

Administración de Sistemas en la Cloud

Guía de estudio de PowerShell

Índice

Esquema	3
Ideas clave	4
7.1. Introducción y objetivos	4
7.2. Qué es PowerShell	4
7.3. <i>Cmdlets</i>	5
7.4. <i>Scripts</i>	11
7.5. Expresiones y otros comandos	20
7.6. Referencias bibliográficas	24
A fondo	26
Test	27

GUÍA DE ESTUDIO DE POWERSHELL

QUÉ ES LA SHELL

Un intérprete de comandos del sistema operativo

Es un lenguaje de programación

FUNCIONALIDADES

Modo interactivo y no interactivo

Objetos .NET

Soporta funciones y ayuda integrada

COMANDOS HABITUALES

- ▲ Get-Help
- ▲ Get-Command
- ▲ Get-Member
- ▲ Get-ChildItem
- ▲ Select-Object
- ▲ Sort-Object
- ▲ Where-Object
- ▲ Get-PSDrive
- ▲ Write-Host

SCRIPTS

- ▲ La política de ejecución puede limitar los *scripts* que se pueden ejecutar
- ▲ Se pueden ejecutar en el ámbito de la consola con «.»
- ▲ Incluyen comentarios con # y <# .. #>
- ▲ Expresiones:
 - ForEach
 - For
 - While
 - If-Else

Esquema

7.1. Introducción y objetivos

Este tema es una guía de las funcionalidades básicas de PowerShell. Explicará el concepto de *cmdlet* y qué características comunes tienen. También se explicarán los *cmdlets* básicos y cómo combinarlos para escribir *scripts*. Para finalizar, se mencionarán algunas de las expresiones de programación presentes en PowerShell.

Los **objetivos** que se pretenden conseguir son:

- ▶ Conocer los fundamentos de los conceptos básicos de PowerShell.
- ▶ Aprender a usar los *cmdlets* básicos.
- ▶ Aprender a combinar los *cmdlets* con expresiones del lenguaje.

7.2. Qué es PowerShell

PowerShell es una **consola de línea de comandos de Windows** diseñada especialmente para administradores. Incluye una *shell* interactiva y un entorno de *scripting* al estilo de consolas para Linux como Bash. El entorno interactivo y los *scripts* pueden usarse de manera independiente o en conjunto (Shepard, 2015).

A diferencia de otras *shells*, cuyos comandos devuelven y aceptan texto, [PowerShell](#) está basada en el *framework* .NET y, como tal, devuelve y acepta objetos de .NET. Este cambio de paradigma ofrece funcionalidades muy útiles para tareas administrativas.

PowerShell introduce el concepto de *cmdlet*, o *commandlet*: un comando propio de la *shell*, al estilo de los comandos *built-in* de Bash. Los *cmdlets* pueden operar en conjunto, de manera que la salida de un comando puede alimentar la entrada de otro, siguiendo el estilo de las tuberías de otras consolas.

PowerShell permite a los administradores acceder al sistema de ficheros, al igual que otras consolas, además de recursos específicos de Windows, como el registro y los almacenes de certificados de seguridad.

Desde su introducción en 2007, PowerShell es el entorno de *scripting* por defecto de la mayoría de *sysadmins*. Es posible automatizar prácticamente cualquier tarea y, cuando no hay un *cmdlet* para una cierta tarea, siempre es posible invocar comandos nativos de Windows.

7.3. Cmdlets

Los comandos de otras consolas tienen nombres y parámetros definidos de manera irregular. Sin embargo, los *cmdlets* de PowerShell siguen un esquema común de **verbo-nombre**. Los verbos no tienen por qué ser un verbo gramaticalmente hablando, sino que expresan la acción que ejecutará el comando sobre un recurso. El recurso, expresado por el segundo componente, describe objetos específicos del sistema operativo, el sistema de ficheros, etc. Por ejemplo, algunos de los *cmdlets* básicos son *Get-Process*, *Stop-Process*, *Get-Service* y *Stop-Service*.

A continuación, en el vídeo *Primeros pasos en PowerShell*, se muestran algunos de los comandos básicos de PowerShell y una explicación básica de cómo guardar y ejecutar *scripts*.



El concepto de verbo y nombre no se limita al texto con el que invocar el comando. El comando `Get-Command`, muy utilizado para obtener ayuda de forma interactiva, lista todos los comandos disponibles en un entorno concreto. Uno de los filtros que acepta es el nombre sobre el que operan los comandos. Es posible obtener la lista de comandos que operan sobre un **recurso concreto**, `Process`, por ejemplo, de forma inmediata:

```
PS C:\> Get-Command -Noun process
```

CommandType	Name	Version	Source
-----	----	-----	-----
Cmdlet	Debug-Process	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Process	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Start-Process	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Stop-Process	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Wait-Process	3.1.0.0	Microsoft.PowerShell.Management

Al contrario que otras interfaces de línea de comandos, PowerShell **procesa los parámetros directamente**, por lo que la estructura de estos es homogénea entre comandos diferentes. Los comandos estándar de la librería siguen una nomenclatura homogénea, lo que facilita su uso. Además, todos los comandos incluyen una serie de parámetros comunes controlados por el motor de PowerShell: `WhatIf`, `Confirm`, `Verbose`, `Debug`, `Warn`, `ErrorAction`, `ErrorVariable`, `OutVariable`, y `OutBuffer`.

Comandos básicos

El primer comando básico ya se ha mencionado: **Get-Command**. Es fundamental para averiguar los comandos disponibles en un equipo. PowerShell es extensible y es habitual que las herramientas de administración incorporen módulos de PowerShell adicionales. Por tanto, la salida de `Get-Command` variará entre un sistema y otro.

Otro comando ideal para obtener información es **Get-Help** (Hill, 2009). Este *cmdlet* proporciona ayuda sobre lo que hace un *cmdlet* específico, sus parámetros e incluso incluye ejemplos con varios casos de uso. Por ejemplo, `Get-Help Get-Command -`

Detailed mostrará una descripción del comando y un listado de los parámetros con el tipo de dato que espera cada uno, tal como muestra la Figura 1.

```

Administrator: Windows PowerShell
PS C:\Users\Administrator> Get-Help Get-Command -Detailed

NAME
    Get-Command

SYNOPSIS
    Gets all commands.

SYNTAX
    Get-Command [[-Name <String[]>] [-ArgumentList <Object[]>] [-All] [-CommandType {Alias | Function | Filter | Cmdlet | ExternalScript | Application | Script | Workflow | Configuration | All}] [-FullyQualifiedModule <ModuleSpecification[]>] [-ListImported] [-Module <String[]>] [-ParameterName <String[]>] [-ParameterType <PSTypeName[]>] [-ShowCommandInfo] [-Syntax] [-TotalCount <Int32>] [<CommonParameters>]

    Get-Command [-ArgumentList <Object[]>] [-All] [-FullyQualifiedModule <ModuleSpecification[]>] [-ListImported] [-Module <String[]>] [-Noun <String[]>] [-ParameterName <String[]>] [-ParameterType <PSTypeName[]>] [-ShowCommandInfo] [-Syntax] [-TotalCount <Int32>] [-Verb <String[]>] [<CommonParameters>]

DESCRIPTION
    The Get-Command cmdlet gets all commands that are installed on the computer, including cmdlets, aliases, functions, workflows, filters, scripts, and applications. Get-Command gets the commands from Windows PowerShell modules and snap-ins and commands that were imported from other sessions. To get only commands that have been imported into the current session, use the ListImported parameter.

    Without parameters, a Get-Command command gets all of the cmdlets, functions, workflows and aliases installed on the computer. A 'Get-Command *' command gets all types of commands, including all of the non-Windows PowerShell files in the Path environment variable ($env:path), which it lists in the Application command type.

    A Get-Command command that uses the exact name of the command, without wildcard characters, automatically imports the module that contains the command so that you can use the command immediately. To enable, disable, and configure automatic importing of modules, use the $PSModuleAutoLoadingPreference preference variable. For more information, see about_Preference_Variables (http://go.microsoft.com/fwlink/?LinkID=113248) in the Microsoft TechNet library. Get-Command gets its data directly from the command code, unlike Get-Help, which gets its information from help topics.

    In Windows PowerShell 2.0, Get-Command gets only commands in current session. It does not get commands from modules that are installed, but not imported. To limit Get-Command in Windows PowerShell 3.0 and later versions to commands in the current session, use the ListImported parameter.

    Starting in Windows PowerShell 5.0, results of the Get-Command cmdlet display a Version column by default. A new Version property has been added to the CommandInfo class.

PARAMETERS
    -All [<SwitchParameter>]
  
```

Figura 1. Comando Get-Help Get-Command. Fuente: elaboración propia.

Windows Server 2019 no incluye todos los ficheros de ayuda por defecto. Get-Help puede redirigir a la web de Microsoft para obtener ayuda específica, pero también es posible descargar todos los ficheros con Update-Help.

Otro de los comandos más utilizados es **Get-Member**. Uno de los conceptos que más cuesta asumir es que prácticamente todo lo que manipulan los *cmdlets* son objetos .NET. Esto significa que, al enviar la salida en una tubería de un comando a otro, el segundo comando recibirá un objeto. El tipo de los objetos o su formato no tienen por qué ser siempre conocidos, y aquí es donde Get-Member aporta su valor: mostrará el detalle de los métodos y atributos del objeto que haya recibido. Por ejemplo, en la Figura 2, Get-Member recibe un objeto de tipo ApplicationInfo. Este objeto tiene

algunos métodos estándar, heredados de una clase padre, y tiene una lista de propiedades, la mayoría de tipo *string*.

```

Administrator: Windows PowerShell
PS C:\Users\Administrator> Get-Command sc.exe | Get-Member

TypeName: System.Management.Automation.ApplicationInfo
-----
Name      MemberType Definition
-----
Equals    Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
ResolveParameter Method      System.Management.Automation.ParameterMetadata ResolveParameter(string name)
ToString   Method      string ToString()
CommandType Property     System.Management.Automation.CommandTypes CommandType {get;}
Definition Property     string Definition {get;}
Extension  Property     string Extension {get;}
Module     Property     psmoduleinfo Module {get;}
ModuleName Property     string ModuleName {get;}
Name       Property     string Name {get;}
OutputType Property     System.Collections.ObjectModel.ReadOnlyCollection[System.Management.Automation.PSTypeName] OutputType {get;}
Parameters Property     System.Collections.Generic.Dictionary[string, System.Management.Automation.ParameterSet] Parameters {get;}
ParameterSets Property     System.Collections.ObjectModel.ReadOnlyCollection[System.Management.Automation.CommandParameterSet] ParameterSets {get;}
Path       Property     string Path {get;}
RemotingCapability Property     System.Management.Automation.RemotingCapability RemotingCapability {get;}
Source     Property     string Source {get;}
Version    Property     version Version {get;}
Visibility Property     System.Management.Automation.SessionStateEntryVisibility Visibility {get;set;}
FileVersionInfo ScriptProperty System.Object FileVersionInfo {get=[System.Diagnostics.FileVersionInfo]::getversioninfo($Path)}
HelpUri    ScriptProperty System.Object HelpUri {get=$cmdletProgressPreference = $ProgressPreference}
  
```

Figura 2. Ejemplo de Get-Member. Fuente: elaboración propia.

Visto que el objeto tiene una propiedad *Version*, es posible acceder a ella igual que en otros lenguajes de programación: con el operador «.» (punto), como muestra la Figura 3.

```

PS C:\Users\Administrator> (Get-Command sc.exe).Version

Major Minor Build Revision
-----
10     0      17763  1
  
```

Figura 3. Acceso a propiedades de un objeto de PowerShell. Fuente: elaboración propia.

Si la salida del comando produce una lista de objetos, *Get-Member* dará información sobre los objetos como tal. Con esta información, es posible acceder al contenido de la lista con otras expresiones nativas de PowerShell, como *foreach*. Esta expresión aplica un bucle sobre una lista. En cada iteración, la variable *\$_* contiene el objeto correspondiente y es posible acceder a él, como en el caso anterior. La Figura 4 muestra un ejemplo en el que la lista contiene objetos de tipo *CmdletInfo*.


```
PS C:\> Get-Command -Noun Process | foreach {$_.Name + " " + $_.Version}
Debug-Process 3.1.0.0
Get-Process 3.1.0.0
Start-Process 3.1.0.0
Stop-Process 3.1.0.0
Wait-Process 3.1.0.0
PS C:\>
```

Figura 4. Get-Command y foreach. Fuente: elaboración propia.

El último comando básico de esta sección es **Get-PSDrive**. En PowerShell, el sistema de ficheros es solo uno de los tipos de unidades que se pueden manipular. Las unidades disponibles se pueden obtener con **Get-PSDrive**, sin parámetros (ver Figura 5).

```
PS C:\> Get-PSDrive
```

Name	Used (GB)	Free (GB)	Provider	Root
Alias			Alias	
C	10.78	20.69	FileSystem	C:\
Cert			Certificate	\
D	0.05	0.00	FileSystem	D:\
Env			Environment	
Function			Function	
HKCU			Registry	HKEY_CURRENT_USER
HKLM			Registry	HKEY_LOCAL_MACHINE
Variable			Variable	
WSMan			WSMan	

Figura 5. Lista de unidades con Get-PSDrive. Fuente: elaboración propia.

Estas unidades pueden manipularse con el mismo subconjunto de *cmdlets*, que se pueden obtener con **Get-Command *-Item*** (ver Figura 6).

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

CommandType	Name	Version	Source
Cmdlet	Clear-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Clear-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Copy-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Copy-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-ItemPropertyValue	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Invoke-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Move-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Move-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	New-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	New-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Remove-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Remove-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Rename-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Rename-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Set-Item	3.1.0.0	Microsoft.PowerShell.Management
Cmdlet	Set-ItemProperty	3.1.0.0	Microsoft.PowerShell.Management

Figura 6. Comandos para manipulación de unidades. Fuente: elaboración propia.

Por ejemplo, la Figura 7 demuestra el uso de `Get-Item` para obtener un listado de los archivos de `c:\Users\Administrator\Downloads` y para obtener las variables de entorno que empiezan con `P`.

```
PS C:\> Get-Item Env:\P*

Name                           Value
----                           -
PSModulePath                   C:\Users\Administrator\Documents\WindowsPowerShell\Modules;C:\Program Files\WindowsPowerShell\Modules;...
PROCESSOR_ARCHITECTURE         AMD64
Path                           C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:...
ProgramFiles(x86)              C:\Program Files (x86)
PROCESSOR_LEVEL                6
PATHEXT                        .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.CPL
PROCESSOR_REVISION             9e09
PROCESSOR_IDENTIFIER           Intel64 Family 6 Model 150 Stepping 9, GenuineIntel
ProgramFiles                   C:\Program Files
ProgramData                    C:\ProgramData
ProgramId6432                  C:\Program Files
PUBLIC                         C:\Users\Public

PS C:\> Get-Item C:\Users\Administrator\Downloads\*

Directory: C:\Users\Administrator\Downloads

Mode                LastWriteTime         Length Name
----                -
-a- - - -          4/19/2020  10:15 AM        1179024 putty.exe
-a- - - -          4/20/2020  12:33 AM         696720 puttygen.exe
-a- - - -          4/22/2020   4:11 AM          2693 ubuntu_key.ppk
```

Figura 7. `Get-Item` en unidad de sistema de archivos y en variables de entorno. Fuente: elaboración propia.

Alias

PowerShell soporta alias, que son nombres alternativos de comandos y *cmdlets* (Shepard, 2015). Por ejemplo, `dir` y `ls` son alias de `Get-ChildItem` y `cd` es un alias de `Set-Location`. Hay alias disponibles para muchos *cmdlets* con funcionalidad similar en entornos DOS o Linux. El objetivo es doble: hacer el código más conciso en la línea de comandos y facilitar la transición de los usuarios habituales de otras *shell*.

Para obtener una lista de los alias definidos en una sesión, se puede hacer uso de `Get-Alias`. Con el parámetro por defecto, `Get-Alias <alias>`, se obtiene el comando al que un alias hace referencia y con `Get-Alias -Definition <cmdlet>` se obtienen los alias definidos para ese *cmdlet* (ver Figura 8).

```
PS C:\> Get-Alias ls

CommandType      Name                               Version      Source
-----
Alias            ls -> Get-ChildItem

PS C:\> Get-Alias -Definition Get-ChildItem

CommandType      Name                               Version      Source
-----
Alias            dir -> Get-ChildItem
Alias            gci -> Get-ChildItem
Alias            ls -> Get-ChildItem
```

Figura 8. Alias de Child-Item. Fuente: elaboración propia.

El *cmdlet* `New-Alias` crea un alias en la sesión actual. Los alias no son permanentes y desaparecen al cerrar la consola, pero es posible guardarlos con `Export-Alias`. Al iniciar sesión, se pueden importar con `Import-Alias`.

```
PS C:\> New-Alias -Name ll -Value Get-ChildItem
PS C:\> ll C:\Users\Administrator\Downloads\

Directory: C:\Users\Administrator\Downloads

Mode                LastWriteTime         Length Name
----
-a----          4/19/2020  10:15 AM        1179024 putty.exe
-a----          4/20/2020  12:33 AM         696720 puttygen.exe
-a----          4/22/2020   4:11 AM           2693 ubuntu_key.ppk

PS C:\> Export-Alias -Path C:\Users\Administrator\alias.csv
PS C:\> Import-Alias -Path C:\Users\Administrator\alias.csv -ErrorAction SilentlyContinue
```

Figura 9. Gestión de alias. Fuente: elaboración propia.

7.4. Scripts

Como se ha mencionado anteriormente, PowerShell ofrece no solo una herramienta interactiva, sino también un **entorno de programación**. Los *scripts* en PowerShell tienen la extensión `.ps1`, independientemente de la versión de PowerShell (Shepard, 2015).

Aunque, independiente del motor de ejecución, la mayoría de las versiones de Windows incluyen un editor llamado PowerShell ISE. Ofrece autocompletado de los nombres de los comandos y de los parámetros, igual que la línea de comandos, así

como ejecución del *script*, *debugging*, coloreado de sintaxis, etc. La Figura 10 muestra el PowerShell ISE con un *script* sencillo que crea una carpeta nueva y cambia a ella.

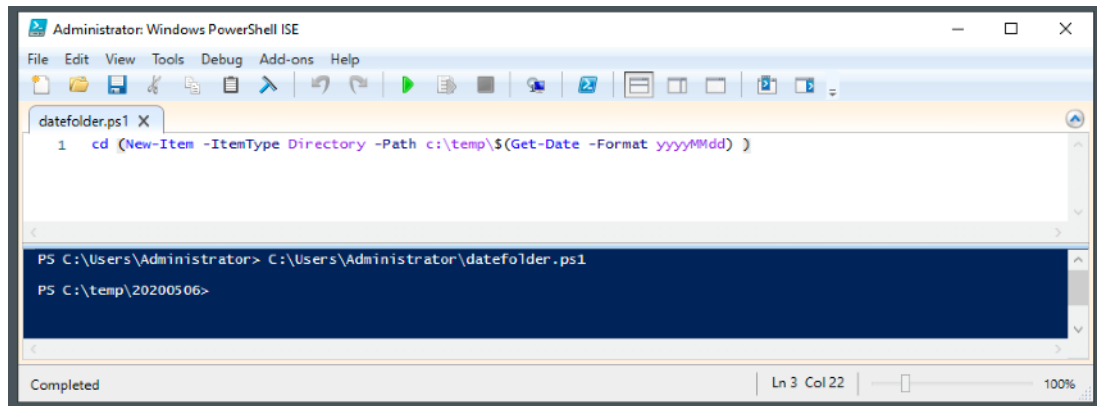


Figura 10. PowerShell ISE. Fuente: elaboración propia.

Política de ejecución

El *script* anterior se ha ejecutado en el propio editor, pero también es posible ejecutarlo en la línea de comandos. Sin embargo, es habitual que ocurra algo parecido a la Figura 11, donde se muestra un mensaje de error que hace referencia a la política de ejecución o *execution policy*.

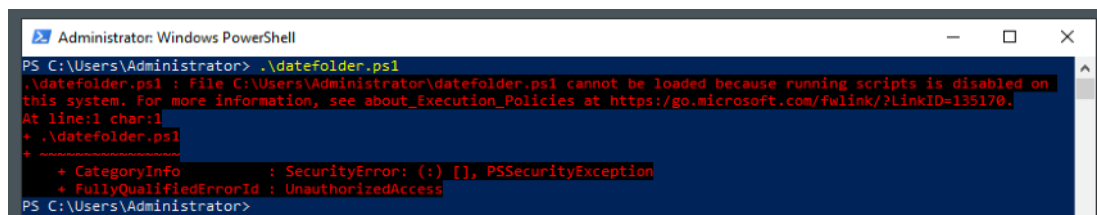


Figura 11. Error por restricción de la política de ejecución. Fuente: elaboración propia.

La política de ejecución es una **medida de seguridad** de PowerShell que ofrece control a los administradores sobre qué *scripts* se pueden ejecutar. Las políticas de ejecución posibles son:

- ▶ *Restricted*: no permite la ejecución de *scripts* en ningún caso. Fue la política por defecto para las versiones anteriores a Windows Server 2012 R2.
- ▶ *All signed*: solo los *scripts* con una firma digital válida se pueden ejecutar.

- ▶ *Remote signed*: los *scripts* de ubicaciones remotas deben tener una firma digital válida para ejecutarlos, pero los *scripts* locales se pueden ejecutar sin restricciones. Es la política por defecto desde Windows Server 2012 R2.
- ▶ *Unrestricted*: cualquier *script* se puede ejecutar sin restricciones.

El comando `Get-ExecutionPolicy` muestra la política actual, mientras que `Set-ExecutionPolicy` permite cambiar de una política a otra (ver Figura 12).

```

Administrator: Windows PowerShell
PS C:\Users\Administrator> .\datefolder.ps1
.\datefolder.ps1 : File C:\Users\Administrator\datefolder.ps1 cannot be loaded because running s
cripts is
disabled on this system. For more information, see about_Execution_Policies at
https://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ .\datefolder.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess

PS C:\Users\Administrator> Get-ExecutionPolicy
Restricted
PS C:\Users\Administrator> Set-ExecutionPolicy RemoteSigned

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the executio
n policy might
expose you to the security risks described in the about_Execution_Policies help topic at
https://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): y
PS C:\Users\Administrator> .\datefolder.ps1
PS C:\temp\20200506>
  
```

Figura 12. Cambio de política de ejecución. Fuente: elaboración propia.

Tipos de *scripts*

Una distinción habitual divide los *scripts* en controladores y herramientas. Los *scripts* **controladores** llaman a otros *scripts*, *cmdlets* y funciones para completar su tarea. Pueden incluir más de una tarea, pero en general no se espera que se vayan a reusar a menudo. Se usan habitualmente, por ejemplo, en el Programador de tareas de Windows.

Por otro lado, los *scripts* **herramientas** ejecutan una única tarea y están pensados, precisamente, para ser reutilizados. Se usarían individualmente, únicamente en una sesión interactiva, pero el objetivo es realmente ofrecer una solución a un paso

concreto dentro de una tarea más amplia. No hay diferencias técnicas entre ambos tipos, sino más bien un estilo diferente de ejecución y documentación elegido por el administrador que lo escribió.

Comentarios

Los *scripts* de PowerShell aceptan comentarios de una línea precedidos por el carácter # y de varias líneas rodeados con los delimitadores <# y #>.

```
# comentario en una línea
Write-Host Hello World
```

```
<#
    comentario en
    múltiples líneas
#>
Write-Host Bye bye
```

Ámbito de los *scripts*

El motor de PowerShell crea un **ámbito** (o *scope*) durante la ejecución de cada *script*. Si el *script* crea o modifica variables o funciones, estos cambios son visibles en el ámbito del *script* y desaparecerán cuando el *script* termine. En este ejemplo, la variable \$var se crea en el ámbito del *script* hello-world.ps1, por lo que al terminar el *script* ya no está disponible:

```
PS C:\> Get-Content .\hello-world.ps1
$var = "Hello World"
Write-Host "La variable contiene:" $var
PS C:\> .\hello-world.ps1
La variable contiene: Hello World
PS C:\> Write-Host $var

PS C:\>
```

PowerShell usa la misma técnica que Bash para ejecutar un *script* en el ámbito de la *shell* actual para mantener todos los objetos creados durante la ejecución: en vez invocar el *script* como cualquier otro ejecutable, se invoca el *script* con el comando «.» (punto). En Bash, el punto era equivalente al comando `source`, esta herramienta suele llamarse *dot-sourcing*. Este método tiene el mismo efecto que ejecutar todos los comandos del *script* en la línea de comandos, uno tras otro. El resultado del ejemplo anterior, usando esta técnica, sería el siguiente:

```
PS C:\> . .\hello-world.ps1
La variable contiene: Hello World
PS C:\> echo $var
Hello World
PS C:\>
```

Parámetros

Ejecutar una serie de comandos puede ser útil, pero es habitual que estos comandos necesiten algún tipo de datos de entrada. PowerShell soporta la definición de múltiples parámetros, el tipo, los valores por defecto y la obligatoriedad. Esta funcionalidad es nativa de PowerShell, por lo que no hay que usar comandos adicionales, como `getopts` en Bash, para pasear los parámetros.

PowerShell incluye este tipo de validaciones, pero eso no implica que el *script* pueda incluir validaciones adicionales específicas para la lógica del código. Por ejemplo, un parámetro puerto puede estar definido como un entero, que PowerShell puede comprobar, pero haría falta una comprobación adicional para verificar que el valor entra dentro de un rango concreto de enteros. El siguiente código muestra un ejemplo de esta funcionalidad:

```
Param(
    [Parameter(Mandatory=$true)]
    [string] $Var1,
    [Parameter(Mandatory=$false)]
    [Int32] $Var2=42
```

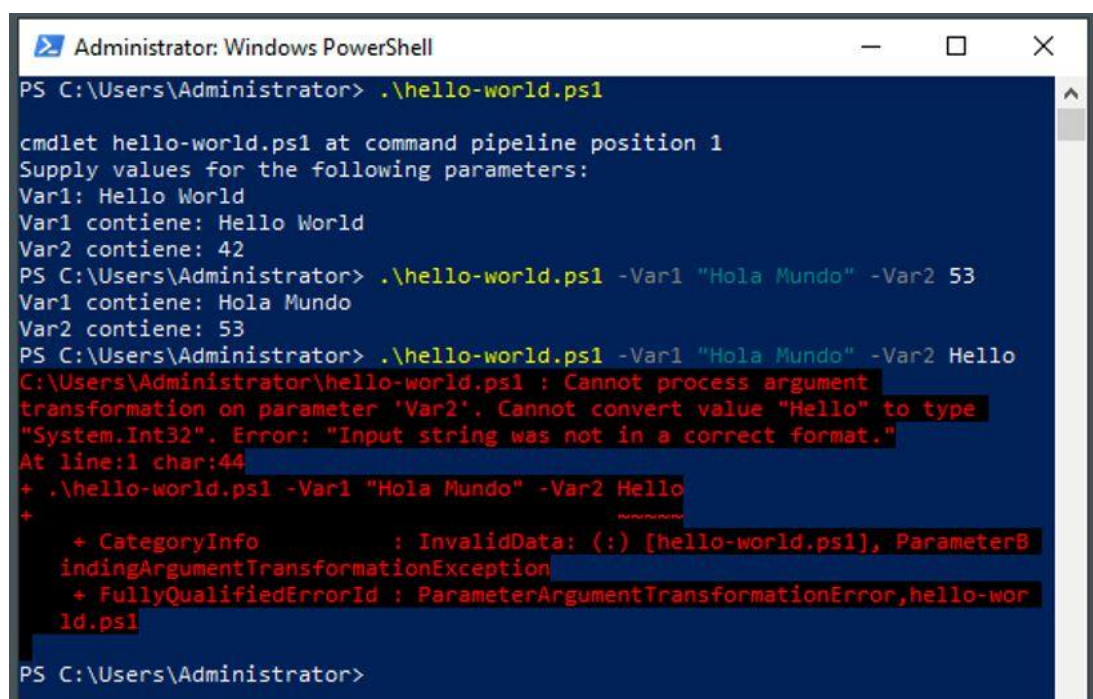

)

```
Write-Host "Var1 contiene:" $Var1
```

```
Write-Host "Var2 contiene:" $Var2
```

El parámetro `Var1` es obligatorio y está definido como una cadena de texto, mientras que `Var2` será un entero con valor 42, si el usuario no especifica otro valor. La Figura 13 muestra varias ejecuciones del *script* en las que PowerShell:

- ▶ Solicita un parámetro obligatorio que se ha especificado durante la ejecución.
- ▶ Sobrescribe el valor por defecto de un valor, si el usuario lo proporciona.
- ▶ Muestra un error si el valor de un parámetro no es conforme al tipo especificado.



```
Administrator: Windows PowerShell
PS C:\Users\Administrator> .\hello-world.ps1

cmdlet hello-world.ps1 at command pipeline position 1
Supply values for the following parameters:
Var1: Hello World
Var1 contiene: Hello World
Var2 contiene: 42
PS C:\Users\Administrator> .\hello-world.ps1 -Var1 "Hola Mundo" -Var2 53
Var1 contiene: Hola Mundo
Var2 contiene: 53
PS C:\Users\Administrator> .\hello-world.ps1 -Var1 "Hola Mundo" -Var2 Hello
C:\Users\Administrator\hello-world.ps1 : Cannot process argument
transformation on parameter 'Var2'. Cannot convert value "Hello" to type
"System.Int32". Error: "Input string was not in a correct format."
At line:1 char:44
+ .\hello-world.ps1 -Var1 "Hola Mundo" -Var2 Hello
+ ~~~~~
+ CategoryInfo          : InvalidData: (:) [hello-world.ps1], ParameterB
indingArgumentTransformationException
+ FullyQualifiedErrorId : ParameterArgumentTransformationError,hello-wor
ld.ps1

PS C:\Users\Administrator>
```

Figura 13. Validación de parámetros en un *script*. Fuente: elaboración propia.

Funciones

Al igual que otros *frameworks* de programación y *scripting*, PowerShell soporta la definición de funciones. La definición de funciones y *scripts* es similar: cualquier línea de código que se pueda escribir en un *script* se puede escribir en una función y las

funciones aceptan parámetros al igual que los *scripts*. No obstante, las funciones permiten reusar código y facilitan su mantenimiento, como en cualquier lenguaje de programación. Además, PowerShell lista todas las funciones disponibles en una de las unidades, `Function:`, tal como muestra la Figura 14.

```
PS C:\Users\Administrator> Get-ChildItem function: | Select-Object -First 10
```

CommandType	Name	Version
Function	A:	
Function	B:	
Function	C:	
Function	cd..	
Function	cd\	
Function	Clear-Host	
Function	ConvertFrom-SddlString	3.1.0.0
Function	D:	
Function	E:	
Function	F:	

Figura 14. Lista de funciones. Fuente: elaboración propia.

La definición de una función se limita a usar la palabra `function`, englobando el código entre llaves. La definición de parámetros es idéntica a la de un *script* y debe estar al principio de la función. El ejemplo de *script* de la sección anterior se puede convertir en una función de la siguiente manera.

```
function Write-Vars {
    Param(
        [Parameter(Mandatory=$true)]
        [string] $Var1,
        [Parameter(Mandatory=$false)]
        [Int32] $Var2=42
    )

    Write-Host "Var1 contiene:" $Var1
    Write-Host "Var2 contiene:" $Var2
}
```

Al ejecutar este *script*, que solo contiene una función, no se imprimirá nada en la consola. El *script* ya no ejecuta los *cmdlets* de cada línea, sino que define una función, que estará disponible en el resto del ámbito del *script*. Sin embargo, si el *script* se

ejecuta con un *dot-source*, la función estará disponible en el ámbito de la consola y se podrá ejecutar como un comando más (ver Figura 15).

```

Administrator: Windows PowerShell

PS C:\Users\Administrator> Write-Vars
Write-Vars : The term 'Write-Vars' is not recognized as the name of a cmdlet, function, script file,
or operable
program. Check the spelling of the name, or if a path was included, verify that the path is correct a
nd try again.
At line:1 char:1
+ Write-Vars
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Write-Vars:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\Users\Administrator> . .\hello-world.ps1
PS C:\Users\Administrator> Get-ChildItem function: | Where-Object { $_.Name -eq "Write-Vars"}

CommandType      Name
-----
Function         Write-Vars

PS C:\Users\Administrator> Write-Vars -Var1 "Hello World"
Var1 contiene: Hello World
Var2 contiene: 42
PS C:\Users\Administrator>
  
```

Figura 15. Definición de función en un *script*. Fuente: elaboración propia.

La simple definición de una función incorpora una ayuda básica, accesible con `Get-Help` (ver Figura 16).

```

PS C:\Users\Administrator> Get-Help Write-Vars

NAME
    Write-Vars

SYNTAX
    Write-Vars [-Var1] <string> [[-Var2] <int>] [<CommonParameters>]

ALIASES
    None

REMARKS
    None
  
```

Figura 16. Ayuda básica de una función personalizada. Fuente: elaboración propia.

PowerShell soporta unos comentarios con un formato especial, para personalizar la ayuda. Las palabras clave como `.Synopsis` y `.DESCRIPTION` delimitan cada una de las secciones de la ayuda. La función del ejemplo anterior se podría comentar de la siguiente manera:

```

<#
.Synopsis
    Funcion de ejemplo
.DESCRIPTION
    Esta funcion imprime los valores de dos parametros, Var1,
    que es una cadena de texto y es obligatorio, y Var2, que
    es un entero y su valor por defecto es 42.
.EXAMPLE
    Write-Vars -Var1 "Hello World"
.EXAMPLE
    Write-Vars -Var1 "Hello World" -Var2 53
.INPUTS
    No espera datos por la entrada estandar.
.OUTPUTS
    Emite el parametro Var2 por la salida estandar.

#>
function Write-Vars {
    Param(
        [Parameter(Mandatory=$true)]
        [string] $Var1,
        [Parameter(Mandatory=$false)]
        [Int32] $Var2=42
    )

    Write-Host "Var1 contiene:" $Var1
    Write-Host "Var2 contiene:" $Var2
    $Var2
}

```

Una vez importada en la consola, la ayuda contendrá las secciones definidas en el comentario (ver Figura 17).

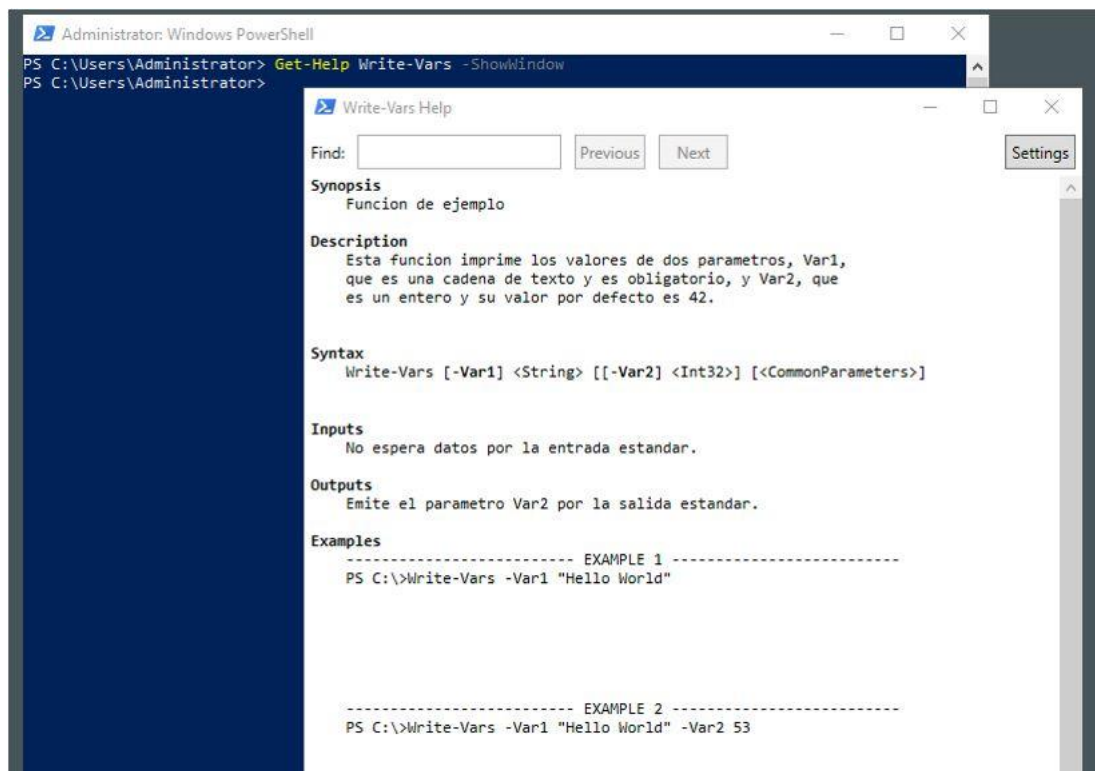


Figura 17. Ayuda personalizada. Fuente: elaboración propia.

7.5. Expresiones y otros comandos

If-Else

Los bloques lógicos *If-Else* funcionan de manera esperada, como en otros lenguajes de programación:

```
If ($Var1 -eq $Var2) {
    Write-Host "Iguales"
} ElseIf ($Var1 -gt $Var2) {
    Write-Host "Mayor"
} Else {
    Write-Host "Menor"
}
```

Los operadores de igualdad, mayor que, etc., se parecen más a los usados en Bash que a los disponibles en otros lenguajes. La Tabla 1 contiene algunos de los más habituales. Se puede consultar la tabla completa [aquí](#).

Comparadores lógicos		
	Significado	Ejemplo
-ne	!=	1 -ne 2
-eq	==	2 -eq (1+1)
-le	<=	2 -le 2
-lt	<	1 -lt 1
-ge	>=	Get-ChildItem Where-Object {\$_.Length -ge 100}
-gt	>	2 -gt 1
-like		Get-ChildItem Where-Object {\$_.Name -like 'D*'}
-notlike		Get-ChildItem Where-Object {\$_.Name -notlike 'D*'}

Tabla 1. Comparadores lógicos. Fuente: elaboración propia.

Bucles

La expresión más habitual para recorrer un bucle es **ForEach**. Esta expresión ejecuta un bloque de código para todos los valores de una colección. Por ejemplo, el siguiente bloque imprime el nombre de todos los ficheros de una carpeta:

```
ForEach ($file in (Get-ChildItem -Path C:\Users\Administrator -File)) {
    Write-Host $file.Name
}
```

No es necesario inicializar la variable si **ForEach** recibe la colección a través de una tubería. En ese caso, la variable `$_` contiene el valor de cada iteración (ya se vio en un ejemplo similar en la sección sobre **Get-Member**). Esta funcionalidad es especialmente útil en modo interactivo.

```
Get-ChildItem -Path C:\Users\Administrator -File | ForEach {
    Write-Host $_.Name
}
```

Las construcciones **While** y **For** funcionan como se esperaría:

```
$i = 1
While ($i -le 5) {
    Write-Host $i
    $i = $i + 1
}

For($j=1; $j -le 5; $j++)
{
    Write-Host $j
}
```

Tuberías

Como se ha mencionado anteriormente, las tuberías permiten **transferir objetos .NET entre comandos**. Dado que son objetos, se pueden manipular con algunos *cmdlets* específicos. Por ejemplo, *Sort-Object* facilita la ordenación de una colección a partir de atributos arbitrarios de los objetos. El siguiente ejemplo ordena los ficheros de un directorio por extensión y tamaño, de manera descendente.

```
PS C:\> Get-ChildItem -Path C:\Users\Administrator -File | Sort-Object -
Property Extension,Length -Descending
```

Directory: C:\Users\Administrator

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a----	5/7/2020 2:53 AM	1148	hello-world.ps1
-a----	5/7/2020 2:54 AM	686	Write-Vars.ps1
-a----	5/6/2020 8:54 AM	80	datefolder.ps1
-a----	4/20/2020 12:37 AM	2693	security_key.ppk
-a----	4/19/2020 10:48 AM	397	putty.log

```
-a----          5/6/2020    8:42 AM          13916 alias.csv
```

El comando **Where-Object**, que ya ha aparecido en algunos ejemplos, se usa para filtrar listas a partir de los atributos de los objetos.

```
PS C:\> Get-ChildItem -Path C:\Users\Administrator -File | Where-Object
{$_ .Length -gt 100 -and $_ .CreationTime -gt '5/1/2020'}
```

Directory: C:\Users\Administrator

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a----	5/6/2020 8:42 AM	13916	alias.csv
-a----	5/7/2020 2:53 AM	1148	hello-world.ps1
-a----	5/7/2020 2:54 AM	686	Write-Vars.ps1

El comando **Select-Object** se puede usar con dos objetivos: limitar el número de objetos de la colección y limitar las propiedades de estos objetos. La limitación en el número de objetos no es un filtrado como el de **Where-Object**, sino un truncado de la lista para obtener los primeros diez elementos, o los diez últimos, por ejemplo.

```
PS C:\> Get-ChildItem -Path C:\Users\Administrator -File | Sort-Object -
Property Length -Descending | Select-Object -First 2 -Skip 1
```

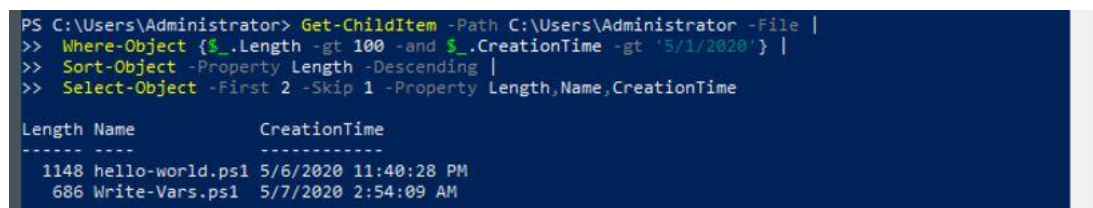
Directory: C:\Users\Administrator

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a----	4/20/2020 12:37 AM	2693	security_key.ppk
-a----	5/7/2020 2:53 AM	1148	hello-world.ps1

Las propiedades se pueden seleccionar para, por ejemplo, guardar en un archivo CSV solo ciertas columnas.

```
PS C:\> Get-ChildItem -Path C:\Users\Administrator -File | Sort-Object -
Property Length -Descending | Select-Object -First 2 -Skip 1 -Property
Length,Name | Export-Csv output.csv
PS C:\Users\Administrator> Get-Content .\output.csv
#TYPE Selected.System.IO.FileInfo
"Length","Name"
"2693","security_key.ppk"
"1148","hello-world.ps1"
```

Y, por supuesto, estos comandos se pueden enlazar en una misma tubería arbitrariamente larga, como en la Figura 18.



```
PS C:\Users\Administrator> Get-ChildItem -Path C:\Users\Administrator -File |
>> Where-Object {$_.Length -gt 100 -and $_.CreationTime -gt '5/1/2020'} |
>> Sort-Object -Property Length -Descending |
>> Select-Object -First 2 -Skip 1 -Property Length,Name,CreationTime

Length Name                CreationTime
-----
1148 hello-world.ps1 5/6/2020 11:40:28 PM
686 Write-Vars.ps1 5/7/2020 2:54:09 AM
```

Figura 18. Tubería con manipulación de objetos de una colección. Fuente: elaboración propia.

7.6. Referencias bibliográficas

Hill, K. (2009, marzo 8). *Effective Windows PowerShell: The Free eBook*. Keith Hill's Blog. <https://rkeithhill.wordpress.com/2009/03/08/effective-windows-powershell-the-free-ebook/>

Microsoft. (2021, marzo 22). *What is PowerShell?* <https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7>

Microsoft. (2021, junio 21). *About_Comparison_Operators*. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators?view=powershell-7

Shepard, M. (2015). *Getting Started with PowerShell*. Packt Publishing.

Documentación de Get-Help

Microsoft. (2021, julio 30). *about_Functions*. https://docs.microsoft.com/es-es/powershell/module/microsoft.powershell.core/about/about_functions?view=powershell-7.1

El comando `Get-Help` no ofrece ayuda solamente sobre comandos concretos, también tiene secciones sobre expresiones del lenguaje, *scripts*, variables, etc.; todas ellas disponible en la misma línea de comandos. En cualquier caso, la documentación *online* también contiene todos estos detalles.

Scripting Blog

Microsoft. (s. f.). *Scripting Blog*. <https://devblogs.microsoft.com/scripting/>

Microsoft. (2019, noviembre 13). *PowerTip: Use Windows PowerShell to display all Environment variables*. <https://devblogs.microsoft.com/scripting/powertip-use-windows-powershell-to-display-all-environment-variables/>

Microsoft. (2019, noviembre 13). *Testing RPC ports with PowerShell (and yes, it's as much fun as it sounds!)*. <https://devblogs.microsoft.com/scripting/testing-rpc-ports-with-powershell-and-yes-its-as-much-fun-as-it-sounds/>

Este blog, orientado a desarrolladores y administradores, ofrece muchos ejemplos prácticos que pueden servir de ideas o sacar de apuros. Algunos son pequeñas píldoras, como *PowerTip: Use Windows PowerShell to display all Environment variables*, mientras que otros analizan en profundidad un tema concreto, como *Testing RPC ports with PowerShell (and yes, it's as much fun as it sounds!)*.

1. ¿Cuál de los siguientes comandos vuelca el listado de ficheros del directorio actual en un fichero de texto?
 - A. `Get-ChildItem > listado.txt.`
 - B. `gci > listado.txt.`
 - C. `gci | Export-Csv listado.txt.`
 - D. Todos los anteriores.

2. ¿Qué expresión lógica indica igualdad entre objetos?
 - A. `-eq.`
 - B. `==.`
 - C. `=.`
 - D. `===.`

3. ¿Qué expresión lógica indica que un objeto es menor o igual a otro?
 - A. `-lt.`
 - B. `-le.`
 - C. `-eq.`
 - D. `-ge.`

4. ¿Qué política de ejecución permite ejecutar cualquier *script*?
 - A. *Remote signed.*
 - B. *Remote only.*
 - C. *Unrestricted.*
 - D. *Restricted.*

5. ¿Qué comando de PowerShell equivale a `man` en Linux?
- A. `Get-Help`.
 - B. `Get-Member`.
 - C. `Get-Manual`.
 - D. `Read-Help`.
6. ¿En qué se diferencian las tuberías de Bash de las de PowerShell?
- A. El carácter de separación es `||` en PowerShell y `|` en Bash.
 - B. En Bash se transfiere texto mientras que en PowerShell se transfieren objetos.
 - C. Funcionan de la misma manera.
 - D. En Bash se transfieren objetos de C mientras que en PowerShell se transfieren objetos .NET.
7. ¿Qué editor integrado incluye por defecto PowerShell?
- A. PowerShell ISE.
 - B. Notepad.
 - C. `vim`.
 - D. Ninguna de las anteriores.
8. ¿Cuál de los siguientes comandos filtra objetos en base a las propiedades?
- A. `Get-Member`.
 - B. `Filter-Object`.
 - C. `Select-Object`.
 - D. `Sort-Object`.

9. ¿Qué significa que Env: sea una unidad?

- A. Que las variables de entorno se pueden manipular con los mismos comandos que un sistema de ficheros, como Get-ChildItem y New-Item.
- B. Que Env: es un alias de Get-Variables.
- C. Nada, las unidades exclusivamente son A:, C:, D:, etc.
- D. Que PowerShell guarda las variables de entorno en una ruta del disco identificado con la unidad Env:.

10. Relaciona cada expresión con su equivalente en otros entornos de programación.

-ne	1
-lt	2
-gt	3
-ge	4

A	>=
B	<
C	!=
D	>