



Administración de Sistemas Operativos, PowerShell



Cervera Rodríguez, Diego

Haz García, Adrián

Pernas Labrada, Miguel

Abril de 2023



UNIVERSIDADE DA CORUÑA

Contenido

Introducción.....	4
Variables y Tipos de Datos	9
Variables automáticas.....	11
Variables de preferencia	13
Tipos de Datos.....	15
Ejecuciones Condicionales y Bucles	17
Ejecuciones condicionales.....	17
El declarador if	17
El Operador Ternario.....	18
La declaración switch	19
Operadores lógicos y comparativos.....	24
Operadores comparativos.....	24
Operadores lógicos.....	26
Bucles	27
El bucle For.....	27
El bucle Foreach	29
Bucle While.....	30
Bucle Do	31
Las declaraciones Break y Continue.....	33
Cmdlets.....	36
Objetos y Pipeline	42
Objetos.....	42
Get-Member.....	42
Select-Object	43
Where-Object.....	44
Los métodos	45
El pipeline	48
Funciones	53
Módulos	61
Scripts.....	67
Valores de salida	69
Escritura de ayuda.....	69
Gestión de errores.....	70
Try/Catch.....	70
Try/Finally.....	71

Variable Automática \$PSItem	72
Ejemplo práctico de script.....	74
Administración con PowerShell	82
Administración de servicios:	82
Stop-Service.....	85
Start-Service	86
Suspend-Service	86
Restart-Service	88
Set-Service.....	88
Administración de procesos.....	90
Get-Process	91
Stop-Process.....	93
Start-Process	93
Wait-Process	95
Administración de usuarios y grupos.....	96
Get-LocalUser	96
Get-LocalGroup	97
*-LocalUser.....	98
*-LocalGroup	100
Anexo I: Instalación y configuración de Vim en Windows.	102
Anexo II: Habilitar la ejecución de Scripts de PowerShell en un Sistema Windows.	103
Referencias.....	106

Introducción.

Los inicios de PowerShell de Windows se remontan a comienzos del siglo XXI, cuando Microsoft comenzó a buscar una alternativa a las limitaciones en términos de funcionalidad y escalabilidad que proporcionaba la línea de comandos CMD.exe de Windows.

En 2002, Jeffrey Snover, un arquitecto técnico en Microsoft comenzó a trabajar en una nueva consola de línea de comandos que pudiera ofrecer una experiencia de usuario más poderosa y flexible en Windows y que aprovechara la potencia de .NET Framework, la plataforma de desarrollo de software de Microsoft.

El proyecto se conoció inicialmente como "Monad" y, después de varios años de desarrollo y pruebas, se lanzó como PowerShell 1.0 en noviembre de 2006, disponible para Windows XP, Windows Server 2003 y versiones posteriores de Windows.

Hoy en día, PowerShell es una herramienta esencial utilizada por administradores de sistemas, desarrolladores y otros profesionales de las TI que tienen el propósito de configurar, administrar y automatizar tareas o realizarlas de forma más controlada a través de la creación de scripts, del uso de variables y parámetros y de la combinación de cmdlets existentes o creados por uno mismo.

Con Powershell, a diferencia de con otros shells, no se trabaja solamente con texto, sino que también tiene una salida basada en objetos, proporcionada por el framework .NET en el que se apoya. Estos objetos son como contenedores que almacenan información de una forma estructurada, lo que permite a los usuarios trabajar con la información de una manera mucho más flexible y poderosa.

Una de las principales razones que convierten a Powershell en una herramienta de administración de sistemas muy flexible y potente es su alta disponibilidad en múltiples plataformas, entornos y sistemas operativos.

En Windows, PowerShell está integrado en todas las versiones modernas, ya sea en la edición Home, Pro o Enterprise, por lo que los usuarios de este sistema operativo pueden comenzar a usar PowerShell sin tener que descargar e instalar ningún software adicional. Además, se puede utilizar tanto en el entorno de línea de comandos como en el entorno de Shell interactivo PowerShell ISE que proporciona una interfaz gráfica de usuario para crear, modificar, depurar y ejecutar scripts. Sin embargo, es importante tener en cuenta

que PowerShell ISE ya no recibe actualizaciones y Microsoft está recomendando el uso de Visual Studio Code como un entorno de desarrollo integrado, más moderno y avanzado.

En Linux y macOS, PowerShell se puede instalar y utilizar mediante la descarga e instalación del paquete disponible en los repositorios de paquetes de varias distribuciones de Linux, incluidas Ubuntu, Debian, CentOS y Fedora, y también se puede descargar directamente desde el sitio web oficial de PowerShell.

Microsoft ha desarrollado una versión móvil de PowerShell para dispositivos iOS y Android, lo que permite a los usuarios ejecutar scripts desde estos dispositivos.

PowerShell también se puede integrar en aplicaciones de servidor como SharePoint y SQL Server, para automatizar tareas de administración y mantenimiento.

También se puede ejecutar en contenedores Docker y en Amazon Web Service (AWS) mediante el servicio AWS Tools para PowerShell, lo que permite ejecutar scripts en estas plataformas.

En resumen, PowerShell está disponible en una amplia gama de plataformas y entornos, lo que lo hace una herramienta de administración de sistemas muy versátil y útil.

PowerShell ha evolucionado y ha tenido varias versiones a lo largo de los años, que se suelen presentar con cada lanzamiento de un sistema operativo. Veamos un resumen de cada una de sus versiones estables con algunas de las novedades que implementaron.

PowerShell 1.0:

Se lanzó en 2006 para los sistemas operativos Windows XP, Windows Vista y Windows Server 2003. Estaba integrado también en Windows Server 2008 como una característica para instalar.

Esta primera versión ya permite administrar varios tipos de objetos y proporciona cmdlets básicos para la manipulación de objetos tales como archivos, claves de registro, etc.

PowerShell 2.0

Se lanzó en 2009 para los sistemas operativos Windows 7 y Windows Server 2008 R2.

Esta nueva versión incorpora un número considerable de cmdlets y de funcionalidades, como la administración remota de máquinas (PowerShell Remoting), la ejecución en

segundo plano de los trabajos, y una interfaz gráfica de usuario: Windows PowerShell ISE, que aporta una gran facilidad para crear, modificar, depurar y ejecutar scripts

PowerShell 3.0

Se lanzó en 2012 para los sistemas operativos Windows 8 y Windows Server 2012.

Esta nueva versión simplificó la sintaxis del lenguaje e introdujo nuevas características como el modo de sesiones sólidas, para recuperarse automáticamente de errores de ejecución o de red; el autocompletado, el soporte para trabajo con varios discos, los Workflows, para realizar tareas más grandes y complejas; las tareas programadas y el cmdlet Show-Command, que permite ver los parámetros de un cmdlet específico.

PowerShell 4.0

Se lanzó en 2013 para los sistemas operativos Windows 8.1 y Windows Server 2012 R2.

Esta nueva versión introdujo características como el Desired State Configuration (DSC), que permite a los administradores de sistemas definir, configurar y mantener el estado deseado de los sistemas de Windows; también introdujo el soporte de autenticación Kerberos para sesiones remotas, comandos de administración de redes y mejoras en el rendimiento y en el soporte de trabajo en equipo.

PowerShell 5.0

Se lanzó en 2016 para los sistemas operativos Windows 10 y Windows Server 2016.

Esta nueva versión introdujo nuevas características como el soporte para clases y enumeraciones, la PowerShell Gallery, que proporciona acceso a una amplia variedad de módulos y scripts desarrollados por la comunidad y por Microsoft; e introdujo mejoras en el DSC.

PowerShell 5.1

Se lanzó en 2017 también para los sistemas operativos Windows 10 y Windows Server 2016.

Esta nueva versión se trató de una actualización importante de PowerShell 5.0 y ofrece numerosos cmdlets nuevos orientados a la seguridad y mejoras en cmdlets existentes, así como mejoras en la funcionalidad de PowerShell Remoting, en la compatibilidad con Unicode y en la capacidad de trabajar con conjuntos de caracteres complejos.

A partir de la siguiente versión, se ha adoptado .NET Core como plataforma de base en lugar del framework .NET.

.NET Core es una versión más modular de .NET que se ha diseñado para ser más escalable y portátil.

PowerShell 6.0 y versiones posteriores se han desarrollado para ser multiplataforma, lo que significa que se puede ejecutar en sistemas operativos diferentes a Windows, como Linux y macOS.

PowerShell 6.0

Se lanzó en 2018 siendo una versión multiplataforma que se puede ejecutar en Windows, Linux y macOS.

Esta nueva versión incluye nuevos cmdlets como Get-Error, Get-Verb y Get-CommandParameter; mejoras en la compatibilidad con JSON, lo que lo hace más adecuado para trabajar con servicios web; incluye la funcionalidad integrada para trabajar con Docker, lo que permite trabajar con contenedores de manera más eficiente y efectiva.

PowerShell 6.1

Se lanzó también en 2018, 6 meses después de la versión anterior y siendo posible ejecutarla en los mismos sistemas operativos.

Esta nueva versión incluyó mejoras en la experiencia del usuario, permitiendo crear perfiles para personalizar la configuración y las preferencias de PowerShell; incluyó nuevos cmdlets de seguridad, mejoras en la compatibilidad con SSH y Jupyter Notebooks, y un soporte mejorado para macOS y Linux.

PowerShell 6.2

Se lanzó en 2019 y se puede ejecutar en Windows, Linux y macOS.

Esta nueva versión incluye mejoras en la seguridad, administración de módulos y paquetes, compatibilidad con DSC y soporte mejorado para macOS y Linux.

PowerShell 7

Se lanzó en 2020 y es compatible con Windows, Linux y macOS.

Esta nueva versión es compatible con versiones anteriores de PowerShell, lo que significa que puede ejecutar scripts y comandos creados en versiones anteriores.

En ella se introdujeron nuevas características, como la capacidad de ejecutar scripts sin escribir "PowerShell" en la línea de comandos, una nueva sintaxis para la creación de matrices y la capacidad de administrar dispositivos IoT, que son objetos cotidianos que se conectan a Internet y que tienen la capacidad de recopilar y transmitir datos.

Otra novedad que incluyó esta versión fue mejoras en la seguridad, como la capacidad de verificar la integridad de los módulos de PowerShell antes de cargarlos.

PowerShell 7.1

Esta nueva versión se lanzó a finales de 2020 y es compatible con Windows, Linux y macOS.

En ella se introdujeron mejoras en el rendimiento y la seguridad, incluyendo la capacidad de cifrar datos de forma más segura. También se incluyó la capacidad de ejecutar scripts de forma interactiva en el terminal, la capacidad de mostrar el historial de ejecución de comandos y la capacidad de establecer un valor de retorno predeterminado para funciones.

PowerShell 7.2

Esta versión es la más reciente, lanzada en mayo de 2022 y compatible con Windows, Linux y macOS.

Entre las características incluidas en esta versión encontramos la capacidad de administrar el registro de eventos de Windows y de establecer prioridades de proceso para comandos. También se incluyeron mejoras en la seguridad, como el soporte para proteger credenciales de usuarios almacenadas en variables, y el soporte para la autenticación Kerberos en Linux. Otras mejoras se produjeron en la experiencia de usuario, mostrando errores y advertencias en el editor de código, la capacidad de agrupar y filtrar resultados de comandos y la capacidad de deshacer y rehacer cambios en el editor de código.

Los requerimientos necesarios para poder ejecutar PowerShell son:

Tener un sistema operativo compatible, entre los que se encuentran Windows, MacOS y Linux. La versión más reciente de PowerShell requiere Windows 10, pero versiones anteriores de PowerShell se pueden ejecutar en versiones antiguas de Windows, como vimos anteriormente.

Tener una versión actualizada de .NET Framework instalada en el sistema, ya que PowerShell está construido sobre esta tecnología.

Tener acceso de administrador en el sistema, ya que si no, es posible que no puedas ejecutar ciertos comandos o scripts.

Tener espacio en disco suficiente. Aunque PowerShell no requiere mucho espacio en disco para instalarse y ejecutarse, pero debes asegurarte de tener suficiente espacio disponible para instalar cualquier módulo o paquete que puedas necesitar.

Tener una cantidad de memoria RAM adecuada. Aunque PowerShell no es especialmente exigente en estos términos, es posible requerir una cantidad significativa de memoria para poder ejecutar ciertos comandos o scripts.

Tener todos los controladores de hardware actualizados en el sistema, especialmente los controladores de la tarjeta gráfica y de red, ya que PowerShell puede interactuar con ellos.

Variables y Tipos de Datos

Una variable, no es ni más ni menos, que un espacio en memoria donde se puede guardar información. Un espacio con un nombre. Por ejemplo, se pueden usar para almacenar los resultados de los comandos y almacenar elementos que se utilizan en comandos y expresiones, como nombres, rutas, configuraciones y valores.

En el caso de PowerShell la forma de identificar esos espacios en memoria es mediante una cadena de texto precedida por \$. Por ejemplo, \$variable.

El símbolo de igual (=) permite asignar valores en *PowerShell*, y debemos leerlo de derecha a izquierda. Es decir, el valor representado a la derecha del símbolo es asignado a la variable que aparezca a la izquierda.

Respecto a los nombres que les puedes asignar a las variables hay algunas observaciones:

No distinguen entre mayúsculas y minúsculas. Es decir que \$VARiable es exactamente lo mismo que \$variable.

Lo más recomendable a la hora de crear tus propios nombres de variables es que utilices exclusivamente caracteres alfanuméricos.

Mejor evitar caracteres especiales y espacios en los nombres de las variables ya que son difíciles de usar.

En el caso de que se quieran incluir *espacios*, habría que encerrar el nombre de la variable entre { }

Se pueden encontrar diferentes tipos de variables en PowerShell:

Variables creadas por el usuario: Las variables creadas por el usuario son creadas y mantenidas por el usuario. De forma predeterminada, las variables que crea en la línea de comandos de PowerShell solo existen mientras la ventana de PowerShell está abierta. Cuando se cierra la ventana de PowerShell, se eliminan las variables. En caso de que queramos guardar una variable, habría que agregarla a nuestro perfil de PowerShell. También se pueden crear variables en scripts con alcance global.

```
PS C:\WINDOWS\system32> $variable1="Hola"
PS C:\WINDOWS\system32> $variable2="Que tal?"
PS C:\WINDOWS\system32> $VARiable3=100
PS C:\WINDOWS\system32> ${VAR iable4}=200
```

Ilustración 1. Ejemplo de creación de variables en PowerShell

```
PS C:\WINDOWS\system32> New-Variable -Name variable5 -Value 300
PS C:\WINDOWS\system32> $variable5
300
```

Ilustración 2. Ejemplo de creación de variable usando un cmdlet

```
PS C:\WINDOWS\system32> $variable1
Hola
PS C:\WINDOWS\system32> $variable2
Que tal?
PS C:\WINDOWS\system32> $variable3
100
PS C:\WINDOWS\system32> ${var iable4}
200
```

Ilustración 3. Muestra del contenido de las variables creadas anteriormente

```
PS C:\WINDOWS\system32> $variable1+$variable2
Hola Que tal?
PS C:\WINDOWS\system32> $variable3+${var iable4}
300
PS C:\WINDOWS\system32> $variable3-${var iable4}
-100
```

Ilustración 4. Ejemplo de operaciones básicas con variables

Variables automáticas: Las variables automáticas almacenan el estado de PowerShell. Estas variables son creadas por PowerShell y PowerShell cambia sus valores según sea necesario para mantener su precisión. En general, estas variables se consideran de solo lectura, aunque es posible escribir en ellas, pero, no es recomendable hacerlo. Existen algunas variables especiales o automáticas o definidas por defecto. Variables que guardan información de estado.

Algunas de las variables automáticas más comunes pueden ser:

- **\$\$** contiene el último token en la última línea recibida por la sesión.

```
PS C:\WINDOWS\system32> $variable1 + $variable2
Hola Que tal?
PS C:\WINDOWS\system32> $$
$variable2
```

Ilustración 5. Variables automáticas: Ejemplo de '\$\$'

- `$^` contiene el primer token en la última línea recibida por la sesión.

```
PS C:\WINDOWS\system32> $variable1 + $variable2
Hola Que tal?
PS C:\WINDOWS\system32> $^
$variable1
```

Ilustración 6. Variables automáticas: Ejemplo de '\$^'

- `$?` esta variable contiene el estado de la ejecución del último comando. Si el resultado de la ejecución del último comando ha sido satisfactorio contendrá True y en otro caso contendrá False.

```
PS C:\WINDOWS\system32> $variable1 + $variable2
Hola Que tal?
PS C:\WINDOWS\system32> $?
True
```

Ilustración 7. Variables automáticas: Ejemplo de '\$?'

- `$Error` contiene un vector, array, de los últimos errores ordenados de más reciente a menos, de forma que el error más reciente es `$Error[0]`.

```
PS C:\WINDOWS\system32> $Error
as : El término 'as' no se reconoce como nombre de un cmdlet, función, archivo de script o programa ejecutable.
Compruebe si escribió correctamente el nombre o, si incluyó una ruta de acceso, compruebe que dicha ruta es correcta e
inténtelo de nuevo.
En línea: 1 Carácter: 1
+ as
+ ~~
+ CategoryInfo          : ObjectNotFound: (as:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

Ilustración 8. Variables automáticas: Ejemplo de '\$Error'

En caso de que se quiera ver el listado completo se puede utilizar el comando:

```
PS > Get-Help about_automatic_variables
```

```

PS C:\WINDOWS\system32> get-help about_automatic_variables

ABOUT_AUTOMATIC_VARIABLES

Short description

Describes variables that store state information for PowerShell. These
variables are created and maintained by PowerShell.

Long description

Conceptually, these variables are considered to be read-only. Even though
they CAN be written to, for backward compatibility they SHOULD NOT be
written to.

Here is a list of the automatic variables in PowerShell:

$$

Contains the last token in the last line received by the session.

$?

Contains the execution status of the last command. It contains TRUE if the
last command succeeded and FALSE if it failed.

```

Ilustración 9. Ayuda sobre Variables Automáticas

Variables de preferencia: Las variables de preferencia almacenan las preferencias del usuario para PowerShell. Estas variables las crea PowerShell y se completan con valores predeterminados. Los usuarios pueden cambiar los valores de estas variables.

Algunas de las variables de preferencia más comunes son:

- **\$ConfirmPreference:** Esta variable determina si se debe solicitar confirmación antes de ejecutar comandos que cambian datos. Los valores posibles son: "None", "Low", "Medium" y "High"
- **\$DebugPreference:** Esta variable determina la cantidad de información de depuración que se mostrará al ejecutar un comando de PowerShell. Los valores posibles son: "SilentlyContinue", "Stop", "Inquire", "Continue" y "ContinueAsync"

- **\$ErrorActionPreference:** Esta variable determina cómo se manejarán los errores en PowerShell. Los valores posibles son: "SilentlyContinue", "Stop", "Continue", "Inquire" y "Ignore"
- **\$WarningPreference:** Esta variable determina cómo se manejarán las advertencias en PowerShell. Los valores posibles son: "SilentlyContinue", "Continue", "Inquire" y "Stop"

En caso de que se quiera ver el listado completo se puede utilizar el comando:

```
PS > Get-Help about_preferences_variables
```

```
PS C:\WINDOWS\system32> get-help about_preference_variables

ABOUT_PREFERENCE_VARIABLES

Short description
Variables that customize the behavior of PowerShell.

Long description
PowerShell includes a set of variables that enable you to customize its
behavior. These preference variables work like the options in GUI-based
systems.

The preference variables affect the PowerShell operating environment and
all commands run in the environment. In many cases, the cmdlets have
parameters that you can use to override the preference behavior for a
specific command.

The following table lists the preference variables and their default
values.
```

Variable	Default Value
\$ConfirmPreference	High
\$DebugPreference	SilentlyContinue
\$ErrorActionPreference	Continue
\$ErrorView	NormalView
\$FormatEnumerationLimit	4
\$InformationPreference	SilentlyContinue
\$LogCommandHealthEvent	\$False (not logged)
\$LogCommandLifecycleEvent	\$False (not logged)
\$LogEngineHealthEvent	\$True (logged)
\$LogEngineLifecycleEvent	\$True (logged)
\$LogProviderLifecycleEvent	\$True (logged)
\$LogProviderHealthEvent	\$True (logged)

Ilustración 10. Ayuda sobre Variables de Preferencia

Tipos de Datos

Los tipos de datos en PowerShell son fundamentales para el manejo de información en los scripts y herramientas de automatización. Cada tipo de datos representa un conjunto específico de valores y operaciones que se pueden realizar con ellos.

La tabla de datos siguiente muestra los tipos de datos más utilizados, aunque existan más:

TIPO	DESCRIPCIÓN
Int, Int32, Int64	Representa enteros en un rango comprendido entre : [-2 147 483 648, 2 147 483 647] El tipo de dato int se almacena como un entero de cuatro bytes (32 bits)
Double	Representa números tan grandes como 10^{308} (positivos o negativos), con una precisión de 15 cifras y tan pequeños como 10^{-323} (64 bits).
Char	Representa cualquiera de los 65 536 caracteres Unicode contenidos en dos bytes (16 bits).
String	Representa una cadena bajo la forma de vector de caracteres Unicode.
Datetime	Representa la fecha y la hora.
Boolean	Representa un valor booleano (true o false).
Array	Representa un array que contiene múltiples valores. Un array puede contener datos de diferentes tipos.

Object, PObject, PSCustomObject	Representa un objeto generalmente personalizado.
--	--

El tipado de los datos se establece de forma automática, pero si se desea se puede hacer de forma manual, se asigna manualmente poniendo el tipo de dato antes de la variable, por tanto, no aceptará otro valor que no sea del tipo indicado:

```
PS C:\WINDOWS\system32> [int]$variable1=100
PS C:\WINDOWS\system32> [int]$variable2="Hola"
No se puede convertir el valor "Hola" al tipo "System.Int32". Error: "La cadena de entrada no tiene el formato correcto."
En línea: 1 Carácter: 1
+ [int]$variable2="Hola"
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException
```

Ilustración 11. Ejemplo del tipado de datos asignado manualmente a una variable y error provocado debido a que el valor dado no correspondía con el tipo de dato de la variable

Hacerlo de forma manual puede ser útil cuando se trabaja con muchas variables.

Si queremos ver el tipo de una variable, se puede hacer a través del módulo:

```
PS > $<Variable>.GetType()
```

```
PS C:\WINDOWS\system32> $variable1.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     Int32                                     System.ValueType
```

Ilustración 12. Ejemplo sacando el tipo de datos de una variable

Ejecuciones Condicionales y Bucles

Ejecuciones condicionales.

Las ejecuciones condicionales permiten correr un bloque de código dependiendo de la evaluación de una condición. Resultan de utilidad cuando un script necesita realizar una lógica u otra, dependiendo del valor de una constante, *flag* o resultado de una evaluación.

Las condiciones están formadas por operaciones booleanas; esto quiere decir que tenemos dos únicas posibilidades: *True* o *False*.

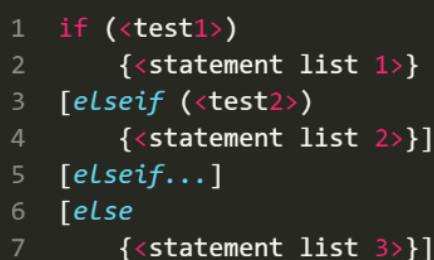
Para realizar esta evaluación, es común emplear *operadores lógicos*, tales como los vistos en la sección de [Operadores lógicos](#).

Los declaradores (o *statements*) condicionales de los que disponemos en PowerShell son:

- `if - elseif - else`
- `switch`

El declarador `if`, es el declarador condicional más básico.

Su sintaxis es la siguiente:

A screenshot of a PowerShell terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is displayed with line numbers 1 through 7 on the left. The code shows the syntax for an if statement: line 1: `if (<test1>)`, line 2: `{<statement list 1>}`, line 3: `[elseif (<test2>)`, line 4: `{<statement list 2>}]`, line 5: `[elseif...]`, line 6: `[else`, line 7: `{<statement list 3>}]`.

```
1  if (<test1>)
2      {<statement list 1>}
3  [elseif (<test2>)
4      {<statement list 2>}]
5  [elseif...]
6  [else
7      {<statement list 3>}]
```

Ilustración 13. Sintaxis de la cláusula `IF`

Se compone de una *keyword* obligatoria (`if`) y una o más opcionales (`else` y `elseif`).

Un ejemplo de código y ejecución sería el siguiente:

```
PS C:\Users\      \tmp> cat .\condicionales.ps1
$condicion = $true
if ( $condicion )
{
    Write-Output "La condicion era verdadera"
}
else
{
    Write-Output "La condicion era falsa"
}
PS C:\Users\      \tmp> .\condicionales.ps1
La condicion era verdadera
PS C:\Users\      \tmp>
```

Ilustración 14. Código de ejemplo de sentencia 'if'

Un código con bloques `elseif` podría ser el siguiente:

```
PS C:\Users\      \tmp> cat .\condicionales2.ps1
$numero = 2
if ( $numero -ge 3 )
{
    Write-Output "El numero [$numero] es mayor o igual que 3"
}
elseif ( $numero -lt 2 )
{
    Write-Output "El numero [$numero] es menor que 2"
}
else
{
    Write-Output "El numero [$numero] es igual a 2"
}
PS C:\Users\      \tmp> .\condicionales2.ps1
El numero [2] es igual a 2
PS C:\Users\      \tmp>
```

Ilustración 15. Segundo ejemplo de sentencia 'if'

El Operador Ternario.

Para las versiones de PowerShell ≥ 7.0 , se introduce una nueva sintaxis usando un operador ternario `<?>`. Su sintaxis es la siguiente:

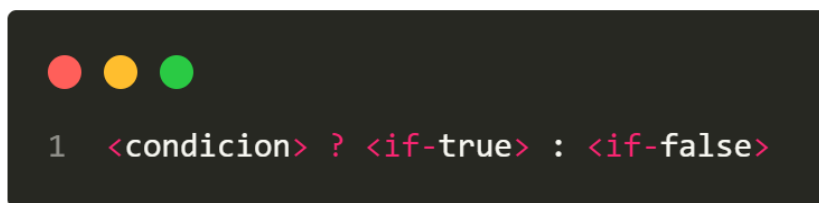


Ilustración 16. Sintaxis sentencia IF con operador ternario. PowerShell >= 7.0

Un ejemplo de ejecución sería:

```
PowerShell > $PSVersionTable

Name                           Value
----                           -
PSVersion                      7.3.3
PSEdition                      Core
GitCommitId                    7.3.3
OS                              Linux 5.4.0-1105-azure #111~18.04.1-Ubuntu SMP Fri Mar 3 22:47:43 UTC 2023
Platform                       Unix
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

Ilustración 17. Información sobre la versión de PowerShell

```
PS /home/ > $mensaje = (Test-Path $path) ? "Path existe" : "Path no encontrado"
Test-Path: Value cannot be null. (Parameter 'The provided Path argument was null or an empty collection.')
PS /home/adrian_haz> $mensaje
Path no encontrado
PS /home/adrian_haz>
```

Ilustración 18. Ejemplo de sentencia IF con operador ternario

Como la variable `$path` no existe, la variable `$mensaje` guarda la segunda cadena (evaluación falsa).

La declaración `switch` es equivalente a una serie de declaraciones `if`, pero más simple. Esta declaración lista cada condición y una acción opcional. Si la condición se satisface, la acción es ejecutada. Si no se cumple ninguna condición y se ha definido el bloque **default**, se ejecutará la acción definida en dicho bloque. Cabe destacar que la declaración `switch` puede emplear las variables automáticas `$_` y `$switch`.

La relación entre un `switch` básico y una concatenación de bloques `if` es la siguiente:

```

1  switch (<test-expression>) {
2      <result1-to-be-matched> { <action> }
3      <result2-to-be-matched> { <action> }
4  }
5
6  if (<result1-to-be-matched> -eq (<test-expression>)) { <action> }
7  if (<result2-to-be-matched> -eq (<test-expression>)) { <action> }

```

Ilustración 19. Equivalencia de una sentencia SWITCH simple y una concatenación de sentencias IF

Aunque internamente un *switch* se trate de una simplificación de una concatenación de bloques *if*, permite realizar un filtrado muy especializado. La sintaxis completa es la siguiente:

```

1  switch [-regex | -wildcard | -exact] [-casesensitive] [(<test-expression>) | -file filename]
2  {
3      <result1-to-be-matched> { <action> }
4      <result2-to-be-matched> { <action> }
5      "string" | number | variable | { <value-scriptblock> } { <action> }
6      ...
7      default { <action> } # Opcional
8  }

```

Ilustración 20. Sintaxis de la sentencia SWITCH

Es importante remarcar que el valor `<result-to-be-matched>` se halla a la izquierda de la expresión de comparación. Por tanto, el resultado del `<test-expression>` se encuentra a la derecha.

La variable automática “\$`_`” contiene el valor de la expresión pasada a la declaración de *switch* y está disponible para evaluación y uso dentro de las declaraciones `<result-to-be-matched>`

Por defecto, la búsqueda que se realiza tiene las siguientes características:

- Es una búsqueda exacta; es decir, viene marcada la *flag* `-exact`
- Es una búsqueda *case-insensitive*; es decir, la *flag* `-casesensitive` está deshabilitada.

Los posibles parámetros son:

- **Wildcard.** Indica que la condición es una cadena *wildcard*. Si la cláusula de coincidencia no es una cadena, el parámetro es ignorado. La comparación es *case-insensitive*.
- **Exact.** Indica que, si la condición es una cadena, debe coincidir de manera exacta.
- **CaseSensitive.** Realiza una coincidencia distinguiendo entre mayúsculas y minúsculas. Si la cláusula de coincidencia no es una cadena, es ignorado.
- **File.** La entrada es un fichero. Si varios parámetros `-file` son incluidos, solo se tendrá en cuenta el último. Cada línea del archivo es leída y evaluada por el *switch*. La comparación es *case-insensitive*.
- **Regex.** Realiza una coincidencia por expresión regular del valor de la condición. Si la cláusula de coincidencia no es una cadena, se ignora. La comparación es *case-insensitive*. La variable automática `$matches` está disponible en el código de coincidencia.

Si se especifican parámetros conflictivos, como **Regex** y **Wildcard**, el último parámetro especificado será el empleado.

Ejemplos de diferentes ejecuciones:

```
PS D:\tmp2\tmp> cat .\switch1.ps1
switch (3)
{
    1 {"$_" es uno."}
    2 {"$_" es dos."}
    3 {"$_" es tres."}
    4 {"$_" es cuatro."}
}
PS D:\tmp2\tmp> .\switch1.ps1
[3] es tres.
PS D:\tmp2\tmp> █
```

Ilustración 21. Ejemplo de código sentencia SWITCH: Coincidencia con un número.

```
PS D:\tmp2\tmp> cat .\switch2.ps1
switch (3)
{
    1 {"$_" es uno."}
    2 {"$_" es dos."}
    3 {"$_" es tres."}
    4 {"$_" es cuatro."}
    3 {"$_" tres de nuevo."}
}
PS D:\tmp2\tmp> .\switch2.ps1
[3] es tres.
[3] tres de nuevo.
PS D:\tmp2\tmp>
```

Ilustración 22. Ejemplo de código sentencia SWITCH: Coincidencia con varios números.

```
PS D:\tmp2\tmp> cat .\switch3.ps1
switch (3)
{
    1 {"$_" es uno."}
    2 {"$_" es dos."}
    3 {"$_" es tres."; Break}
    4 {"$_" es cuatro."}
    3 {"$_" tres de nuevo."}
}
PS D:\tmp2\tmp> .\switch3.ps1
[3] es tres.
PS D:\tmp2\tmp> █
```

Ilustración 23. Ejemplo de código sentencia SWITCH: Coincidencia con varios números, pero se incluye la sentencia Break.

```

PS D:\tmp2\tmp> cat .\switch4.ps1
switch (1, 5)
{
    1 {"[$_] es uno."}
    2 {"[$_] es dos."}
    3 {"[$_] es tres."}
    4 {"[$_] es cuatro."}
    5 {"[$_] es cinco."}
}
PS D:\tmp2\tmp> .\switch4.ps1
[1] es uno.
[5] es cinco.
PS D:\tmp2\tmp>

```

Ilustración 24. Ejemplo de código sentencia SWITCH: Evaluación de varios valores. Como se puede observar, se ejecuta uno a cada tiempo.

```

PS D:\tmp2\tmp> cat .\switch5.ps1
switch ("seis")
{
    1 {"[$_] es uno." ; Break}
    2 {"[$_] es dos."; Break}
    3 {"[$_] es tres."; Break}
    4 {"[$_] es cuatro."; Break}
    5 {"[$_] es cinco."; Break}
    "se*" {"[$_] coincide con se*."}
    Default {
        "No hay coincidencias con [$_]
    }
}
PS D:\tmp2\tmp> .\switch5.ps1
No hay coincidencias con [seis]
PS D:\tmp2\tmp>

```

Ilustración 25. Ejemplo de código sentencia SWITCH: Intento de coincidencia sin el parámetro "Wildcard"

```

PS D:\tmp2\tmp> cat .\switch6.ps1
switch -Wildcard ("seis")
{
    1 {"[$_] es uno." ; Break}
    2 {"[$_] es dos."; Break}
    3 {"[$_] es tres."; Break}
    4 {"[$_] es cuatro."; Break}
    5 {"[$_] es cinco."; Break}
    "se*" {"[$_] coincide con [se*]}
    Default {
        "No hay coincidencias con [$_]
    }
}
PS D:\tmp2\tmp> .\switch6.ps1
[seis] coincide con [se*]
PS D:\tmp2\tmp>

```

Ilustración 26. Ejemplo de código sentencia SWITCH: Coincidencia usando el parámetro Wildcard.

```

PS D:\tmp2\tmp> cat .\switch7.ps1
$email = 'antonio.yanez@udc.es'
$email2 = 'antonio.yanez@usc.gal'
$url = 'https://www.dc.fi.udc.es/~afyanez/Docencia/2023'
switch -Regex ($url, $email, $email2)
{
    '^w+\.\w+@(udc|usc|edu)\.es|gal$' { "[$_] es una direccion de correo electronico academica" }
    '^ftp:\/\/.*$' { "[$_] es una direccion ftp" }
    '^(http[s]?)\:\/\/.*$' { "[$_] es una direccion web, que utiliza [$(($matches[1]))]" }
}
PS D:\tmp2\tmp> .\switch7.ps1
[https://www.dc.fi.udc.es/~afyanez/Docencia/2023] es una direccion web, que utiliza [https]
[antonio.yanez@udc.es] es una direccion de correo electronico academica
[antonio.yanez@usc.gal] es una direccion de correo electronico academica
PS D:\tmp2\tmp>

```

Ilustración 27. Ejemplo de código sentencia SWITCH: Coincidencia utilizando el parámetro Regex. Uso de la variable automática \$matches

Operadores lógicos y comparativos.

Como se ha podido observar, en las expresiones condicionales es común emplear tanto operadores lógicos como comparativos.

Ambos tipos de operadores son un subconjunto de los diversos operadores disponibles en PowerShell.

Operadores comparativos.

Esta clase de operadores permiten, como su nombre indica, comparar valores o incluso filtrar valores que coincidan con un patrón especificado. Hay cinco tipos de operadores:

Tipo	Equality (igualdad)	Matching (coincidencia)	Replacement (reemplazo)	Containment (contención)	Type (tipo)
Estándar	-eq => Igual -ne => No igual -gt => Mayor estricto	-like => La cadena coincide con la Wildcard. -notlike => La cadena no coincide con la Wildcard.	-replace => Reemplaza la string que coincide con el patrón Regex	-contains => La colección contiene el valor. -notcontains => La colección no contiene el valor.	-is => Ambos objetos son del mismo tipo. -isnot => Los objetos

	-ge => Mayor o igual -lt => Menor estricto -le => Menor o igual	-match => La cadena coincide con el Regex -notmatch => La cadena no coincide con el Regex.		-in => El valor está en la colección. -notin => El valor no está en la colección.	no son del mismo tipo.
Case-sensitive	-ceq, -cne, -cgt, -cge, -clt, -cle	-clike, -cnotlike, -cmatch, -cnotmatch	-creplace	-ccontains, -cnotcontains	NaN
Case-Insensitive	-ieq, -ine, -igt, -ige, -ilt, -ile	-ilike, -inotlike, -imatch, -inotmatch	-ireplace	-icontains, -inotcontains	NaN

Cuando la entrada de un operador es un valor escalar, el operador devuelve un valor de tipo *Boolean*. Cuando la entrada es una colección, el operador devuelve los elementos de la colección que coinciden con el valor situado a la derecha del operador. A esta norma hay varias excepciones:

- Los operadores de tipo *Containment* y *Type* siempre devuelven un *Boolean*.
- El operador `-replace` devuelve el resultado del reemplazo.
- Los operadores `-match` y `-notmatch` también generan información bajo la variable automática `$Matches`, salvo que la parte izquierda de la expresión sea una colección.

Además, existe la siguiente regla: **Los valores a la derecha del operador serán transformados al tipo del valor de la izquierda**, para su posterior comparación. Es por esto que el siguiente ejemplo genera dos salidas distintas:

```
PS D:\tmp2\tmp> 1 -eq "1.0"
True
PS D:\tmp2\tmp> "1.0" -eq 1
False
PS D:\tmp2\tmp>
```


Ilustración 28. Diferencia de salidas por la conversión de tipos. En la primera se hace una conversión de String a Int. En la segunda, de Int a String

Operadores lógicos.

Los operadores lógicos conectan expresiones y declaraciones, permitiendo comprobar múltiples condiciones en una única expresión. PowerShell soporta los siguientes operadores lógicos:

- AND lógico (`-and`)
- OR lógico (`-or`)
- OR EXCLUSIVO lógico (`-xor`)
- NOT lógico (`-not`) o (`!`)

La sintaxis de estos operadores es la siguiente:



```

1 <statement> {-AND | -OR | -XOR} <statement>
2
3 {! | -NOT} <statement>

```

Ilustración 29. Sintaxis de los operadores lógicos.


Bucles

Continuando la lógica de las ejecuciones condicionales, aparecen los bucles. Un bucle es una estructura que permite ejecutar un bloque de código mientras se satisfaga (o no) una condición. Esta condición resulta de una expresión con resultado de tipo *Boolean*.

PowerShell dispone de diversos tipos de bucles.

El bucle For.

El bucle *For* es una construcción del lenguaje que se utiliza para crear un bucle que ejecute comandos en un bloque de código, mientras se evalúa una condición específica a `$true`. Su sintaxis es la siguiente:



```

1 for (<Init>; <Condition>; <Repeat>) {
2     <Statement list>
3 }

```

Ilustración 30. Sintaxis del bucle For

Específicamente, sus elementos son:

- `<Init>` => Representa uno o más comandos que son ejecutados antes del comienzo del bucle.
- `<Condition>` => PowerShell evalúa la condición antes de cada iteración del bucle. Si el resultado de la condición es `$true`, corre; si es `$false`, sale del bucle.

- <Repeat> => Representa uno o más comandos, separados por comas, que son ejecutados cada vez que el bucle se repite.
- <Statement list> => El conjunto de comandos que se encuentran en el bloque de código del bucle.

Los ejemplos de ejecución de bucles *For* son triviales, así a todo, merece la pena observar el uso de las operaciones de asignación para múltiples valores.

```
PS D:\tmp2\tmp> cat .\for1.ps1
for (($i = 0), ($j = 0); $i -lt 5; $i++)
{
    "`$i:$i"
    "`$j:$j"
}
PS D:\tmp2\tmp> .\for1.ps1
$i:0
$j:0
$i:1
$j:0
$i:2
$j:0
$i:3
$j:0
$i:4
$j:0
PS D:\tmp2\tmp>
```

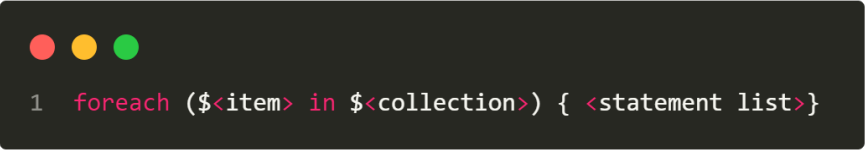
Ilustración 31. Ejemplo de ejecución de bucle For: Asignación múltiple empleando comas

```
PS D:\tmp2\tmp> cat .\for2.ps1
for ($($i = 0;$j = 0); $i -lt 5; $($i++;$j++))
{
    "`$i:$i"
    "`$j:$j"
}
PS D:\tmp2\tmp> .\for2.ps1
$i:0
$j:0
$i:1
$j:1
$i:2
$j:2
$i:3
$j:3
$i:4
$j:4
PS D:\tmp2\tmp> █
```

Ilustración 32. Ejemplo de ejecución de bucle For: Asignación y variación múltiple usando una subexpresión

El bucle Foreach

Este bucle permite iterar los valores contenidos en una colección de elementos. Su sintaxis es:



```
1 foreach ($<item> in $<collection>) { <statement list>}
```

Ilustración 33. Sintaxis del bucle Foreach

La descripción de sus elementos es la siguiente:

- La parte encerrada entre paréntesis representa la variable y la colección a iterar. PowerShell crea la variable **\$<item>** de forma automática cuando el bucle corre. Antes de cada iteración, la variable se inicializa a un valor de la colección.
- El **<statement list>** representa la serie de comandos a ejecutar para cada elemento de la colección.

Con el bucle *foreach*, es posible emplear *cmdlets* que devuelvan una colección de elementos. De esta forma se puede iterar cada uno de estos elementos.

Cabe remarcar la existencia de la variable automática **\$foreach**, la cual existe únicamente cuando corre el bucle. Esta variable es categorizada como *enumeradora*, y puede ser usada para iterar sobre los elementos procesados por los bloques de código. Los métodos que proporciona son:

- **MoveNext =>** Este método avanza el enumerador al siguiente elemento de la colección. Devuelve *\$true* si se realiza la operación de manera exitosa; *\$false* en cualquier otro caso. Para evitar que se muestre el resultado de esta operación por pantalla (*output stream*), se puede hacer un pipe a *Out-Null*.
`$foreach.MoveNext() | Out-Null`
- **Reset =>** Establece el enumerador a su posición inicial, el cual es **anterior** al primer elemento de la colección.
- **Current =>** Consigue el elemento de la colección de la posición actual del enumerador.

Ejemplos de ejecución:

```
PS D:\tmp2\tmp> cat .\foreach1.ps1
$ssoo = "freebsd", "openbsd", "solaris", "fedora", "ubuntu", "netbsd"
foreach ($so in $ssoo)
{
    Write-Host $so
}
PS D:\tmp2\tmp> .\foreach1.ps1
freebsd
openbsd
solaris
fedora
ubuntu
netbsd
PS D:\tmp2\tmp> █
```

Ilustración 34.. Ejemplo de bucle Foreach: Ejemplo básico

```
PS D:\tmp2\tmp> cat .\foreach2.ps1
foreach ($archivo in Get-ChildItem)
{
    if ($archivo.length -ge 10KB)
    {
        Write-Host $archivo -> [($archivo.length)]
    }
}
PS D:\tmp2\tmp> .\foreach2.ps1
.condicionales2.ps1.un~ -> [ 10591 ]
.switch7.ps1.un~ -> [ 28443 ]
PS D:\tmp2\tmp> █
```

Ilustración 35. Ejemplo de bucle Foreach: Uso de cmdlets

Bucle While

La declaración *While* permite ejecutar un bloque de código mientras se satisfaga una condición; es decir, se evalúe a `$true`. Este bucle es más sencillo de construir que un bucle *For*, ya que su sintaxis es más directa.

Su sintaxis es la siguiente:

```
1 while (<condition>) { <statement list> }
```

Ilustración 36. Sintaxis del bucle While

Cuando se ejecuta este tipo de bucle, PowerShell evalúa la <condition> **ANTES** de entrar en el bloque de código definido en <statement list>. La condición se resuelve como verdadera o falsa. Siempre y cuando la condición sea verdadera, se ejecutará el bloque de código.

Algunos ejemplos de ejecución:

```
PS D:\tmp2\tmp> cat .\while.ps1
$num = 0

while ($num -ne 3)
{
    $num++
    Write-Host $num
}
PS D:\tmp2\tmp> .\while.ps1
1
2
3
PS D:\tmp2\tmp> █
```

Ilustración 37. Ejemplo bucle While: Ejemplo básico

```
PS D:\tmp2\tmp> cat .\while2.ps1
$num = 0

while ($num -ne 5)
{
    if ($num -eq 1) { $num = $num + 3 ; Continue }
    $num++
    Write-Host $num
}
PS D:\tmp2\tmp> .\while2.ps1
1
5
PS D:\tmp2\tmp> █
```

Ilustración 38. Ejemplo bucle While: Uso de una palabra clave

Bucle Do

La declaración *Do* funciona con el keyword *While* o *Until* para correr un bloque de código, sujeto a una condición. La diferencia entre un bucle *While* y un *Do-While* es que, en el segundo, el código siempre se va a correr, ya que la condición se revisa **DESPUÉS** de la ejecución del bloque.

La diferencia entre un bucle *Do-Until* y *Do-While* es que el primero continua si la condición se evalúa a `$false`, mientras que la condición del segundo a `$true`.

La sintaxis de este bucle es la siguiente:



```
1  do { <statement list> } until (<condition>)
2
3  do { <statement list> } while (<condition>)
```

Ilustración 39. Sintaxis del bucle Do

Ejemplos de ejecución:

```
PS D:\tmp2\tmp> cat .\do-while.ps1
$valor = 5
$multiplicacion = 1
do
{
    $multiplicacion = $multiplicacion * $valor
    $valor--
}
while ($valor -gt 0)

Write-Host $multiplicacion
PS D:\tmp2\tmp> .\do-while.ps1
120
PS D:\tmp2\tmp> █
```

Ilustración 40. Ejemplo Do-While


```

PS D:\tmp2\tmp> cat .\do-until.ps1
$valor = 5
$multiplicacion = 1
do
{
    $multiplicacion = $multiplicacion * $valor
    $valor--
}
until ($valor -eq 0)

Write-Host $multiplicacion
PS D:\tmp2\tmp> .\do-until.ps1
120
PS D:\tmp2\tmp> █

```

Ilustración 41. Ejemplo Do-Until: Misma lógica que el ejemplo anterior, pero adaptado al Do-Until

Las declaraciones *Break* y *Continue*.

Estas declaraciones se pueden emplear en los bucles *For*, *Foreach*, *Do* y *While*. Además, se pueden utilizar en declaraciones del tipo *Switch* y *Trap*.

La declaración *Break* permite ignorar el resto de la declaración, saliendo del bloque de código. Esto quiere decir que no se ejecutarán las demás iteraciones, o coincidencias, y se pasará a la siguiente parte del script.

En cambio, el declarante *Continue*, ejecuta directamente la siguiente iteración, o coincidencia, de la declaración. Es útil para evitar elementos dentro de colecciones, por ejemplo.

No se debe utilizar ni *Break* ni *Continue* fuera de bucles, *switch* o *trap*. Esto se debe a que PowerShell buscaría en la *call stack* una estructura de la que salir o continuar. Si no se encuentra tal estructura, se terminaría el programa abruptamente.

Ejemplos de ejecución:

```
PS D:\tmp2\tmp> cat .\break1.ps1
$num = 10

for($i = 2; $i -lt 10; $i++)
{
    $num = $num+$i
    if ($i -eq 5) { Break }
}

Write-Host $num
Write-Host $i
PS D:\tmp2\tmp> .\break1.ps1
24
5
PS D:\tmp2\tmp> █
```

Ilustración 42. Ejemplo de Break: En un bucle For

```
PS D:\tmp2\tmp> cat .\break2.ps1
$cadena = "Hola, buenas tardes"
$cadena2 = "Hola, buenas noches"

switch -Wildcard ($cadena, $cadena2)
{
    "Hola, buenas*" {"[$_] coincide con [Hola, buenas*]}
    "Hola, bue*" {"[$_] coincide con [Hola, bue*]}
    "Hola,*" {"[$_] coincide con [Hola,*]"; Break }
    "Hola, buenas tardes" {"[$_] coincide con [Hola, buenas tardes]}
}
PS D:\tmp2\tmp> .\break2.ps1
[Hola, buenas tardes] coincide con [Hola, buenas*]
[Hola, buenas tardes] coincide con [Hola, bue*]
[Hola, buenas tardes] coincide con [Hola,*]
PS D:\tmp2\tmp> █
```

Ilustración 43. Ejemplo de Break: En un Switch

```

PS D:\tmp2\tmp> cat .\continue1.ps1
$num = 10

for($i = 2; $i -lt 10; $i++)
{
    if ($i -eq 5) { Continue }
    $num = $num+$i
}

Write-Host $num
Write-Host $i
PS D:\tmp2\tmp> .\continue1.ps1
49
10
PS D:\tmp2\tmp> █

```

Ilustración 44. Ejemplo de Continue: En un bucle For

```

PS D:\tmp2\tmp> cat .\continue2.ps1
$cadena = "Hola, buenas tardes"
$cadena2 = "Hola, buenas noches"

switch -Wildcard ($cadena, $cadena2)
{
    "Hola, buenas*" {"[$_] coincide con [Hola, buenas*]}
    "Hola, bue*" {"[$_] coincide con [Hola, bue*]"; Continue}
    "Hola,*" {"[$_] coincide con [Hola,*]}
    "Hola, buenas tardes" {"[$_] coincide con [Hola, buenas tardes]}
}
PS D:\tmp2\tmp> .\continue2.ps1
[Hola, buenas tardes] coincide con [Hola, buenas*]
[Hola, buenas tardes] coincide con [Hola, bue*]
[Hola, buenas noches] coincide con [Hola, buenas*]
[Hola, buenas noches] coincide con [Hola, bue*]
PS D:\tmp2\tmp> █

```

Ilustración 45. Ejemplo de Continue: En un Switch

Cmdlets.

En PowerShell, un *cmdlet* (*command-let*) es una pequeña función que realiza una tarea específica. Los *cmdlets* son los bloques de construcción fundamentales en PowerShell.

Los *cmdlets* están diseñados para ser fáciles de usar y de combinar con otros *cmdlets* para crear scripts complejos. Los *cmdlets* deben ser simples y estar diseñados para ser utilizados en compañía de otros *cmdlets*. Es por ello por lo que realizan una acción y suelen devolver un objeto de *Microsoft .NET* al siguiente comando en canalización (|).

Los cmdlets se diferencian de los comandos de otros entornos de *shell* por los siguientes motivos:

- Son instancias de clases de .NET; no son ejecutables independientes.
- Se pueden crear a partir de pocas líneas de código.
- No suelen realizar su propio análisis, presentación de errores o formato de salida. El tiempo de ejecución de PowerShell controla el análisis, la presentación de errores y el formato de salida.
- Procesan objetos de entrada de la canalización, en lugar de secuencias de texto. Suelen entregar objetos como salida a la canalización.
- Están orientados a registros porque procesan un único objeto a la vez.

Estos comandos son reconocidos por su formato de nombre. Se nombran siempre de la misma manera: Un verbo y un nombre, separadas ambas palabras por un guión (-). El verbo identifica la acción que realiza el cmdlet y el nombre identifica el recurso en el que el cmdlet realiza la acción.

PowerShell incluye muchos *cmdlets* integrados que realizan una amplia variedad de tareas, desde la gestión de archivos y directorios hasta el control de servicios y procesos del sistema. También es posible crear un *cmdlet* personalizado empleando lenguajes como C#. De esta manera se puede aumentar la funcionalidad del sistema.

Sus nombres son muy descriptivos. Explican qué hacen (*Get*, *Set*, *Format*, *Out...*) y sobre qué sujeto lo hacen. Algunos ejemplos de *cmdlets* son:

- `Get-ChildItem`: Devuelve una lista de los elementos secundarios de una ubicación especificada, como archivos y carpetas.
- `Get-Process`: Muestra una lista de los procesos que se están ejecutando en el sistema.
- `New-Item`: Crea un nuevo archivo o carpeta en una ubicación especificada.
- `Set-Content`: Escribe contenido en un archivo especificado.
- `Remove-Item`: Elimina un archivo o carpeta especificado.
- `Get-Service`: Muestra una lista de los servicios que se están ejecutando en el sistema.
- `Set-Service`: Cambia el estado de un servicio especificado, como iniciarlo o detenerlo.
- `Get-NetIPAddress`: Devuelve una lista de las direcciones IP del sistema.
- `Test-Connection`: Realiza un ping a una dirección IP especificada para comprobar si está disponible.
- `Invoke-WebRequest`: Realiza una solicitud HTTP o HTTPS a una dirección URL especificada.
- `Get-Help`: Muestra ayuda acerca de un cmdlet

Para obtener información sobre los cmdlets del sistema, se puede emplear el cmdlet `Get-Command`. Con el siguiente comando, se muestran todos los cmdlets agrupados por recurso.

```
Get-Command -Type Cmdlet | Sort-Object -Property Noun |
Format-Table -GroupBy Noun
```

La salida de la ejecución anterior sería:

```
PS C:\> Get-Command -Type Cmdlet | Sort-Object -Property Noun | Format-Table -GroupBy Noun
```

Noun: Acl			
CommandType	Name	Version	Source
-----	----	-----	-----
Cmdlet	Set-Acl	3.0.0.0	Microsoft.PowerShell.Security
Cmdlet	Get-Acl	3.0.0.0	Microsoft.PowerShell.Security

Noun: Alias			
CommandType	Name	Version	Source
-----	----	-----	-----
Cmdlet	Set-Alias	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Import-Alias	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Get-Alias	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	New-Alias	3.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Export-Alias	3.1.0.0	Microsoft.PowerShell.Utility

Ilustración 46. Listado de los cmdlets del sistema

Hay una serie de términos que se repiten en los manuales de los *cmdlets*:

- **Atributo:** Atributo de .NET que se emplea para declarar una clase de *cmdlet* como *cmdlet*. Aunque PowerShell utiliza otros atributos que son opcionales, se requiere le atributo *Cmdlet*.
- **Parámetro:** Propiedades públicas que definen los parámetros que están disponibles para el usuario o aplicación que ejecuta el cmdlet. Los cmdlets pueden tener parámetros obligatorios, con nombre, posicionales y *modificadores*. Los parámetros *switch* permiten definir parámetros que se evalúan solo si los parámetros se especifican en la llamada.
- **Conjunto de parámetros:** Grupo de parámetros que pueden usarse en el mismo comando para realizar una acción específica. Un cmdlet puede tener varios conjuntos de parámetros, pero cada conjunto de parámetros debe tener al menos un parámetro que sea único.
- **Parámetro dinámico:** Parámetro que se agrega al cmdlet en tiempo de ejecución. Normalmente, los parámetros dinámicos se agregan al cmdlet cuando otro parámetro se establece en un valor específico.
- **Métodos de procesamiento de entrada:** La clase *System.Management.Automation.Cmdlet* proporciona los siguientes métodos virtuales que se usan para procesar registros. Todas las clases de cmdlet derivadas deben invalidar uno o varios de los tres primeros métodos:

- System.Management.Automation.Cmdlet.BeginProcessing: Se usa para proporcionar funcionalidad opcional de procesamiento previo y único para el cmdlet.
 - System.Management.Automation.Cmdlet.ProcessRecord: Se usa para proporcionar la funcionalidad de procesamiento de registro por registro para el cmdlet. Es posible que se llame a este método más de una vez.
 - System.Management.Automation.Cmdlet.EndProcessing: Proporciona una funcionalidad opcional de procesamiento posterior al procesamiento para el cmdlet.
 - System.Management.Automation.Cmdlet.StopProcessing: Detener el procesamiento.
- **Característica ShouldProcess**: PowerShell permite crear cmdlets que solicitan al usuario comentarios antes de que el cmdlet haga un cambio en el sistema. Para esta característica, el cmdlet debe declarar que admite la característica al declarar el atributo *Cmdlet* y el cmdlet debe llamar a los métodos **ShouldProcess**: `System.Management.Automation.Cmdlet.ShouldProcess` y `System.Management.Automation.Cmdlet.ShouldContinue`, desde dentro de un método de procesamiento de entrada.
 - **Transacción**: Grupo lógico de comandos que se tratan como una única tarea. Se produce un error para toda la transacción si surge uno en alguna de las subtareas. El usuario tiene la opción de aceptar o rechazar las acciones realizadas dentro de la transacción hasta el punto de fallo. Para participar en una transacción, el cmdlet debe declarar que admite transacciones cuando se declara el atributo *Cmdlet*.

Existen diversas formas de crear un *cmdlet*. Algunas opciones son:

- Escribir un script de PowerShell y convertirlo en un *cmdlet* mediante la función **New-Module**.
- Escribir un *cmdlet* empleando un lenguaje de programación .NET, como C#. A continuación, compilarlo y cargarlo en PowerShell utilizando el *cmdlet* **Import-Module**.

- Utilizar el modelo de objetos de administración de Windows (WMI¹) o el modelo de objetos de *Common Information Model* (CIM).

Tal y como se ha indicado anteriormente, PowerShell utiliza un par de nombres verbo y sustantivo para dar nombre a los cmdlets.

- **Sustantivo:** Especifica el recurso sobre los que actúa el cmdlet. Es el que diferencia los cmdlets de otros cmdlets. Si el nombre del recurso fuese muy genérico (por ejemplo, *Server*), se tendría que usar un **prefijo** (SQLServer). La combinación de un sustantivo específico con un verbo más general permite una rápida localización, evitando además la duplicidad de nombres.
- **Verbo:** Al especificar un verbo, las directrices de desarrollo requieren que se use uno de los verbos predefinidos proporcionados por PowerShell. Esta regla garantizará la coherencia entre los cmdlets escritos y de Microsoft.

Cuando se asigne un nombre, se debe usar la *Pascal case*.

El ya mencionado cmdlet `Get-Command` presenta una serie de opciones muy útiles para obtener información acerca de los cmdlets del sistema:

- El listado de cmdlets ya probado.

```
Get-Command -Type Cmdlet | Sort-Object -Property Noun |
Format-Table -GroupBy Noun
```

¹ **Windows Management Instrumental (WMI)** es una tecnología de Microsoft que proporciona un marco estándar para el acceso y la administración de recursos en un sistema operativo Windows. Se basa en el modelo de objetos de *Common Information Model* (CIM), que proporciona una forma común de representar los recursos de hardware y software en un sistema, y permite a los administradores de sistemas realizar tareas de administración y monitoreo de manera centralizada.
<https://learn.microsoft.com/es-es/windows/win32/wmisdk/wmi-start-page>

- Obtener la sintaxis de un cmdlet concreto.

```
Get-Command -Name <cmdlet> -Args Cert: -Syntax
```

```
PS C:\> Get-Command -Name Get-Childitem -Args Cert: -Syntax

Get-ChildItem [[-Path <string[]>] [-Filter <string>] [-Include <string[]>] [-Exclude <string[]>] [-Recurse] [-Depth <uint32>] [-Force] [-Name] [-UseTransaction] [-CodeSigningCert] [-DocumentEncryptionCert] [-SSLServerAuthentication] [-DnsName <DnsNameRepresentation>] [-Eku <string[]>] [-ExpiringInDays <int>] [<CommonParameters>]]

Get-ChildItem [[-Filter <string>] -LiteralPath <string[]> [-Include <string[]>] [-Exclude <string[]>] [-Recurse] [-Depth <uint32>] [-Force] [-Name] [-UseTransaction] [-CodeSigningCert] [-DocumentEncryptionCert] [-SSLServerAuthentication] [-DnsName <DnsNameRepresentation>] [-Eku <string[]>] [-ExpiringInDays <int>] [<CommonParameters>]]

PS C:\>
```

Ilustración 47. Obtener la sintaxis de un cmdlet

- Obtener el cmdlet de un alias.

```
Get-Command -Name <Alias>
```

```
PS C:\> Get-Command -Name dir

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

PS C:\> █
```

Ilustración 48. Obtener el cmdlet bajo un alias

- Obtener los cmdlets de un recurso.

```
Get-Command -Noun <Recurso>
```

```
PS C:\WINDOWS\system32> Get-Command -Noun WSMANInstance

CommandType      Name                      Version      Source
-----
Cmdlet            Get-WSManInstance        3.0.0.0      Microsoft.WSMan.Management
Cmdlet            New-WSManInstance        3.0.0.0      Microsoft.WSMan.Management
Cmdlet            Remove-WSManInstance     3.0.0.0      Microsoft.WSMan.Management
Cmdlet            Set-WSManInstance        3.0.0.0      Microsoft.WSMan.Management

PS C:\WINDOWS\system32> █
```

Ilustración 49. Cmdlets de un recurso específico

Objetos y Pipeline

Como ya se comentó anteriormente, con PowerShell no se trabaja solamente con texto, sino que también tiene una salida basada en objetos, proporcionada por el framework .NET en el que se apoya.

Objetos.

Un objeto es una estructura de datos que contiene propiedades y métodos que se pueden acceder y manipular.

Para trabajar con estos objetos de una forma estandarizada y coherente, se pueden utilizar cmdlets específicos, lo que hace que el trabajo sea más fácil y eficiente.

Entre los cmdlets básicos para trabajar con objetos en PowerShell encontramos:

Get-Member

Se utiliza para obtener información acerca de los miembros (propiedades y métodos) de un objeto.

Por ejemplo, se puede obtener información sobre qué propiedades y métodos tiene un objeto de tipo servicio:

```
PS D:\Desktop> Get-Service -Name "LSM" | Get-Member

TypeName: System.ServiceProcess.ServiceController

Name            MemberType      Definition
----            -
Name            AliasProperty  Name = ServiceName
RequiredServices AliasProperty  RequiredServices = ServicesDependedOn
Disposed        Event          System.EventHandler Disposed(System.Object, System.EventArgs)
Close           Method         void Close()
Continue        Method         void Continue()
CreateObjRef    Method         System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Dispose         Method         void Dispose(), void IDisposable.Dispose()
Equals          Method         bool Equals(System.Object obj)
ExecuteCommand  Method         void ExecuteCommand(int command)
GetHashCode     Method         int GetHashCode()
GetLifetimeService Method       System.Object GetLifetimeService()
GetType         Method         type GetType()
InitializeLifetimeService Method       System.Object InitializeLifetimeService()
Pause           Method         void Pause()
Refresh         Method         void Refresh()
Start           Method         void Start(), void Start(string[] args)
Stop            Method         void Stop()
WaitForStatus   Method         void WaitForStatus(System.ServiceProcess.ServiceControllerStatus
CanPauseAndContinue Property       bool CanPauseAndContinue {get;}
CanShutdown     Property       bool CanShutdown {get;}
CanStop         Property       bool CanStop {get;}
Container       Property       System.ComponentModel.IContainer Container {get;}
```

Ilustración 50. Get-Member: Información de un objeto

Si lo que se quisiese obtener fuese solo información sobre las propiedades de un objeto, habría que añadir a este cmdlet el parámetro -MemberType:

```
PS D:\Desktop> Get-Service -Name "LSM" | Get-Member -MemberType Property

TypeName: System.ServiceProcess.ServiceController

Name            MemberType      Definition
----            -
CanPauseAndContinue Property       bool CanPauseAndContinue {get;}
CanShutdown     Property       bool CanShutdown {get;}
CanStop         Property       bool CanStop {get;}
Container       Property       System.ComponentModel.IContainer Container {get;}
```

Ilustración 51. Get-Member: Propiedades de un objeto

Para las propiedades, la columna “Definition” nos proporciona información muy útil que se estructura en 3 partes:

- El tipo de dato que devuelve la propiedad (bool, string, otro objeto, ...).
- El mismo nombre de la propiedad.
- Y si es una propiedad solo de lectura {get;} o también de escritura {get;set;}.

Para obtener solo información sobre los métodos de un objeto podemos utilizar el mismo parámetro:

```
PS D:\Desktop> Get-Item .\test.txt | Get-Member -MemberType Method

TypeName: System.IO.FileInfo

Name                MemberType Definition
-----
AppendText          Method      System.IO.StreamWriter AppendText()
CopyTo              Method      System.IO.FileInfo CopyTo(string destFileName), System.IO.FileInfo CopyTo(string destFileName, bool overwrite)
Create              Method      System.IO.FileStream Create()
CreateObjRef        Method      System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
CreateText           Method      System.IO.StreamWriter CreateText()
```

Ilustración 52. Get-Member: Información sobre los métodos de un objeto.

Para los métodos, la columna “Definition” también nos proporciona información muy útil que se estructura en 3 partes:

- El tipo de dato que devuelve el método (void, string, otro objeto, ...).
- El mismo nombre del método.
- Entre paréntesis, el tipo de dato que recibe el método por parámetro.

Select-Object

Se utiliza para seleccionar y mostrar únicamente el valor de las propiedades de un objeto que se desean.

Por ejemplo, en la imagen siguiente se ha indicado que se muestren simplemente el valor de las propiedades “Name” y “Length”, si no se hubiera utilizado este cmdlet también se mostrarían otras propiedades como “Mode” y “LastWriteTime”.

```
PS D:\Desktop> Get-Item .\test.txt | Select-Object Name, Length

Name      Length
-----
test.txt  0
```

Ilustración 53. Select-Object: Filtrado de columnas

Este cmdlet también se puede utilizar para seleccionar un subconjunto de objetos de una colección. Esto es útil cuando se trabaja con colecciones grandes y se desea reducir el número de objetos en la salida.

Existen varios parámetros que nos permiten acotar una salida amplia de objetos, como puede ser el parámetro `-First <int>`, que nos devuelve el número `<int>` de objetos desde el inicio de la colección, el parámetro `-Last <int>`, que nos devuelve el número `<int>` de objetos desde el final de la colección o el parámetro `Skip <int>`, que se salta los `<int>` primeros objetos de la colección y nos devuelve el resto. Veamos algún ejemplo práctico:

```
PS D:\Desktop> Get-Service | Select-Object -Last 5
```

Status	Name	DisplayName
Stopped	XblAuthManager	Administración de autenticación de ...
Stopped	XblGameSave	Partida guardada en Xbox Live
Stopped	XboxGipSvc	Xbox Accessory Management Service
Stopped	XboxNetApiSvc	Servicio de red de Xbox Live
Running	ZeroConfigService	Intel(R) PROSet/Wireless Zero Confi...

Ilustración 55. Select-Object: Obtener las últimas 5 filas

```
PS D:\Desktop> Get-Service | Select-Object -First 5
```

Status	Name	DisplayName
Stopped	AarSvc_38f150e	Agent Activation Runtime_38f150e
Running	AdobeARMSvc	Adobe Acrobat Update Service
Stopped	AJRouter	Servicio de enrutador de AllJoyn
Stopped	ALG	Servicio de puerta de enlace de niv...
Stopped	AppIDSvc	Identidad de aplicación

Ilustración 54. Select-Object: Obtener las primeras 5 filas

Where-Object

Se utiliza para filtrar y seleccionar únicamente los objetos que cumplen con una condición específica. Por ejemplo, en la siguiente imagen se seleccionan únicamente los objetos de tipo servicios que están en estado “Running”.

```
PS D:\Desktop> Get-Service | Where-Object {$_.Status -eq "Running"}

Status  Name                DisplayName
-----
Running AdobeARMService    Adobe Acrobat Update Service
Running Appinfo       Información de la aplicación
Running AppMgmt       Administración de aplicaciones
Running AppXSvc       Servicio de implementación de AppX ...
Running AudioEndpointBu...  Compilador de extremo de audio de W...
Running Audiosrv      Audio de Windows
Running BFE           Motor de filtrado de base
```

Ilustración 56. Where-Object: Filtrado por estado

Si bien existen muchos otros cmdlets que pueden ser útiles para trabajar con objetos, nos centraremos en explorar las propiedades y métodos disponibles para los objetos y cómo se pueden utilizar para cumplir nuestros objetivos.

Las propiedades son atributos que definen las características de un objeto y se puede acceder a sus valores y manipularlos utilizando la notación de punto (.) y el nombre de la propiedad.

Por ejemplo, se puede acceder de esta manera al valor de la propiedad “IsReadOnly” de un objeto de tipo archivo y modificar dicho valor:

```
PS D:\Desktop> (Get-Item .\test.txt).IsReadOnly
False
PS D:\Desktop> (Get-Item .\test.txt).IsReadOnly = 1
PS D:\Desktop> (Get-Item .\test.txt).IsReadOnly
True
```

Ilustración 57. Acceso a los atributos de un objeto

Los métodos son acciones que se pueden realizar sobre un objeto y se invocan de manera similar a las propiedades, usando la notación de punto, el nombre del método y terminando con el operador de invocación (), que dependiendo de su definición puede llevar o no parámetros.

Estas acciones pueden variar desde simples operaciones de lectura y escritura hasta operaciones más avanzadas, como la ejecución de comandos o la manipulación de archivos, así que vamos a ver un ejemplo en el que se use el método `CopyTo(string path)` para crear un archivo a partir de otro y el método `Delete()` para borrar un archivo:

```
PS D:\Desktop> Get-ChildItem *.txt

Directorio: D:\Desktop

Mode                LastWriteTime         Length Name
----                -
-a----            06/04/2023    19:50             100 test.txt

PS D:\Desktop> (Get-Item .\test.txt).CopyTo("D:\Desktop\prueba.txt")

Mode                LastWriteTime         Length Name
----                -
-a----            06/04/2023    19:50             100 prueba.txt

PS D:\Desktop> (Get-Item .\test.txt).Delete()
PS D:\Desktop> Get-ChildItem *.txt

Directorio: D:\Desktop

Mode                LastWriteTime         Length Name
----                -
-a----            06/04/2023    19:50             100 prueba.txt
```

Ilustración 58. Ejemplos de uso de métodos de un objeto

Además de utilizar objetos predefinidos en PowerShell, otra de las funcionalidades que proporciona es la creación de nuevos objetos personalizados. Esto se puede realizar de varias maneras, así que vamos a ver 3 de ellas:

1. Se puede utilizar la clase base `PSObject` para que con el cmdlet `New-Object` se pueda crear un objeto vacío que se pueda ir expandiendo dinámicamente con la agregación de propiedades y métodos mediante el cmdlet `Add-Member`:

```
PS D:\Desktop> $miObjeto = New-Object PSObject
PS D:\Desktop> $miObjeto | Add-Member -MemberType NoteProperty -Name Nombre -Value "Miguel"
PS D:\Desktop> $miObjeto | Add-Member -MemberType NoteProperty -Name Edad -Value 23
PS D:\Desktop> $miObjeto | Add-Member -MemberType ScriptMethod -Name Saludar -Value { Write-Host "¡Hola Mundo!" }
```

Ilustración 59. Expansión de un objeto de forma dinámica.

En este ejemplo, primero creamos un objeto vacío utilizando el cmdlet `New-Object` y lo asignamos a la variable `$miObjeto`. Luego, utilizamos el cmdlet `Add-`

Member dos veces para agregar propiedades y una vez para agregar un método que al ser invocado escribirá un texto por consola.

2. Otra forma de crear un objeto nuevo de cero similar a la anterior es añadiendo las propiedades a través del paso de una tabla hash durante la creación del objeto:

```
PS D:\Desktop> $miObjeto = New-Object -TypeName PSObject -Property @{
>>     Nombre = "Miguel"
>>     Edad = 23
>> }
PS D:\Desktop> $miObjeto | Add-Member -MemberType ScriptMethod -Name Saludar -Value { Write-Host "¡Hola Mundo!" }
PS D:\Desktop> $miObjeto | Get-Member

TypeName: System.Management.Automation.PSCustomObject

Name      MemberType Definition
-----
Equals     Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
ToString   Method      string ToString()
Edad       NoteProperty int Edad=23
Nombre     NoteProperty string Nombre=Miguel
Saludar    ScriptMethod System.Object Saludar();
```

Ilustración 60. Adición de propiedades utilizando una 'Hash Table'

3. Una tercera forma de crear un objeto nuevo de cero es utilizar el acelerador de tipo `PSCustomObject` delante de la definición de la tabla hash, lo que ahorra la creación del objeto con el cmdlet `New-Object`.

Cuando se usa este acelerador de tipo, se crea automática y dinámicamente una nueva clase que no se almacena en el disco. Es una forma rápida y conveniente de crear objetos que no requieren una definición de clase previa. Vamos a ver un ejemplo de ello:

```

PS D:\Desktop> $miObjeto = [PSCustomObject] @{
>>     Nombre = "Miguel"
>>     Edad = 23
>> }
PS D:\Desktop> $miObjeto | Add-Member -MemberType ScriptMethod -Name Saludar -Value { Write-Host "¡Hola Mundo!" }
PS D:\Desktop> $miObjeto | Get-Member

TypeName: System.Management.Automation.PSCustomObject

Name      MemberType Definition
-----
Equals     Method      bool Equals(System.Object obj)
GetHashCode Method      int GetHashCode()
GetType    Method      type GetType()
ToString   Method      string ToString()
Edad       NoteProperty int Edad=23
Nombre     NoteProperty string Nombre=Miguel
Saludar    ScriptMethod System.Object Saludar();

```

Ilustración 61. Creación de objeto utilizando el acelerador de tipo PSCustomObject

El **pipeline** es una característica que permite encadenar comandos y operaciones para procesar datos de manera eficiente. Esto significa que la salida de un comando se puede pasar directamente como entrada al siguiente comando, lo que permite realizar múltiples tareas en una sola línea de código.

Sin embargo, no todos los comandos se pueden encadenar. Por tanto, para saber si se puede encadenar un comando se tiene que utilizar la ayuda. En concreto, se puede utilizar el cmdlet `Get-Help`, acompañado del parámetro `-Full` y el comando del que se quiere obtener información. De esta manera, prestando atención a las secciones `INPUTS` y `OUTPUTS` se puede saber si admite la posibilidad de encadenar y qué tipos de datos admite.

Vamos a ver un caso en el que sí sea posible la concatenación de comandos:

Si se quiere parar un proceso en ejecución a partir del nombre de dicho proceso, se puede intentar encadenar los comandos `Get-Process` y `Stop-Process` respectivamente de esta manera:

```
Get-Process -Name Acrobat | Stop-Process
```

Ilustración 62. Parar un proceso empleando pipelines y cmdlets

Para que el encadenamiento sea posible, la salida del cmdlet `Get-Process` tiene que ser compatible con la entrada del cmdlet `Stop-Process`, así que vamos a analizarlas con el cmdlet de ayuda `Get-Help`:

```
Get-Help -Full Get-Process

SALIDAS
    System.Diagnostics.ProcessModule
    System.Diagnostics.FileVersionInfo
    System.Diagnostics.Process
```

Ilustración 64. Salidas de `Get-Process`

```
Get-Help -Full Stop-Process

ENTRADAS
    System.String[]
    System.Int32[]
    System.Diagnostics.Process[]
```

Ilustración 63. Entradas de `Stop-Process`

Un objeto `System.Diagnostics.Process` representa un proceso en ejecución en el sistema y como se puede observar es una de las posibles salidas del cmdlet `Get-Process` y una de las posibles entradas del cmdlet `Stop-Process`.

Por lo tanto, el encadenamiento de ambos comandos es posible y produce el efecto deseado:

```
PS D:\Desktop> Get-Process
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
457	55	46748	80204	0,94	3380	7	Acrobat
512	19	38488	52152	0,41	7316	7	Acrobat
342	20	8884	24408	0,09	2072	7	AcroCEF
357	22	41584	77988	0,45	5644	7	AcroCEF

```
PS D:\Desktop> Get-Process -Name Acrobat | Stop-Process
```

```
PS D:\Desktop> Get-Process
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
140	11	11336	11820	0,09	13380	7	ai
361	21	8884	6692	1,48	4028	7	ApplicationFrameHost
128	8	1596	588		5064	0	armsvc

Ilustración 65. Muestra del funcionamiento de la ejecución.

Vamos a ver un caso en el que no sea posible la concatenación de comandos:

Por ejemplo, el cmdlet `Get-Clipboard` se utiliza para obtener el contenido del portapapeles, por lo que no recibe ningún objeto como entrada. Si lo intentas encadenar mediante pipeline con un cmdlet que devuelva por salida algún tipo de objeto, se producirá un error:

```
Get-Help -Full Get-ChildItem

SALIDAS
    System.IO.FileInfo
    System.IO.DirectoryInfo
```

```
Get-Help -Full Get-Clipboard

ENTRADAS
    Ninguno
```

```

PS D:\Desktop> Get-ChildItem *.txt | Get-Clipboard
System.String[]
Get-Clipboard : El objeto de entrada no se puede enlazar a ninguno de los parámetros del comando, bien porque el
comando no admite la entrada de canalización o porque la entrada y sus propiedades no coinciden con ninguno de los
parámetros que aceptan entradas de canalización.
En línea: 1 Carácter: 23
+ Get-ChildItem *.txt | Get-Clipboard
+ ~~~~~
+ CategoryInfo          : InvalidArgument: (D:\Desktop\*.txt:PSObject) [Get-Clipboard], ParameterBindingException
+ FullyQualifiedErrorId : InputObjectNotBound,Microsoft.PowerShell.Commands.GetClipboardCommand

```

Ilustración 66. Salidas de Get-ChildItem ; Entradas de Get-Clipboard ; Error de canalización con pipes

A pesar de no haberlo especificado hasta ahora, es importante tener en cuenta que existen **dos tipos de encadenación con pipeline**: por **valor** y por **nombre** de la propiedad.

Vamos a hacer uso del cmdlet Stop-Service para ver las diferencias de estos dos tipos.

Como se puede observar gracias al cmdlet Get-Help, Stop-Service puede recibir por entrada 2 tipos de valores: O cadenas de texto u objetos de tipo ServiceController

```

Get-Help -Full Stop-Service
ENTRADAS
System.String[]
System.ServiceProcess.ServiceController[]

```

Ilustración 67. Entradas del cmdlet Get-Help

Si analizamos los parámetros de este cmdlet, vemos que el parámetro -Name acepta entrada de canalización tanto por valor como por nombre de la propiedad, en cambio el parámetro -InputObject acepta entrada de canalización por valor para objetos ServiceController:

```

-InputObject <ServiceController[]>

¿Requerido?           true
¿Posición?             0
¿Aceptar canalización? true (ByValue)
Nombre de conjunto de parámetros      InputObject
Alias                  Ninguno
¿Dinámico?            false

-Name <string[]>

¿Requerido?           true
¿Posición?             0
¿Aceptar canalización? true (ByValue, ByPropertyName)
Nombre de conjunto de parámetros      Default
Alias                  ServiceName
¿Dinámico?            false

```

Ilustración 68. Entrada de canalización del cmdlet Stop-Service

Visto esto, sabemos que si se canaliza el resultado de un comando que produce un objeto de tipo ServiceController a Stop-Service, se vincula esa entrada al parámetro

-InputObject. En cambio, si se canaliza el resultado de un comando que produce una salida de cadena a Stop-Service, lo vincula al parámetro -Name. Este es el funcionamiento de encadenación con pipeline por valor.

Vamos a ver estos 2 ejemplos de canalización por valor:

En el primero, se canaliza a Stop-Service la salida que produce el cmdlet Get-Service, que es de tipo ServiceController:

```
PS D:\Desktop> Get-Service
Running Spooler Cola de impresión

PS D:\Desktop> Get-Service Spooler | Stop-Service

PS D:\Desktop> Get-Service
Stopped Spooler Cola de impresión
```

Ilustración 69. Primer ejemplo de canalización por valor

En el segundo, se canaliza a Stop-Service una cadena de texto:

```
PS D:\Desktop> Get-Service
Running Spooler Cola de impresión

PS D:\Desktop> "Spooler" | Stop-Service

PS D:\Desktop> Get-Service
PS D:\Desktop> Get-Service
```

Ilustración 70. Segundo ejemplo de canalización por valor

El funcionamiento de encadenación con pipeline por nombre de la propiedad se puede explicar con el comando -Name. Si se canaliza el resultado de un comando que no produce un objeto de tipo ServiceController ni de tipo cadena, pero produce una salida que contiene una propiedad llamada Name, entonces se vincula la propiedad Name de la salida al parámetro -Name de Stop-Service

Vamos a ver con un ejemplo la canalización por nombre.

```
PS D:\Desktop> Get-Service
Running Spooler          Cola de impresión

PS D:\Desktop> $miObjeto = [PSCustomObject] @{
>>   Name = "Spooler"
>> }
PS D:\Desktop> $miObjeto | Stop-Service

PS D:\Desktop> Get-Service
PS D:\Desktop> Get-Service
```

Ilustración 71. Ejemplo de canalización por nombre

Funciones

En PowerShell podemos hacer uso de las llamadas funciones. Una función es un bloque de código con un nombre definido, que permite “llamar” por este nombre a ese bloque de código, una o varias veces en nuestro script, para evitar la repetición de un conjunto de instrucciones continuamente.

Una de las razones por las que se usan mucho las funciones, es la capacidad de poder organizar la escritura del código. Permite la capacidad de llamar bloques de script varias veces, reduciendo la cantidad de código escrito para llevar a cabo la tarea final. Esto sin mencionar la mejora de la lectura del código, sea para entenderlo fácilmente o para poder encontrar un posible error.

Una función está compuesta por diferentes elementos: un nombre, un tipo de ámbito(opcional), uno o varios argumentos(opcional) y un bloque de instrucciones.

```
Function [<ámbito> :] <nombre de función> (<argumento>)
{
    param (<lista de parámetros>)
    # bloque de instrucciones
}
```

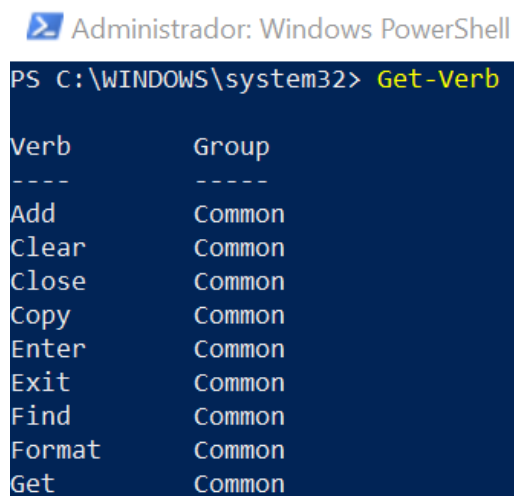
Ilustración 72. Sintaxis de una función básica

La nomenclatura de una función es importante, se deben de usar verbos aprobados seguidos de un guion y un nombre en singular, también es recomendable agregar un prefijo al nombre. Por ejemplo:

<VerboAprobado>-<Prefijo><NombreSingular>.

El prefijo es usado para evitar conflictos en los nombres.

En PowerShell hay una lista de verbos que se pueden obtener ejecutando el comando “Get-Verb”.



```

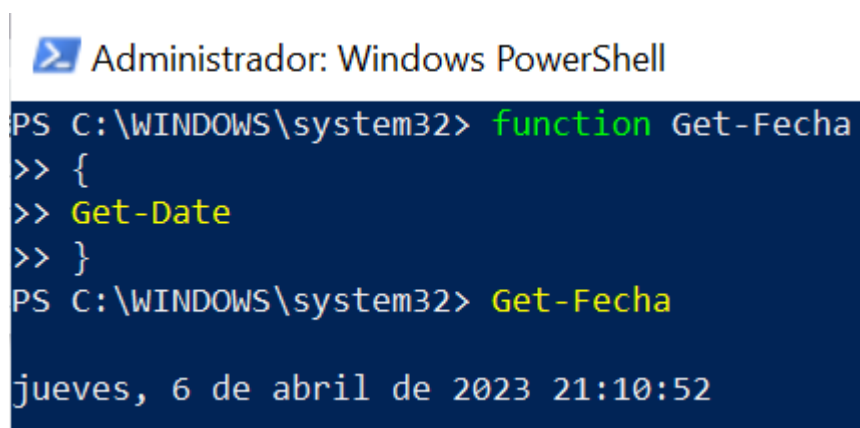
Administrador: Windows PowerShell
PS C:\WINDOWS\system32> Get-Verb

Verb      Group
----      -
Add       Common
Clear     Common
Close     Common
Copy      Common
Enter     Common
Exit      Common
Find      Common
Format    Common
Get       Common
  
```

Ilustración 73. Lista de verbos aprobados, usando el cmdlet Get-Verb

Es importante elegir un verbo aprobado en PowerShell al agregar funciones a un módulo. Si elige un verbo no aprobado, el módulo generará un mensaje de advertencia en el momento de la carga. Ese mensaje de advertencia hará que las funciones no parezcan profesionales. Los verbos no aprobados también limitan la detectabilidad de las funciones.

Para realizar una función simple se declara con la palabra clave de la función, seguida del nombre de dicha función y, a continuación, una llave de apertura y otra de cierre. El código que ejecutará la función se encuentra dentro de esas llaves.



```

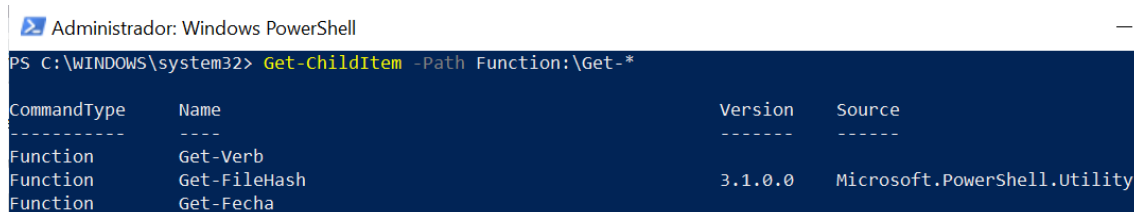
Administrador: Windows PowerShell
PS C:\WINDOWS\system32> function Get-Fecha
>> {
>> Get-Date
>> }
PS C:\WINDOWS\system32> Get-Fecha

jueves, 6 de abril de 2023 21:10:52
  
```

Ilustración 74. Ejemplo de una función simple que nos devuelve la hora

Una vez cargadas en la memoria, se pueden ver las funciones con el comando:

```
PS > Get-ChildItem -Path Function:\*-*
```



```

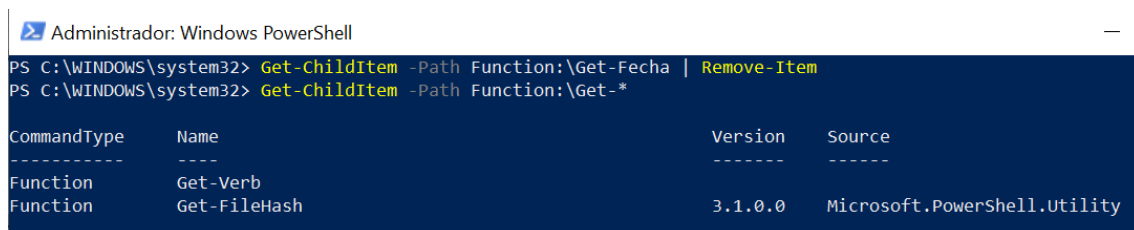
Administrador: Windows PowerShell
PS C:\WINDOWS\system32> Get-ChildItem -Path Function:\Get-*

CommandType      Name                                Version      Source
-----
Function          Get-Verb
Function          Get-FileHash                        3.1.0.0      Microsoft.PowerShell.Utility
Function          Get-Fecha
  
```

Ilustración 75. Ejecución del comando para encontrar la función creada en el ejemplo anterior

En caso de que las queramos eliminar, podemos cerrar PowerShell o ejecutar el comando:

```
PS > Get-ChildItem -Path Function:\NombreFunc | Remove-Item
```



```

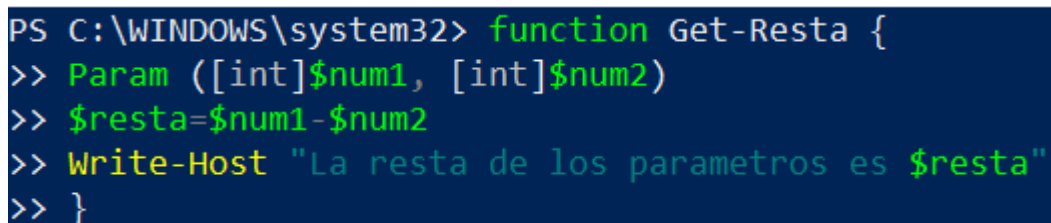
Administrador: Windows PowerShell
PS C:\WINDOWS\system32> Get-ChildItem -Path Function:\Get-Fecha | Remove-Item
PS C:\WINDOWS\system32> Get-ChildItem -Path Function:\Get-*

CommandType      Name                                Version      Source
-----
Function          Get-Verb
Function          Get-FileHash                        3.1.0.0      Microsoft.PowerShell.Utility
  
```

Ilustración 76. Ejecución del comando de borrado

Es mejor usar parámetros y variables que asignar valores de forma estática. Los parámetros de función de PowerShell hacen que las funciones sean más potentes al aprovechar las características y los argumentos para limitar a los usuarios el ingreso de valores particulares. En cuanto a la asignación de nombres a los parámetros, utilice el mismo nombre que los cmdlets predeterminados. Se debe poder ejecutar el código siguiendo una entrada válida para permitir que éste cuente con una posible ruta de acceso, es decir, es necesario validar la entrada bien al principio.

Es importante cerciorarse de las variables que utilizan los parámetros especificando el tipo de datos entrantes.



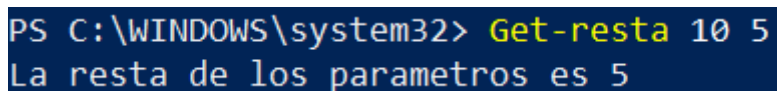
```

PS C:\WINDOWS\system32> function Get-Resta {
>> Param ([int]$num1, [int]$num2)
>> $resta=$num1-$num2
>> Write-Host "La resta de los parametros es $resta"
>> }
  
```

Ilustración 77. Ejemplo de función básica con parámetros especificando el tipo de datos

Una función puede tener cualquier número de parámetros, hay dos tipos de parámetros en PowerShell, los parámetros posicionales y los parámetros nombrados.

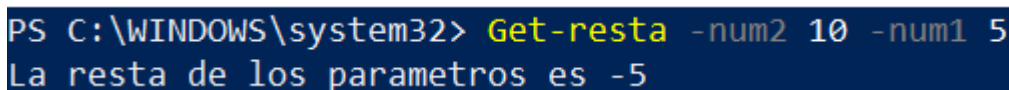
Los parámetros posicionales son aquellos que se definen en el orden en que se especifican en la línea de comando. Por ejemplo, si se tiene una función que toma dos parámetros posicionales, el primer parámetro será el primero que se especifique en la línea de comando y el segundo será el segundo. El uso de los parámetros posicionales puede ser útil cuando se quiere ejecutar una función de forma rápida sin necesidad de especificar nombres de parámetros.



```
PS C:\WINDOWS\system32> Get-resta 10 5
La resta de los parametros es 5
```

Ilustración 78. Ejecución de una función básica con parámetros posicionales

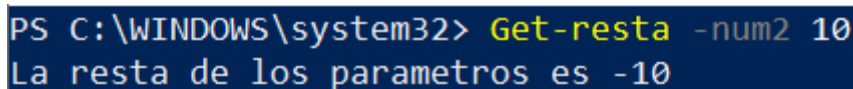
Los parámetros nombrados, por otro lado, se especifican por su nombre y no por su posición en la línea de comando. El uso de los parámetros nombrados permite una mayor claridad en la especificación de los argumentos de una función, ya que no dependen del orden en que se escriban en la línea de comando. Para especificar un parámetro nombrado, se utiliza el formato "-nombreparametro valor"



```
PS C:\WINDOWS\system32> Get-resta -num2 10 -num1 5
La resta de los parametros es -5
```

Ilustración 79. Ejecución de una función básica con parámetros nombrados

Además de los parámetros posicionales y nombrados, también hay parámetros opcionales y requeridos. Los parámetros opcionales no son necesarios para que la función se ejecute correctamente y tienen valores predeterminados si no se especifican.



```
PS C:\WINDOWS\system32> Get-resta -num2 10
La resta de los parametros es -10
```

Ilustración 80. Ejecución de una función básica con parámetros opcionales

Por otro lado, los parámetros requeridos son necesarios para que la función se ejecute correctamente y deben ser especificados en la línea de comando.

```
PS C:\WINDOWS\system32> function Get-Resta {
>> Param ([Parameter(Mandatory)][int]$num1, [int]$num2)
>> $resta=$num1-$num2
>> Write-Host "La resta de los parametros es $resta"
>> }
PS C:\WINDOWS\system32> Get-Resta -num2 10

cmdlet Get-Resta en la posición 1 de la canalización de comandos
Proporcione valores para los parámetros siguientes:
num1: 5
La resta de los parametros es -5
```

Ilustración 81. Ejemplo de una función básica con un parámetro requerido


Convertir una función de PowerShell en una avanzada es realmente sencillo. Una de las diferencias entre una función y una función avanzada es que las avanzadas tienen una serie de parámetros comunes que se agregan a la función automáticamente. En estos parámetros comunes se incluyen parámetros como `Verbose` y `Debug`.

Se puede agregar `CmdletBinding` para convertir la función en una avanzada. El hecho de agregar `CmdletBinding` agrega también los parámetros comunes automáticamente. `CmdletBinding` requiere un bloque `param`, pero el bloque `param` puede estar vacío.

```
PS C:\WINDOWS\system32> function Get-Resta {
>> [CmdletBinding()]
>> Param ([int]$num1, [int]$num2)
>> $resta=$num1-$num2
>> Write-Host "La resta de los parametros es $resta"
>> }
```

Ilustración 82. Ejemplo de función avanzada


Al explorar en profundidad los parámetros con `Get-Command`, se muestran los nombres de los parámetros reales, incluidos los comunes.

 Administrador: Windows PowerShell

```
PS C:\WINDOWS\system32> (Get-Command -Name Get-Resta).Parameters.Keys
num1
num2
Verbose
Debug
ErrorAction
WarningAction
InformationAction
ErrorVariable
WarningVariable
InformationVariable
OutVariable
OutBuffer
PipelineVariable
```

Ilustración 83. Uso del comando `Get-Command` para mostrar todos los parámetros de la función

Aunque los comentarios son muy útiles, en términos de programación, los usuarios finales no los verán nunca, a no ser que examinen el propio código.

 Administrador: Windows PowerShell

```
PS C:\WINDOWS\system32> function Get-Resta {
>> [CmdletBinding()]
>> Param ([int]$num1, [int]$num2)
>> $resta=$num1-$num2 #Operacion que realiza la resta
>> Write-Host "La resta de los parametros es $resta"
>> }
```

Ilustración 84. Ejemplo de una función con un comentario

Si se requiere que la función exponga cierto tipo de información, se deberá usar `Write-Verbose` en lugar de un comentario insertado.

Esto provocará que cuando se llame a la función colocando el parámetro **Verbose**, mostrará el resultado detallado.

Administrador: Windows PowerShell

```
PS C:\WINDOWS\system32> function Get-Resta {  
>> [CmdletBinding()]  
>> Param ([int]$num1, [int]$num2)  
>> $resta=$num1-$num2  
>> Write-Verbose -Message "Operacion que va a relizar una resta de $num1 y $num2"  
>> Write-Host "La resta de los parametros es $resta"  
>> }  
PS C:\WINDOWS\system32> Get-Resta 10 5 -Verbose  
DETALLADO: Operacion que va a relizar una resta de 10 y 5  
La resta de los parametros es 5
```

Ilustración 85. Ejemplo de una función con Write-Verbose

Como en muchos otros lenguajes, las funciones pueden generar errores si no se manejan adecuadamente. Algunos de los errores comunes que se pueden encontrar al trabajar con funciones en PowerShell son:

- **Error de sintaxis:** Este error se produce cuando hay un error de sintaxis en la función, como un error de gramática o una palabra clave no reconocida. Para solucionarlo, es necesario corregir el código para que se ajuste a la sintaxis correcta.
- **Error de tiempo de ejecución:** Este error se produce durante la ejecución de la función, cuando hay un problema en la lógica de la función, como una referencia a una variable no definida o un error en una operación de cálculo. Para solucionarlo, es necesario identificar y corregir el problema en el código.
- **Error de validación de parámetros:** Este error se produce cuando se proporciona un valor de parámetro que no cumple con los requisitos definidos en el parámetro avanzado de la función. Para solucionarlo, es necesario proporcionar un valor de parámetro válido o cambiar la definición del parámetro para permitir valores diferentes.
- **Error de excepción:** Este error se produce cuando se produce una excepción no controlada en la función, como una llamada a un método que no existe o una operación de archivo que no tiene permisos suficientes. Para solucionarlo, es necesario identificar y corregir el problema en el código, o agregar una captura de excepciones para manejar el error adecuadamente.

El tratamiento de errores en PowerShell es fundamental para crear scripts y funciones robustas y confiables. PowerShell proporciona varias formas de manejar errores:

- El uso del `try/catch` para capturar excepciones y manejar errores en tiempo de ejecución.
- El uso del cmdlet “Trap” se utiliza para capturar errores de nivel de Shell.
- El uso de `if/else`, `switch` y `foreach` para manejar errores de flujo.

Módulos

Los módulos han aparecido con la versión 2.0 de PowerShell, un módulo es una unidad independiente y reutilizable que permite particionar, organizar y abstraer el código de PowerShell. Un módulo puede contener uno o varios miembros de módulo, que son comandos (como cmdlets y funciones) y elementos (como variables y alias). Los nombres de estos miembros se pueden mantener en privado en el módulo o se pueden exportar a la sesión en la que se importa el módulo.

A partir de PowerShell 3.0, PowerShell importa los módulos automáticamente la primera vez que se ejecuta cualquier comando en un módulo instalado. Ahora se pueden utilizar los comandos de un módulo sin realizar ninguna instalación o configuración de perfil, por lo que no es necesario administrar los módulos después de instalarlos en el equipo.

Solo se importan automáticamente los módulos almacenados en la ubicación especificada por la variable de entorno *PSModulePath*. Los módulos de otras ubicaciones deben importarse mediante la ejecución del `Import-Module` cmdlet.

Si queremos consultar los módulos de PowerShell cargados en nuestra sesión o queremos listar los disponibles dentro de las distintas ubicaciones de nuestro path de módulos deberemos valernos del cmdlet `Get-Module`²

```
PS C:\Users\ > Get-Module
```

ModuleType	Version	Name	ExportedCommands
Manifest	3.1.0.0	Microsoft.PowerShell.Utility	{Add-Member, Add-Type, Clear-Variable, Compare-Object...}
Script	2.0.0	PSReadline	{Get-PSReadLineKeyHandler, Get-PSReadLineOption, Remove-PS...}

Ilustración 86. Ejecución del comando Get-Module

Si añadimos al comando el parámetro `-ListAvailable` este nos devolverá una lista con todos los módulos disponibles para su carga clasificado según la ubicación de estos dentro de nuestro path.

² Microsoft, Get-Module, <https://docs.microsoft.com/en-us/powershell/module/Microsoft.PowerShell.Core/Get-Module?view=powershell-5.1>

```

Windows PowerShell
PS C:\Users\ > Get-Module -ListAvailable

Directorio: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version      Name                               ExportedCommands
-----
Script      1.0.1      Microsoft.PowerShell.Operation.V... {Get-OperationValidation, Invoke-OperationValidation}
Binary      1.0.0.1    PackageManagement                 {Find-Package, Get-Package, Get-PackageProvider, Get-Packa...
Script      3.4.0      Pester                           {Describe, Context, It, Should...}
Script      1.0.0.1    PowerShellGet                    {Install-Module, Find-Module, Save-Module, Update-Module...}
Script      2.0.0      PSReadline                       {Get-PSReadLineKeyHandler, Set-PSReadLineKeyHandler, Remov...

Directorio: C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules

ModuleType Version      Name                               ExportedCommands
-----
Manifest    1.0.0.0     AppBackgroundTask                 {Disable-AppBackgroundTaskDiagnosticLog, Enable-AppBackgro...
Manifest    2.0.0.0     AppLocker                        {Get-AppLockerFileInformation, Get-AppLockerPolicy, New-Ap...
Manifest    1.0.0.0     AppvClient                       {Add-AppvClientConnectionGroup, Add-AppvClientPackage, Add...
Manifest    2.0.1.0     Appx                             {Add-AppxPackage, Get-AppxPackage, Get-AppxPackageManifest...
Script      1.0.0.0     AssignedAccess                   {Clear-AssignedAccess, Get-AssignedAccess, Set-AssignedAcc...
Manifest    1.0.0.0     BitLocker                       {Unlock-BitLocker, Suspend-BitLocker, Resume-BitLocker, Re...
Manifest    2.0.0.0     BitsTransfer                     {Add-BitsFile, Complete-BitsTransfer, Get-BitsTransfer, Re...
Manifest    1.0.0.0     BranchCache                     {Add-BCDataCacheExtension, Clear-BCCache, Disable-BC, Disa...

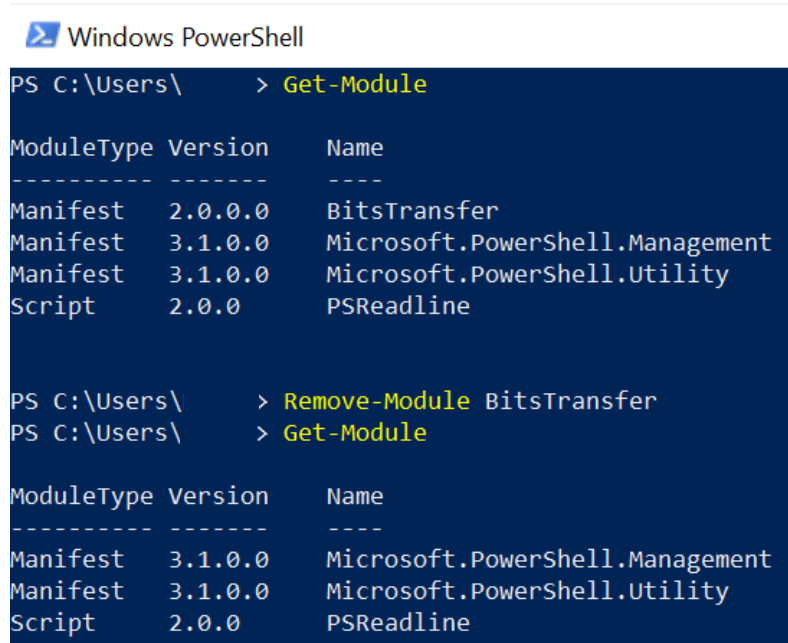
```

Ilustración 87. Ejecución del comando `Get-Module -ListAvailable`

Nos podemos encontrar cuatro tipos de módulos:

- **Módulo de manifiesto:** Un módulo de tipo manifiesto está compuesto de un archivo (**.psd1**) que contiene diversa información relativa a un módulo como su modo de ejecución, el autor, el número de versión...
- **Módulo de script:** Un módulo de script es un archivo (**.psm1**) que contiene cualquier código válido de Windows PowerShell, de tipo script. Los desarrolladores y administradores de scripts pueden usar este tipo de módulo para crear módulos cuyos miembros incluyen funciones, variables, etc. Un módulo de script puede ser tan simple como una sola función.
- **Módulo binario:** Un módulo de tipo binario es un módulo que contiene código compilado (archivo **.dll**). Los desarrolladores de cmdlets pueden usar este tipo de módulo para compartir cmdlets, proveedores y mucho más. Permite crear cmdlets que son más rápidos o usan características (como multithreading) que no son tan fáciles de codificar en scripts
- **Módulo Dinámico:** Un *módulo dinámico* es aquel que no se encuentra almacenado en una carpeta, sino creado en tiempo de ejecución y que se mantiene hasta que se cierra la sesión de PowerShell.

Para quitar un módulo ya cargado en nuestra sesión podemos usar el cmdlet `Remove-Module`. Esta opción puede sernos muy útil si queremos cargar, por ejemplo, una nueva versión de un módulo que ya teníamos cargado.



```

Windows PowerShell
PS C:\Users\ > Get-Module

ModuleType Version      Name
-----
Manifest 2.0.0.0 BitsTransfer
Manifest 3.1.0.0 Microsoft.PowerShell.Management
Manifest 3.1.0.0 Microsoft.PowerShell.Utility
Script 2.0.0 PSReadline

PS C:\Users\ > Remove-Module BitsTransfer
PS C:\Users\ > Get-Module

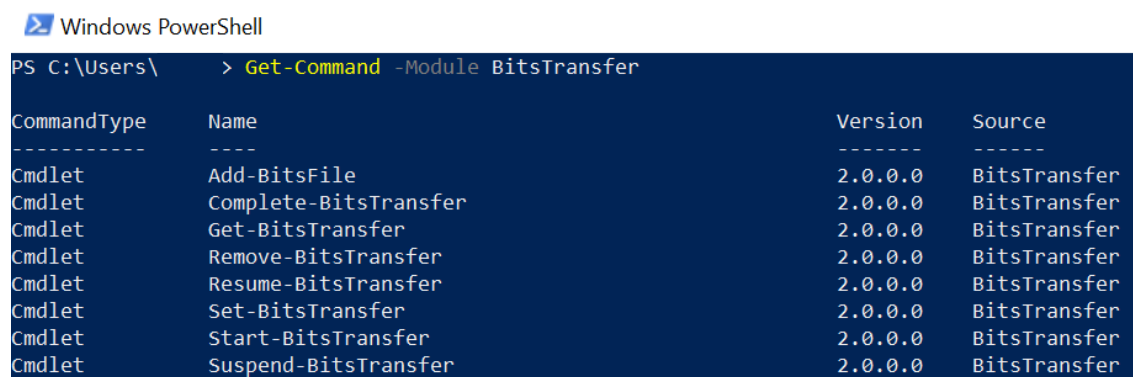
ModuleType Version      Name
-----
Manifest 3.1.0.0 Microsoft.PowerShell.Management
Manifest 3.1.0.0 Microsoft.PowerShell.Utility
Script 2.0.0 PSReadline

```

Ilustración 88. Ejemplo de borrado de módulo.

Los módulos se pueden anidar; es decir, un módulo puede importar otro módulo. Un módulo que tiene asociados módulos anidados es un módulo raíz.

Se puede usar el cmdlet `Get-Command` para buscar los comandos de un módulo.



```

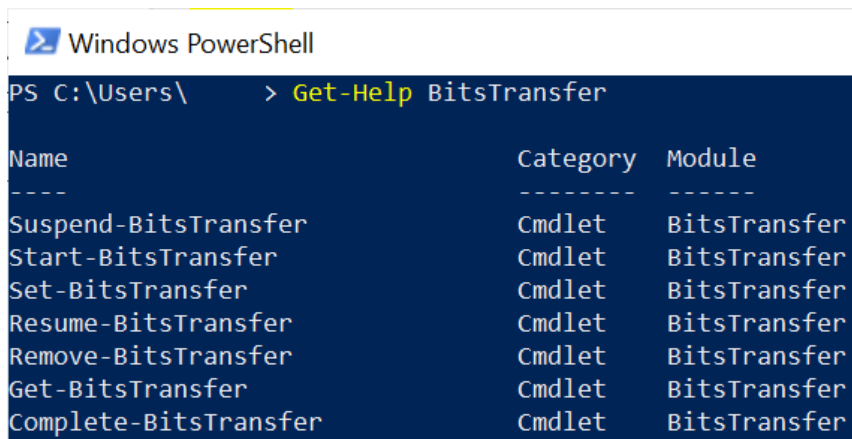
Windows PowerShell
PS C:\Users\ > Get-Command -Module BitsTransfer

CommandType Name Version Source
-----
Cmdlet Add-BitsFile 2.0.0.0 BitsTransfer
Cmdlet Complete-BitsTransfer 2.0.0.0 BitsTransfer
Cmdlet Get-BitsTransfer 2.0.0.0 BitsTransfer
Cmdlet Remove-BitsTransfer 2.0.0.0 BitsTransfer
Cmdlet Resume-BitsTransfer 2.0.0.0 BitsTransfer
Cmdlet Set-BitsTransfer 2.0.0.0 BitsTransfer
Cmdlet Start-BitsTransfer 2.0.0.0 BitsTransfer
Cmdlet Suspend-BitsTransfer 2.0.0.0 BitsTransfer

```

Ilustración 89. Ejecución del comando `Get-Command`

Si el módulo contiene archivos de Ayuda para los comandos que exporta, el `Get-Help` mostrará los temas de Ayuda. Se usa el mismo formato de comando que se usaría para obtener ayuda para cualquier comando de PowerShell.



```

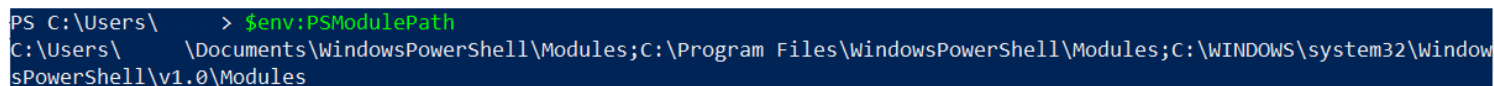
Windows PowerShell
PS C:\Users\ > Get-Help BitsTransfer

Name                Category  Module
----                -
Suspend-BitsTransfer Cmdlet    BitsTransfer
Start-BitsTransfer  Cmdlet    BitsTransfer
Set-BitsTransfer     Cmdlet    BitsTransfer
Resume-BitsTransfer  Cmdlet    BitsTransfer
Remove-BitsTransfer  Cmdlet    BitsTransfer
Get-BitsTransfer     Cmdlet    BitsTransfer
Complete-BitsTransfer Cmdlet    BitsTransfer
  
```

Ilustración 90. Ejecución del comando `Get-Help`

A partir de PowerShell 3.0, puede descargar archivos de Ayuda para un módulo y descargar actualizaciones en los archivos de Ayuda para que nunca estén obsoletos.

Cuando un módulo no está instalado en las ubicaciones especificadas, es necesario importarlo.

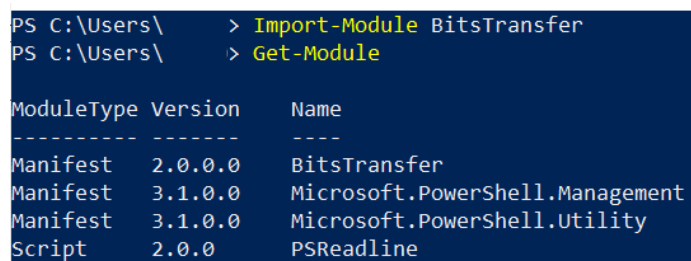


```

PS C:\Users\ > $env:PSModulePath
C:\Users\ \Documents\WindowsPowerShell\Modules;C:\Program Files\WindowsPowerShell\Modules;C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
  
```

Ilustración 91. Ejecución del comando `PSModulePath`

Para importar módulos en una ubicación `PSModulePath` en la sesión actual, se utiliza el `Import-Module`.



```

PS C:\Users\ > Import-Module BitsTransfer
PS C:\Users\ > Get-Module

ModuleType Version  Name
-----
Manifest  2.0.0.0  BitsTransfer
Manifest  3.1.0.0  Microsoft.PowerShell.Management
Manifest  3.1.0.0  Microsoft.PowerShell.Utility
Script    2.0.0    PSReadline
  
```

Ilustración 92. Ejemplo de cómo importar un módulo

Cuando más de un comando en la sesión tiene el mismo nombre, se producen conflictos de nombre. La importación de un módulo provoca un conflicto de nombres si los comandos del módulo tienen los mismos nombres que los comandos o elementos de la sesión.

Los conflictos de nombres pueden ocultar o reemplazar los comandos.

Se oculta un comando si no es el comando que se ejecuta cuando se escribe el nombre de comando, pero puede ejecutarlo mediante otro método, por ejemplo, calificando el nombre de comando con el nombre del módulo o complemento en el que se originó.

Para detectar conflictos de nombres, se usa el parámetro **All** del `Get-Command cmdlet`. A partir de PowerShell 3.0, `Get-Command` solo obtiene los comandos que se ejecutan al escribir el nombre del comando. El parámetro **All** obtiene todos los comandos con el nombre específico de la sesión.

Para evitar conflictos de nombres, se pueden usar los parámetros **NoClobber** o **Prefix** del `Import-Module cmdlet`. El parámetro **Prefix** agrega un prefijo a los nombres de los comandos importados para que sean únicos en la sesión. El parámetro **NoClobber** no importa ningún comando que oculte o reemplace los comandos existentes en la sesión.

También se pueden usar los parámetros **Alias**, **Cmdlet**, **Function** y **Variable** de `Import-Module` para seleccionar solo los comandos que se desea importar y se puede excluir comandos que causen conflictos de nombres en la sesión.

Incluso si un comando está oculto, se puede ejecutar calificando el nombre del comando con el nombre del módulo o complemento en el que se originó.

Las reglas de precedencia de comandos de PowerShell determinan qué comando se ejecuta cuando la sesión incluye comandos con el mismo nombre.

Por ejemplo, cuando una sesión incluye una función y un cmdlet con el mismo nombre, PowerShell ejecuta la función de forma predeterminada. Cuando la sesión incluye comandos del mismo tipo con el mismo nombre, como dos cmdlets con el mismo nombre, de forma predeterminada ejecuta el comando agregado más recientemente.

Los comandos que exporta un módulo deben seguir las reglas de nomenclatura de comandos de PowerShell. Si el módulo que importa exporta cmdlets o funciones que tienen verbos no aprobados en sus nombres, el `Import-Module` cmdlet muestra un mensaje de advertencia.

El módulo completo se importa de todos modos, incluidos los comandos que no cumplen las especificaciones. Aunque el mensaje se muestra a los usuarios del módulo, el autor del módulo debe corregir el problema de nomenclatura.

Para suprimir el mensaje de advertencia, use el parámetro **DisableNameChecking** del `Import-Module` cmdlet.

Scripts

Un script de PowerShell es un archivo de texto que contiene uno o varios comandos de PowerShell. La extensión empleada para los scripts de PowerShell es **.ps1**

Su ejecución es muy parecida a la de un cmdlet. Se necesita escribir la ruta de acceso del script, junto a los parámetros exigidos por el mismo. Se pueden ejecutar en el equipo local o en sesiones remotas en otros equipos.

La ventaja de los scripts es que permite guardar una serie de comandos para su posterior uso, así como facilitar el que llegue a otros usuarios.

El principal objetivo de un script es el de automatizar una tarea rutinaria, de forma que cuando se necesite realizar esa tarea se ejecute un script, en vez de cada paso de forma manual. Ya que los scripts pueden estar orientados a cualquier ámbito, su complejidad puede cambiar enormemente, dependiendo de la tarea que desempeñen.

Los scripts tienen características adicionales, como comentarios especiales, uso de parámetros, compatibilidad con secciones de datos y firma digital para la seguridad. También se pueden escribir temas de Ayuda para scripts y para cualquier función del script.

PowerShell es un lenguaje de script **interpretado**, lo que significa que no se compila en código de máquina antes de la ejecución. En su lugar, el tiempo de ejecución de PowerShell **interpreta y ejecuta** los comandos y scripts en tiempo real.

Cuando se ejecuta un comando o script de PowerShell, el intérprete **analiza y evalúa** el código, línea por línea, mediante una serie de reglas y procesos que siguen una sintaxis y semántica específicas.

En general, el proceso de análisis y evaluación de PowerShell se divide en tres etapas:

1. **Análisis léxico:** El intérprete divide el código en unidades léxicas, o **tokens**, que representan *keywords*, identificadores, operadores y otros elementos sintácticos del lenguaje.
2. **Análisis sintáctico:** El intérprete utiliza los tokens extraídos en el análisis léxico para construir una estructura sintáctica del script, verificando que la secuencia de tokens sea válida según la gramática de PowerShell.

3. **Evaluación semántica:** El intérprete ejecuta el script, línea por línea, evaluando cada expresión y comando según las reglas semánticas.

Algunas de las reglas seguidas por el intérprete de PowerShell durante el proceso de análisis y evaluación incluye:

- Sintaxis de cmdlets.
- Tipado dinámico. PowerShell es un lenguaje de tipado dinámico.
- Alias. PowerShell admite el uso de alias para comandos. Es por ello por lo que debe revisar si el token se trata de un cmdlet o de un alias del mismo.
- Expansión de variables y subexpresiones. PowerShell expande automáticamente las variables y subexpresiones incluidas en una cadena de comando, sustituyéndolas por su valor real antes de la evaluación.
- Control de flujo. El intérprete debe analizar si el token se trata de una *keyword* perteneciente a una sentencia de control de flujo.

Si existiese un conflicto de nombres, el intérprete de PowerShell mantendría el siguiente orden:

1. Alias
2. Función
3. Cmdlet
4. Archivos ejecutables externos

Es muy interesante conocer este orden a la hora de realizar scripts.

Un script de PowerShell puede llegar a tener la misma estructura que un script de Shell. Se pueden desgranar los siguientes componentes:

- Lectura de parámetros (si hubiesen).
- Creación de variables.
- Controles de flujo.

- Código.
- Salida (si hubiese).
- Gestión de errores.
- Escritura de ayuda.

Los únicos puntos no tratados hasta ahora son: La salida, la escritura de ayuda y la gestión de errores.

Valores de salida. De forma predeterminada, los scripts no devuelven un estado de salida cuando finaliza el script. Debe usar la instrucción '**exit**' para devolver un código de salida de un script. Por defecto, la instrucción '**exit**' devuelve 0. Un código de salida distinto de cero suele indicar un error.

Dentro de los valores que se pueden devolver, hay discrepancias dentro de los distintos Sistemas Operativos que soporta PowerShell:

- **Windows.** Cualquier número entre [int]::MinValue – [int]::MaxValue
- **Unix.** Cualquier número positivo entre [byte]::MinValue – [byte]::MaxValue (0-255).

Escritura de ayuda. Define lo que aparecerá si se llama al cmdlet sobre la función escrita.

Hay dos métodos para realizarlo:

- **Comment-Based** Ayuda para scripts. PowerShell tiene una sintaxis propia para escribir mensaje de ayuda desde los comentarios de una función.
- **XML-Based** Ayuda para scripts. Empleando XML, se puede crear el tipo de ayuda usado normalmente en los cmdlets. Esta opción es la necesaria si el formato de ayuda va a estar disponible en más de un idioma.

Gestión de errores

Antes de comenzar, se debe explicar una terminología básica:

- **Excepción.** Evento que se crea cuando el control de errores normal no puede resolver el problema.
- **Throw y Catch.** Para controlar una excepción iniciada, se debe capturar. Si se inicia una excepción y no se captura, el script deja de ejecutarse.
- **Pila de llamadas.** La pila de llamadas es la lista de funciones que se han llamado entre sí. Cuando se llama a una función, se agrega a la pila o a la parte superior de la lista. Cuando la función termina o devuelve un resultado, se quita de la pila. Cuando se inicia una excepción, se comprueba esa pila de llamadas para que un controlador de excepciones la capture.
- **Errores de terminación y no terminación.** Por lo general, una excepción es un error de terminación. Una excepción iniciada se captura o termina la ejecución actual. De forma predeterminada, Write-Error genera un error de no terminación, agregando un error al flujo de salida sin iniciar una excepción. Esto quiere decir que Write-Error no desencadena una operación catch.

Try/Catch

El bloque `Try/Catch` permite probar una sección de código y, si se produce un error, capturarlo. Un pequeño ejemplo:

```

PS D:\tmp> cat .\try-catch.ps1
try
{
    Write-Output "Todo bien"
}
catch
{
    Write-Output "Algo lanzo una excepcion"
    Write-Output $_
}

try
{
    Start-Something -ErrorAction Stop
}
catch
{
    Write-Output "Algo genero una excepcion o uso Write-Error"
    Write-Output $_
}

PS D:\tmp> .\try-catch.ps1
Todo bien
Algo genero una excepcion o uso Write-Error
Start-Something : The term 'Start-Something' 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 and try
again.
At D:\tmp\try-catch.ps1:13 char:2
+ Start-Something -ErrorAction Stop
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Start-Something:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS D:\tmp>

```

Ilustración 93. Ejemplo de bloque try/catch

Con la variable automática `$_`, se imprime el error capturado.

Try/Finally

Hay ocasiones en las que no es necesario controlar un error, pero sí ejecutar código tanto si se produce una excepción como si no. Un ejemplo sería el cerrar la conexión con una base de datos abierta en el bloque try.

```

try-finally.ps1 (D:\tmp) - VIM
$comando = [System.Data.SqlClient.SqlCommand]::New(queryString, connection)
try
{
    $comando.Connection.Open()
    $comando.ExecuteNonQuery()
}
finally
{
    Write-Error "Ha habido un problema con la ejecución de la query. Cerrando la conexión"
    $comando.Connection.Close()
}

```

Ilustración 94. Ejemplo bloque try/finally

Como no se captura la excepción, se sigue propagando a la pila de llamadas.

Es totalmente válido emplear un bloque conjunto de **Try/Catch/Finally**

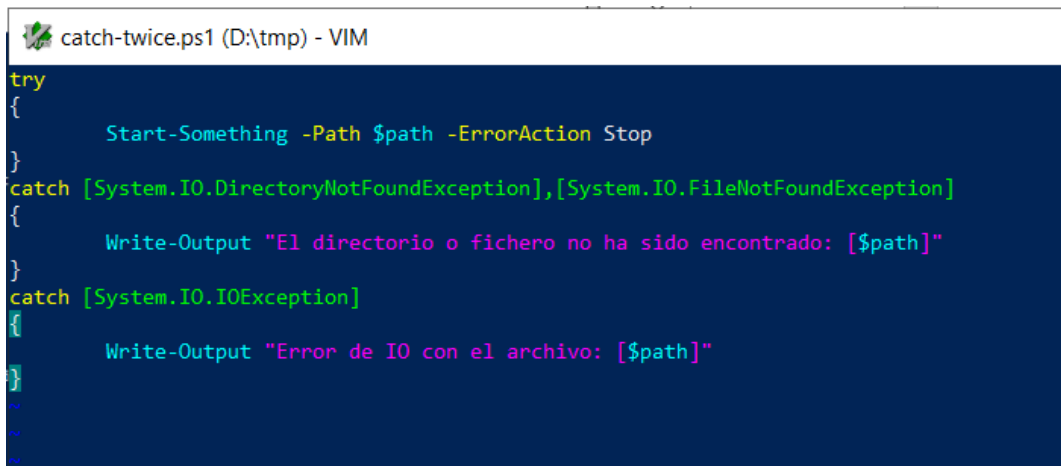
Variable Automática `$PSItem`

Para mostrar información acerca del error capturado, existe la variable automática **`$PSItem`** (o `$_`) de tipo `ErrorRecord` que contiene los detalles sobre la excepción.

Entre sus métodos, encontramos:

- `$PSItem.ToString()` => Produce la misma salida que llamando a la variable dentro de una cadena. Muestra una versión más limpia del mensaje.
- `$PSItem.InvocationInfo` => Esta propiedad contiene información adicional recopilada por PowerShell sobre la función o el script en el que se inició la excepción.
- `$PSItem.ScriptStackTrace` => Muestra el orden de las llamadas de función que le han llevado al código donde se generó la excepción. Si se invocan varios scripts, se realiza un seguimiento de las llamadas.
- `$PSItem.Exception` => La excepción real que se inició.
 - `$PSItem.Exception.Message` => Es el mensaje general que describe la excepción.
 - `$PSItem.Exception.InnerException` => Las excepciones pueden contener excepciones internas. Esto suele suceder cuando el código al que se llama captura una excepción e inicia una diferente.
 - `$PSItem.Exception.StackTrace` => Es el seguimiento de la pila de llamadas que da lugar a la excepción.

En PowerShell es posible capturar varios tipos de excepción a la vez. Simplemente hay que especificar el tipo de error esperado en la sentencia `catch`.

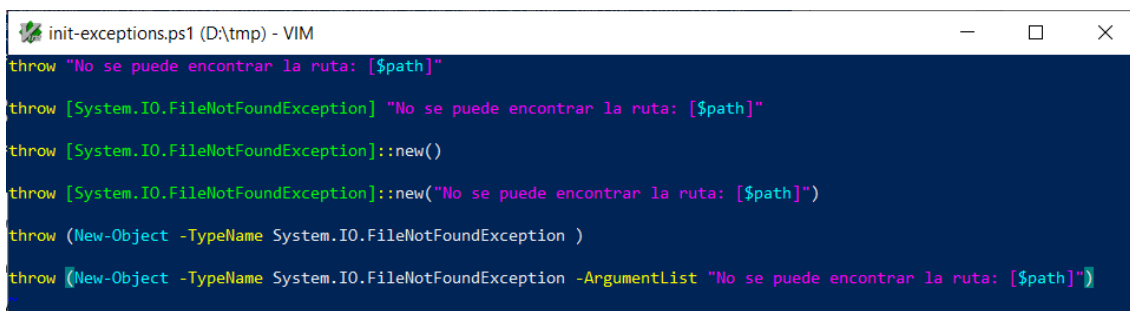


```
catch-twice.ps1 (D:\tmp) - VIM
try
{
    Start-Something -Path $path -ErrorAction Stop
}
catch [System.IO.DirectoryNotFoundException],[System.IO.FileNotFoundException]
{
    Write-Output "El directorio o fichero no ha sido encontrado: [$path]"
}
catch [System.IO.IOException]
{
    Write-Output "Error de IO con el archivo: [$path]"
}
```

Ilustración 95. Recogida de varios tipos de excepción distintos.

Se puede agregar un bloque catch genérico para que capture aquellas excepciones que no coincidiesen con las recogidas por los bloques previos.

Puede resultar de gran valor el inicializar las excepciones con un tipo determinado. En PowerShell se puede conseguir de diversas maneras. Se ejemplifican algunas:



```
init-exceptions.ps1 (D:\tmp) - VIM
throw "No se puede encontrar la ruta: [$path]"
throw [System.IO.FileNotFoundException] "No se puede encontrar la ruta: [$path]"
throw [System.IO.FileNotFoundException]::new()
throw [System.IO.FileNotFoundException]::new("No se puede encontrar la ruta: [$path]")
throw (New-Object -TypeName System.IO.FileNotFoundException )
throw (New-Object -TypeName System.IO.FileNotFoundException -ArgumentList "No se puede encontrar la ruta: [$path]")
```

Ilustración 96. Diversas formas de lanzar una excepción.

Existe un elemento que hay que destacar, a mayores de los ya mencionados. Es la cláusula `trap`. `Trap` permite ejecutar su código cuando se produce una excepción, y luego continuar el código normal. Si se producen varias excepciones, se llama al código una y otra vez.

Un pequeño ejemplo sería:

```

PS D:\tmp> cat .\trap.ps1
trap
{
    Write-Output $PSItem.ToString()
}
throw [System.Exception]::new('primero')
throw [System.Exception]::new('segundo')
throw [System.Exception]::new('tercero')
PS D:\tmp> .\trap.ps1
primero
primero
At D:\tmp\trap.ps1:5 char:1
+ throw [System.Exception]::new('primero')
+ ~~~~~
+ CategoryInfo          : OperationStopped: (:) [], Exception
+ FullyQualifiedErrorId : primero

segundo
segundo
At D:\tmp\trap.ps1:6 char:1
+ throw [System.Exception]::new('segundo')
+ ~~~~~
+ CategoryInfo          : OperationStopped: (:) [], Exception
+ FullyQualifiedErrorId : segundo

tercero
tercero
At D:\tmp\trap.ps1:7 char:1
+ throw [System.Exception]::new('tercero')
+ ~~~~~
+ CategoryInfo          : OperationStopped: (:) [], Exception
+ FullyQualifiedErrorId : tercero

PS D:\tmp>

```

Ilustración 97. Ejemplo del uso de trap

Ejemplo práctico de script.

Para formalizar todo lo comentado hasta ahora, se va a crear un script para el backup del Registro de Windows³.

Antes de comenzar el ejemplo, **conviene repasar los pasos explicados en el [Anexo II](#)**.

En él se detalla cómo habilitar el sistema para que pueda ejecutar scripts propios, ya que por motivos de seguridad viene deshabilitado por defecto.

³ El Registro de Windows es una base de datos jerárquica central, usada para almacenar información necesaria para configurar el sistema para uno o varios usuarios, aplicaciones y dispositivos de hardware. Microsoft, "Windows Registry for Advanced Users" <https://learn.microsoft.com/es-es/troubleshoot/windows-server/performance/windows-registry-advanced-users>

Comenzamos desarrollando la funcionalidad: Realizar el backup en la ruta especificada como parámetro.

```
# Función para realizar un backup del registro del sistema
function Backup-Registry {
    Param(
        [Parameter(Mandatory = $true)]
        [string]$rutaBackup
    )

    # Crear la ruta de destino del backup si no existe
    if (!(Test-Path -Path $rutaBackup)) {
        New-Item -ItemType Directory -Path $rutaBackup | Out-Null
    }

    # Generar un nombre único para el archivo de backup
    $nombreArchivo = "Backup-Registry_" + (Get-Date -Format "yyyy-MM-dd_HH-mm-ss") + ".reg"
    $rutaArchivo = Join-Path -Path $rutaBackup -ChildPath $nombreArchivo

    # Realizar el backup del registro del sistema y guardarlo en el
    # archivo de destino
    try {
        Write-Host "Realizando backup del registro del sistema en
$rutaArchivo..."
        reg export HKLM $rutaArchivo
        Write-Host "El backup del registro del sistema se ha
realizado con éxito."
    }
    catch {
        Write-Host "Se ha producido un error al realizar el backup
del registro del sistema: $_"
    }
}
```

Ilustración 98. Creación del backup

Una vez incluida la primera funcionalidad, se pueden agregar más:

- Creación de un LOG cada vez que se llame a la función.
- Mantenimiento de únicamente diez backups.
- Ejecución diaria, a las 2:00 am.

Las funcionalidades descritas se reflejan de la siguiente forma en el código:

```
# Escribir en el archivo de log
$logDirectory = "$env:APPDATA\RegistryBackup"
$logFile = Join-Path $logDirectory "backup-registry_log.txt"
$logEntry = "$(Get-Date) - $env:USERNAME - Backup -
$backupPath"
if (!(Test-Path $logDirectory)) {
    New-Item -ItemType Directory -Path $logDirectory | Out-Null
}
Add-Content -Path $logFile -Value $logEntry
```

Ilustración 99. Nueva funcionalidad: Escribir en archivo de log

```
# Verificar si hay más de $backupCount backups en el directorio y
eliminar los más antiguos si es necesario
$backupCount = 10
$backups = Get-ChildItem $backupDirectory -Filter *.reg | Sort-
Object LastWriteTime -Descending
if ($backups.Count -gt $backupCount) {
    $backupsToDelete = $backups[$backupCount..($backups.Count -
1)]
    $backupsToDelete | Remove-Item -Force
}
```

Ilustración 100. Nueva funcionalidad: Comprobar que solo existan 10 backups

Para poder configurar la ejecución diaria, antes se debe crear un módulo que contenga este script, para que así se pueda llamar a cada una de las funciones.

Para ello, se tienen que realizar los siguientes pasos:

1. Crear una carpeta con el nombre del módulo en la carpeta de módulos de PowerShell. Para obtener la ruta de módulos, se ejecuta el comando **\$env:PSModulePath**
2. Mover el script a la carpeta creada.
3. Crear un archivo **.psd1**, dentro de la carpeta, con el siguiente contenido:

```
@{
    ModuleVersion = '1.0.0'
    PowerShellVersion = '5.1'
    RootModule = 'Backup-Registry.ps1'
    Description = 'Módulo para realizar backups del registro del
sistema de Windows'
    Author = 'Alice'
    FunctionsToExport = @('Backup-Registry')
}
```

Ilustración 101. Contenido del archivo BackupRegistry.psd1

4. Importar el módulo. Esto cargará el comando en la sesión actual.

```
PS C:\Program Files\WindowsPowerShell\Modules\BackupRegistry> ls

Directory: C:\Program Files\WindowsPowerShell\Modules\BackupRegistry

Mode                LastWriteTime         Length Name
----                -
-a----            4/16/2023   9:21 PM           1024 .BackupRegistry.psd1.un~
-a----            4/16/2023   9:10 PM           2375 Backup-Registry.ps1
-a----            4/16/2023   9:21 PM             292 BackupRegistry.psd1
-a----            4/16/2023   9:21 PM             292 BackupRegistry.psd1~

PS C:\Program Files\WindowsPowerShell\Modules\BackupRegistry> Import-Module BackupRegistry
PS C:\Program Files\WindowsPowerShell\Modules\BackupRegistry>
```

Ilustración 102. Importación del módulo

Se puede emplear la opción **-Force** para sobrescribir la importación anterior.

Con el cmdlet `Get-Help` podemos observar que disponemos de la función que compone el módulo:

```
PS D:\> Get-Help Backup-Registry

NAME
    Backup-Registry

SYNTAX
    Backup-Registry [-rutaBackup] <string> [<CommonParameters>]

ALIASES
    None

REMARKS
    None
```

Ilustración 103. Función importada

Procedemos a comprobar si el módulo funciona como se esperaba:

```
PS D:\> Backup-Registry -rutaBackup 'D:\tmp\Backups\Registro\'
Realizando backup del registro del sistema en D:\tmp\Backups\Registro\RegistryBackup-20230416-214933.reg...
The operation completed successfully.
Backup del registro realizado correctamente
PS D:\> ls .\tmp\Backups\Registro\

Directory: D:\tmp\Backups\Registro

Mode                LastWriteTime         Length Name
----                -
-a----            4/16/2023   9:47 PM        383375246 RegistryBackup-20230416-214719.reg
-a----            4/16/2023   9:49 PM        391552198 RegistryBackup-20230416-214933.reg

PS D:\>
```

Ilustración 104. Comprobación de la función de exportado

Editando el módulo para que solo admita tres copias de seguridad y reimportándolo, comprobamos si se mantienen solo este número de backups:

```
PS C:\Program Files\WindowsPowerShell\Modules\BackupRegistry> vim .\Backup-Registry.ps1
PS C:\Program Files\WindowsPowerShell\Modules\BackupRegistry> Import-Module BackupRegistry -Force
PS C:\Program Files\WindowsPowerShell\Modules\BackupRegistry> Backup-Registry -rutaBackup 'D:\tmp\Backups\Registro\'
Realizando backup del registro del sistema en D:\tmp\Backups\Registro\RegistryBackup-20230416-220059.reg...
The operation completed successfully.
Backup del registro realizado correctamente
PS C:\Program Files\WindowsPowerShell\Modules\BackupRegistry> ls 'D:\tmp\Backups\Registro\'

Directory: D:\tmp\Backups\Registro

Mode                LastWriteTime         Length Name
----                -
-a----            4/16/2023   9:58 PM        383379670 RegistryBackup-20230416-215840.reg
-a----            4/16/2023  10:00 PM        383379670 RegistryBackup-20230416-220022.reg
-a----            4/16/2023  10:01 PM        383379670 RegistryBackup-20230416-220059.reg
```

Ilustración 105. Comprobación de la eliminación automática de backups

Como se puede observar, se ha eliminado el de las 21:47h

Ahora, debemos implementar la tercera y última funcionalidad: La automatización de la ejecución del backup.

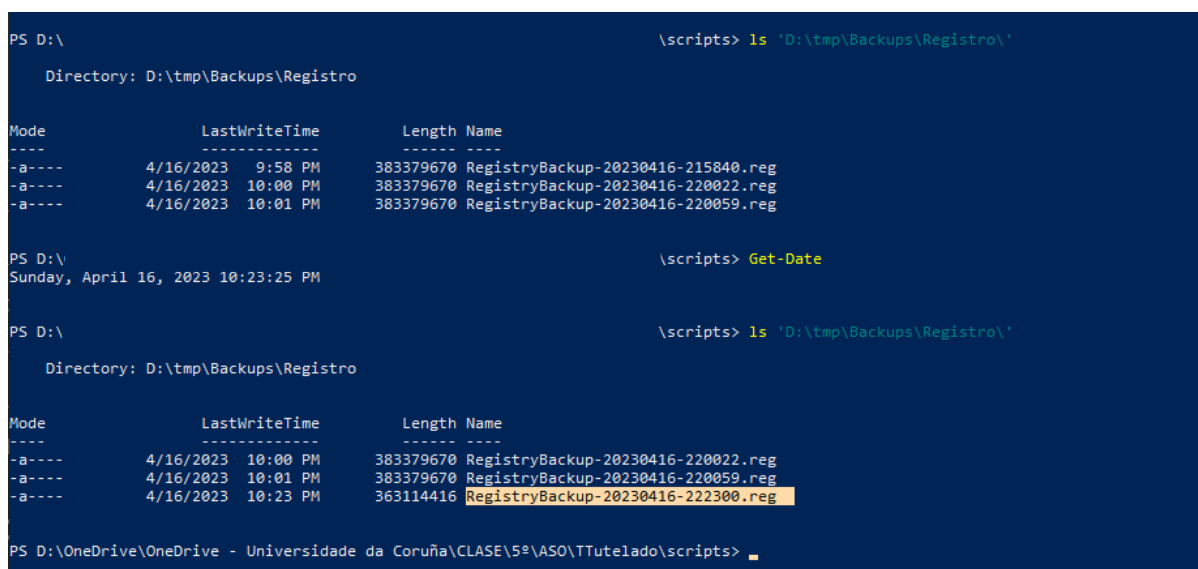
Para ello se empleará el cmdlet `Register-ScheduledTask`

```
# Configuración de la tarea
$Time = New-ScheduledTaskTrigger -At 02:00 -Daily
# Acción de la tarea
$PS = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument
"-Command `\"Import-Module BackupRegistry -Force; Backup-Registry -
rutaBackup 'D:\tmp\Backups\Registro'\`\""
# Crear la tarea programada
Register-ScheduledTask -TaskName "Ejecutar Backup del Registro del
Sistema" -Trigger $Time -Action $PS
```

Ilustración 106. Programación de la tarea

Para hacer una prueba, se configurará para que realice una única ejecución para las 22:23h del 16/04/2023.

Tal y como se esperaba, aparece un archivo con la hora y el día con el que se había configurado el job.



```
PS D:\ \scripts> ls 'D:\tmp\Backups\Registro\'

Directory: D:\tmp\Backups\Registro

Mode                LastWriteTime         Length Name
----                -
-a----          4/16/2023   9:58 PM           383379670 RegistryBackup-20230416-215840.reg
-a----          4/16/2023  10:00 PM           383379670 RegistryBackup-20230416-220022.reg
-a----          4/16/2023  10:01 PM           383379670 RegistryBackup-20230416-220059.reg

PS D:\ \scripts> Get-Date
Sunday, April 16, 2023 10:23:25 PM

PS D:\ \scripts> ls 'D:\tmp\Backups\Registro\'

Directory: D:\tmp\Backups\Registro

Mode                LastWriteTime         Length Name
----                -
-a----          4/16/2023  10:00 PM           383379670 RegistryBackup-20230416-220022.reg
-a----          4/16/2023  10:01 PM           383379670 RegistryBackup-20230416-220059.reg
-a----          4/16/2023  10:23 PM           363114416 RegistryBackup-20230416-222300.reg

PS D:\OneDrive\OneDrive - Universidade da Coruña\CLASE\5º\ASO\TTutelado\scripts>
```

Ilustración 107. Comprobación de ejecución del script de forma automática

Para listar las tareas programadas, se puede emplear el cmdlet `Get-ScheduledTask`

Si queremos eliminar una tarea programada, se puede emplear el cmdlet `Unregister-ScheduledTask`. Veamos un rápido ejemplo:

```
PS D:\> \scripts> Get-ScheduledTask
```

TaskPath	TaskName	State
\\	Adobe Acrobat Update Task	Ready
\\	AdobeAAMUpdater-1.0-DESKTOP-67...	Ready
\\	Ejecutar Backup del Registro d...	Ready
\\	GamingOSDAutoStartUp	Disabled
\\	GoogleUpdateTaskMachineCore	Ready

Ilustración 108. Lista de tareas programadas.

Se quiere eliminar la tarea 'Ejecutar Backup del Registro del Sistema'. Ejecutamos:

```
PS > Unregister-ScheduledTask 'Ejecutar Backup del Registro del Sistema'
```

```
PS D:\> \scripts> Unregister-ScheduledTask 'Ejecutar Backup del Registro del Sistema'
```

Confirm
Are you sure you want to perform this action?
Performing operation 'Delete' on Target '\Ejecutar Backup del Registro del Sistema'.
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): Y
PS D:\> \scripts>

Ilustración 109. Eliminar tarea programada

Una vez ejecutado y confirmado, se comprueba si efectivamente se ha eliminado la tarea:

```
PS D:\> \scripts> Get-ScheduledTask
```

TaskPath	TaskName	State
\\	Adobe Acrobat Update Task	Ready
\\	AdobeAAMUpdater-1.0-DESKTOP-67...	Ready
\\	GamingOSDAutoStartUp	Disabled
\\	GoogleUpdateTaskMachineCore	Ready
\\	GoogleUpdateTaskMachineUA	Ready
\\	Intel PTT EK Recertification	Ready
\\	MicrosoftEdgeUpdateTaskMachine...	Ready
\\	MicrosoftEdgeUpdateTaskMachineUA	Ready
\\	MonitorMysticLight	Ready
\\	MSISW_Host	Running

Ilustración 110. Volver a listar las tareas programadas

Como se puede observar, la tarea se ha eliminado correctamente.

En resumen, en este ejemplo hemos realizado lo siguiente:

- Crear un script de backup del Registro del Sistema.
- Añadir el script como módulo.
- Probar el módulo.
- Crear una tarea programada, para que se ejecute la función todos los días a las 2 de la madrugada.
- Eliminar la tarea programada.

Administración con PowerShell

PowerShell es una herramienta de administración de sistemas que ha ganado popularidad en los últimos años debido a su capacidad para automatizar tareas repetitivas y administrar sistemas de manera eficiente y efectiva.

En este punto, se explorarán varios tipos de administración de sistemas, incluyendo la administración de procesos, servicios, usuarios y grupos, discos y almacenamiento, redes y servidores. Cada uno de estos tipos de administración es crucial para el funcionamiento eficiente de un sistema informático y requiere habilidades y herramientas específicas para su gestión.

En particular, este apartado se centrará en cómo PowerShell puede ser utilizado para administrar estos diferentes aspectos del sistema, examinando las funciones y comandos específicos que están disponibles para cada tipo de administración.

Administración de servicios:

Los servicios son procesos en segundo plano que se ejecutan en un sistema y proporcionan funcionalidades y características específicas para los usuarios y aplicaciones.

Con PowerShell, se pueden administrar los servicios de una manera fácil y eficiente. Se pueden iniciar, detener, pausar y reiniciar servicios utilizando comandos específicos, así como configurar su comportamiento y estado de inicio.

El comando más esencial para obtener los servicios en un equipo local es el cmdlet `Get-Service`. Si se usa el este comando sin parámetros, se devolverán todos los servicios del sistema, independientemente de cuál sea su estado.

```
PS C:\WINDOWS\system32> Get-Service
```

Status	Name	DisplayName
Stopped	AarSvc_6d66f8e	Agent Activation Runtime_6d66f8e
Running	AdobeARMservice	Adobe Acrobat Update Service
Stopped	AJRouter	Servicio de enrutador de AllJoyn
Stopped	ALG	Servicio de puerta de enlace de niv...
Stopped	AppIDSvc	Identidad de aplicación
Running	Appinfo	Información de la aplicación
Running	AppMgmt	Administración de aplicaciones
Stopped	AppReadiness	Preparación de aplicaciones
Stopped	AppVClient	Microsoft App-V Client
Running	AppXSvc	Servicio de implementación de AppX ...
Stopped	AssignedAccessM...	Servicio AssignedAccessManager
Running	AudioEndpointBu...	Compilador de extremo de audio de W...
Running	Audiosrv	Audio de Windows
Stopped	autotimesvc	Hora de la red de telefonía móvil
Stopped	AxInstSV	Instalador de ActiveX (AxInstSV)
Stopped	BcastDVRUserSer...	Servicio de usuario de difusión y G...
Stopped	BDESVC	Servicio Cifrado de unidad BitLocker
Stopped	BEService	BattleEye Service
Running	BFE	Motor de filtrado de base

Ilustración 111. Ejemplo de uso de `Get-Service`

Si lo que se quiere obtener es un servicio concreto, se puede filtrar la búsqueda del cmdlet haciendo uso de sus parámetros, como por ejemplo el parámetro `-Name` que indica el nombre del servicio o el parámetro `-DisplayName` que proporciona un nombre más intuitivo con palabras clave para encontrar el tipo de servicio.

```
PS C:\WINDOWS\system32> Get-Service -Name Spooler
```

Status	Name	DisplayName
Running	Spooler	Cola de impresión

```
PS C:\WINDOWS\system32> Get-Service -DisplayName Hora*
```

Status	Name	DisplayName
Stopped	autotimesvc	Hora de la red de telefonía móvil
Running	W32Time	Hora de Windows

Ilustración 112. Ejemplos de filtrado de servicios

Para encontrar un servicio concreto, también se puede filtrar la búsqueda por el valor de una de sus propiedades, por ejemplo, si se quiere listar solo los servicios en estado “Running”, se filtra la búsqueda por el valor de su propiedad `Status`:

```
PS C:\WINDOWS\system32> Get-Service | Where-Object {$_.Status -eq "Running"}

Status Name                DisplayName
-----
Running AdobeARMService      Adobe Acrobat Update Service
Running Appinfo         Información de la aplicación
Running AppMgmt         Administración de aplicaciones
Running AudioEndpointBu... Compilador de extremo de audio de W...
Running Audiosrv        Audio de Windows
Running BFE             Motor de filtrado de base
Running BrokerInfrastru... Servicio de infraestructura de tare...
Running BTAGService     Servicio de puerta de enlace de aud...
Running BthAvctpSvc     Servicio AVCTP
Running bthserv         Servicio de compatibilidad con Blue...
Running camsvc          Servicio Administrador de funcional...
```

Ilustración 113. Filtrado de servicios en ejecución

O si se quiere listar solo los servicios que se inicien automáticamente, se puede filtrar la búsqueda por el valor de la propiedad `StartType`:

```
PS C:\WINDOWS\system32> Get-Service |
>> Where-Object {$_.StartType -eq "Automatic"} |
>> Select-Object Name, StartType

Name                StartType
-----
AdobeARMService     Automatic
AudioEndpointBuilder Automatic
Audiosrv             Automatic
BFE                 Automatic
BrokerInfrastructure Automatic
CDPSvc              Automatic
CDPUserService_6d66f8e Automatic
ClickToRunSvc       Automatic
CoreMessagingRegistrar Automatic
cplspcon            Automatic
CryptSvc            Automatic
DcomLaunch          Automatic
DeviceAssociationService Automatic
Dhcp                Automatic
DiagTrack            Automatic
```

Ilustración 114. Filtrado de servicios por tipo de inicio

El cmdlet `Get-Service` tiene dos parámetros que son muy útiles para la administración de servicios:

- Uno de ellos es el parámetro `DependentServices`, que obtiene servicios que dependen del servicio que se indica. Por ejemplo, si se utiliza este parámetro con el servicio `Spooler`, que administra la cola de impresión, se obtiene que un servicio dependiente de él es el servicio de Fax:

```
PS C:\WINDOWS\system32> Get-Service -DependentServices Spooler
```

Status	Name	DisplayName
Stopped	Fax	Fax

Ilustración 115. Obtención de servicios que dependen del indicado

- Otro parámetro útil es `RequiredServices`, que obtiene servicios de los que depende el servicio que se le indica. Por ejemplo, si se utiliza este parámetro con el servicio Fax, que permite enviar y recibir faxes, se obtiene que depende de varios servicios, como el servicio Spooler, que administra la cola de impresión o el servicio TapiSrv, que permite que las aplicaciones interactúen con los dispositivos de comunicación:

```
PS C:\WINDOWS\system32> Get-Service -RequiredServices Fax
```

Status	Name	DisplayName
Running	RpcSs	Llamada a procedimiento remoto (RPC)
Stopped	TapiSrv	Telefonía
Running	Spooler	Cola de impresión

Ilustración 116. Obtención de dependencias

A continuación, se pondrá atención a los comandos básicos de administración del estado de servicios. Veremos los comandos esenciales para la administración de servicios, que permiten detener, iniciar, suspender y reanudar servicios en PowerShell.

Stop-Service.

Se puede hacer uso del cmdlet `Stop-Service`, que se utiliza para detener un servicio inmediatamente y también cualquier tarea en curso que esté realizando.

Algunos de los parámetros interesantes de este cmdlet son:

- `-Name`: sirve para especificar el nombre del servicio que se desea detener.
- `-PassThru`: sirve para que la ejecución del comando devuelva un objeto que represente al servicio iniciado.

- -Confirm: sirve para que el sistema solicite confirmación del usuario antes de detener el servicio.

```
PS C:\WINDOWS\system32> Stop-Service -Name Spooler -Confirm -PassThru

Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Stop-Service" en el destino "Cola de impresión (Spooler)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):

Status      Name           DisplayName
-----
Stopped     Spooler        Cola de impresión
```

Ilustración 117. Ejemplo de parado de servicio

Es importante tener en cuenta que algunos servicios pueden estar configurados para reiniciarse automáticamente después de que se detengan. Por lo tanto, el comando `Stop-Service` no impedirá que el servicio se reinicie automáticamente en estos casos. Para evitar que el servicio se reinicie, se debe configurar la propiedad `StartType` del servicio a `Disabled`, actividad que se abordará más adelante.

Start-Service

Se puede hacer uso del cmdlet `Start-Service`, que se utiliza para iniciar un servicio y que comience a ejecutarse.

```
PS C:\WINDOWS\system32> Start-Service -Name Spooler -Confirm -PassThru

Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Start-Service" en el destino "Cola de impresión (Spooler)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):

Status      Name           DisplayName
-----
Running     Spooler        Cola de impresión
```

Ilustración 118. Ejemplo de comienzo de servicio

Es importante tener en cuenta que, como se vio antes, **algunos servicios pueden tener dependencias de otros servicios**. En estos casos, es posible que el cmdlet `Start-Service` no inicie el servicio correctamente si sus dependencias no están disponibles o no se han iniciado previamente.

Suspend-Service

Se puede hacer uso del cmdlet `Suspend-Service`, que se utiliza para suspender un servicio, es decir, pausarlo temporalmente en lugar de detenerlo por completo. Esto significa que el servicio todavía estará disponible, pero no realizará ninguna tarea

mientras esté suspendido. Si se reanuda después de suspenderlo, continuará desde donde lo dejó antes de la suspensión.

```
PS C:\WINDOWS\system32> Suspend-Service -Name stisvc -Confirm -PassThru
Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Suspend-Service" en el destino "Adquisición de imágenes de Windows (WIA) (stisvc)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):

Status   Name           DisplayName
-----
Paused   stisvc         Adquisición de imágenes de Windows ...
```

Ilustración 119. Ejemplo de suspensión de servicio

Es importante tener en cuenta que no todos los servicios admiten la suspensión. Además, como se vio antes, algunos servicios pueden tener dependencias de otros servicios o componentes del sistema. En estos casos, es posible que la suspensión del servicio no sea recomendable o puede causar problemas en el funcionamiento de otros servicios o componentes del sistema.

Una manera de saber qué servicios pueden ser suspendidos, es analizando el valor de la propiedad `CanPauseAndContinue` de los servicios. Por ejemplo, si se quiere ver una lista de los servicios que sí que pueden ser suspendidos se ejecuta lo siguiente:

```
PS C:\WINDOWS\system32> Get-Service | Where-Object CanPauseAndContinue -eq True

Status   Name           DisplayName
-----
Running  LanmanWorkstation Estación de trabajo
Running  MYSQL80        MYSQL80
Running  stisvc         Adquisición de imágenes de Windows ...
Running  TapiSrv        Telefonía
Running  Winmgmt        Instrumental de administración de W...
```

Ilustración 120. Obtención de servicios que pueden ser suspendidos

Por lo tanto, si se intenta suspender un servicio que no pertenece a esa lista se producirá un error:

```
PS C:\WINDOWS\system32> Suspend-Service -Name Spooler
Suspend-Service : No se puede suspender el servicio 'Cola de impresión (Spooler)' porque éste no admite la suspensión ni la reanudación.
En línea: 1 Carácter: 1
+ Suspend-Service -Name Spooler
+ ~~~~~
+ CategoryInfo          : CloseError: (System.ServiceProcess.ServiceController:ServiceController) [Suspend-Service], ServiceCommandException
+ FullyQualifiedErrorId : CouldNotSuspendServiceNotSupported,Microsoft.PowerShell.Commands.SuspendServiceCommand
```

Ilustración 121. Ejemplo de intento de suspensión de un servicio que no lo permite

Restart-Service

Se puede hacer uso del cmdlet `Restart-Service`, que se utiliza para reiniciar un servicio. Este comando detiene y luego inicia el servicio especificado, lo que puede ser útil si el servicio no está respondiendo correctamente o si se han realizado cambios en la configuración del servicio que requieren un reinicio para que tengan efecto.

```
PS C:\WINDOWS\system32> Restart-Service -Name WSearch -Confirm -PassThru

Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Restart-Service" en el destino "Windows Search (WSearch)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):

Status      Name      DisplayName
-----
Running WSearch  Windows Search
```

Ilustración 122. Ejemplo de reinicio de servicio

Es importante tener en cuenta que, al reiniciar un servicio, cualquier tarea en curso que se esté realizando por el servicio se interrumpirá. Además, si el servicio tiene dependencias de otros servicios o componentes del sistema, estos también se verán afectados por el reinicio del servicio.

Set-Service

Finalizando con la administración de servicios, se puede hacer uso del cmdlet `Set-Service`, que se utiliza para cambiar la configuración de un servicio, realizando tareas como cambiar su nombre descriptivo, su descripción, su tipo de inicio e iniciar, detener o suspender el servicio.

Veamos un ejemplo para cada uno de estos posibles cambios:

Para cambiar el nombre descriptivo de un servicio concreto, existe para este cmdlet el parámetro `-DisplayName`:

```
PS C:\WINDOWS\system32> Set-Service -Name dcsvc -DisplayName "Servicio de virtualización de credenciales de seguridad distribuidas"

Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Set-Service" en el destino "dcsvc (dcsvc)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):

Status      Name      DisplayName
-----
Stopped dcsvc    Servicio de virtualización de crede...
```

Ilustración 123. Cambio del nombre descriptivo de un servicio

Para cambiar el tipo de inicio de un servicio concreto, existe para este cmdlet el parámetro `-StartupType`:

```
PS C:\WINDOWS\system32> Set-Service -Name BITS -StartupType Automatic -Confirm -PassThru | Select-Object Name, StartType
Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Set-Service" en el destino "Servicio de transferencia inteligente en segundo plano (BITS)"
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):

Name StartType
----
BITS Automatic
```

Ilustración 124. Cambio de tipo de inicio de un servicio

Para cambiar la descripción de un servicio concreto, existe para este cmdlet el parámetro `-Description`:

```
PS C:\WINDOWS\system32> Set-Service -Name BITS -Description "Transfiere archivos en segundo plano mediante el uso de ancho de banda de red inactivo."
Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Set-Service" en el destino "Servicio de transferencia inteligente en segundo plano (BITS) (BITS)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):
```

Ilustración 125. Cambio de descripción de un servicio

Pero en este caso existe una curiosidad y es que los servicios no cuentan con una propiedad que sea la descripción, por lo tanto, la descripción de un servicio no se puede mostrar con el cmdlet `Select-Object`, sino que hay que ejecutar el siguiente comando para poder verla:

```
PS C:\WINDOWS\system32> Get-CimInstance Win32_Service -Filter 'Name = "BITS"' | Format-List Name, Description
Name           : BITS
Description    : Transfiere archivos en segundo plano mediante el uso de ancho de banda de red inactivo.
```

Ilustración 126. Obtención de la descripción de un servicio

Para iniciar, parar o suspender un servicio concreto, existe para este cmdlet el parámetro `-Status`:

```
PS C:\WINDOWS\system32> Set-Service -Name Spooler -Status Running -Confirm -PassThru
Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Set-Service" en el destino "Cola de impresión (Spooler)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterm

Status Name DisplayName
-----
Running Spooler Cola de impresión
```

Ilustración 127. Cambio de estado de un servicio con `Set-Service`: Iniciar servicio

```

PS C:\WINDOWS\system32> Set-Service -Name stisvc -Status Paused -Confirm -PassThru

Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Set-Service" en el destino "Adquisición de imágenes de Windows ..."
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspende [?] Ayuda (el valor predeterminado es S)

Status      Name      DisplayName
-----
Paused      stisvc     Adquisición de imágenes de Windows ...

```

Ilustración 129. Cambio de estado de un servicio con Set-Service: Pausar servicio

```

PS C:\WINDOWS\system32> Set-Service -Name BITS -Status Stopped -Confirm -PassThru

Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Set-Service" en el destino "Servicio de transferencia inteligente de alto rendimiento"
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspende [?] Ayuda (el valor predeterminado es S)

Status      Name      DisplayName
-----
Stopped      BITS      Servicio de transferencia inteligente de alto rendimiento

```

Ilustración 128. Cambio de estado de un servicio con Set-Service: Parar servicio

Administración de procesos

Cuando un programa se inicia, el sistema operativo crea un proceso que lo representa. Este proceso es una entidad virtual que contiene información sobre el programa en ejecución, como su estado, identidad, prioridad, recursos utilizados y otros detalles relevantes.

La administración de procesos es una tarea importante en la administración de sistemas, ya que permite a los administradores monitorear y controlar el estado de los procesos, que pueden estar activos, en espera o en pausa, y se pueden crear, detener, pausar o reiniciar mediante herramientas de administración del sistema.

Get-Process

Para obtener información detallada sobre los procesos en ejecución, existe en PowerShell el cmdlet `Get-Process`. Si no se le añade ningún parámetro, muestra una lista de todos los procesos en ejecución en el sistema, con información detallada sobre cada uno, como el nombre (`ProcessName`), el identificador (`Id`), la carga de trabajo de la CPU, la memoria que está utilizando y otros detalles relevantes:

```
PS C:\WINDOWS\system32> Get-Process
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	--	--	-----
140	11	11428	22712	0,08	1524	9	ai
330	20	9024	29168	0,11	5116	9	ApplicationFrameHost
128	8	1620	1836	0,03	5064	0	armsvc
186	12	10604	17836	0,19	14612	0	audiodg
220	14	9216	20344	0,25	88	9	chrome
176	9	2216	7980	0,05	1700	9	chrome
283	17	8388	22616	0,58	2380	9	chrome
369	20	64968	113292	13,80	9504	9	chrome
319	23	19716	45916	13,27	11236	9	chrome

Ilustración 130. Ejemplo de ejecución del cmdlet `Get-Process`

Para obtener información detallada sobre un proceso en ejecución concreto se puede filtrar la búsqueda por varios parámetros, como por ejemplo el nombre del proceso o su identificador:

```
PS C:\WINDOWS\system32> Get-Process -Name Acrobat
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	--	--	-----
649	60	129264	164152	2,67	7964	9	Acrobat
738	27	40928	66388	1,19	12088	9	Acrobat


```
PS C:\WINDOWS\system32> Get-Process -Name Search*
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	--	--	-----
1449	167	209836	297928	12,45	14860	9	SearchApp
148	8	1532	7772	0,02	13544	0	SearchFilterHost
955	73	33916	44176	8,17	15840	0	SearchIndexer
271	10	1896	8580	0,06	16792	9	SearchProtocolHost


```
PS C:\WINDOWS\system32> Get-Process -Id 13948
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	--	--	-----
2214	74	138780	251988	85,42	13948	9	WINWORD

Ilustración 131. Diversos filtrados de procesos

Para `Get-Process` existen muchos otros parámetros interesantes, como, por ejemplo:

- `-FileVersionInfo`: muestra información de versión del módulo principal del proceso, que su archivo .exe

```
PS C:\WINDOWS\system32> Get-Process WINWORD -FileVersionInfo
```

ProductVersion	FileVersion	FileName
16.0.16227.20258	16.0.16227.20258	C:\Program Files\Microsoft Office\Root\Office16\WINWORD.EXE

Ilustración 132. `Get-Process`: Información del módulo principal del proceso.

- `-IncludeUserName`: muestra quién es el propietario del proceso.

```
PS C:\WINDOWS\system32> Get-Process WINWORD -IncludeUserName
```

Handles	WS(K)	CPU(s)	Id	UserName	ProcessName
2538	240824	101,64	13948	DESKTOP-1FIPQM0\Miguel	WINWORD

Ilustración 133. `Get-Process`: Información del propietario del proceso

- `-Module`: muestra los módulos cargados por el proceso.

```
PS C:\WINDOWS\system32> Get-Process WINWORD -Module
```

Size(K)	ModuleName	FileName
1604	WINWORD.EXE	C:\Program Files\Microsoft Office\Root\Office16\WINWORD.EXE
2016	ntdll.dll	C:\WINDOWS\SYSTEM32\ntdll.dll
764	KERNEL32.DLL	C:\WINDOWS\System32\KERNEL32.DLL
2888	KERNELBASE.dll	C:\WINDOWS\System32\KERNELBASE.dll
576	apphelp.dll	C:\WINDOWS\SYSTEM32\apphelp.dll
696	ADVAPI32.dll	C:\WINDOWS\System32\ADVAPI32.dll
632	msvcrt.dll	C:\WINDOWS\System32\msvcrt.dll
624	sechost.dll	C:\WINDOWS\System32\sechost.dll
1168	RPCRT4.dll	C:\WINDOWS\System32\RPCRT4.dll
1952	AppVirtSubsystems64.dll	C:\Program Files\Microsoft Office\Root\Office16\AppVirtSubsystems64.dll

Ilustración 134. `Get-Process`: Información de los módulos cargados por el proceso

Stop-Process

Para detener uno o varios procesos en ejecución, existe en PowerShell el cmdlet `Stop-Process`. Se puede especificar el proceso a detener por su nombre, su identificador o pasando un objeto de tipo proceso a `Stop-Process`. Veamos un ejemplo de cada uno de estos casos:

```
PS C:\WINDOWS\system32> Stop-Process -Name Acrobat -Confirm -PassThru

Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Stop-Process" en el destino "Acrobat (7964)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):

Handles NPM(K) PM(K) WS(K) CPU(s) Id SI ProcessName
-----
639 63 129832 164440 3,05 7964 9 Acrobat

PS C:\WINDOWS\system32> Stop-Process -Id 10940 -Confirm -PassThru

Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Stop-Process" en el destino "Acrobat (10940)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):

Handles NPM(K) PM(K) WS(K) CPU(s) Id SI ProcessName
-----
590 62 128536 164020 3,08 10940 9 Acrobat

PS C:\WINDOWS\system32> Get-Process -Name Acrobat | Stop-Process -Confirm -PassThru

Confirmar
¿Está seguro de que desea realizar esta acción?
Se está realizando la operación "Stop-Process" en el destino "Acrobat (6064)".
[S] Sí [O] Sí a todo [N] No [T] No a todo [U] Suspender [?] Ayuda (el valor predeterminado es "S"):

Handles NPM(K) PM(K) WS(K) CPU(s) Id SI ProcessName
-----
598 62 131720 166592 2,42 6064 9 Acrobat
```

Ilustración 135. Ejemplos de diversas formas de parado de un proceso

Como se puede observar en las imágenes, el cmdlet `Stop-Process` cuenta también con dos parámetros interesantes:

- El parámetro `-Confirm` indica que antes de detener el proceso tiene que pedir confirmación al usuario.
- El parámetro `-PassThru` pasa el objeto de tipo proceso que acaba de detener. Sin este parámetro, no habría ninguna salida después de un `Stop-Process`

Start-Process

Para iniciar un nuevo proceso (aplicación o script), existe en PowerShell el cmdlet `Start-Process`. Este cmdlet es muy útil para no tener que abrir manualmente la

aplicación o el script y tiene una amplia gama de opciones y parámetros que se pueden utilizar para personalizar su funcionamiento. Veamos algunos de ellos:

- El parámetro `-FilePath` especifica la ruta completa del archivo ejecutable que se va a iniciar con el proceso.
- El parámetro `-PassThru` devuelve un objeto que representa el proceso que se inició.

```
PS D:\Desktop> Start-Process -FilePath "C:\Windows\System32\notepad.exe" -PassThru
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
35	4	904	2428	0,00	2684	10	notepad

Ilustración 136. Start-Process: Ejemplo de parámetro `-PassThru`

- El parámetro `-ArgumentList` especifica los argumentos que se van a pasar al proceso que se está iniciando.
- El parámetro `-WorkingDirectory` especifica el directorio de trabajo para el proceso que se está iniciando. Esto significa que cualquier archivo que se abra o se cree se guardará en el directorio especificado.

```
PS C:\Windows\System32> Start-Process -FilePath "cmd.exe" -ArgumentList "/c mkdir NuevaCarpeta" -WorkingDirectory "D:\Documents\FIC\Q6\AS0" -PassThru
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
19	3	1548	2040	0,00	604	10	cmd

Ilustración 137. Start-Process: Ejemplo de parámetro `-WorkingDirectory`

- El parámetro `-WindowStyle` especifica el estilo de la ventana del proceso que se está iniciando. Este parámetro acepta los valores "Normal", "Maximized", "Minimized" y "Hidden".

```
PS C:\Windows\System32> Start-Process -FilePath "notepad.exe" -WindowStyle "Maximized" -PassThru
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
0	2	552	1556	0,02	5368	10	notepad

Ilustración 138. . Start-Process: Ejemplo de parámetro `-WindowStyle`

- El parámetro `-Verb` se utiliza para especificar la acción que se llevará a cabo en el archivo o recurso especificado. El valor que se especifica para este parámetro debe ser un verbo de acción válido, como "Open", "Print", "Edit", "Copy" o "Paste", entre otros.

```
PS C:\Windows\System32> Start-Process -FilePath "D:\Documents\FIC\Q6\AS0\TT\TT.txt" -Verb Print -PassThru
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
34	7	1648	3696	0,02	13784	10	notepad

Ilustración 139. . Start-Process: Ejemplo de parámetro -Verb

Wait-Process

Para esperar a que un proceso en ejecución se detenga, existe en PowerShell el cmdlet `Wait-Process`. Este cmdlet suprime el símbolo del sistema hasta que se detengan los procesos. Se puede especificar el proceso a detener a través de los parámetros `-Name`, `-Id` o canalizando un objeto de tipo proceso. Para indicar el tiempo máximo que el cmdlet debe esperar a que se complete el proceso antes de continuar se utiliza el parámetro `-Timeout`

```
PS C:\Windows\System32> Get-Process -Name notep*
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
263	16	3812	15912	0,06	6208	10	notepad

```
PS C:\Windows\System32> Wait-Process -Name notepad
PS C:\Windows\System32> Get-Process -Name notep*
```

```
PS C:\Windows\System32> Get-Process -Name notepad
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
263	16	3848	15908	0,08	11568	10	notepad

```
PS C:\Windows\System32> Wait-Process -Id 11568
PS C:\Windows\System32> Get-Process -Name notep*
```

```
PS C:\Windows\System32> Get-Process -Name notep*
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
263	16	3852	15960	0,16	6552	10	notepad

```
PS C:\Windows\System32> Get-Process -Name notepad | Wait-Process
```

Ilustración 140. Realización de un Wait-Process por diversas vías: Nombre ; ID ; Canalización

Administración de usuarios y grupos

La administración de usuarios y grupos es fundamental para garantizar la seguridad y el control de acceso a los recursos de la red. En esta sección, se explorará cómo PowerShell se puede utilizar para administrar usuarios y grupos locales en un equipo con Windows. Se verá cómo los cmdlets `Get-LocalUser`, `Get-LocalGroup` y `Get-LocalGroupMember` permiten obtener información detallada de los usuarios y grupos, y cómo los cmdlets `New-LocalUser` y `Add-LocalGroupMember` permiten crear y administrar usuarios y grupos.

Get-LocalUser

Para obtener información detallada de los usuarios locales en un equipo, existe en PowerShell el cmdlet `Get-LocalUser`. Si no se le añade ningún parámetro, muestra una lista de todas las cuentas de usuario locales con información sobre cada una, como el nombre (`Name`), si está habilitada o no (`Enabled`) y una pequeña descripción (`Description`):

```
PS C:\WINDOWS\system32> Get-LocalUser
```

Name	Enabled	Description
Administrador	False	Cuenta integrada para la administración del equipo o dominio
DefaultAccount	False	Cuenta de usuario administrada por el sistema.
Invitado	False	Cuenta integrada para el acceso como invitado al equipo o dominio
Miguel	True	
WDAGUtilityAccount	False	Una cuenta de usuario que el sistema administra y usa para escenarios de Protección

Ilustración 141. Información que proporciona el cmdlet Get-LocalUser

Para obtener información detallada sobre una cuenta de usuario concreta se puede filtrar la búsqueda por parámetros como su nombre o su identificador de seguridad (SID):


```
PS C:\WINDOWS\system32> Get-LocalUser -Name Miguel | Select-Object *
AccountExpires      :
Description         :
Enabled             : True
FullName            :
PasswordChangeableDate : 01/05/2020 3:12:41
PasswordExpires     :
UserMayChangePassword : True
PasswordRequired    : False
PasswordLastSet     : 01/05/2020 3:12:41
LastLogon           : 10/04/2023 11:21:54
Name                : Miguel
SID                 : S-1-5-21-619924196-4045554399-1956444398-1001
PrincipalSource      : Local
ObjectClass          : Usuario
```

Ilustración 143. Información de usuario local, filtrando por nombre

```
PS C:\WINDOWS\system32> Get-LocalUser -SID S-1-5-21-619924196-4045554399-1956444398-500 | Select-Object *
AccountExpires      :
Description         : Cuenta integrada para la administración del equipo o dominio
Enabled             : False
FullName            :
PasswordChangeableDate :
PasswordExpires     :
UserMayChangePassword : True
PasswordRequired    : True
PasswordLastSet     :
LastLogon           :
Name                : Administrador
SID                 : S-1-5-21-619924196-4045554399-1956444398-500
PrincipalSource      : Local
ObjectClass          : Usuario
```

Ilustración 142. Información de usuario local, filtrando por SID

Get-LocalGroup

Para obtener información detallada de los grupos locales en un equipo, existe en PowerShell el cmdlet `Get-LocalGroup`. Si no se le añade ningún parámetro, muestra una lista de todos los grupos locales con información sobre cada uno, como el nombre (Name) y una pequeña descripción (Description):

```
PS C:\WINDOWS\system32> Get-LocalGroup
Name                Description
-----
docker-users        Users of Docker Desktop
Administradores     Los administradores tienen acceso completo y sin restricciones al equipo o dominio
Administradores de Hyper-V Los miembros de este grupo tienen acceso completo y sin restricciones a todas las
Duplicadores        Pueden replicar archivos en un dominio
IIS_IUSRS           Grupo integrado usado por Internet Information Services.
Invitados           De forma predeterminada, los invitados tienen el mismo acceso que los miembros de
Lectores del registro de eventos Los miembros de este grupo pueden leer registros de eventos del equipo local.
Operadores criptográficos Los miembros tienen autorización para realizar operaciones criptográficas.
Operadores de asistencia de control de acceso Los miembros de este grupo pueden consultar de forma remota los atributos de auto
Operadores de configuración de red Los miembros en este equipo pueden tener algunos privilegios administrativos para
Operadores de copia de seguridad Los operadores de copia de seguridad pueden invalidar restricciones de seguridad
Propietarios del dispositivo Los miembros de este grupo pueden cambiar la configuración de todo el sistema.
System Managed Accounts Group Los miembros de este grupo los administra el sistema.
Usuarios            Los usuarios no pueden hacer cambios accidentales o intencionados en el sistema y
Usuarios avanzados  Los usuarios avanzados se incluyen para la compatibilidad con versiones anteriores
Usuarios COM distribuidos Los miembros pueden iniciar, activar y usar objetos de COM distribuido en este eq
Usuarios de administración remota Los miembros de este grupo pueden acceder a los recursos de WMI mediante protocolo
Usuarios de escritorio remoto A los miembros de este grupo se les concede el derecho de iniciar sesión remotame
```

Ilustración 144. Información mostrada por defecto por el cmdlet `Get-LocalGroup`

Para obtener información detallada sobre un grupo concreto se puede filtrar la búsqueda por parámetros como su nombre o su identificador de seguridad (SID):

```
PS C:\WINDOWS\system32> Get-LocalGroup -Name Administradores | Select-Object *

Description      : Los administradores tienen acceso completo y sin restricciones al equipo o dominio
Name             : Administradores
SID              : S-1-5-32-544
PrincipalSource  : Local
ObjectClass      : Grupo
```

Ilustración 145. Obtención de información de un grupo local, filtrando por nombre

```
PS C:\WINDOWS\system32> Get-LocalGroup -SID S-1-5-32-545 | Select-Object *

Description      : Los usuarios no pueden hacer cambios accidentales o intencionados en el sistema y pueden ejecutar la mayoría de aplicaciones
Name             : Usuarios
SID              : S-1-5-32-545
PrincipalSource  : Local
ObjectClass      : Grupo
```

Ilustración 146. Obtención de información de un grupo local, filtrando por identificador de seguridad

*-LocalUser

New-LocalUser. Para crear una cuenta de usuario local en un equipo, existe en PowerShell el cmdlet `New-LocalUser`. Es importante tener en cuenta que, para crear una cuenta de usuario local con este cmdlet, se deben tener los permisos necesarios en el equipo para crear nuevas cuentas de usuario. Por lo tanto, o se ejecuta el cmdlet en una sesión de PowerShell con privilegios elevados o como un usuario que tenga permisos de administrador en el equipo.

Este cmdlet cuenta con varios parámetros que pueden ser útiles en diferentes situaciones. Los parámetros más interesantes son:

- **Name:** Este parámetro se utiliza para establecer el nombre de la cuenta de usuario local que se está creando.
- **NoPassword:** Este parámetro permite crear una cuenta de usuario local sin una contraseña.
- **Password:** Este parámetro permite establecer una contraseña para la cuenta de usuario local que se está creando. Se utiliza el cmdlet `ConvertTo-SecureString`, que convierte la contraseña en un objeto de tipo `SecureString` utilizando un algoritmo de cifrado específico.

- **Description:** Este parámetro permite establecer una descripción para la cuenta de usuario local.

```
PS C:\WINDOWS\system32> New-LocalUser -Name "Usuario1" -Description "Usuario de prueba 1" -NoPassword
```

Name	Enabled	Description
-----	-----	-----
Usuario1	True	Usuario de prueba 1

Ilustración 148. Creación de usuario local, sin contraseña

```
PS C:\WINDOWS\system32> New-LocalUser -Name "Usuario2" -Description "Usuario de prueba 2" -Password (ConvertTo-SecureString -AsPlainText "12345" -Force)
```

Name	Enabled	Description
-----	-----	-----
Usuario2	True	Usuario de prueba 2

Ilustración 147. Creación de usuario local, empleando contraseña

Remove-LocalUser. Para eliminar un usuario local en un equipo, existe en PowerShell el cmdlet `Remove-LocalUser`. Es recomendable verificar que no haya archivos, carpetas o recursos importantes asociados con la cuenta de usuario antes de eliminarla, ya que esto puede afectar el acceso a los recursos del equipo. Se puede especificar el usuario local a eliminar por su nombre a través del parámetro `-Name` o pasando un objeto de tipo usuario local a este cmdlet.

```
PS C:\WINDOWS\system32> Get-LocalUser -Name "Usuario1"
```

Name	Enabled	Description
-----	-----	-----
Usuario1	True	Usuario de prueba 1

```
PS C:\WINDOWS\system32> Remove-LocalUser -Name "Usuario1"
```

```
PS C:\WINDOWS\system32> Get-LocalUser -Name "Usuario1"
```

```
Get-LocalUser : No se encontró el usuario Usuario1.
```

```
PS C:\WINDOWS\system32> Get-LocalUser -Name "Usuario2"
```

Name	Enabled	Description
-----	-----	-----
Usuario2	True	Usuario de prueba 2

```
PS C:\WINDOWS\system32> Get-LocalUser -Name "Usuario2" | Remove-LocalUser
```

```
PS C:\WINDOWS\system32> Get-LocalUser -Name "Usuario2"
```

```
Get-LocalUser : No se encontró el usuario Usuario2.
```

Ilustración 149. Eliminación de usuarios locales del sistema

*-LocalGroup

New-LocalGroup. Para crear un grupo local en un equipo, existe en PowerShell el cmdlet `New-LocalGroup`. Es importante tener en cuenta que, para crear un grupo local con este cmdlet, hay que asegurarse de que el nombre del grupo sea único en el equipo y que cumpla con las restricciones de caracteres y longitud establecidas por el sistema operativo.

Este cmdlet cuenta con los parámetros `-Name`, que establece el nombre del grupo local que se está creando, y el parámetro `-Description`, que establece una descripción.

```
PS C:\WINDOWS\system32> New-LocalGroup -Name 'Grupo1' -Description 'Grupo de prueba 1'

Name      Description
----      -
Grupo1    Grupo de prueba 1
```

Ilustración 150. Creación de grupo local

Add-LocalGroupMember. Para agregar miembros a un grupo local en un equipo, existe en PowerShell el cmdlet `Add-LocalGroupMember`. Este cmdlet utiliza los parámetros `-Group` para indicar el nombre del grupo local al que se desea agregar un miembro, y `-Member` para indicar el nombre del usuario local que se desea agregar.

```
PS C:\WINDOWS\system32> Add-LocalGroupMember -Group Grupo1 -Member Usuario2 -Verbose
DETALLADO: Se está realizando la operación "Agregar miembro DESKTOP-1FIPQM0\Usuario2" en el destino "Grupo1".
```

Ilustración 151. Adición de miembro a un grupo local

Get-LocalGroupMember. Para obtener una lista de los miembros de un grupo local en un equipo, existe en PowerShell el cmdlet `Get-LocalGroupMember`. Este cmdlet utiliza el parámetro `-Group` para indicar el nombre del grupo local del que se desea listar los miembros.

```
PS C:\WINDOWS\system32> Get-LocalGroupMember Grupo1

ObjectClass Name                                PrincipalSource
-----
Usuario      DESKTOP-1FIPQM0\Usuario1 Local
Usuario      DESKTOP-1FIPQM0\Usuario2 Local
```

Ilustración 152. Obtención de los grupos locales del sistema

Remove-LocalGroupMember. Para eliminar miembros de un grupo local en un equipo, existe en PowerShell el cmdlet `Remove-LocalGroupMember`. Este cmdlet utiliza los parámetros `-Group` para indicar el nombre del grupo local del que se desea eliminar un miembro, y `-Member` para indicar el nombre del usuario local que se desea eliminar.

```
PS C:\WINDOWS\system32> Remove-LocalGroupMember -Group Grupo1 -Member Usuario1
PS C:\WINDOWS\system32> Remove-LocalGroupMember -Group Grupo1 -Member Usuario2
PS C:\WINDOWS\system32> Get-LocalGroupMember Grupo1
PS C:\WINDOWS\system32>
```

Ilustración 153. Eliminación de miembros de un grupo

Remove-LocalGroup. Para eliminar un grupo local en un equipo, existe en PowerShell el cmdlet `Remove-LocalGroup`. Es importante tener en cuenta que antes de eliminar un grupo local se recomienda verificar que no haya usuarios importantes en el grupo antes de eliminarlo, ya que esto puede afectar el acceso a los recursos del equipo. Se puede especificar el grupo local a eliminar por su nombre a través del parámetro `-Name` o pasando un objeto de tipo grupo local a este cmdlet.

```
PS C:\WINDOWS\system32> Get-LocalGroup -Name "Grupo1"

Name      Description
----      -
Grupo1    Grupo de prueba 1

PS C:\WINDOWS\system32> Remove-LocalGroup -Name "Grupo1"
PS C:\WINDOWS\system32> Get-LocalGroup -Name "Grupo1"
Get-LocalGroup : No se encontró el grupo Grupo1.
```

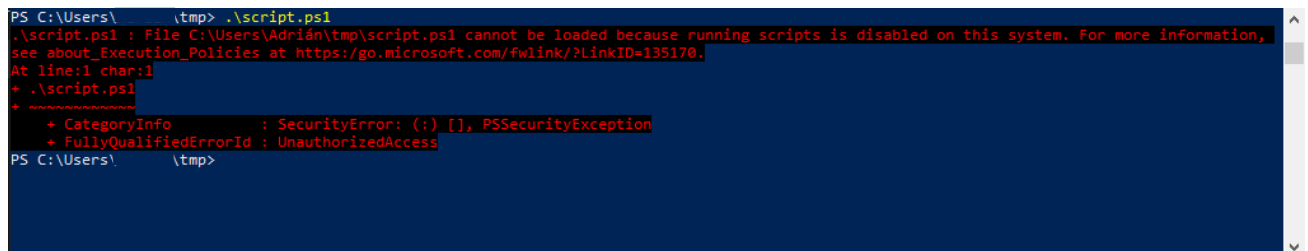
Ilustración 154. Eliminación de grupo local

Anexo I: Instalación y configuración de Vim en Windows.

1. Dirigirse a la web <https://www.vim.org>
2. En la web, ir a **Download**.
3. Seleccionar: “PC: MS-DOS and MS-Windows”
4. Ejecutar el binario. Realizar la instalación al gusto.
5. Actualizar el *Path*, para que PowerShell pueda encontrar el programa.
 - a. Abrir **Panel de Control > Sistema**
 - b. **Editar las variables de entorno del sistema.**
 - c. **Avanzado > Variables de Entorno.**
 - d. **Variables del Sistema** > Seleccionamos *Path* > **Editar...**
 - e. Añadir, como una nueva entrada, la carpeta que incluye el ejecutable *Vim.exe*. En este caso es **C:\Program Files(x86)\Vim\vim90**
 - f. Salir, guardando los cambios.
6. Abrir PowerShell y ver si efectivamente se puede usar el comando **vim**.

Anexo II: Habilitar la ejecución de Scripts de PowerShell en un Sistema Windows.

Problema: Se genera el siguiente error al ejecutar un archivo .ps1 desde PowerShell.



```
PS C:\Users\Adrián\ tmp> .\script.ps1
.\script.ps1 : File C:\Users\Adrián\ tmp\script.ps1 cannot be loaded because running scripts 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
+ .\script.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\Adrián\ tmp>
```

Ilustración 155. Ejemplo de error en la ejecución del script

“[Nombre del Script] cannot be loaded because running scripts is disabled on the system...”

Las políticas de ejecución de PowerShell son una característica de seguridad para controlar las condiciones bajo las cuales PowerShell carga archivos de configuración y ejecuta scripts. Las políticas de ejecución para el sistema local y el usuario actual se guardan en el **registro**. Los *cmdlets* relacionados son **Set-ExecutionPolicy** y **Get-ExecutionPolicy**

PowerShell cuenta con diversas políticas de ejecución:

- **AllSigned**. Para que un script se ejecute o se cargue un archivo de configuración, este fichero debe estar firmado por un publicador de confianza.
- **Bypass**. No existe bloqueo ni mensajes de advertencia. Está pensado para casos de uso en el que PowerShell pertenece a una aplicación más grande, la cual tiene su propio modelo de seguridad.
- **Default**. Para cliente Windows es **Restricted**. Para Windows servers es **RemoteSigned**.
- **RemoteSigned**. Necesita la firma digital de un publicador de confianza para aquellos scripts descargados desde internet. Para aquellos creados en local, no se necesita ninguna firma.

- Si se quisiese ejecutar igualmente un script descargado y no firmado, se podría desbloquear el script, empleando el *cmdlet* `Unblock-File`.
- **Restricted.** Permite ejecutar comandos individuales, pero no scripts. No se permite ejecutar archivos de configuración (**.ps1xml**), ni archivos de módulos (**.psm1**), ni perfiles de PowerShell (**.ps1**).
- **Undefined.** No hay política de ejecución establecida en el *scope* actual. Si esta es la política para todos los *scopes*, la política efectiva es la misma que la **Default**, para cada uno de los casos.
- **Unrestricted.** Es la política por defecto para aquellos sistemas que no sean Windows. No puede ser cambiada de ninguna forma. Los scripts no firmados pueden ser ejecutados. Se produce un mensaje de advertencia cada vez que se ejecuta un script o carga un archivo de configuración.

Cada una de estas políticas afectan a un *scope* concreto, pudiendo ser:

- **MachinePolicy.** Establecida por una *Group Policy*, para todos los usuarios del sistema.
- **UserPolicy.** Establecida por una *Group Policy*, para el usuario actual del sistema.
- **Process.** Solo afecta a la sesión de PowerShell actual. La política de ejecución se guarda en la variable de entorno **\$env:PSExecutionPolicyPreference**, y no en el registro. Cuando termina la sesión, la variable y el valor son desechados.
- **CurrentUser.** Afecta solo al usuario actual. Se guarda en la subclave **HKEY_CURRENT_USER** del registro.
- **LocalMachine.** El por defecto al establecer una política de ejecución. Afecta a todos los usuarios del sistema actual. Se guarda en la subclave **HKEY_LOCAL_MACHINE** del registro.

Para poder ejecutar scripts, se cambiará el valor para el *scope* que se considere. En este ejemplo se realizará para el usuario actual:

1. Abrir una terminal de PowerShell **como administrador**.
2. Listar las ExecutionPolicy de cada scope, empleando el *cmdlet* Get-ExecutionPolicy

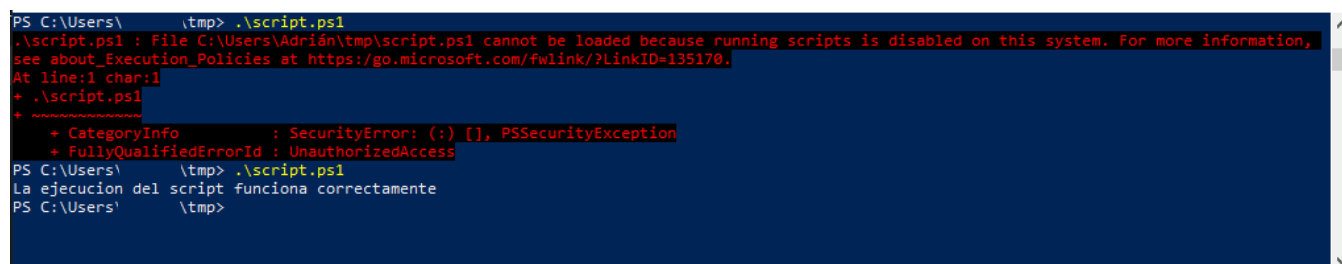
```
PS > Get-ExecutionPolicy -List
```

3. Cambiar la ExecutionPolicy para el *scope* Process, usando el *cmdlet* Set-ExecutionPolicy

```
PS > Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -
Scope CurrentUser
```

4. Aceptar.
5. Se vuelve a listar las políticas, para ver si se ha realizado el cambio correctamente.

Se vuelve a ejecutar el script desde la terminal sin privilegios de administrador, para ver si funcionó correctamente la actualización de políticas de ejecución.



```
PS C:\Users\Adrián\tmp> .\script.ps1
.\script.ps1 : File C:\Users\Adrián\tmp\script.ps1 cannot be loaded because running scripts 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
+ .\script.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess

PS C:\Users\Adrián\tmp> .\script.ps1
La ejecución del script funciona correctamente
PS C:\Users\Adrián\tmp>
```

Ilustración 156. Reejecución del script. Funciona correctamente

Referencias

Adam Bertram, “Your Getting Started Guide to Powershell Functions”

<https://adamtheautomator.com/powershell-functions/>

Atareao, “Objetos en PowerShell” <https://atareao.es/tutorial/powershell/objetos-en-powershell/>

Atareao, “Pipes en PowerShell o el arte de encadenar comandos”

<https://atareao.es/tutorial/powershell/pipes-en-powershell/>

Atareao, “Variables en PowerShell” <https://atareao.es/tutorial/powershell/variables-en-powershell/>

Juan Antonio Soto, “¿Qué es PowerShell y para qué sirve?”. Geeknetic,

<https://www.geeknetic.es/PowerShell/que-es-y-para-que-sirve>

Microsoft, “about_Automatic_Variables” [https://learn.microsoft.com/en-](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-7.3)

[us/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-7.3](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-7.3)

Microsoft, “about_Break” [https://learn.microsoft.com/en-](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_break?view=powershell-7.3)

[us/powershell/module/microsoft.powershell.core/about/about_break?view=powershell-7.3](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_break?view=powershell-7.3)

Microsoft, “about_Command_Precedence” [https://learn.microsoft.com/es-](https://learn.microsoft.com/es-es/powershell/module/microsoft.powershell.core/about/about_command_precedence?view=powershell-7.3)

[es/powershell/module/microsoft.powershell.core/about/about_command_precedence?view=powershell-7.3](https://learn.microsoft.com/es-es/powershell/module/microsoft.powershell.core/about/about_command_precedence?view=powershell-7.3)

Microsoft, “about_Comparison_Operators” [https://learn.microsoft.com/en-](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators?view=powershell-7.3)

[us/powershell/module/microsoft.powershell.core/about/about_comparison_operators?view=powershell-7.3](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators?view=powershell-7.3)

Microsoft, “about_Continue” [https://learn.microsoft.com/en-](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_continue?view=powershell-7.3)

[us/powershell/module/microsoft.powershell.core/about/about_continue?view=powershell-7.3](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_continue?view=powershell-7.3)

Microsoft, “about_Do” [https://learn.microsoft.com/en-](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_do?view=powershell-7.3)

[us/powershell/module/microsoft.powershell.core/about/about_do?view=powershell-7.3](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_do?view=powershell-7.3)

Microsoft, “about_For” [https://learn.microsoft.com/en-](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_for?view=powershell-7.3)

[us/powershell/module/microsoft.powershell.core/about/about_for?view=powershell-7.3](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_for?view=powershell-7.3)

Microsoft, “about_Foreach” [https://learn.microsoft.com/en-](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_foreach?view=powershell-7.3)

[us/powershell/module/microsoft.powershell.core/about/about_foreach?view=powershell-7.3](https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_foreach?view=powershell-7.3)

- Microsoft, “about_If” https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_if?view=powershell-7.3
- Microsoft, “about_Logical_Operators” https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logical_operators?view=powershell-7.3
- Microsoft, “about_Modules” https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_modules?view=powershell-7.3
- Microsoft, “about_Parsing” https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_parsing?view=powershell-7.3
- Microsoft, “about_Scripts” https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scripts?view=powershell-7.3
- Microsoft, “about_Switch” https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch?view=powershell-7.3
- Microsoft, “about_Variables” https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_variables?view=powershell-7.3
- Microsoft, “about_While” https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_while?source=recommendations&view=powershell-7.3
- Microsoft, “Capítulo 9: Funciones” <https://learn.microsoft.com/es-us/powershell/scripting/learn/ps101/09-functions?view=powershell-7.3>
- Microsoft, “Everything you wanted to know about exceptions” <https://learn.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-exceptions?view=powershell-7.3>
- Microsoft, “Managing processes with Process cmdlets” <https://learn.microsoft.com/en-us/powershell/scripting/samples/managing-processes-with-process-cmdlets?view=powershell-5.1>
- Microsoft, “Managing services” <https://learn.microsoft.com/en-us/powershell/scripting/samples/managing-services?view=powershell-5.1>
- Microsoft, “ScheduledTasks” <https://learn.microsoft.com/en-us/powershell/module/scheduledtasks/?view=windowsserver2022-ps>

Microsoft, “Register-ScheduledJob” <https://learn.microsoft.com/en-us/powershell/module/psscheduledjob/register-scheduledjob?view=powershell-5.1>

Microsoft, “Update-Module” <https://learn.microsoft.com/en-us/powershell/module/powershellget/update-module?view=powershellget-2.x>

Microsoft, “Windows registry information for advanced users” <https://learn.microsoft.com/en-us/troubleshoot/windows-server/performance/windows-registry-advanced-users>

Microsoft, “11. Módulos” <https://learn.microsoft.com/es-es/powershell/scripting/lang-spec/chapter-11?view=powershell-7.3>

Microsoft, “2. Lexical Structure” <https://learn.microsoft.com/en-us/powershell/scripting/lang-spec/chapter-02?view=powershell-7.3>

Mike F Robbins, “PowerShell 101”

Modesto San Juan, “[PowerShell] Variables automáticas” <http://www.modestosanjuan.com/powershell-variables-automaticas/>

PowerShell By Example, “Functions” <https://powershellbyexample.dev/post/functions/>

Robert Sheldon, “PowerShell 101”

Rocío GR, “Qué es Windows Powershell y cómo puedes sacarle partido”. AdslZone, <https://www.adslzone.net/esenciales/preguntas/que-es-powershell/>

Sobrebits, “Introducción a los módulos de PowerShell” <https://sobrebits.com/introduccion-a-los-modulos-de-powershell/>

Sobrebits, “3 formas distintas de crear objetos en PowerShell” <https://sobrebits.com/3-formas-distintas-de-crear-objetos-en-powershell/>

Soka, “Administración de usuarios y grupos locales con Powershell” <https://soka.gitlab.io/blog/post/2020-01-04-admin-usuarios-y-grupos-locales-windows-ps/>

Somebooks, “Capítulo 3: Variables PowerShell” <http://somebooks.es/capitulo-3-variables-powershell/2/>

Wikipedia contributors. “PowerShell”. En *Wikipedia, The Free Encyclopedia*. Último acceso 4 abr 2023 a las 02:21, desde <https://es.wikipedia.org/wiki/PowerShell>

Windows PowerShell - Administrar puestos cliente Windows, “Las distintas versiones de Windows PowerShell” <https://www.ediciones-eni.com/open/mediabook.aspx?idR=bfb063fa0d5be7a81d94d54272aca25a>

Windows PowerShell - Los fundamentos del lenguaje, “Los módulos” <https://www.ediciones-eni.com/open/mediabook.aspx?idR=60a7ad80c93135ab2cd424bb4af5c2c0>

Windows PowerShell – Los fundamentos del lenguaje, “Tipos de datos” <https://www.ediciones-eni.com/open/mediabook.aspx?idR=d528d26834ce7caaad4452136296afbb>

Windows PowerShell – Los fundamentos del lenguaje, “1. Estructura de una función” <https://www.ediciones-eni.com/open/mediabook.aspx?idR=1f95297d73bc98c8f6440405251e1ca1>

W3schools.io, “Install vim editor on Window.” <https://www.w3schools.io/editor/vim-install/>