteriormente: `Get-Host` o `Select-Object`.

La lista de cmdlets es extensísima, por lo que es imposible revisar ni tan siquiera una parte importante de los mismos en un curso de estas características. Sin embargo, sí que veremos los más habituales y sobre todo, aquellos que nos pueden permitir encontrar el cmdlet que necesitamos, ver qué propiedades tiene dicho cmdlet y averiguar cómo se utiliza.

2.5.2. Get-Command

El cmdlet `Get-Command` sirve para obtener un listado de los cmdlets que existen en PowerShell. Si lo introducimos veremos una larga secuencia de cmdlets lo cual no es muy útil. En cambio, sí puede resultarnos útil cuando utilizamos la opción `-Noun` para averiguar los cmdlets relacionados con un determinado elemento del sistema:

```
> Get-Command -Noun process
```

El cmdlet anterior nos muestra un listado de los cmdlets y funciones disponibles en el sistema para gestionar procesos.

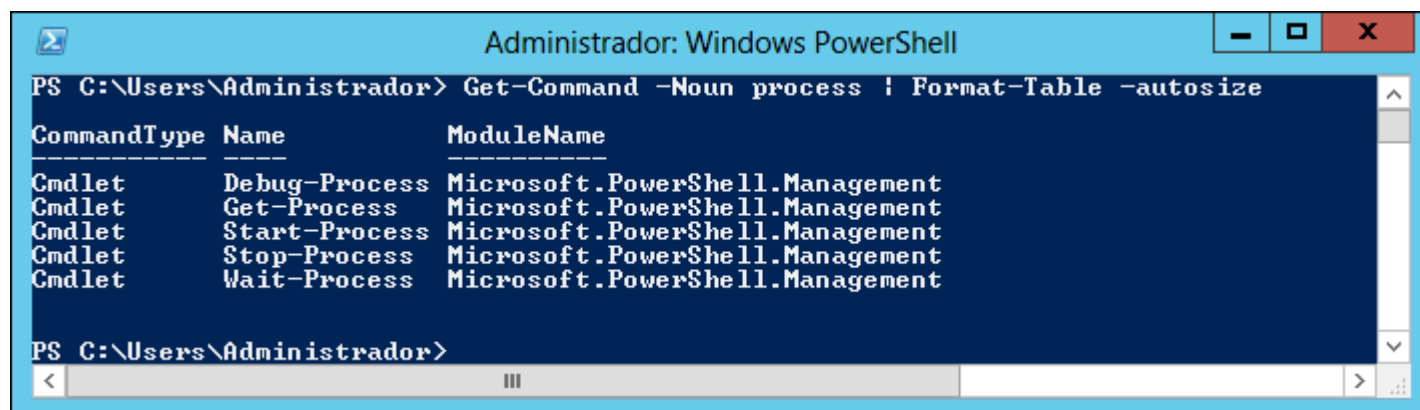


Figura 2.1-1. Resultado de la ejecución de `Get-Command -Noun`.

Sin embargo, si no solo quisiéramos hallar cmdlets, sino por ejemplo también ficheros `.exe`, utilizaríamos la opción `-Name`:

```
> Get-Command -Name *file*
```

```

PS C:\Users\Administrador> Get-Command -Name *file* | Format-Table -autosize
CommandType Name                                     ModuleName
-----
Function    Close-SmbOpenFile                          SmbShare
Function    Disable-NetIPHttpsProfile                  NetworkTransition
Function    Enable-NetIPHttpsProfile                  NetworkTransition
Function    Get-FileIntegrity                         Storage
Function    Get-NetConnectionProfile                  NetConnection
Function    Get-NetFirewallProfile                    NetSecurity
Function    Get-NfsOpenFile                           NFS
Function    Get-RDFileTypeAssociation                  RemoteDesktop
Function    Get-SmbOpenFile                           SmbShare
Function    Get-SupportedFileSystems                  Storage
Function    Publish-BCFileContent                     BranchCache
Function    Repair-FileIntegrity                      Storage
Function    Revoke-NfsOpenFile                        NFS
Function    Set-FileIntegrity                         Storage
Function    Set-NetConnectionProfile                  NetConnection
Function    Set-NetFirewallProfile                    NetSecurity
Function    Set-RDFileTypeAssociation                  RemoteDesktop
Cmdlet      Add-BitsFile                              BitsTransfer
Cmdlet      Get-AppLockerFileInformation               AppLocker
Cmdlet      New-ADDCCloneConfigFile                    ActiveDirectory
Cmdlet      New-PSSessionConfigurationFile             Microsoft.PowerShell.Core
Cmdlet      Out-File                                   Microsoft.PowerShell.Utility
Cmdlet      Test-PSSessionConfigurationFile             Microsoft.PowerShell.Core
Cmdlet      Unblock-File                              Microsoft.PowerShell.Utility
Application forfiles.exe
Application openfiles.exe

PS C:\Users\Administrador>

```

Figura2.1-2. Resultado de la ejecución de `Get-Command -Name`.

En el ejemplo anterior buscamos todos los elementos (funciones, cmdlets, aplicaciones, etc.) que contienen `file` en su nombre. Como puede observarse en los ejemplos anteriores, pueden utilizarse comodines del tipo `*`. Comprabad como difiere el resultado en la ejecución de:

```
> Get-Command -Noun file
```

y

```
> Get-Command -Name *file
```

2.5.3. Get-Member

El cmdlet `Get-Member` indica las propiedades y métodos soportados por un objeto, veamos cómo puede sernos útil. Supongamos que queramos almacenar en una variable la referencia a un directorio concreto. Como PowerShell trata **todos los elementos como objetos**, utilizaremos el cmdlet `Get-Item` para asignar el objeto directorio a la variable que queremos crear:

```
> $carpeta = Get-Item C:\Users\Administrador
```

Para comprobar que ahora en la variable `$carpeta` no solo tenemos la ruta del directorio, sino todas las propiedades del objeto, escribiremos lo siguiente:

```
> $carpeta | Get-Member
```

```

Administrador: Windows PowerShell
PS C:\Users\Administrador> $carpeta | Get-Member

TypeName: System.IO.DirectoryInfo

Name                MemberType          Definition
-----
Mode                CodeProperty        System.String Mode{get=Mode;}
Create              Method               void Create(), void Create(System.Se
CreateObjRef        Method               System.Runtime.Remoting.ObjRef Creat
CreateSubdirectory  Method               System.IO.DirectoryInfo CreateSubdir
Delete              Method               void Delete(), void Delete(bool recu
EnumerateDirectories Method               System.Collections.Generic.IEnumerab
EnumerateFiles       Method               System.Collections.Generic.IEnumerab
EnumerateFileSystemInfos Method               System.Collections.Generic.IEnumerab
Equals              Method               bool Equals(System.Object obj)
GetAccessControl     Method               System.Security.AccessControl.Direct
GetDirectories       Method               System.IO.DirectoryInfo[] GetDirecto
GetFiles             Method               System.IO.FileInfo[] GetFiles(string
GetFileSystemInfos   Method               System.IO.FileSystemInfo[] GetFileSy
GetHashCode          Method               int GetHashCode()
GetLifetimeService  Method               System.Object GetLifetimeService()
GetObjectData        Method               void GetObjectData(System.Runtime.Se
GetType              Method               type GetType()
InitializeLifetimeService Method               System.Object InitializeLifetimeServ
MoveTo               Method               void MoveTo(string destDirName)
Refresh             Method               void Refresh()
SetAccessControl     Method               void SetAccessControl(System.Securit
ToString             Method               string ToString()
PSChildName          NoteProperty         System.String PSChildName=Administra
PSDrive              NoteProperty         System.Management.Automation.PSDrive
PSIsContainer        NoteProperty         System.Boolean PSIsContainer=True
PSParentPath         NoteProperty         System.String PSParentPath=Microsoft
PSPath               NoteProperty         System.String PSPath=Microsoft.Power
PSPProvider          NoteProperty         System.Management.Automation.Provide
Attributes           Property              System.IO.FileAttributes Attributes
CreationTime          Property              datetime CreationTime {get;set;}
CreationTimeUtc       Property              datetime CreationTimeUtc {get;set;}
Exists                Property              bool Exists {get;}
Extension             Property              string Extension {get;}
FullName              Property              string FullName {get;}
LastAccessTime        Property              datetime LastAccessTime {get;set;}
LastAccessTimeUtc     Property              datetime LastAccessTimeUtc {get;set;}
LastWriteTime         Property              datetime LastWriteTime {get;set;}
LastWriteTimeUtc      Property              datetime LastWriteTimeUtc {get;set;}
Name                  Property              string Name {get;}
Parent                Property              System.IO.DirectoryInfo Parent {get;}
Root                  Property              System.IO.DirectoryInfo Root {get;}
BaseName              ScriptProperty        System.Object BaseName {get=$this.Na
PS C:\Users\Administrador>

```

Figura2.1-3. Propiedades del objeto C:\Users\Administradores.

Veamos cómo podemos aprovechar alguna de las propiedades del objeto anterior. Si nos fijamos en el listado que hemos obtenido, veremos que existe una propiedad denominada `parent`. Si escribimos:

```
> $carpeta.parent
```

La salida será como sigue:

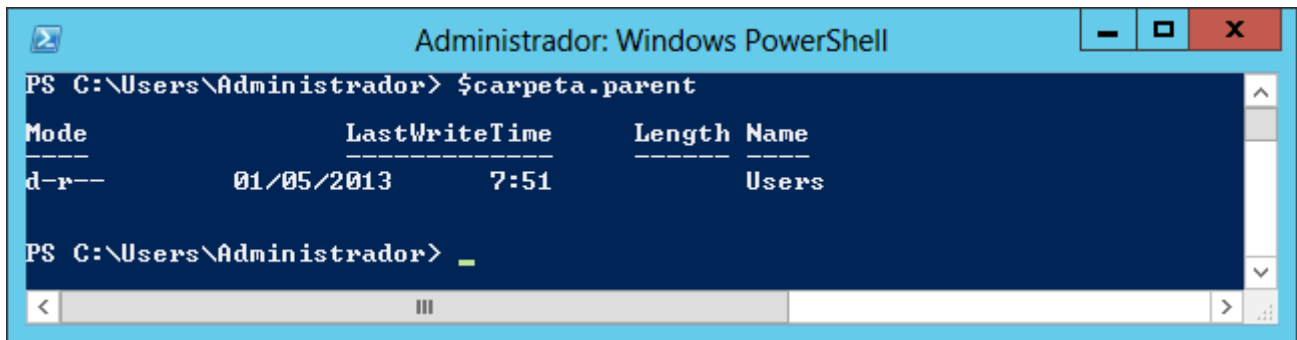


Figura 2.1-4. Propiedad parent de \$carpeta.

Podemos comprobar que se nos muestra el nombre del directorio del que cuelga \$carpeta. Con PowerShell también podemos obtener de la misma manera propiedades del directorio 'padre':

> `$carpeta.parent.parent`

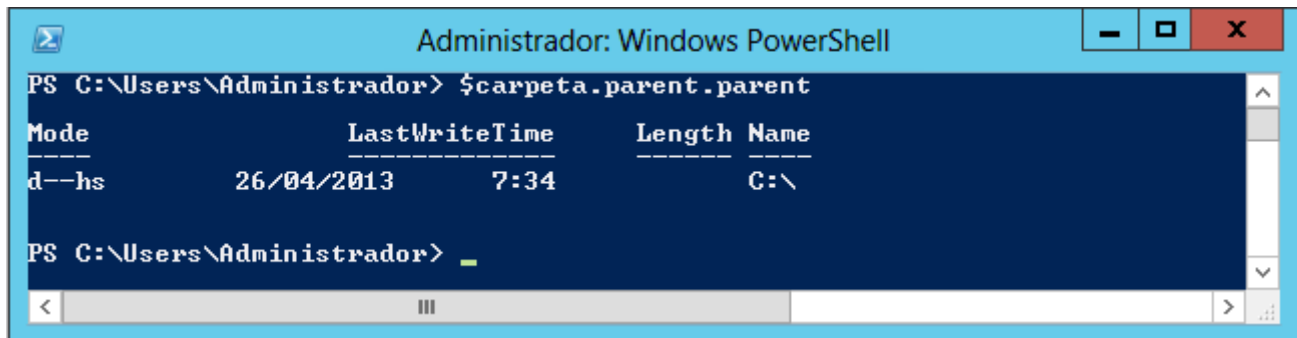


Figura 2.1-5. Propiedad parent de \$carpeta.parent.

Si solo quisiéramos obtener el nombre del directorio de nivel superior, utilizaríamos la propiedad name:

> `$carpeta.parent.name`

Por otra parte, también podemos obtener información adicional, como por ejemplo cuándo se realizó el último acceso al directorio aprovechando la propiedad LastAccessTime:

> `$carpeta.LastAccessTime`

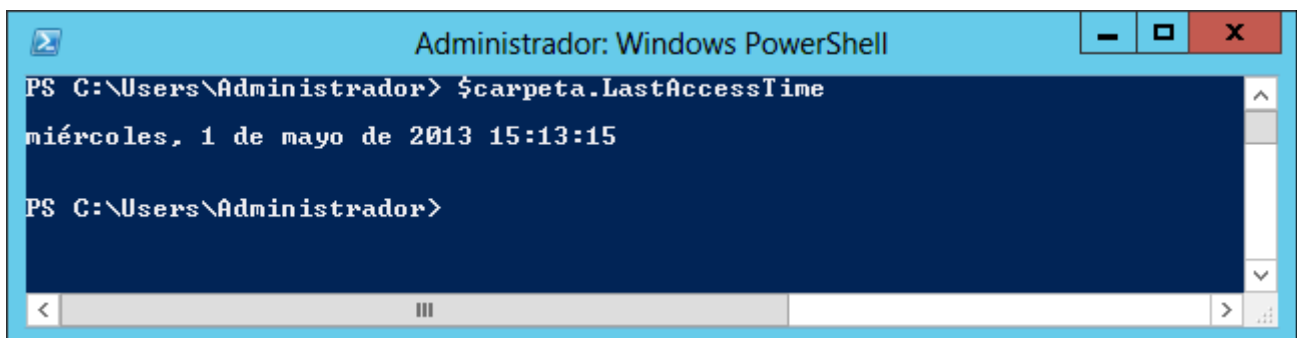


Figura 2.1-6. Fecha del último acceso a \$carpeta.

También podemos realizar consultas sobre el tipo de objeto. En el siguiente ejemplo obtenemos el valor de la propiedad `PsIsContainer`, que nos indica si el objeto es una carpeta, devolviendo `True` o `False`:

```
> $carpeta.PsIsContainer
```

Para finalizar, podemos utilizar el método `delete()` para borrar no solo la referencia al objeto, sino el objeto en sí (**no lo ejecutaremos sobre `C:\Users\Administrador`, ya que es el directorio personal del administrador**):

```
> $carpeta.delete()
```

2.5.4. Get-Help

El cmdlet `Get-Help` nos va a permitir obtener información acerca de cómo utilizar un determinado cmdlet. Antes de empezar a utilizarlo, actualizaremos el módulo de ayuda escribiendo el siguiente cmdlet:

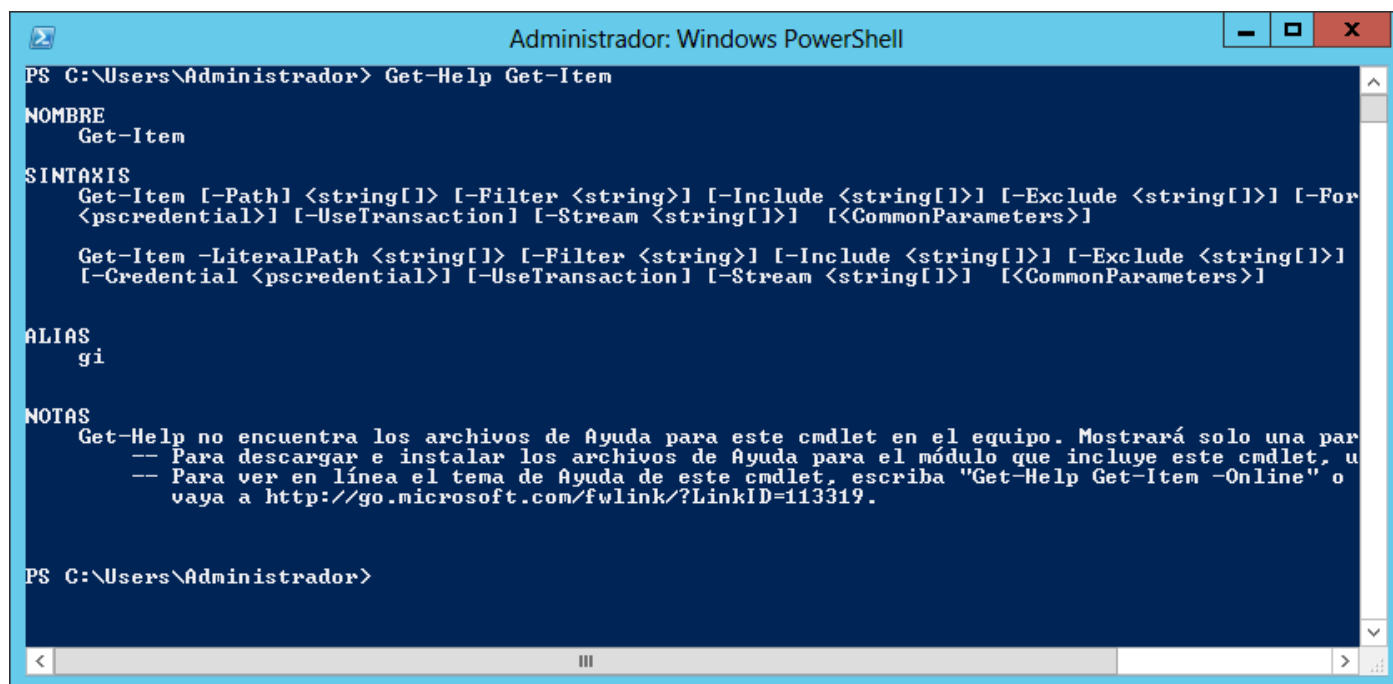
```
> Update-Help
```

Para poder actualizar correctamente el módulo de ayuda, necesitaremos que nuestro equipo tenga conexión a Internet.

La utilización habitual de `Get-Help` sería `Get-Help CMDLET_A_CONSULTAR`, como por ejemplo:

```
> Get-Help Get-Item
```

Cuya salida sería:



```
Administrador: Windows PowerShell
PS C:\Users\Administrador> Get-Help Get-Item
NOMBRE
    Get-Item
SINTAXIS
    Get-Item [-Path] <string[]> [-Filter <string>] [-Include <string[]>] [-Exclude <string[]>] [-For
    <pscredential>] [-UseTransaction] [-Stream <string[]>] [<CommonParameters>]

    Get-Item -LiteralPath <string[]> [-Filter <string>] [-Include <string[]>] [-Exclude <string[]>]
    [-Credential <pscredential>] [-UseTransaction] [-Stream <string[]>] [<CommonParameters>]
ALIAS
    gi
NOTAS
    Get-Help no encuentra los archivos de Ayuda para este cmdlet en el equipo. Mostrará solo una par
    -- Para descargar e instalar los archivos de Ayuda para el módulo que incluye este cmdlet, u
    -- Para ver en línea el tema de Ayuda de este cmdlet, escriba "Get-Help Get-Item -Online" o
    vaya a http://go.microsoft.com/fwlink/?LinkID=113319.
PS C:\Users\Administrador>
```

Figura 2.1-7. Resultado de la ejecución de `Get-Help Get-Item`.

Como se puede ver, se nos ofrece un pequeño resumen de la funcionalidad del cmdlet, la sintaxis con sus opciones, una descripción detallada, otros cmdlets relacionados y finalmente algunas observaciones adicionales.

Una opción sumamente interesante del cmdlet `Get-Help` consiste en la utilización del argumento `-examples` para mostrarnos cómo puede utilizarse un determinado cmdlet:

```
> Get-Help Get-Item -examples
```

Los tres cmdlets revisados hasta ahora (`Get-Command`, `Get-Member` y `Get-Help`) nos van a permitir manejarnos bastante bien con PowerShell sin tener un conocimiento exacto a priori ni del cmdlet a utilizar ni de su sintaxis u opciones. Sin embargo, iremos introduciendo en esta sección y en las posteriores, los cmdlets que nos van a resultar más útiles en las tareas de administración de nuestro sistema.

2.5.5. Get-ChildItem

Habitualmente, en los sistemas Windows, para obtener un listado de los directorios y ficheros, utilizábamos el comando `dir`. Este comando también funciona en PowerShell, pero en realidad es un alias del cmdlet `Get-ChildItem`. De hecho, si escribimos en PowerShell el siguiente cmdlet:

```
> Get-Alias dir
```

Obtendremos algo similar a lo siguiente:

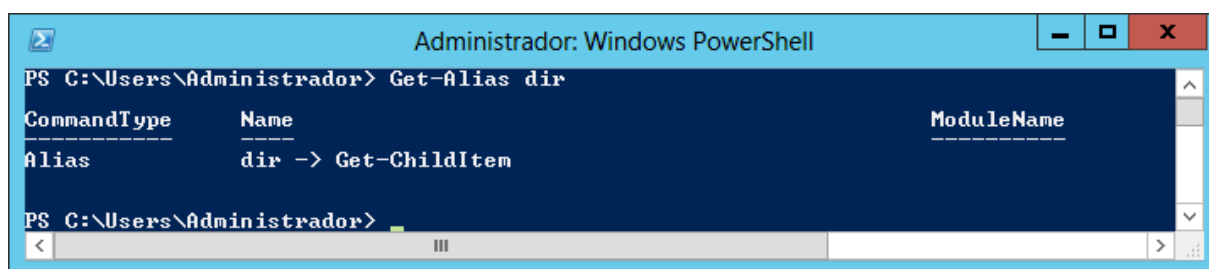


Figura 2.1-7. Cmdlet al que apunta `dir`.

Con `Get-Alias` lo que hemos hecho es obtener el cmdlet correspondiente al alias `dir`. Si queremos obtener un listado de los elementos de un directorio escribiremos:

```
> Get-ChildItem
```

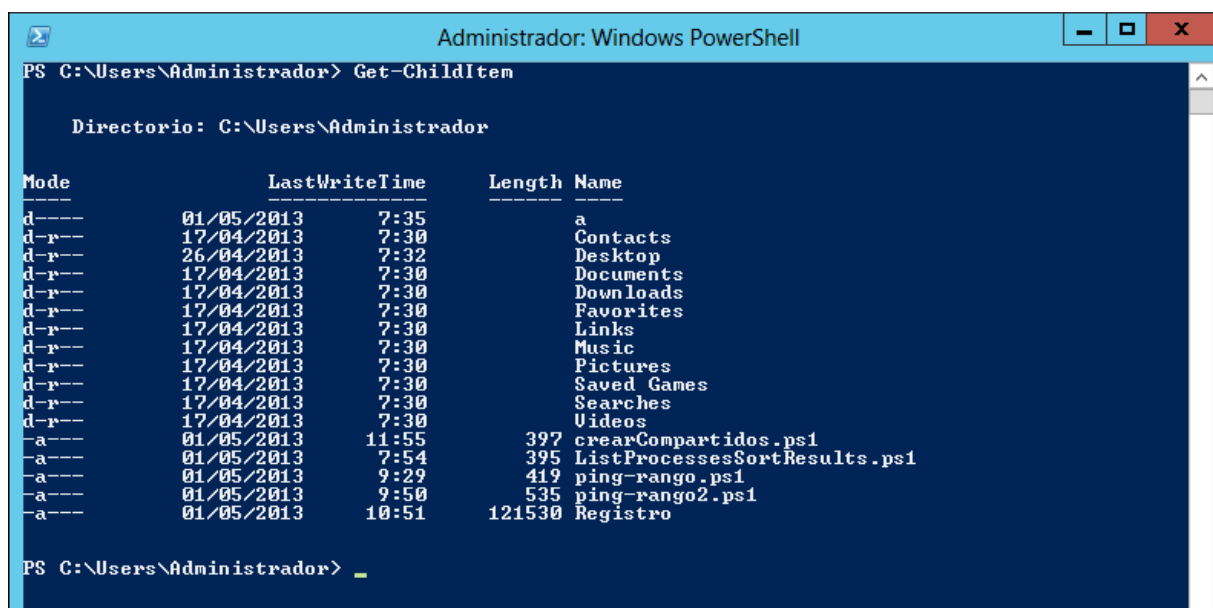


Figura 2.1-8. Elementos del directorio actual.

Si queremos que **también** se muestren los ficheros y directorios del sistema añadiremos la opción `-force`:

```
> Get-ChildItem -Force
```

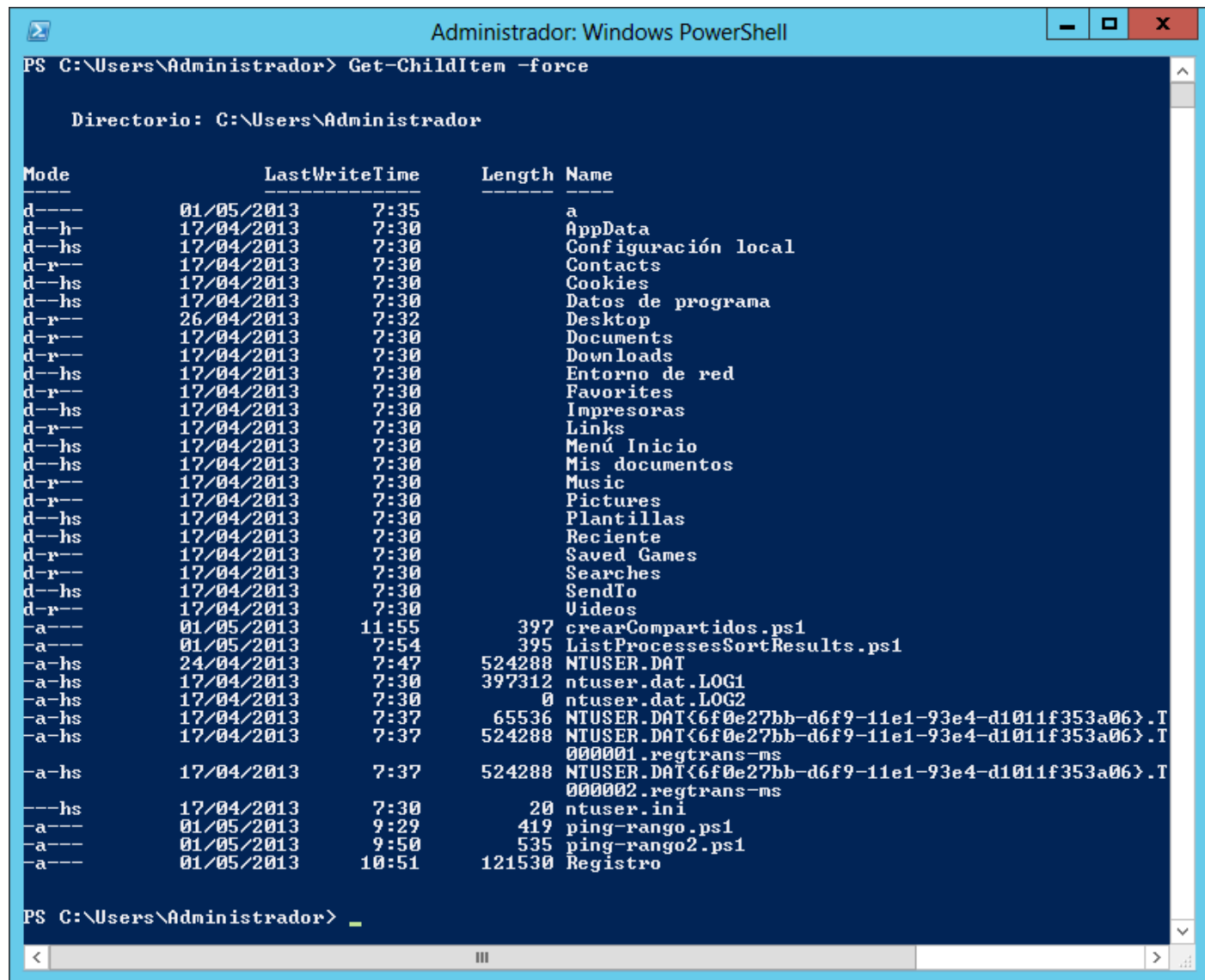


Figura 2.1-9. **Todos** los elementos del directorio actual.

2.5.6. New-Item

El cmdlet `New-Item` nos va a permitir crear una carpeta o un fichero vacío. Para ello tendremos que especificar la ruta (mediante `-path`) y el tipo de elemento (mediante `-itemtype`). Por ejemplo, con la siguiente instrucción crearemos una carpeta llamada `nueva_carpeta` en el directorio actual.:

```
> new-item -path ./nueva_carpeta -itemtype directory
```

Para crear un fichero dentro de la carpeta anterior escribiremos lo siguiente:

```
> new-item -path ./nueva_carpeta/nuevo_fichero -itemtype file
```

2.5.7. Write-Host

Otro cmdlet que utilizaremos muy a menudo será `Write-Host`, el cual nos permite mostrar por pantalla un determinado mensaje, como puede ser texto, o la salida de ejecución de una instrucción o de un script. Por ejemplo, podemos mostrar por pantalla un mensaje clásico:

```
> Write-Host "Hola Mundo"
```

También podemos mostrar la salida de otros cmdlets:

```
> Write-Host "La última vez que se accedió al directorio" $home " fue el " (Get-Item $home).LastAccessTime
```

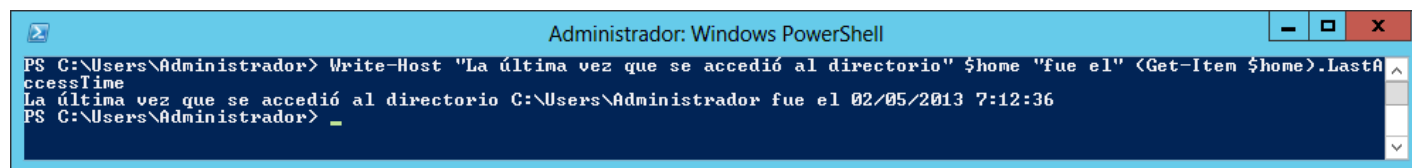


Figura 2.1-10. Salida del comando anterior.

2.6. Variables y Constantes

Variables

En PowerShell, no es necesario declarar las variables antes de utilizarlas, en realidad se declaran cuando se les asigna un valor. Además, las variables deben ir precedidas del símbolo del dólar (\$). Por ejemplo, si escribimos lo siguiente:

```
> $numero = 5
```

habremos creado una variable numérica (llamada `$numero`) al asignarle el valor '5'. Si por ejemplo ejecutamos lo siguiente:

```
> $texto = "Hola Mundo"
```

estaremos creando una variable que contiene la cadena "Hola Mundo".

PowerShell trabaja de una manera bastante eficiente identificando los tipos de datos asignados a las variables. Respecto a la creación de variables, la única salvedad que hay que tener en cuenta reside en que existen una serie de variables del sistema. Algunas de las que utilizaremos a lo largo del tema son:

Nombre	Utilización
<code>\$home</code>	Directorio personal del usuario.
<code>\$args</code>	Contiene los argumentos que se le pasan a un script.
<code>\$?</code>	Contiene el estado (éxito/fallo) de la ejecución de la última instrucción.
<code>\$_</code>	Objeto actual, se utiliza en los filtros <code>Where-Object</code> , <code>ForEach-Object</code> y <code>switch</code> que utilizaremos en los scripts.

Constantes

La declaración de constantes es algo más compleja, ya que requiere la utilización del cmdlet `Set-Variable`, especificando con el argumento `-option` que se trata de una constante. Veamos un ejemplo:

```
> Set-Variable -name CDRM -value 5 -option constant
```

En el ejemplo anterior, con el argumento `-name` le proporcionamos un nombre a la constante, con el argumento `-value` le otorgamos un valor, y finalmente, como se ha explicado anteriormente, con `-option` indicamos que se trata de una constante. Para crear la constante no ha hecho falta incluir el símbolo del dólar, sin embargo, para poder utilizar la constante **sí** será necesario que el símbolo del dólar preceda al nombre de la constante:

```
> Write-Host $CDRM
```

Veamos un ejemplo en el que puede ser interesante la utilización de constantes. La clase WMI `Win32_LogicalDisk` nos permite listar las unidades disponibles en nuestro sistema:

```
> Get-WmiObject -class Win32_LogicalDisk
```

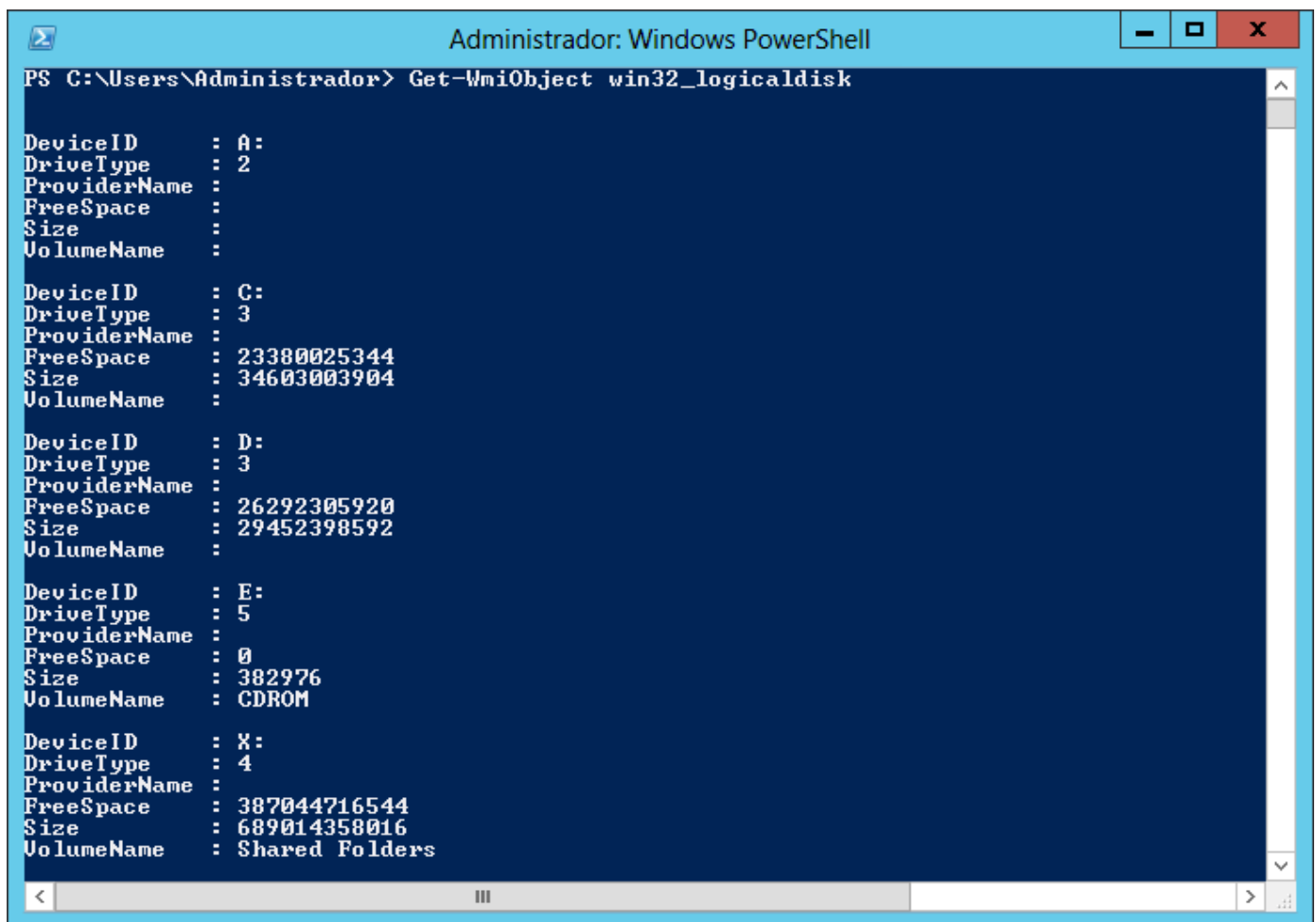
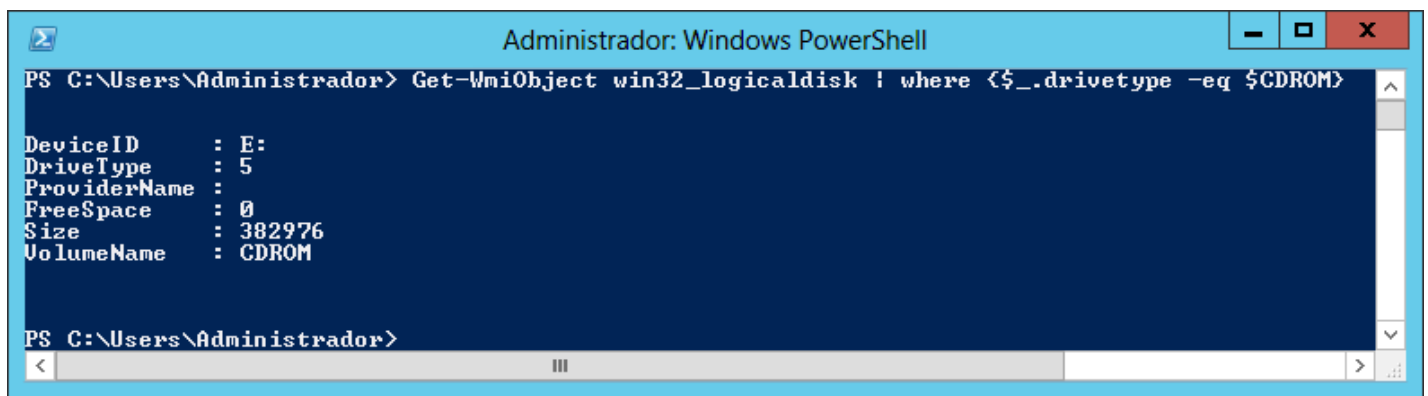


Figura 2.2-1. Unidades dadas de alta en el sistema.

Como se puede apreciar en la figura anterior, el sistema utiliza la propiedad `DriveType` para indicar el tipo de dispositivo. Concretamente el valor de `DriveType` igual a 2 indica que se trata de un dispositivo extraíble, el 3 indica un disco duro local, el 4 una unidad de red, y el 5 una unidad óptica. Por tanto podemos utilizar la constante que hemos creado anteriormente para mostrar las unidades ópticas del sistema:

```
> Get-WMIObject -class win32_logicaldisk | where {$_.drivetype -eq $CDROM}
```



```
Administrador: Windows PowerShell
PS C:\Users\Administrador> Get-WmiObject win32_logicaldisk | where {$_.drivetype -eq $CDROM}

DeviceID      : E:
DriveType     : 5
ProviderName  :
FreeSpace     : 0
Size          : 382976
VolumeName    : CDR0M

PS C:\Users\Administrador>
```

Figura 2.2-2 Unidades ópticas en el sistema.

En la siguiente sección veremos con más detalle las consultas a WMI con `Get-WMIObject` que hemos introducido en este ejemplo.

2.7. Consultas a WMI

Podemos definir WMI (Windows Management Instrumentation) de una manera simplificada como una base de datos que nos permite interactuar con los dispositivos instrumentados, tanto físicos como lógicos del sistema. Esta interacción en realidad consiste en:

1. La obtención de información de dichos dispositivos instrumentados (es decir, que posean unos registros de información a los que WMI pueda acceder).
2. En la escritura de los parámetros de configuración de estos dispositivos.

Quizá se aclare algo más este concepto con la figura 2.3-1, donde se muestra la arquitectura de WMI.



Figura 2.3-1. Arquitectura WMI.

Como se puede ver en la figura anterior, la arquitectura WMI consta de tres actores principales:

1. La aplicación que trata de obtener o escribir información.
2. La infraestructura WMI que es el repositorio de información.
3. El componente físico o lógico que va a ser consultado o administrado -según la dirección del flujo de información-.

WMI está estructurado en clases, algunas de las más utilizadas son:

- Win32_Service: permite obtener las propiedades de cada uno de los servicios.
- Win32_NetworkAdapter: permite configurar la red.
- Win32_Process: permite gestionar los procesos del sistema operativo.
- Win32_Pingstatus: permite obtener los valores devueltos por ping clásico.
- Win32_Share: permite acceder a los datos de gestión de los recursos compartidos.

PowerShell nos va a permitir interactuar con el WMI de una manera muy sencilla: para obtener la información del sistema mediante consultas a WMI lo haremos a través de declaraciones WQL (*WMI Query Language*) del tipo `select * from Clase_WMI`.

Por ejemplo, como acabamos de ver, si utilizáramos una consulta como `select * from Win32_Share` con el cmdlet `Get-WMIObject` podríamos obtener todas las propiedades de cada uno de los recursos compartidos por el equipo:

```
> Get-WMIObject -query "select * from Win32_Share"
```

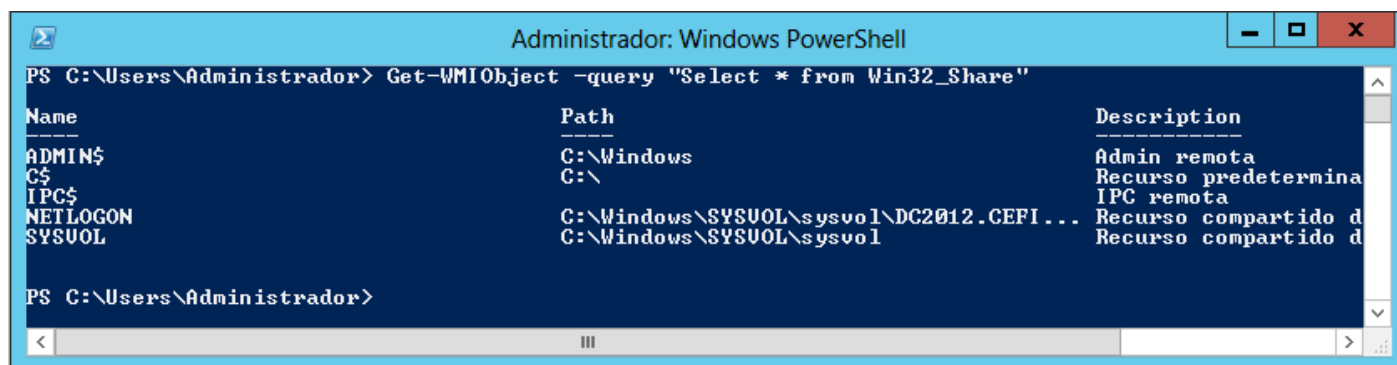


Figura 2.3-1. Recursos compartidos.

Como se puede observar, la sentencia WQL es muy similar a las SQL, ya que de hecho WQL es un subconjunto del anterior.

De todas maneras, veremos esta manera de trabajar con más detalle a lo largo de los siguientes puntos en los que iremos introduciendo ejemplos completos de utilización de algunas de las clases WMI que hemos comentado y que nos permitirán acceder a información del sistema de una manera muy potente y relativamente sencilla.

3. Scripts con PowerShell

Por defecto, la utilización de scripts no está permitida en PowerShell (sic) por cuestiones de seguridad. Sin embargo, podemos configurar esa política para permitir su ejecución según diferentes niveles de seguridad. Concretamente, en PowerShell se contemplan 4 niveles:

Nivel	Significado
Restricted	No se puede ejecutar ningún script.
AllSigned	Se ejecutarán los scripts que esté firmados por un autor de confianza
RemoteSigned	Los scripts descargados de Internet deben estar firmados por un autor de confianza
Unrestricted	Se pueden ejecutar todos los scripts, sin embargo los scripts descargados de Internet solicitarán permiso para su ejecución

Visto lo anterior, si queremos, en un entorno de pruebas, ejecutar nuestros scripts, deberemos escribir lo siguiente en la consola:

```
> Set-ExecutionPolicy Unrestricted
```

Para conocer el nivel de restricción de la ejecución que tenemos establecido, podemos escribir el siguiente cmdlet:

```
> Get-ExecutionPolicy
```

A partir de ahora estableceremos el nivel de seguridad en `Unrestricted`.

3.1. Creación de Scripts

Para crear un script no tenemos más que abrir un editor de texto, e ir añadiendo los cmdlets, alias, funciones, etc., que necesitemos para realizar las tareas que tengamos previstas. Este fichero deberá guardarse con la extensión `.ps1`, para indicarle al sistema que se trata de un script de PowerShell. Veamos un ejemplo:

```
#Este es mi primer script con PowerShell

Clear-Host

Write-Host "El directorio actual es:"
Get-Location | Write-Host

Write-Host "Cambiemos de directorio..."

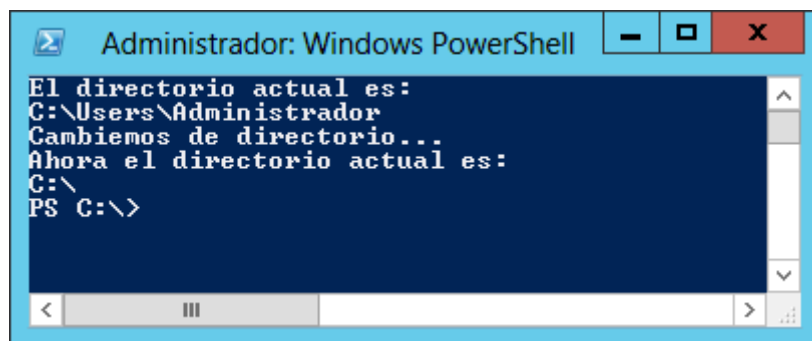
Start-Sleep -second 1
Set-Location -path C:\

Write-host "Ahora el directorio actual es:"
Get-Location | Write-Host
```

Lo guardaremos como `primer_script.ps1` y lo ejecutaremos desde PowerShell de la siguiente manera (el script debe hallarse en el directorio actual de trabajo):

```
> ./primer_script.ps1
```

Obtendremos la siguiente salida:



```
Administrador: Windows PowerShell
El directorio actual es:
C:\Users\Administrador
Cambiemos de directorio...
Ahora el directorio actual es:
C:\
PS C:\>
```

Figura 3- Resultado de ejecutar `primer_script.ps1`.

Revisemos ahora el código que hemos escrito. En primer lugar introducimos un comentario con `#`. El primer cmdlet (`Clear-Host`), se limita a 'limpiar' la pantalla. Existe un alias para este cmdlet que es `cls`. Como vimos en secciones anteriores, `Write-Host` muestra un texto por pantalla, y por tanto `Get-Location | Write-Host` muestra por pantalla el directorio actual. `Start-Sleep` simplemente establece un retraso en la ejecución del siguiente conjunto de instrucciones (un segundo: `-second 1`), y finalmente con `Set-Location -path C:\` cambiamos el directorio de trabajo al que se le indique como argumento `-path`, en el ejemplo `C:\`.

3.2. Entorno de Scripts Integrado

Los sistemas Windows que tienen instalado PowerShell, tienen incorporada una característica que nos va a permitir escribir nuestros scripts de una manera mucho más cómoda y sencilla que utilizar el editor de textos: el Entorno de Scripts Integrado (ISE). Es una aplicación que nos va ayudar en la tarea de escribir scripts mediante funcionalidades como la coloración sintáctica, la visualización de los números de líneas, el depurador integrado, la ayuda en modo gráfico y el autocompletado de cmdlets.

Para abrir PowerShell ISE, bastará con escribir su nombre en la ventana de búsqueda, y automáticamente nos aparecerán dos accesos directos (figura 3.1-1), el primero abrirá directamente PowerShell ISE, y el segundo lo abrirá en modo de compatibilidad con x86.

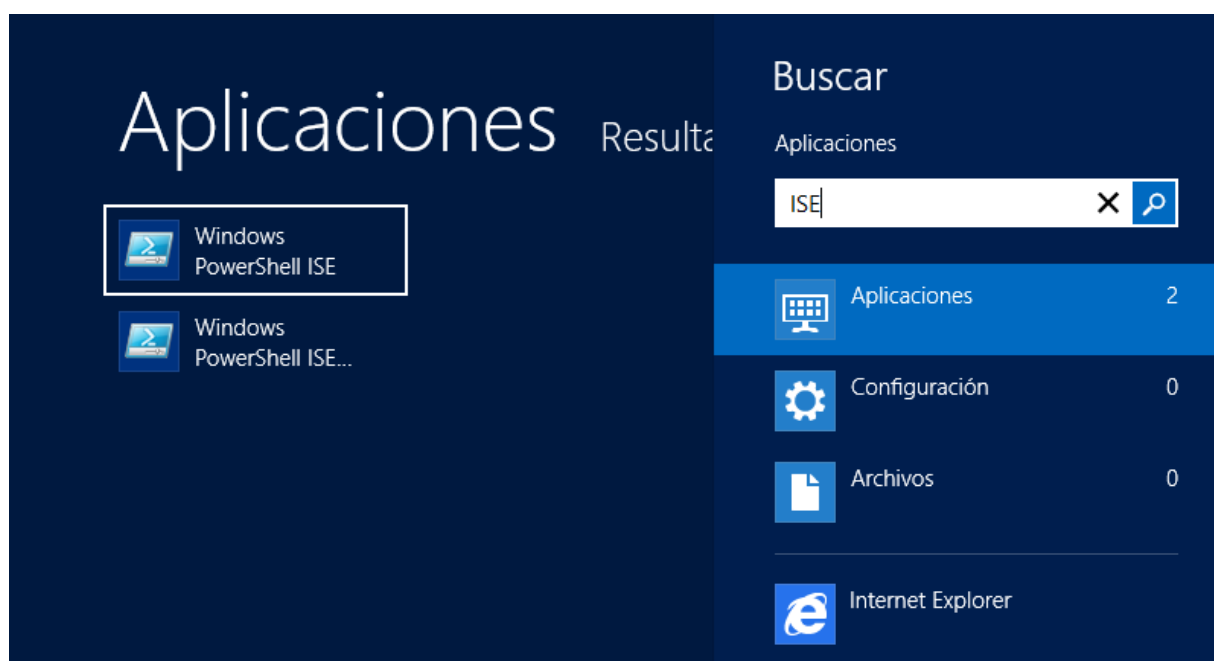


Figura 3.1-1. Acceso a PowerShell ISE.

Al abrirlo aparecerá una ventana como la de la figura 3.1-2, cuyas partes principales consisten en un menú de edición, depuración y ejecución en la parte superior, un editor en la parte principal, una consola en la parte inferior y un panel de cmdlets en el lado derecho.

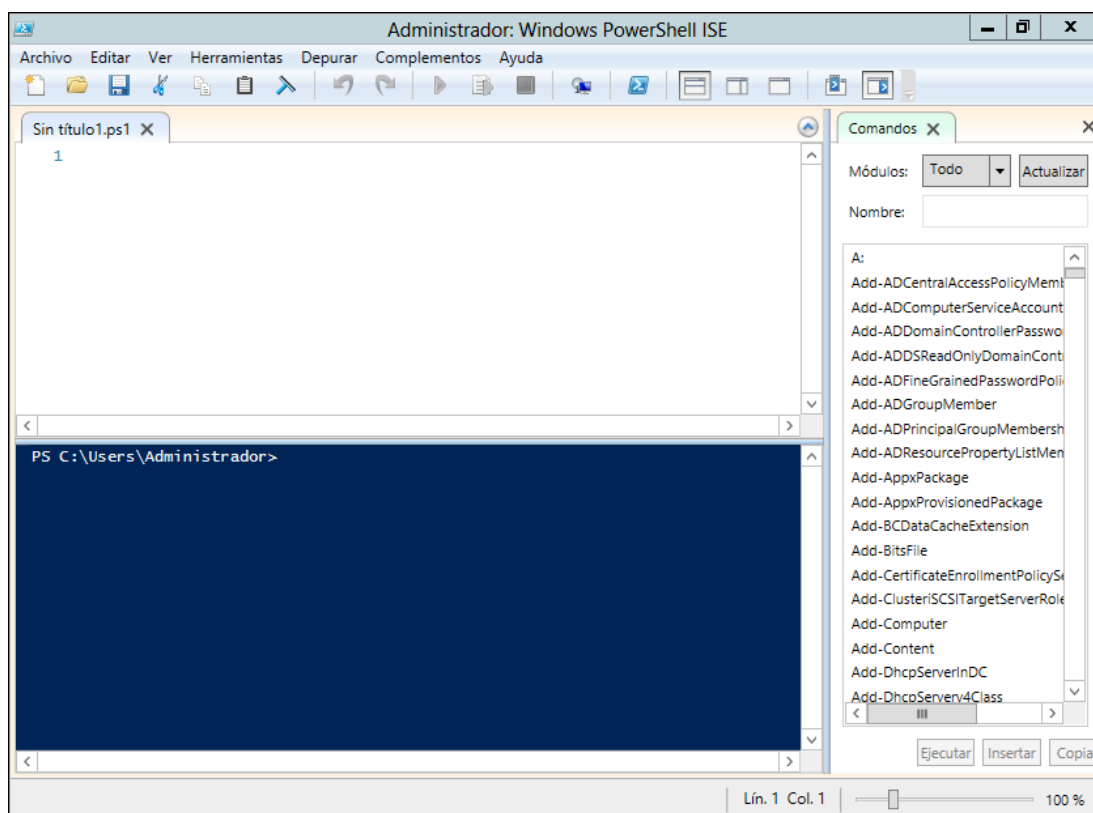


Figura 3.1-2. PowerShell ISE.

Como se indicó anteriormente, a medida que vamos desarrollando nuestro trabajo, se van numerando las líneas, aparece la coloración sintáctica y además se autocompletan los cmdlets (3.1-3).

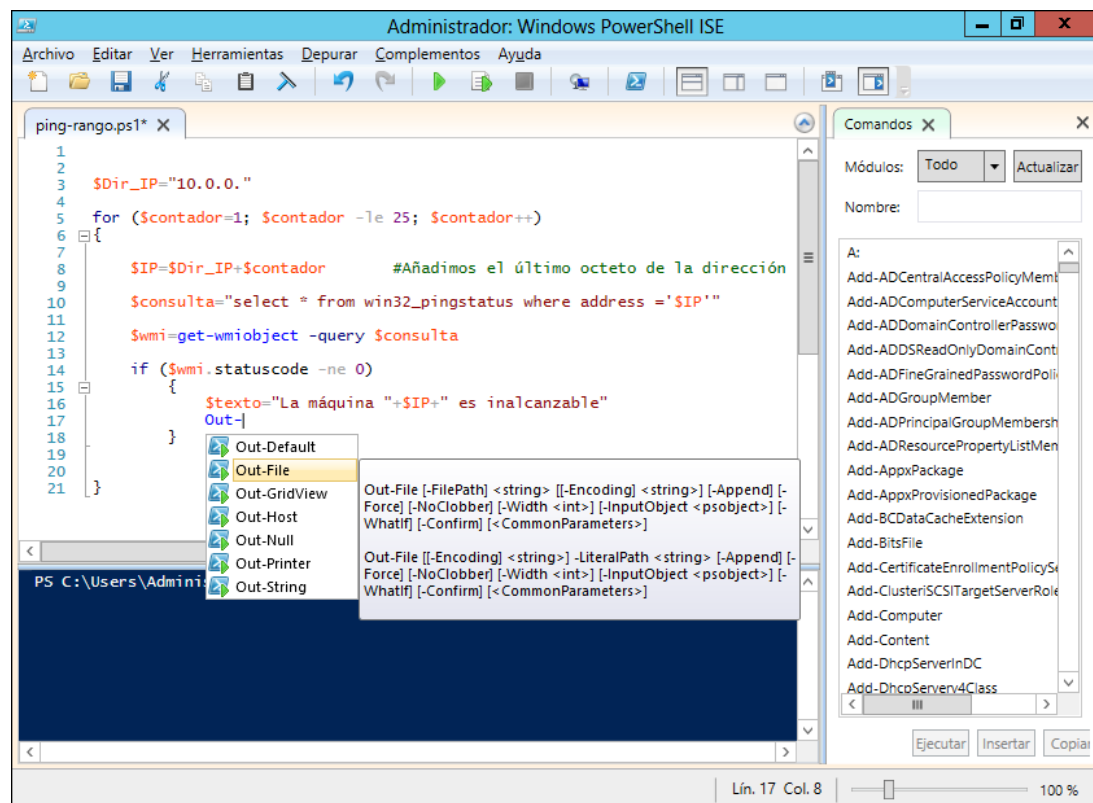


Figura 3.1-3. Algunas de las funcionalidades de ISE.

En el panel de la ayuda de los cmdlets, podemos seleccionar uno de todos los que existen en PowerShell, ver sus opciones, e incluso configurar su funcionamiento, e ISE nos proporcionará la sintaxis correspondiente a la funcionalidad que le hemos indicado y nos mostrará por consola el resultado de su ejecución.

Por ejemplo, en la figura 3.2-4 podemos ver cuál sería la sintaxis que tendríamos que utilizar si quisiéramos que el cmdlet Write-Host mostrara por pantalla una determinada cadena de texto con fondo azul claro y texto rojo.

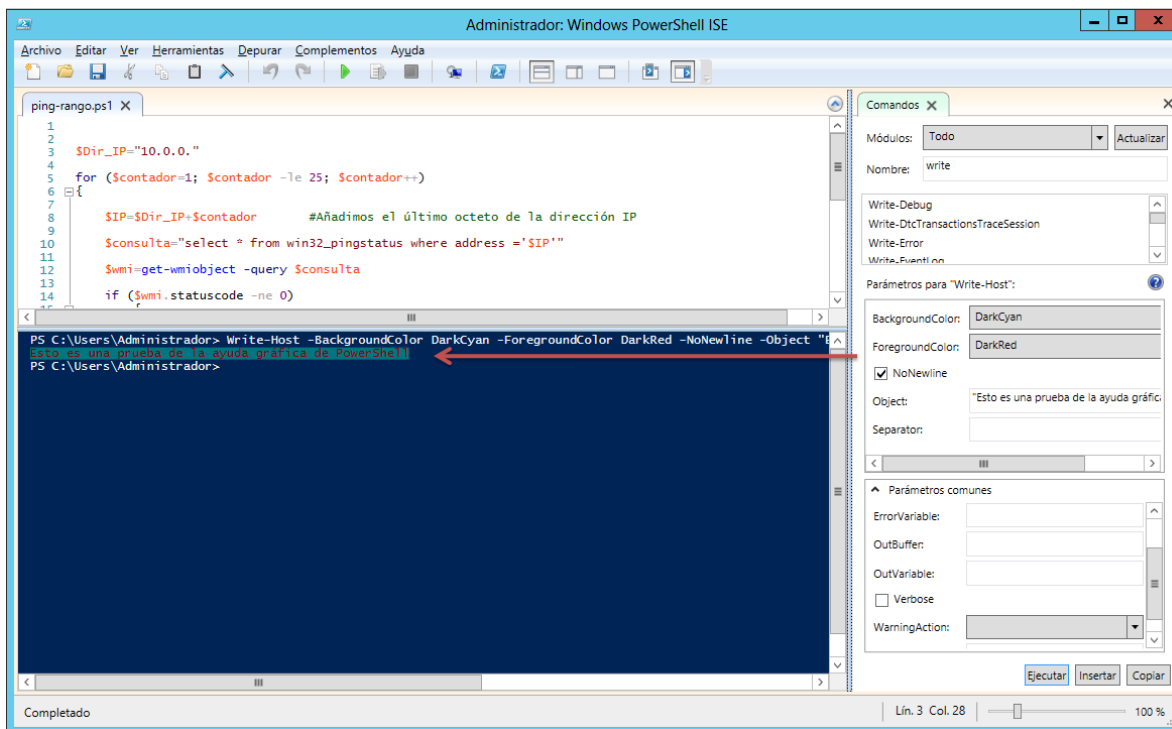


Figura 3.1-4. Ayuda gráfica de los cmdlets.

Otra de las funcionalidades que puede sernos de especial utilidad consiste en la ejecución de los scripts por pasos y la introducción de puntos de interrupción para poder controlar los puntos donde se producen errores, pudiendo detener la ejecución y controlar el valor de las variables mediante la consola (figura 3.1-5).

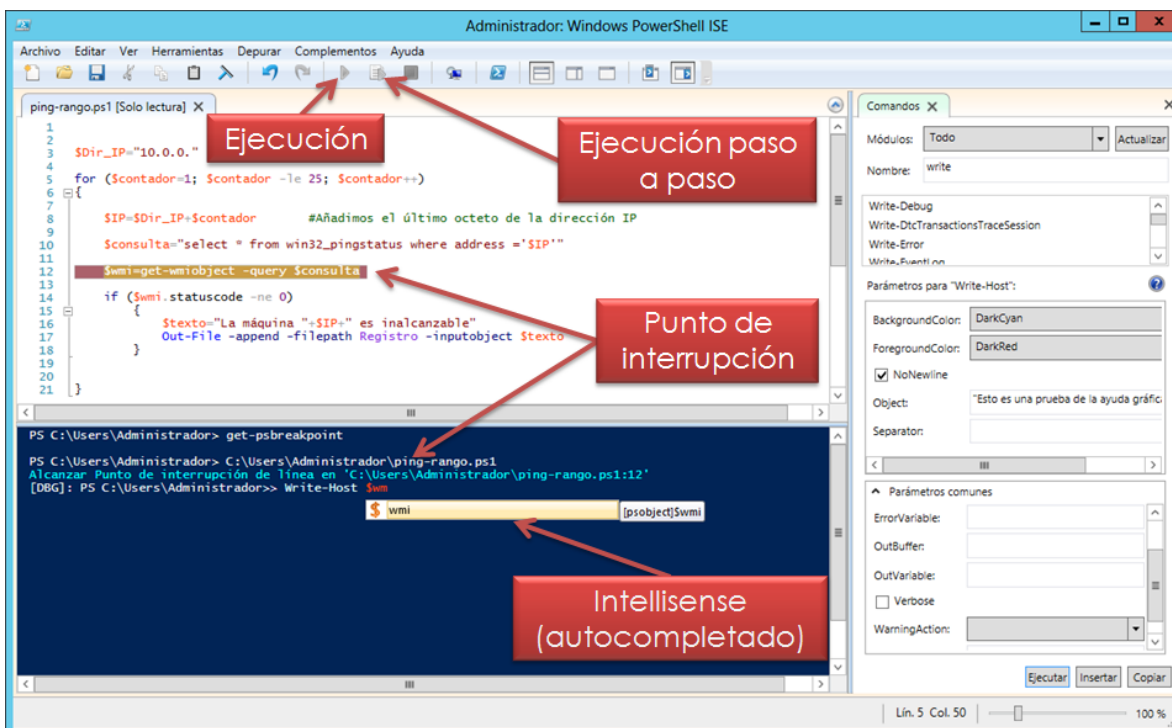


Figura 3.1-5. Ejecución controlada de los scripts.

3.3. Estructuras Condicionales

3.3.1. if-elseif-else

Como sabemos, una estructura condicional de tipo `if` permite bifurcar el flujo de ejecución en función del valor de una condición. La sintaxis clásica sería como sigue:

```
if (condicion)
{
    #instrucciones si se cumple la condición
}
else
{
    #instrucciones si NO se cumple la condición
}
```

Para evaluar las condiciones se suelen utilizar operadores que recuerdan a los empleados en los diferentes shells de GNU/Linux. Los más habituales son:

Operador	Significado
<code>-eq</code>	Igual a
<code>-lt</code>	Menor que
<code>-gt</code>	Mayor que
<code>-le</code>	Menor o igual que
<code>-ge</code>	Mayor que
<code>-ne</code>	Distinto de
<code>-not y !</code>	No lógico
<code>-and</code>	Y lógico
<code>-or</code>	O lógico

Por tanto, si escribimos algo como:

```
if ($variable -gt 5)
{
    Write-Host "El valor de la variable es mayor que 5"
}
else
{
    Write-Host "El valor de la variable es menor o igual a 5"
}
```

se nos mostrará por pantalla el primer texto o el segundo, en función del valor de `$variable`.

La estructura anterior puede hacerse algo más compleja mediante la introducción de la cláusula `elseif`, que permite comprobar otra condición, si la anterior no se ha cumplido. Veamos un ejemplo.

```
if ($variable -gt 5)
{
    Write-Host "El valor de la variable es mayor que 5"
}
elseif ($variable -gt 3)
{
    Write-Host "El valor de la variable es mayor que tres y menor o igual a 5"
}
else
{
    Write-Host "El valor de la variable es menor o igual a 3"
}
```

En el ejemplo anterior, si `$variable` es mayor que cinco se ejecuta la primera instrucción, si vale 4 o 5 se ejecuta la segunda instrucción, y finalmente, si no corresponde a ninguno de los otros casos, se ejecuta la última instrucción.

3.3.2. switch

Como en otros lenguajes de programación/scripting podemos utilizar la instrucción `switch` para ejecutar unas instrucciones u otras, en función del valor de una variable.

Veamos el siguiente ejemplo extraído y adaptado de 'Ed Wilson, Windows PowerShell. Scripting Guide, Microsoft Press, 2008'. En él, dependiendo de la propiedad `domainrole` de la clase `Win32_computersystem`, podemos conocer la función que desempeña el equipo dentro de la red. Mediante el empleo de `switch` podemos 'traducir' el valor de esta propiedad:

```
$WMI=Get-WMIObject WIN32_ComputerSystem
Write-Host "El equipo " $WMI.name " actúa como: "
switch($WMI.domainrole)
{
    0 { Write-Host "Equipo aislado"}
    1 { Write-Host "Cliente miembro del dominio"}
    2 { Write-Host "Servidor aislado"}
    3 { Write-Host "Servidor miembro del dominio"}
    4 { Write-Host "Controlador de dominio de respaldo"}
    5 { Write-Host "Controlador de dominio principal"}
    default { Write-Host "Ideterminado"}
}
```

3.4. Bucles

Las estructuras repetitivas como los bucles son de gran importancia al escribir scripts, ya que permiten ejecutar un determinado número de veces (o hasta que se cumpla una condición) una serie de instrucciones, que pueden ir desde la creación/eliminación de usuarios, al testeo de equipos de la red, o el movimiento masivo de ficheros y directorios.

3.4.1. While

Con `while`, mientras se satisfaga la condición del bucle, se repetirá el conjunto de instrucciones que especifiquemos.

La sintaxis genérica es tan sencilla como:

```
while (condicion)
{
    #instrucciones mientras se cumpla la condición
}
```

Comprobemos su funcionamiento mediante un ejemplo. Escribiremos un pequeño script que muestre por pantalla la tabla de multiplicar (hasta el 10) de un número que se introducirá por el teclado:

```
Write-Host "Introduzca el número del cual desea mostrar la tabla de multiplicar"
$numero=Read-Host
$contador = 0

while($contador -le 10)
{
    $resultado=$contador*$numero
    Write-Host $numero " * " $contador " es igual a" $resultado
    $contador++
}
```

Si nos fijamos en el script anterior, en primer lugar mostramos un mensaje por pantalla `Write-Host`, a continuación leemos el número introducido por el usuario (`Read-Host`), inicializamos `$contador` a 0, y mientras este sea **menor o igual a 10** generaremos la tabla de multiplicar. Es interesante darse cuenta de que PowerShell admite los incrementos y los decrementos del tipo `++` y `--` respectivamente.

El ejemplo anterior se encuentra disponible [aquí](#).

3.4.2. Do-While

Una estructura muy similar a la anterior es `do-while`, cuya sintaxis es:

```
do
{
    #instrucciones mientras se cumpla la condición
}
while (condicion)
```

A diferencia del caso anterior, en este tipo de bucle, la condición se evalúa al final, por tanto, aunque la condición no se cumpla, el bucle se ejecutará **al menos una vez**. Dependiendo de la solución que necesitemos, optaremos por un tipo de bucle u otro.

Caso Práctico con While

Supongamos que en nuestra red tenemos 25 máquinas con IPv4 estática y queremos monitorizar de una manera muy burda si se hallan conectados. Para ello queremos escribir un pequeño script que haga un ping de manera correlativa a las 25 máquinas desde 192.168.0.1 hasta 192.168.0.25, de manera indefinida, y que cuando no consigamos hacer ping a una de ellas, se almacene la información en un fichero de registro.

```
#ping con while

$Dir_IP="192.168.0."
$parar=0
$contador=1

while ($parar -eq 0)
{

    $IP=$Dir_IP+$contador #Añadimos el último octeto de la dirección IP
    $consulta="select * from Win32_PingStatus where address ='$IP'"
    $wmi=Get-WMIObject -query $consulta

    if ($wmi.statuscode -ne 0)
    {
        $texto="La máquina "+$IP+" es inalcanzable a las "+(get-date).toShortTimeString()
        Out-File -append -filepath Registro -inputobject $texto
    }

    if ( $contador -lt 25)
    {
        $contador++
    }
    else
    {
        $contador=1
    }
}
```

Las principales novedades que hemos introducido en este script se limitan a la obtención del resultado del ping con la consulta `select * from Win32_PingStatus where address ='$IP'`, la adaptación del formato de la fecha con `(get-date).toShortTimeString()` y la escritura sobre un fichero con `Out-File`. Al introducir el parámetro `-append` en el cmd `Out-File`, lo que hacemos es **añadir** el contenido (`-inputobject`) al fichero (`-filepath`), no sobrescribirlo.

3.4.3. For

Con este tipo de bucle, ejecutamos un conjunto de instrucciones un número de veces fijado *a priori*.

La sintaxis es como sigue:

```
for (valor_inicial_del_contador; condicion_de_parada; incremento_del_contador)
{
    #instrucciones mientras no se llegue al final del contador
}
```

Veamos cómo podríamos implementar de nuevo la tabla de multiplicar, pero esta vez con un `for`.
`Write-Host "Introduzca el número del cual desea mostrar la tabla de multiplicar"`
`$numero=Read-Host`

```
for($contador=0; $contador -le 10; $contador++)
{
    $resultado=$contador*$numero
    Write-Host $numero " * " $contador " es igual a" $resultado
}
```

En este caso la sintaxis queda ligeramente más compacta que en el caso anterior con `while`. Como podemos ver, el primer argumento del `for` consiste en el valor inicial de la variable de control del bucle (`$contador=0`). A continuación establecemos la condición en la que finalizará este, en el ejemplo actual cuando `$contador` sea menor o igual a 10. El último de los argumentos de `for` consiste en el incremento de

la variable de control, o dicho de otra manera, cómo va a modificarse `$contador` en cada iteración del bucle.

Caso Práctico con For

A continuación resolveremos el caso práctico que planteamos con el `while` en el que hacíamos ping a un rango de direcciones IP para saber si los equipos estaban conectados, pero en lugar de plantear un bucle sin fin, únicamente hará ping a cada máquina una vez.

```
#ping con while
$Dir_IP="192.168.0."
for ($contador=1; $contador -le 25; $contador++)
{
    $IP=$Dir_IP+$contador #Añadimos el último octeto de la dirección IP
    $consulta="select * from Win32_PingStatus where address ='$IP'"
    $wmi=Get-WMIObject -query $consulta

    if ($wmi.statuscode -ne 0)
    {
        $texto="La máquina "+$IP+" es inalcanzable"
        Out-File -append -filepath Registro -inputobject $texto
    }
}
```

3.4.4. ForEach-Object

El cmdlet `ForEach-Object` permite recorrer **secuencialmente** una estructura de datos, como puede ser un vector, un fichero o las distintas propiedades de un objeto. Una de sus principales utilidades reside en recorrer todas las líneas de un archivo, donde puede haber usuarios, máquinas o cualquier otro elemento del sistema, y realizar operaciones sobre dichos elementos.

Una forma muy habitual de utilizarlo en PowerShell es como salida de una tubería en cuya entrada se encuentra un vector, de esta manera con `ForEach-Object` realizaremos una serie de tareas sobre cada uno de los elementos de ese array. En el ejemplo que se muestra a continuación, se carga con `Get-Content` el contenido del fichero `maquinas.txt` en la variable `$direcciones_IP`. A continuación `ForEach-Object` va leyendo cada uno de los elementos de `$direcciones_IP` almacenando en la variable `$maquina` el valor del **elemento actual** (`$_`), esto es 10.0.1.1. para la primera pasada, 10.0.1.2 para la segunda pasada, etc. A continuación, con `Get-WMIObject` hacemos ping al equipo con dirección `$maquina`, y en función de la propiedad `statuscode` sabremos si la máquina responde o no, en cuyo caso lo indicaremos en el fichero `registro.txt`.

```
$direcciones_IP = Get-Content -path maquinas.txt
$direcciones_IP | foreach-object{
    $maquina=$_
    $estado=Get-WMIObject -query "select * from Win32_PingStatus where address =
'$maquina'"
    if ($estado.statuscode -ne 0)
    {
        $texto="La máquina "+$maquina+" es inalcanzable"
        Out-File -append -FilePath Registro.txt -InputObject $texto
    }
}
```

3.5. Argumentos y parámetros

3.5.1. Argumentos

Un porcentaje muy elevado de los scripts que escribamos requerirán que se les pasen valores al invocarlos. Estos valores se llaman **argumentos** y su utilización podría ser algo así:

```
> ./script_a_ejecutar argumento1 argumento2 ... argumentoN
```

Una vez dentro del script podremos utilizar estos argumentos mediante la variable del sistema `$args` la cual consiste en un vector donde cada posición corresponde a un argumento. Así `$args[0]` contiene el primer argumento, `$args[1]` contiene el segundo, etc.

Podemos aprovechar el cmdlet `ForEach-Object` que vimos en la sección 4.3.3 para ejemplificar el manejo de los argumentos:

```
#Script argumentos.ps1
$contador = 0
$args | ForEach-Object{
    Write-Host "El argumento número" $contador "es" $_
    $contador++
}
```

En el ejemplo anterior, en lugar de mostrar los argumentos mediante `$args[0]`, `$args[1]`, etc., como a priori no sabemos el número de argumentos del que dispondremos, lo que hacemos es recorrer con `ForEach-Object` cada uno de los elementos del vector de argumentos, y mostrarlos utilizando `$_`, que correspondía al elemento actual en el que se hallaba `ForEach-Object`.

3.5.2. Parámetros

Una manera mucho más ordenada de enviar valores a un script consiste en la utilización de parámetros, en los que asignamos un valor concreto a una variable de la que podrá disponer el script. La sintaxis genérica sería como sigue:

```
> ./script_a_ejecutar -parametro1 valor1 -parametro2 valor2 .... -parametroN valorN
```

Todo esto se verá mucho mejor con un ejemplo. Supongamos que queremos escribir un script que necesitará como parámetros la dirección IP de un equipo, y el nombre del mismo. Para poder obtenerlos adecuadamente el script deberá tener una línea de definición de parámetros como esta:

```
param([string]$IP, [string]$maquina)
```

Donde `[string]` define el tipo de datos que se espera en la variable, aunque como ya hemos visto, la asignación de tipos de datos a las variables funciona muy bien de manera automática en PowerShell, por lo que no es imprescindible hacer la declaración expresa del tipo de la variable.

Por tanto la llamada al script sería, por ejemplo, así:

```
./script_parametros -IP "192.168.0.1" -maquina "W7-A"
```

Una de las ventajas de utilizar parámetros, es que no hace falta respetar el orden de introducción de los mismos, ya que el valor va precedido del nombre del mismo. Por tanto, la llamada anterior sería equivalente a esta:

```
./script_parametros -maquina "W7-A" -IP "192.168.0.1"
```

Podemos comprobar el funcionamiento de las llamadas anteriores con un script tan simple como el siguiente:

```
param([string]$IP, [string]$maquina)
Write-Host "El valor del parámetro IP es" $IP
Write-Host "El valor del parámetro máquina es" $maquina
```

También podemos definir un valor por defecto a los parámetros, por si el usuario no lo indica al invocar al script. Para ello debe proporcionarse un valor a la variable en el momento de la definición. Por ejemplo:

```
param([string]$IP="127.0.0.1", [string]$maquina)
```

En este caso una llamada como la siguiente:

```
./script_parametros -maquina "W7-A"
```

no produciría error en el siguiente script, ya que tiene una dirección IP por defecto:

```
param([string]$IP="127.0.0.1", [string]$maquina)
Write-Host "El valor del parámetro IP es" $IP
Write-Host "El valor del parámetro máquina es" $maquina
```

3.6. Funciones

Como en la gran mayoría de los lenguajes, en PowerShell podemos (y debemos para hacer nuestro código más robusto e inteligible) utilizar funciones, las cuales no son más que un conjunto de instrucciones que puede reutilizarse repetidas veces durante la ejecución del script.

La declaración de las funciones más básica sería así:

```
function nombre (argumentos)
{
param (lista_de_parámetros)
    #instrucciones
}
```

Caso Práctico con Funciones

Como hemos visto anteriormente, aunque hacer ping con la clase Win32_PingStatus es muy potente, también es un proceso de redacción muy pesado. Supongamos que en nuestro script vamos a realizar varias llamadas a Win32_PingStatus, podríamos pensar en crear una función que tomase como parámetro de entrada la dirección IP, y cuyo resultado fuese un valor indicando si el proceso ha tenido éxito o no. Dicha función podría ser como sigue:

```
function falso_ping
{
    param([string]$IP)
    $estado=get-WMIObject -query "select * from Win32_PingStatus where address = '$IP'"
    Return $estado.statuscode
}
```

En la penúltima línea se ha incluido Return para que la función devuelva un valor cuando sea llamada. En este caso devolverá el estado del ping. El script completo que podría utilizar la función anterior sería el siguiente:

```
#Script ejemplo_funciones.ps1
#Este script contiene una función que hace ping y devuelve 0 si todo
#ha ido bien
function falso_ping #Función que devuelve 0 si el ping a una IP ha sido correcto
{
    param([string]$IP)
    $estado=get-WMIObject -query "select * from Win32_PingStatus where address = '$IP'"
    Return $estado.statuscode
}
#Parte principal del script
Write-Host "Vamos a hacer ping con una función"
$direccion=Read-Host "Introduzca la dirección IP(v4) a la que quiere hacer ping"
if ( (falso_ping -IP $direccion) -eq 0)
{
    Write-Host "La máquina" $direccion "está en marcha"
}
else
{
    Write-Host "La máquina" $direccion "no está accesible"
}
```

4. Compartición de Recursos

Para obtener información acerca de los recursos compartidos utilizaremos la clase WMI `Win32_Share`, junto con el cmdlet `Get-WmiObject`. Para obtener un listado de todos los recursos compartidos por el sistema escribiremos lo siguiente:

```
> Get-WmiObject -class win32_share
```

Obteniendo algo similar a lo siguiente:

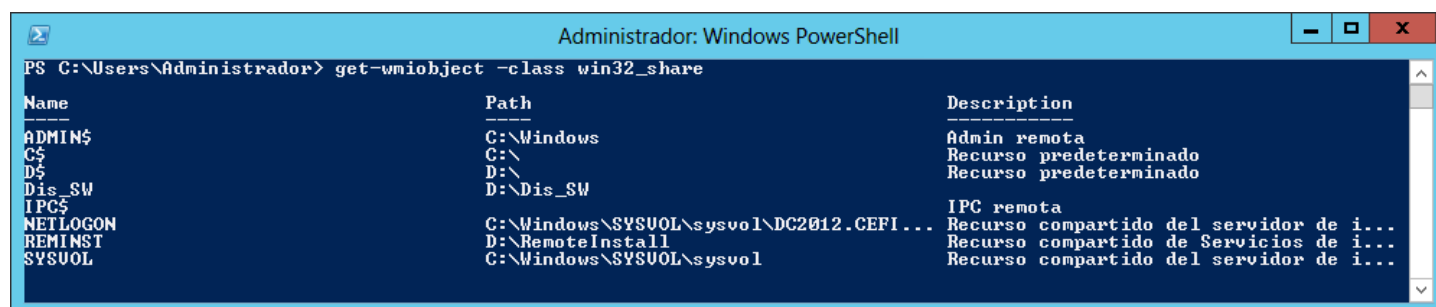


Figura 4-1. Recursos compartidos en el sistema.

Como en casos anteriores, si queremos conocer las propiedades de una clase WMI, ejecutaremos la siguiente instrucción:

```
> Get-WmiObject -class Win32_Share | Get-Member
```

Obteniendo un listado como el siguiente:

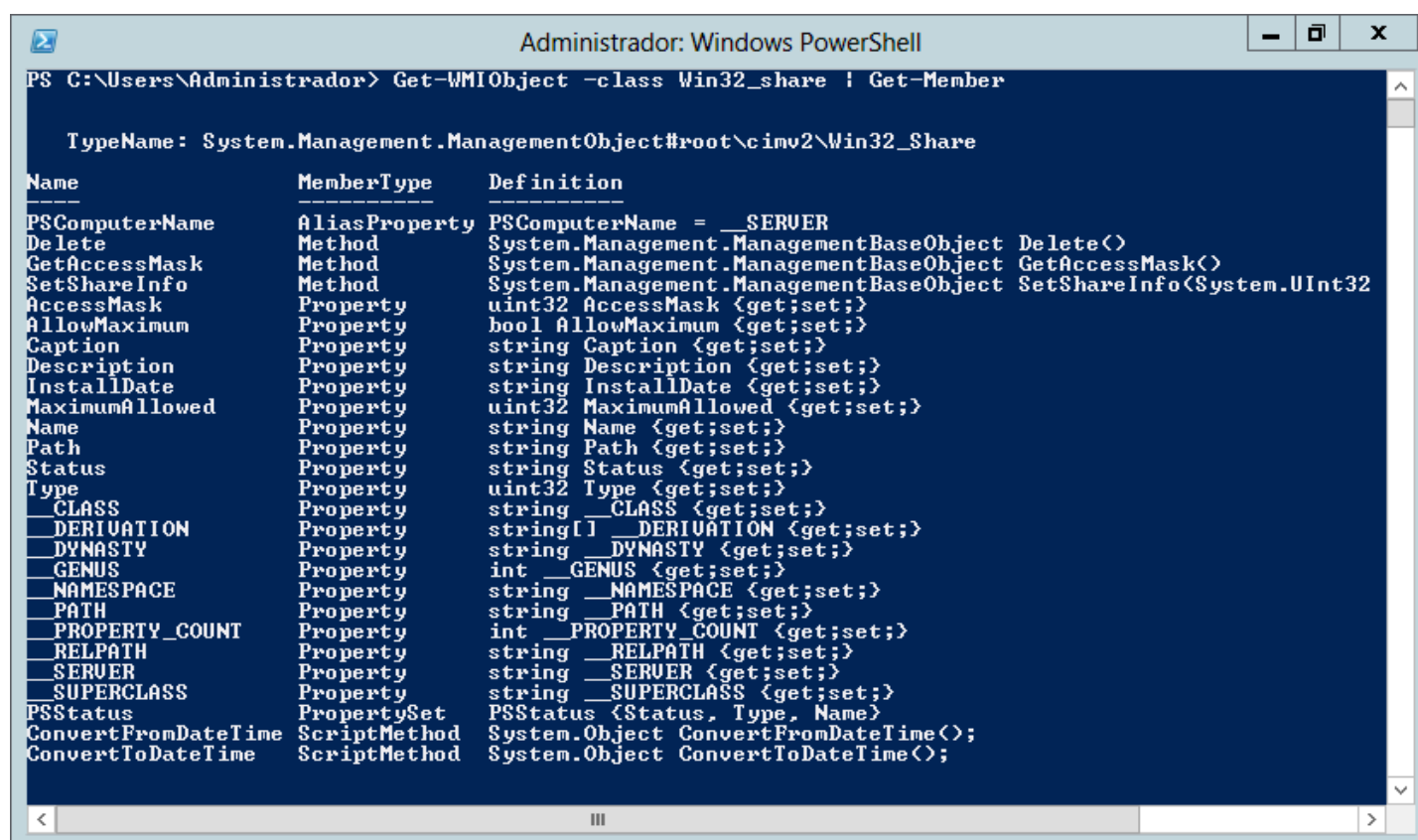


Figura 4-2. Propiedades de la clase win32_share.

Si queremos obtener un listado de los recursos compartidos no administrativos (generados automáticamente por el sistema) escribiremos:

```
> Get-WMIObject -class win32_share -filter "type < '4'"
```

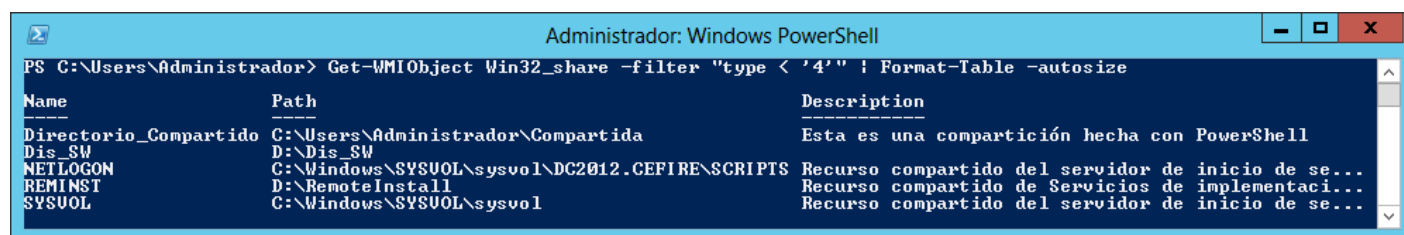


Figura 4-3. Recursos compartidos no administrativos.

La codificación de los distintos tipos de recursos se recoge en la siguiente tabla (obtenida de [Technet](#)):

Valor	Significado
0 (0x0)	Unidad de disco
1	Cola de Impresión
2	Dispositivo
3	IPC
2147483648	Unidad de Disco Administrativa
2147483649	Cola de Impresión Administrativa
2147483650	Dispositivo Administrativo
2147483651	IPC Administrativo

Los tres aspectos más importantes que pueden administrarse en la compartición de recursos son el número máximo de usuarios permitidos, la descripción del recurso y la configuración de seguridad.

Para crear un recurso compartido, utilizaremos el método `create` de la clase `Win32_Share` utilizando el tipo `[wmiClass]`. Este método admite cuatro parámetros:

- La ruta del directorio a compartir.
- El nombre con el que se compartirá el recurso.
- El tipo de recurso a compartir (0 para carpetas).
- El número máximo de usuarios simultáneos admitidos.
- La descripción del recurso compartido.

Supongamos que queremos compartir un directorio que se halla en la ruta `C:\Administrador\carpeta`. Declaremos las variables que nos definen las propiedades del recurso compartido:

```
$ruta="C:\Administrador\carpeta"
$clase_WIM="Win32_Share"
$usuarios=5
$nombre_compartido="Directorio Compartido"
$descripcion="Esta es una compartición hecha con PowerShell"
$tipo=0 #Tipo 0 es para directorios
```

El siguiente paso consiste en utilizar el método `create` de la clase `Win32_Share` con las variables que acabamos de crear.

```
$objetoWMI=[wmiClass]$clase_WIM
$error_devuelto=$objetoWMI.create($ruta, $nombre_compartido, $tipo, $usuarios,
$descripcion)
Write-Host $error_devuelto.returnValue
```

Como el método `create` devuelve un código de error, lo capturaremos con la variable `$error_devuelto` para poder averiguar la causa del fallo en caso de que este se produzca. En el siguiente [enlace](#) encontraréis una relación de los errores que devuelve el método anterior con su significado.

Si comprobamos desde la interfaz gráfica las propiedades de uso compartido de `C:\Administrador\carpeta` veremos que efectivamente se ha habilitado la compartición, con el nombre de recurso compartido que habíamos establecido, el número de usuarios simultáneos limitado a 5 y el comentario que habíamos redactado.

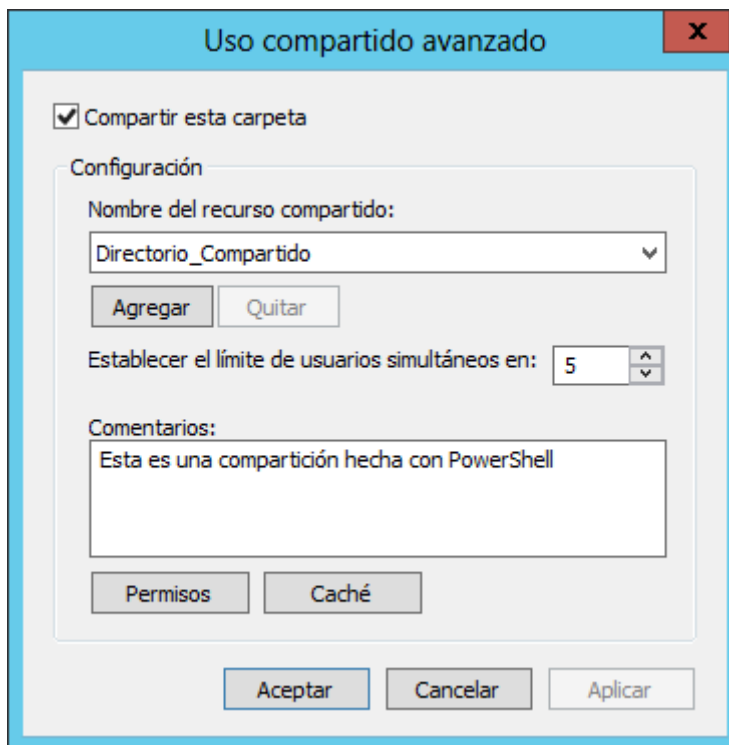


Figura 4-4. Aspectos administrables de la compartición de directorios.

Aprovechando todo lo que hemos visto (y alguna cosa más), podemos redactar un script al que podamos introducir como parámetros la ruta (relativa) de una nueva carpeta para que el script la cree en el directorio actual si no existe (`if (!(Test-Path $ruta))`) y la comparta, el nombre del recurso compartido, el número de usuarios simultáneos máximo y una descripción. Dicho script quedaría como sigue:

```
#Script crear_y_compartir_recurso.ps1
#Versión: 3.0
#Definimos los parámetros que requerirá el script.
#Tanto $usuarios como $descripcion tienen valores por defecto,
#por si no se introducen por parte del usuario.
param($ruta, $nombre_compartido, $usuarios=10, $descripcion="Creado automáticamente con PowerShell")
#Ahora comprobamos si existe la carpeta, y si el resultado
#es negativo, la creamos
$ruta=(Get-Location).path+"\ "+$ruta #Creamos la ruta absoluta a partir de la relativa
if ( !(Test-Path $ruta) )
{
    New-Item -path $ruta -type directory #Creamos el directorio
}
$clase_WIM="Win32_Share"
$objetoWMI=[wmiClass]$clase_WIM
$error_devuelto=$objetoWMI.create($ruta, $nombre_compartido, 0, $usuarios, $descripcion)
if ( $error_devuelto.returnValue -ne 0)
{
    Write-Host "Se ha producido el error" $error_devuelto.returnValue "en la compartición del recurso"
}
```

Si ejecutamos el anterior script por ejemplo con la llamada que se muestra a continuación, veremos que efectivamente se crea la carpeta (si no existía antes), y establece las opciones de compartición definidas en la llamada:

```
> ./crear_y_compartir_recurso -ruta "Nueva_carpeta" -nombre_compartido "Nueva"
```

4.1. Eliminación de Carpetas Compartidas

Del mismo modo que utilizábamos el método `create` para compartir carpetas, ahora utilizaremos el método `delete` para eliminar la compartición de dichos recursos.

La sintaxis será como sigue:

```
$clase_WIM="Win32_Share"
$objetoWMI=Get-WmiObject -Class $clase_WIM -filter "Name='Nombre_de_red_del_recurso'"
$objetoWMI.delete()
```

Veamos este método en un script. Concretamente escribiremos un script que **tome como parámetros el nombre de red del recurso compartido** y elimine dicha compartición. Además, aprovechando la orientación a objetos, recuperará la ruta de la carpeta compartida en el sistema local y nos preguntará si deseamos eliminarla. El script quedaría como sigue:

```
#Script eliminar_compartidos.ps1

#Version 3.0
#
#Este script acepta como parámetro el nombre de red
#de una carpeta compartida y elimina la compartición
#También pregunta si se desea eliminar la carpeta del sistema
param($nombre)
#En primer lugar mostramos los recursos compartidos del sistema
Get-WmiObject -query "Select * from Win32_share"
Write-Host "Compruebe que efectivamente existe el recurso compartido"
Start-Sleep -s 5 #Detenemos la ejecución 5 segundos

#Obtenemos el objeto correspondiente al recurso compartido
$clase_WMI="Win32_Share"
$objetoWMI=Get-WmiObject -Class $clase_WMI -filter "Name='$nombre'"

Write-Host "¿Desea eliminar también la carpeta del sistema? [S/N]"
$sn=Read-Host

if ($sn -eq "S")
{
    Remove-Item -Path $objetoWMI.path
}

#Finalmente eliminamos la compartición
$objetoWMI.delete()

#Mostramos de nuevo el listado de recursos para comprobar
#que todo se ha ejecutado correctamente
Get-WmiObject -query "Select * from Win32_share"
Write-Host "Compruebe que ya no existe el recurso compartido"
```

5. Gestión de Archivos

5.1. Guardar Datos en un Fichero

Como hemos visto anteriormente, el cmdlet utilizado para guardar información en un fichero es `Out-File`, el cual tiene una funcionalidad muy parecida a los operadores de redirección (`>` y `>>`). Los parámetros más habituales son:

- `-FilePath`: Indica el archivo de destino.
- `-Append`: Añade el contenido al archivo.
- `-InputObject`: Indica el objeto a escribir en el archivo

Además, el cmdlet anterior acepta la entrada de datos mediante una tubería, por ejemplo podríamos escribir lo siguiente:

```
> Get-Date | Out-File $home\fecha.txt -Append
```

Lo cual sería equivalente a:

```
> Out-File $home\fecha.txt -Append -InputObject (Get-Date)
```

5.2. Recuperar Datos de un Fichero de Texto

`Get-Content` permite leer el contenido de un determinado fichero de texto, almacenándolo en una estructura en forma de vector bidimensional con la que es muy fácil trabajar.

Veamos un ejemplo, supongamos que tenemos un fichero con nombres de usuario. Leámoslo con `Get-Content` y lo almacenémoslo en la variable `$usuarios`:

```
> $usuarios=Get-Content nombres.txt
```

Ahora podemos acceder a las diferentes líneas del archivo, por ejemplo:

```
> Write-Host $usuarios[2]
```

```
antonio
```

De la misma manera podemos utilizar el cmdlet `ForEach-Object` para recorrerlo línea a línea:

```
$usuarios=Get-Content nombres.txt
$usuarios | foreach-object{
    Write-Host $_
}
```

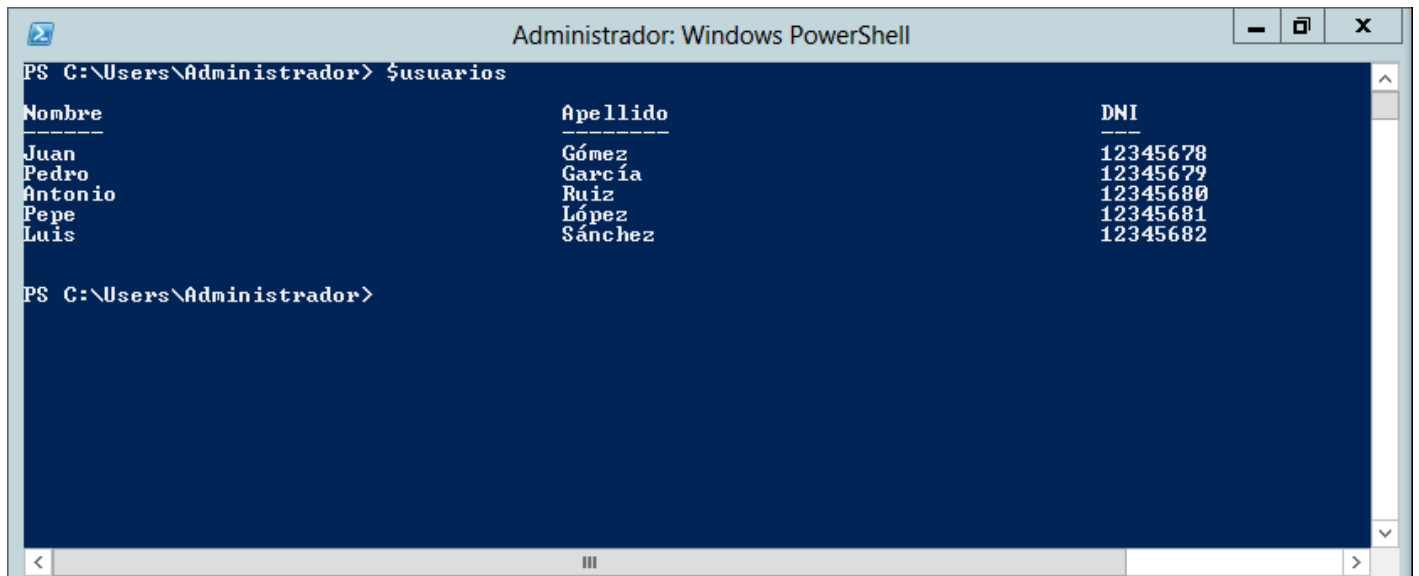
5.3. Recuperar Datos de un Fichero de CSV

Los archivos de tipo CSV (*Comma Separated Values*) tienen separados los campos mediante comas, además una característica que vamos a poder explotar es que la primera fila suele corresponder a una cabecera que nos indica el tipo de información que contiene dicho campo. En el fichero `nombres.csv` podéis encontrar un ejemplo de este tipo de archivo.

Podemos almacenarlo en una variable llamada, por ejemplo, `$usuarios` con el cmdlet `Import-Csv`:

```
> $usuarios=Import-Csv nombres.csv
```


Si mostramos el contenido por pantalla escribiendo simplemente `$usuarios`, obtendremos lo siguiente:



```
PS C:\Users\Administrador> $usuarios
```

Nombre	Apellido	DNI
Juan	Gómez	12345678
Pedro	García	12345679
Antonio	Ruiz	12345680
Pepe	López	12345681
Luis	Sánchez	12345682

```
PS C:\Users\Administrador>
```

Figura 5-1. Usuarios del fichero nombres.csv.

Además podemos hacer referencia a los elementos de la siguiente manera:

```
> $usuarios[0].DNI
```

```
12345678
```

Como se puede ver en el ejemplo anterior, podemos utilizar las cabeceras como una propiedad del objeto para acceder a dicho campo.