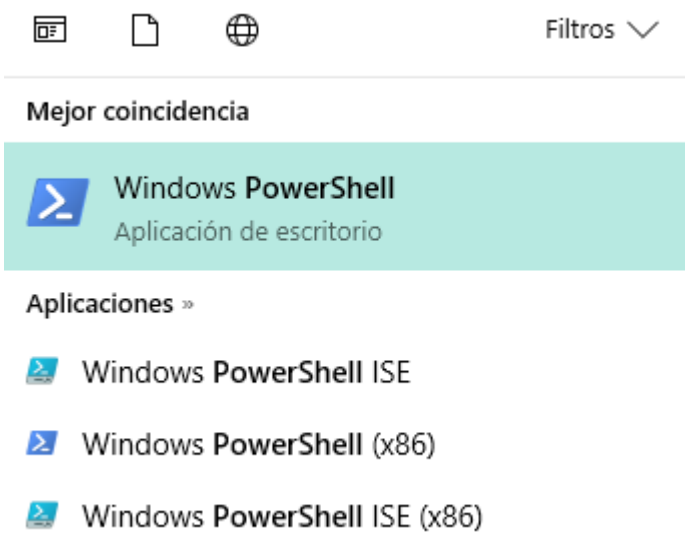
¿Qué és?

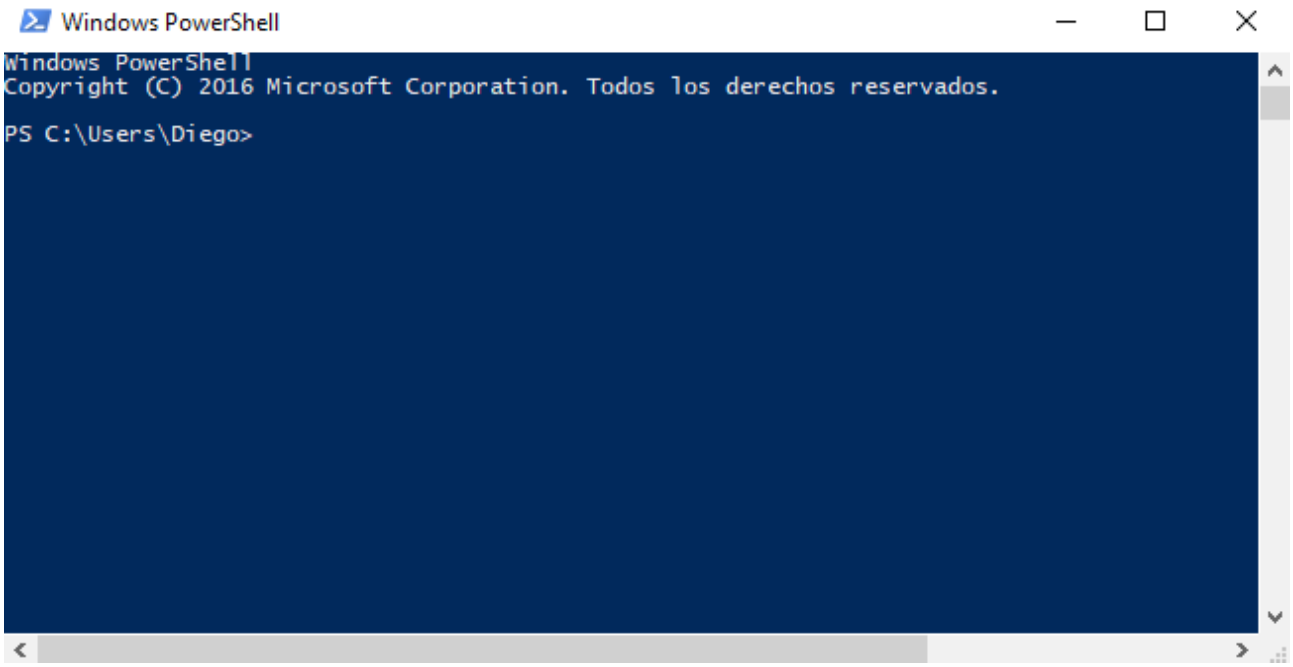
Windows PowerShell es una interfaz de consola (CLI) con posibilidad de escritura y unión de comandos por medio de instrucciones (scripts en inglés). Es mucho más rica e interactiva que sus predecesores, desde DOS hasta Windows 7. Esta interfaz de consola está diseñada para su uso por parte de administradores de sistemas, con el propósito de automatizar tareas o realizarlas de forma más controlada. Originalmente denominada como MONAD en 2003, su nombre oficial cambió al actual cuando fue lanzada al público el 25 de abril de 2006. El 15 de agosto de 2016, Microsoft abrió el código de PowerShell en GitHub.

¿Cómo usarlo?

Para utilizar PowerShell, en Windows 10, se puede realizar una búsqueda con Cortana escribiendo “powershell”. Tras esto, se puede elegir la ventana de comandos tradicional “Windows PowerShell”, o bien, utilizar la versión ISE (Integrated Scripting Environment) con un entorno de ventana, para desarrollo más avanzado. Nosotros como realizar una toma de contacto con powershell eligiremos el modo texto, es decir, el resaltado en la siguiente imagen,



Tras pulsar en dicha búsqueda nos aparecerá en la pantalla esta ventana,



Los cmdlets y su formato

Windows PowerShell utiliza un sistema de nombres con la estructura "verbo-sustantivo" para reducr la memorización de comandos: el nombre de cada cmdlet consta de un verbo estándar y un sustantivo concreto unidos por un guión. Los verbos de Windows PowerShell no siempre están en inglés, pero expresan acciones concretas en Windows PowerShell. Los sustantivos son muy parecidos a los de cualquier idioma, ya que describen tipos de objetos concretos que son importantes para la administración del sistema. Resulta muy fácil entender cómo estos nombres que constan de dos partes reducen el esfuerzo de aprendizaje si observamos varios ejemplos de verbos y sustantivos.
Los sustantivos están menos limitados, pero deben describir siempre a qué se aplica un comando. Windows PowerShell incluye comandos como Get-Process, Stop-Process, Get-Service y Stop-Service.

Puede obtener una lista de todos los comandos que incluyen un verbo concreto con el parámetro -Verb de Get-Command (trataremos Get-Command en profundidad en la siguiente sección). Por ejemplo, para ver todos los cmdlets que utilizan el verbo Get, escriba:

```
PS> Get-Command -Verb Get
CommandType     Name                                Definition
-----------     ----                                ----------
Cmdlet          Get-Acl                             Get-Acl [[-Path] <String[]>]...
Cmdlet          Get-Alias                           Get-Alias [[-Name] <String[]...
Cmdlet          Get-AuthenticodeSignature           Get-AuthenticodeSignature [-...
Cmdlet          Get-ChildItem                       Get-ChildItem [[-Path] <Stri...
...
```

Powershell ofrece un sistema de ayuda para saber qué hacen los cmdlets, mediante el cmdlet “Get-Help <cmdlet>”. A continuación se muestra la ayuda del comando que lista el contenido del directorio actual,

```
PS> Get-Help Get-ChildItem
```

Un aspecto a tener en cuenta, es que la primera vez que vayamos a usar Powershell hemos de actualizar la información de ayuda. Para ello, hemos de ejecutar el siguiente cmdlet, habiendo ejecutado como Administrador la consola de Powershell,

```
PS> Update-Help
```

Los cmdlets utilizan parámetros estándar

Los nombres de parámetros se utilizan siempre con un guión (-) como prefijo, para que Windows PowerShell los identifique claramente como parámetros. En el ejemplo de **Get-Command -Name Clear-Host**, el nombre del parámetro es **Name**, pero se escribe como **-Name**.

Windows PowerShell incluye varios parámetros conocidos como los parámetroscomunes. Dado que se controlan mediante el motor de Windows PowerShell, estosparámetros se comportan de la misma manera siempre que un cmdlet los implementa. Los parámetros comunes son **WhatIf**, **Confirm**, **Verbose**, **Debug**, **Warn**, **ErrorAction**, **ErrorVariable**, **OutVariable** y **OutBuffer**.

Los elementos que definen un comando son,

Completar nombres automáticamente

Los shells de línea de comandos a menudo proporcionan una manera de completar automáticamente los nombres de archivos largos o comandos, lo que agiliza la escritura de comandos y proporciona sugerencias. Windows PowerShell permite completar nombres de archivos y cmdlets con la tecla **Tabulador**.

Puede completar varias veces con el tabulador en una misma línea. Por ejemplo, para completar el nombre del cmdlet **Get-Content** con el tabulador, escriba:

```
PS> Get-Con<Tabulador>
```

Cuando presione la tecla Tabulador, el nombre del comando se ampliará hasta:

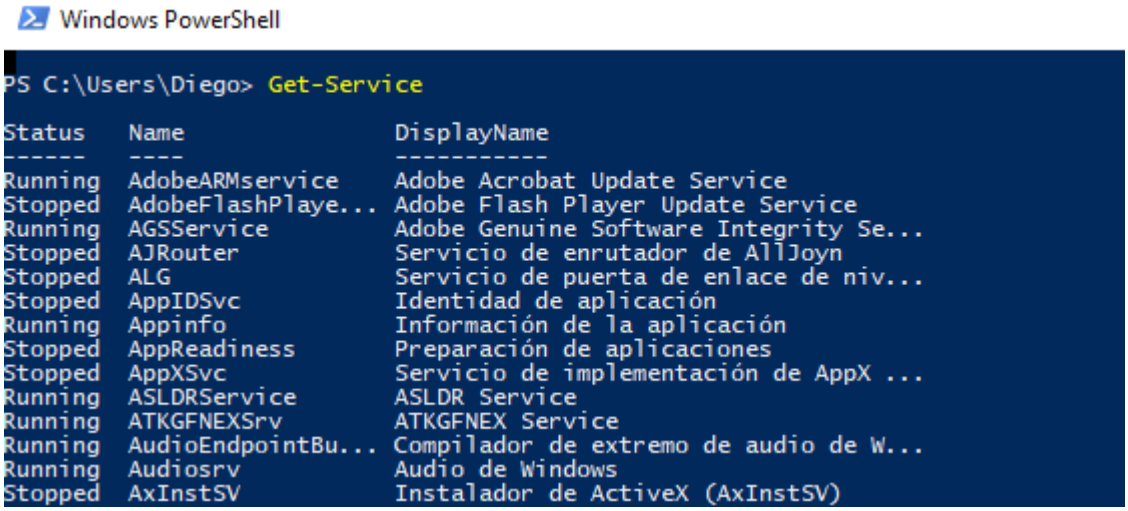
```
PS> Get-Content
```

Canalización de objetos

Las canalizaciones actúan como una serie de tubos conectados. Los elementos que se desplazan por la canalización pasan a través de cada tubo. Para crear una canalización en Windows PowerShell, se conectan comandos con el operador de canalización "|" y el resultado de cada comando se utiliza como entrada del siguiente.

Vamos a ver un ejemplo paso a paso.

```
PS> Get-Service
```



Esto nos da como resultado el conjunto formado por todos los servicios Windows de la máquina. Lo que vemos por pantalla es la representación en modo texto de esos objetos.

Ahora vamos a filtrar esos objetos, seleccionando solo aquellos cuyo estado sea igual a “running”,

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

Windows PowerShell

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

Status   Name               DisplayName
------   ----               -----------
Running  AdobeARMservice    Adobe Acrobat Update Service
Running  AGSService         Adobe Genuine Software Integrity Se...
Running  Appinfo            Información de la aplicación
Running  ASLDRService       ASLDR Service
Running  ATKGFNEXSrv        ATKGFNEX Service
Running  AudioEndpointBu... Compilador de extremo de audio de W...
Running  Audiosrv           Audio de Windows
Running  BFE                Motor de filtrado de base
Running  BITS               Servicio de transferencia inteligen...
Running  BrokerInfrastru... Servicio de infraestructura de tare...
Running  bthserv            Servicio de compatibilidad con Blue...
Running  CDPSvc             Servicio de plataforma de dispositi...
Running  CDPUserSvc_171a... Servicio de usuario de plataforma d...
Running  ClickToRunSvc      Servicio Hacer clic y ejecutar de M...
Running  CoreMessagingRe... CoreMessaging
Running  CryptSvc           Servicios de cifrado
```

El siguiente paso es que ese resultado que hemos obtenido lo queremos ordenado de otra forma. Entonces ejecutamos,

```
PS> Get-Service | Where-Object {$_.Status -eq "Running"} | Sort-Object -Property DisplayName
```

Windows PowerShell

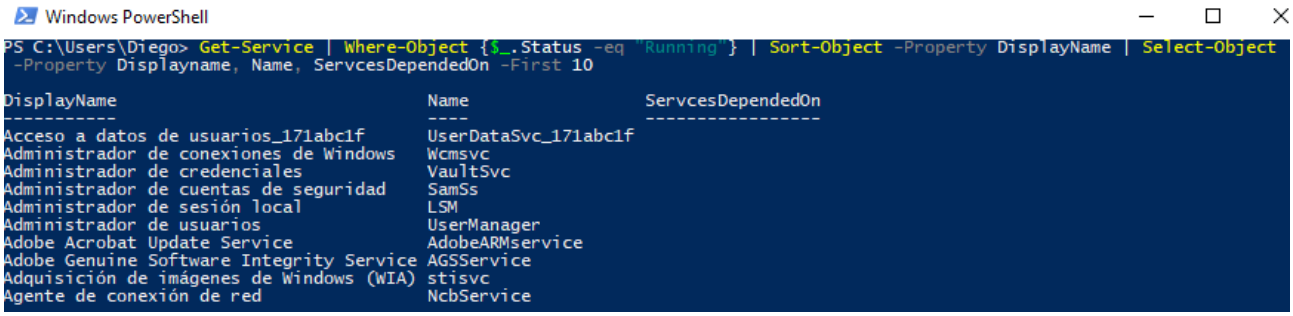
```
PS C:\Users\Diego>
PS C:\Users\Diego> Get-Service | Where-Object {$_.Status -eq "Running"} | Sort-Object -Property DisplayName

Status   Name               DisplayName
------   ----               -----------
Running  UserDataSvc_171... Acceso a datos de usuarios_171abc1f
Running  Wcmsvc             Administrador de conexiones de Windows
Running  VaultSvc           Administrador de credenciales
Running  SamSs              Administrador de cuentas de seguridad
Running  LSM                Administrador de sesión local
Running  UserManager        Administrador de usuarios
Running  AdobeARMservice    Adobe Acrobat Update Service
Running  AGSService         Adobe Genuine Software Integrity Se...
Running  stisvc             Adquisición de imágenes de Windows ...
```

Ahora lo que hemos hecho es pasar el resultado del conjunto de objetos al comando Sort-Object que los ordena alfabéticamente basándose en la propiedad Displayname. Vemos que la salida del comando no ha cambiado mucho.

Una vez más, lo que vamos a hacer es limitar el ´numero d líneas que nos selecciona a 10 únicamente,

```
PS> Get-Service | Where-Object {$_.Status -eq "Running"} | Sort-Object -Property DisplayName |
Select-Object -Property Displayname, Name, ServcesDependedOn -First 10
```



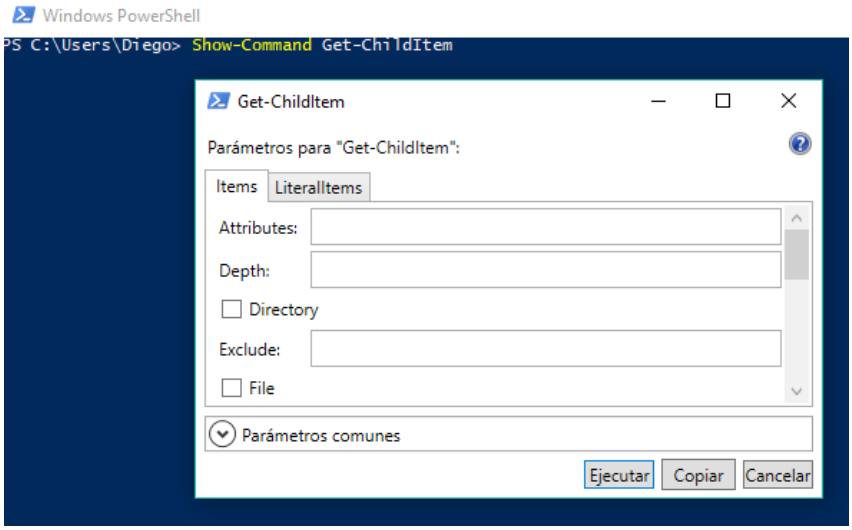
Por último, en lugar de sacar el resultado por pantalla lo vamos a exportar a un archivo en formato CSV.

```
PS> Get-Service | Where-Object {$_.Status -eq "Running"} | Sort-Object -Property DisplayName | Select-Object -Property Displayname, Name, ServcesDependedOn -First 10 | Export-CSV servicios.csv
```

Opciones de comandos en ventana

Para tener una interfaz gráfica donde nos muestre todos los posibles parámetros de un comando, así como la posibilidad de indicarlo, tenemos el comando

```
PS> Show-Command <nombre de comando>
```



Esta forma de introducir los valores de los parámetros nos permite ignorar los errores tipográficos que podamos cometer escribiendo. Tras especificar los parámetros que queramos, podemos pinchar en “Ejecutar” y obtendremos la sintaxis necesaria de nuestro comando.

Comandos básicos de salida

Por defecto el resultado final de un comando se muestra por pantalla representándolo en modo texto, pero tenemos algunas opciones para cambiar ese comportamiento.

Tenemos los cmdlets con verbo Write y Out:

- Write-Host, Write-Output, Write-Verbose, Write-Error, Write-Warning…
- Out-Default, Out-File, Out-GridView…

Normalmente se suele utilizar Write-Host y Write-Ouput para sacar por pantalla alguna información. Pero hay una gran diferencia, Write-Host escribe directamente en pantalla el resultado, mientras que Write-Output genera como salida un objeto (o varios) que puede ser canalizado a otros comandos.

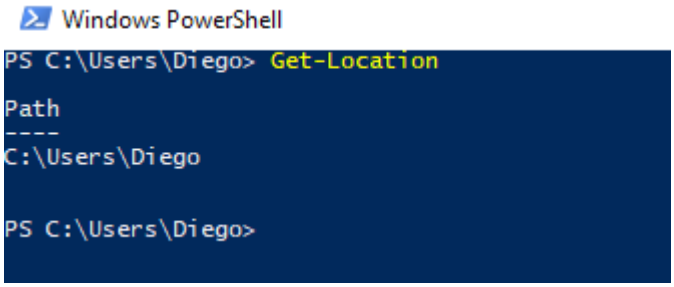
Los cmdlets Out-XXX son muy sencillos ya que redirigen la salida tal y como se indica en el comando:

- Out-Default: redirige a la salida por defecto
- Out-File: redirige a un archivo
- Out-GridView: abre una ventana en forma de tabla
- Out-Printer: lo redirige a la impresora por defecto

Obtener y establecer la ubicación actual

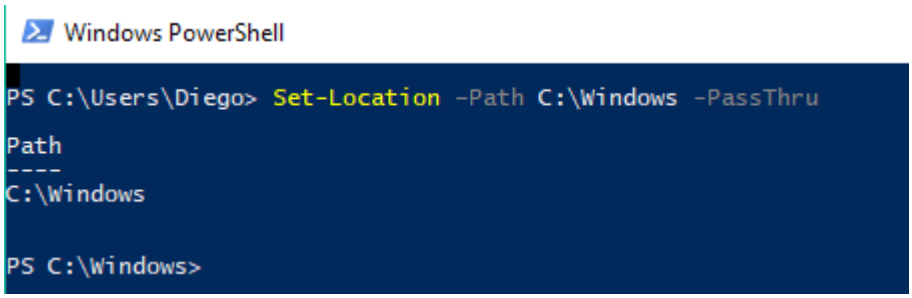
Para determinar la ruta de acceso a la ubicación actual, escriba el comando,

```
PS> Get-Location
```



Así, mismo, para cambiar de ruta y especificar otra de la actual, se utiliza,

```
PS> Set-Location –Path C:\Windows -PassThru
```



El parámetro -PassThru se puede utilizar con muchos comandos Set de WindowsPowerShell para obtener información sobre el resultado en casos en los que no se muestran resultados de manera predeterminada.

Operadores

Vamos a ver ahora los diferentes tipos de operadores de los que PowerShell nos permite utilizar.

Comparación

Permiten comparar varios objetos y su resultado es True o False. Por defecto PowerShell no es CaseSensitve. Los operadores son,

Operador de comparación	Significado	Ejemplo (devuelve el valor True)
-eq	Es igual a	1 -eq 1
-ne	Es distinto de	1 -ne 2
-lt	Es menor que	1 -lt 2
-le	Es menor o igual que	1 -le 2
-gt	Es mayor que	2 -gt 1
-ge	Es mayor o igual que	2 -ge 1
-like	Es como (comparación de caracteres comodín para texto)	“Negu” -like “N*”
-notlike	No es como (comparación de caracteres comodín para texto)	“Negu” -notlike “P*”
-contains	Contiene	1,2,3 -contains 1
-notcontains	No contiene	1,2,3 -notcontains 4

Aritméticos

Operador	Significado	Ejemplo
*	Multiplicación	$a * 5
/	Division	$a / 5
+	Suma	$a + 5
–	Menos	$a – 5
%	Resto	$a % 5

Lógicos

Operador	Descripción	Ejemplo (devuelve el valor True)
-And	Todas las partes de la expresión tienen que ser True	(5 -gt 1) -And (5 -lt 10)
-Or	Alguna de las partes de la expresión tiene que ser True	(5 -gt 1) -Or (5 -lt 1)
-Xor	Exclusión lógica. Es True cuando una parte es True y otra False	(5 -gt 1) -Xor (5 -lt 1)
-Not (!)	Negación	-Not (5 -lt 1)

Asignación

Permiten asignar a una variable el resultado de una operación.

Operador	Significado	Ejemplo
=	Asigna un valor	$a = 5
+=	Suma el valor indicado al ya existente	$a += 5
-=	Resta el valor indicado al ya existente	$a -= 5
*=	Multiplica el valor indicado al ya existente	$a *= 5
/=	Divide por el valor indicado el ya existnte	$a /= 5
++	Incremente el valor en 1	$a++
—	Decrementa el valor en 1	$a–

De tipo

Operador	Significado	Ejemplo
Is	Comprueba si un objeto es de un cierto tipo de objeto	$a -is [int]
IsNot	Comprueba si un objeto no es de un cierto tipo de objeto	$a -isnot [int]
As	Fuerza a tratar un objeto como si fuese un tipo concreto de objeto	$a -as [string]

Otros

Operador	Significado	Ejemplo
&	Permiten ejecutar un texto como si fuese un comando	$a="notepad" &$a
..	Especifica un rango de números	1..5
KB, xMB, xGB	Permite especificar unidades en Bytes, KiloBytes, MegaBytes...	5MB

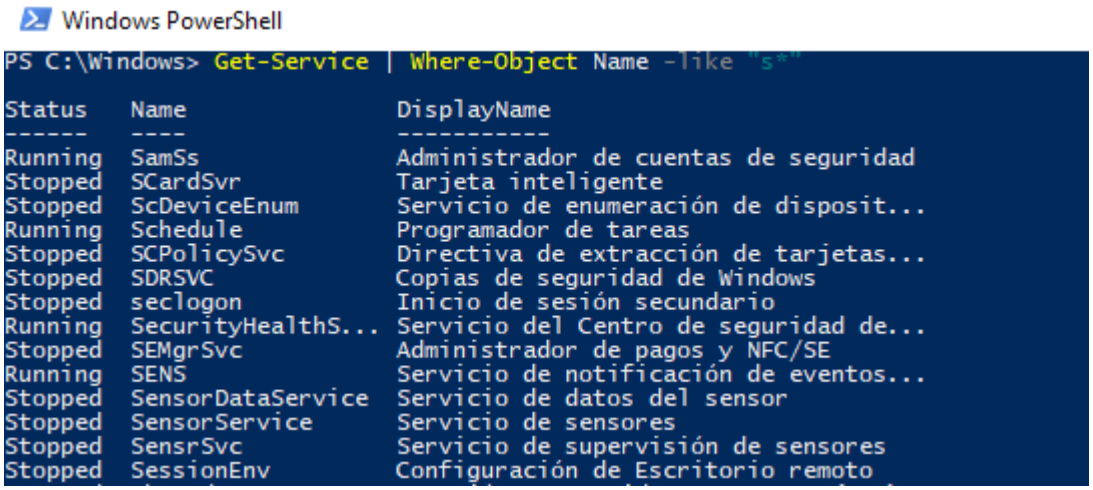
Comandos básicos

Where-Object

Es el principal comando que utilizamos para filtrar los resultados obtenidos por otro cmdlet. Básicamente lo que se hace es comprobar alguna de las propiedades del objeto de entrada y redirigirlo a la salida sólo si cumple la condición indicada.

Veamos un ejemplo. Si queremos obtener los servicios cuyo nombre empieza por “s” podemos hacer los siguiente:

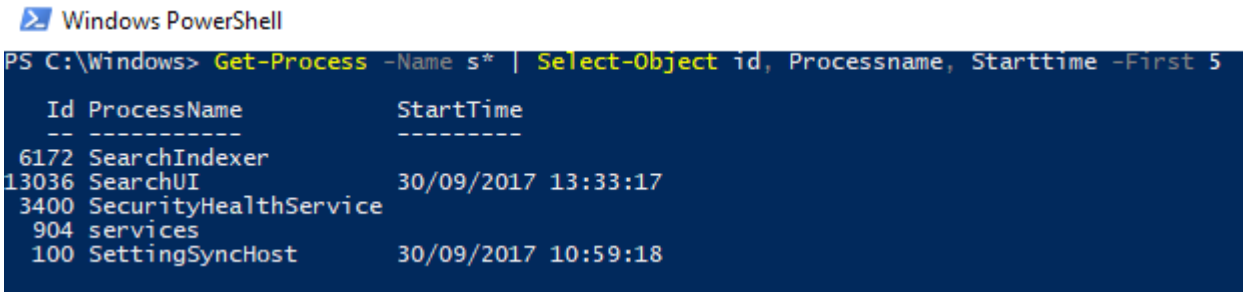
```
PS> Get-Service | Where-Object Name -like "s*"
```



Select-Object

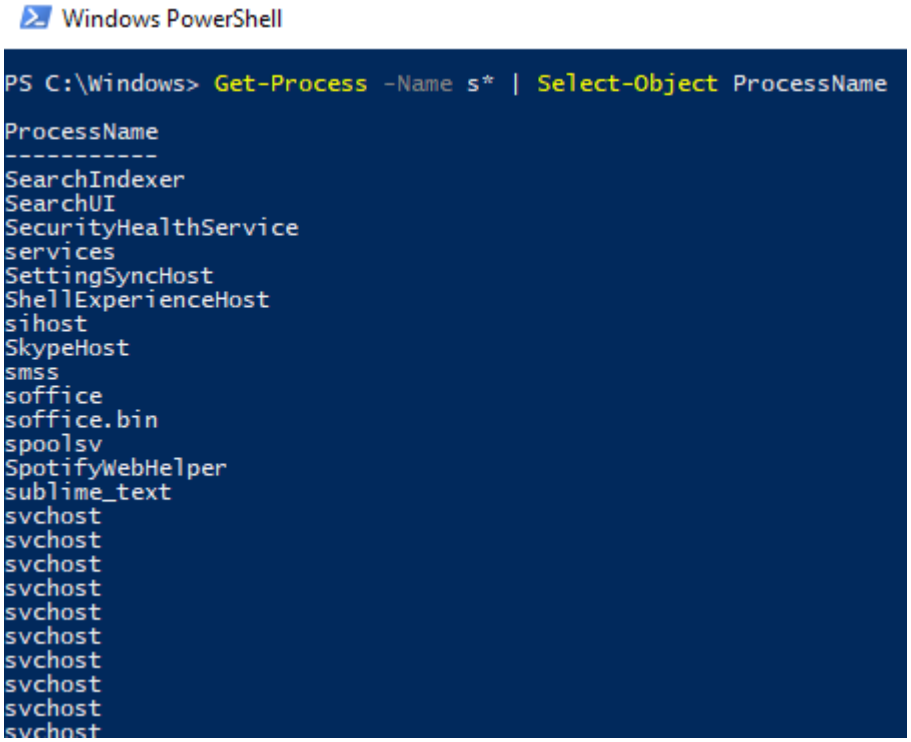
Con este cmdlet lo que podemos es seleccionar los campos del objeto u objetos del resultado final que queremos obtener, de forma que no tengamos el objeto con todas sus propiedades sino solo las que nosotros necesitamos. También nos permite obtener un número concreto de objetos, por ejemplo, los 10 primeros de un resultado.

```
PS> Get-Process -Name s* | Select-Object id, Processname, Starttime -First 5
```



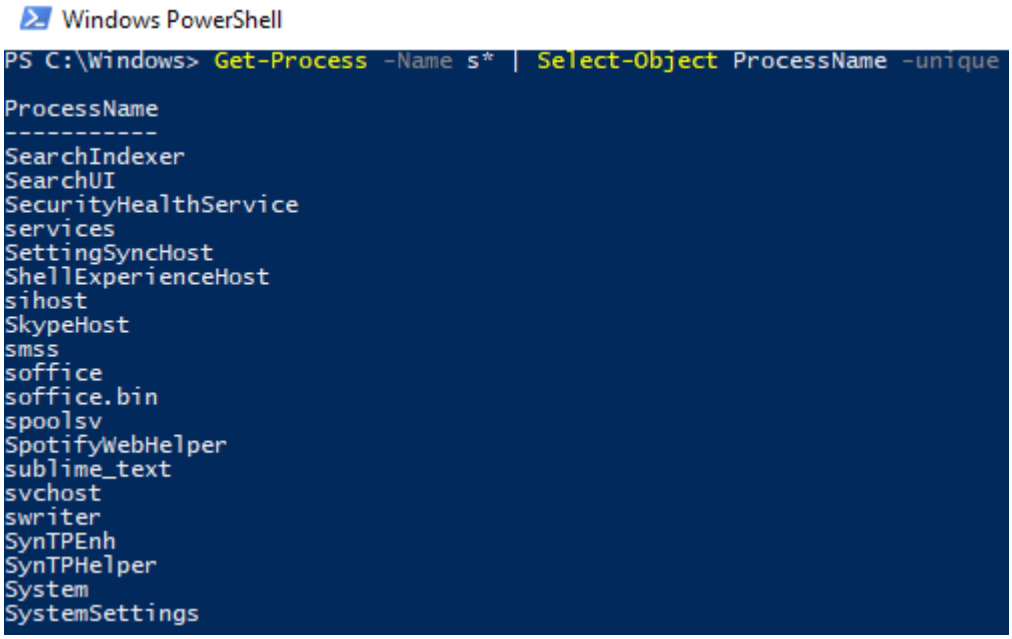
Podemos eliminar resultados repetidos con el parámetro **-Unique**. Por ejemplo,

```
PS> Get-Process -Name s* | Select-Object ProcessName
```



Y ahora con el parámetro unique,

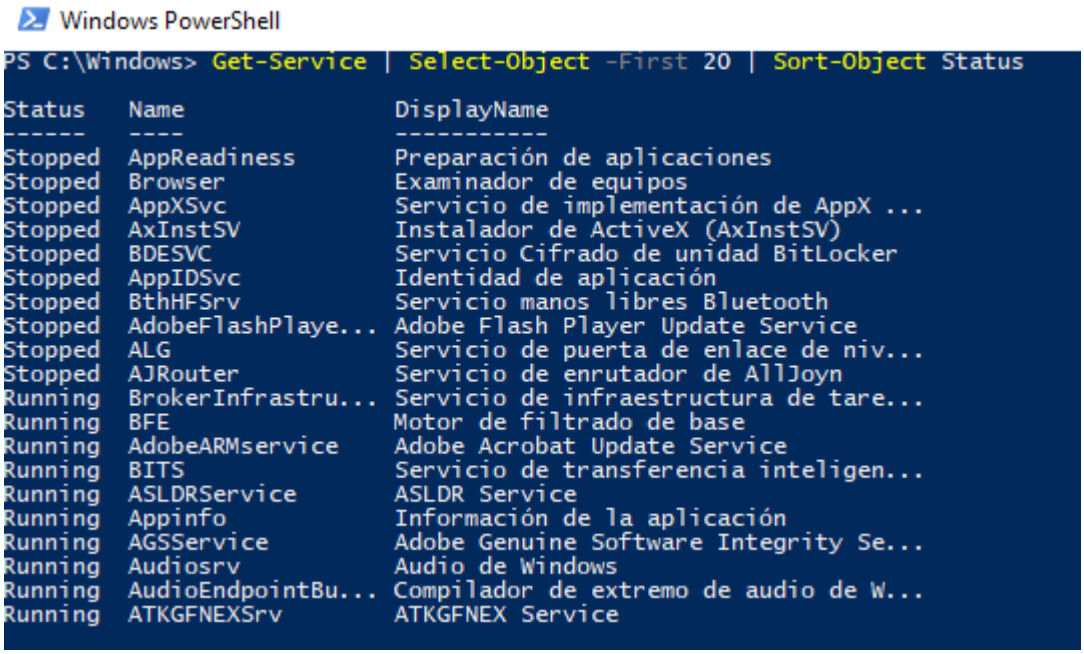
```
PS> Get-Process -Name s* | Select-Object ProcessName -unique
```



Sort-Object

Nos permite ordenar un conjunto de objetos según la propiedad indicada. Por defecto los ordena de forma ascendente tanto alfabética como numéricamente. Podemos indicar varias propiedades a partir de las cuales ordenar los objetos,

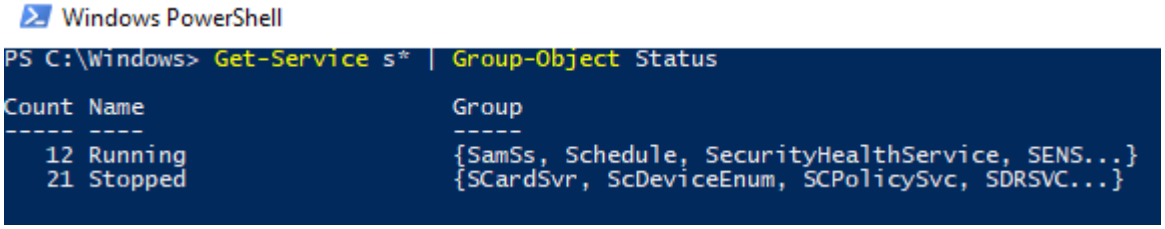
```
PS> Get-Service | Select-Object -First 20 | Sort-Object Status
```



Group-Object

Nos permite organizar y agrupar los objetos basándonos en una propiedad.

```
PS> Get-Service s* | Group-Object Status
```

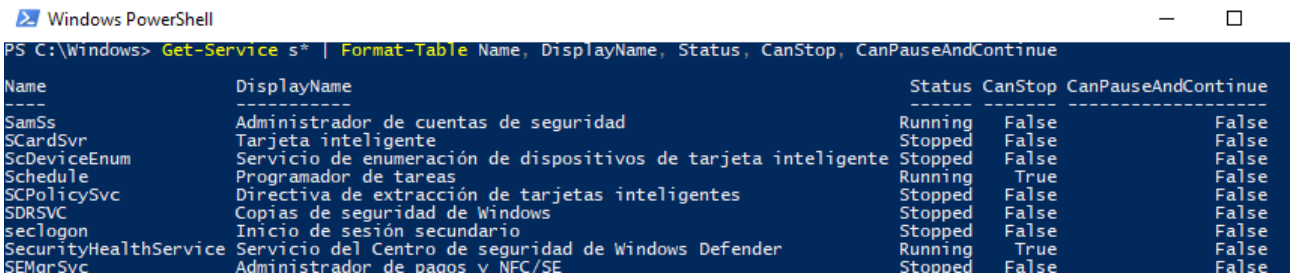


Formato de salida

Format-table

Muestra el resultado en formato de tabla. Dependiendo de la cantidad de información a mostrar puede que oculte parte de ella.

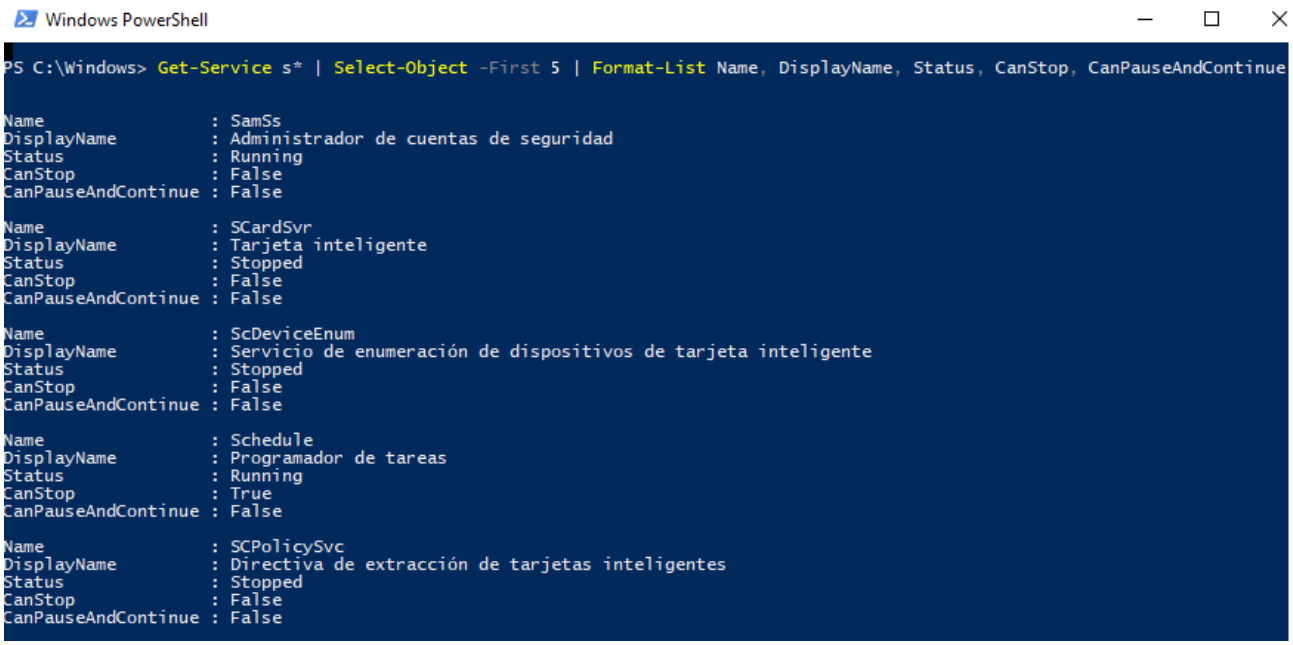
```
PS> Get-Service s* | Format-Table Name, DisplayName, Status, CanStop, CanPauseAndContinue
```



Format-list

En este caso muestra la información en formato lista.

```
PS> Get-Service s* | Sort-Object -First 5 | Format-List Name, DisplayName, Status, CanStop, CanPauseAndContinue
```



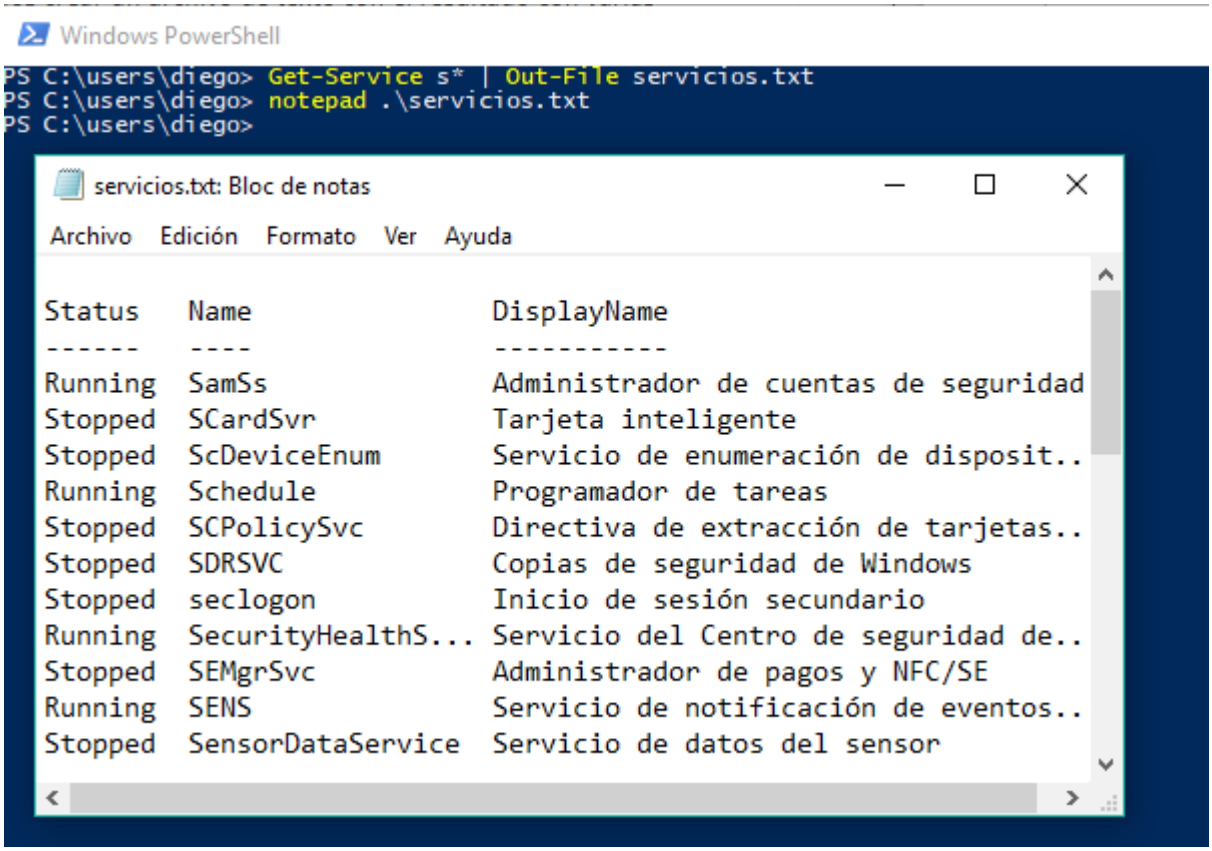
Importar y Exportar datos

Archivos de texto

Es la forma más sencilla de importar y exportar datos, utilizando archivos de texto planos. Podemos crear un archivo de texto con el resultado con varias opciones:

- Out-File

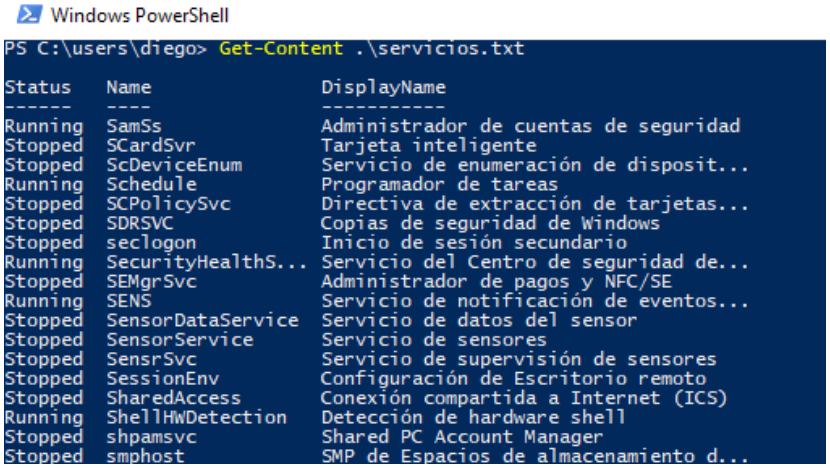
```
PS> Get-Service s* | Out-File servicios.txt
PS> notepad servicios.txt
```



- Redirigiendo la salida con “>” (o “>>” si queremos añadir al final del fichero)

Para obtener la información de archivos de texto, utilizamos el cmdlet Get-Content.

```
PS> Get-Content .\servicios.txt
```



CSV (Comman Separated value)

Es uno de los formatos más extendidos y sencillos a la hora de trabajar con archivos de texto. Utiliza la coma para separar los campos (también se pueden utilizar otros caracteres)

Tenemos los siguientes comandos:

Import-CSV: con powershell podemos importar cualquier archivo CSV. Por defecto el delimitador es la coma, pero se puede indicar otro.La cabecera del archivo pasan a ser los nombres de las propiedades pero también se pueden indicar otros. Todos los valores son importados como cadenas de texto salvo, que PowerShell identifique el contenido como uno de los objetos ya definidos.

Veamos un ejemplo sencillo.

Tenemos un archivo de texto en formato CSV con varios nombres de servidores, direcciones IP y sistema operativo

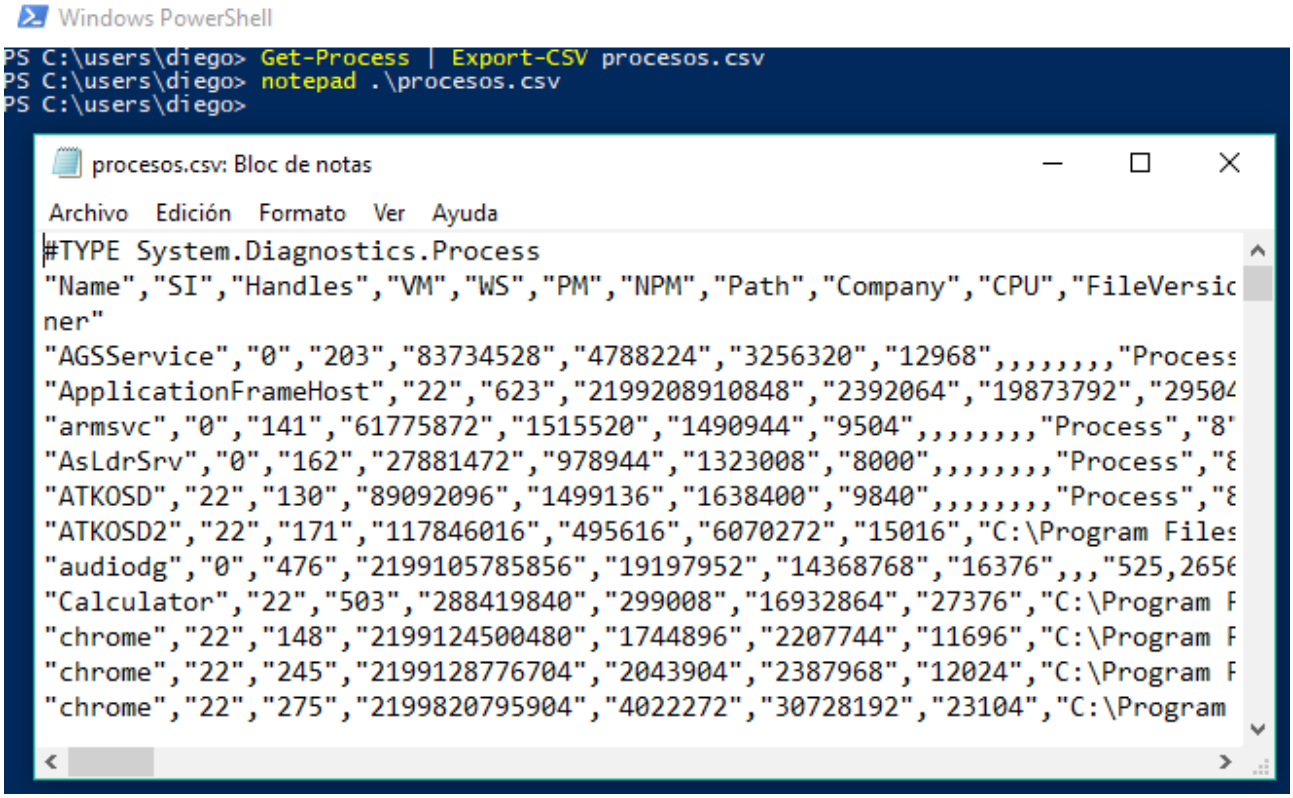
Si queremos obtener esa información en una variable es tan sencillo como utilizar Import-CSV

```
PS> $servidores = Import-Csv servidores.csv
```

Export-CSV: en este caso permite crear archivos CSV a partir de un objeto o conjunto de objetos. Hay que tener en cuenta que las propiedades del objeto se convierten en la cabecera del archivo CSV y que los arrays y objetos no se exportan, por lo que puede que no obtengamos la salida deseada.

También indicar que como primera linea del archivo CSV se añade el tipo de objeto que se ha exportado como comentario. Se puede quitar con el parámetro **NoTypeInformation**

```
PS> Get-Process | Export-CSV procesos.csv
PS> notepad .\procesos.csv
```

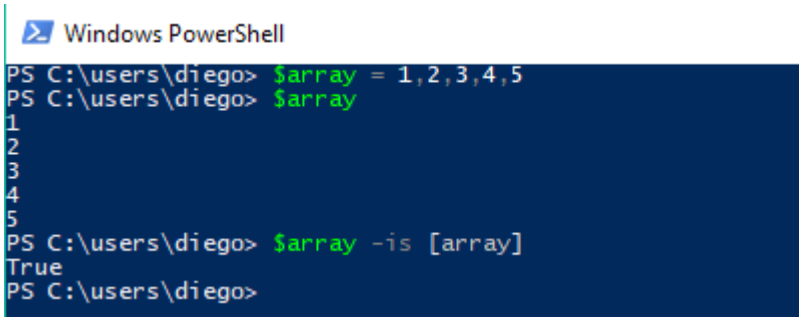


Arrays y Hashtables

Arrays

Un array es una colección de objetos. Los objetos de la colección no tienen por que ser del mismo tipo.

PowerShell trata cualquier lista separada por comas como una tabla.

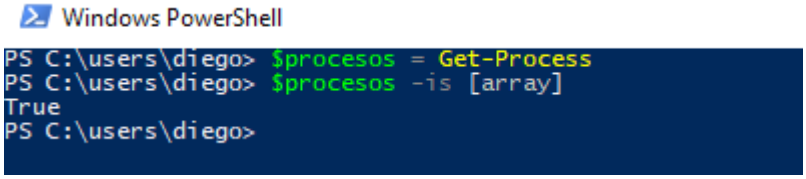


El siguiente comando lo que hace es comprobar si una variable contiene una tabla,

```
PS> $array -is [array]
```

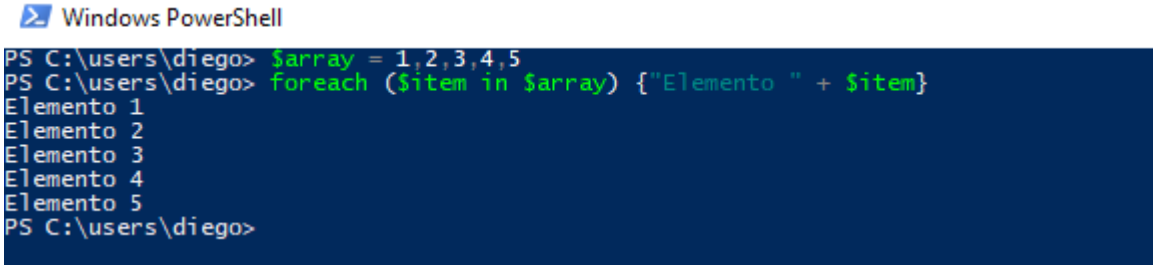
Podemos ver que cualquier cmdlet puede devolver un array como resultado,

```
PS> $procesos = Get-Process
PS> $procesos -is [array]
```



Parar recorrer cada uno de los elementos tenemos el cmdlet foreach

```
PS> $array = 1,2,3,4,5
PS> foreach ($item in $array) {"Elemento " + $item}
```

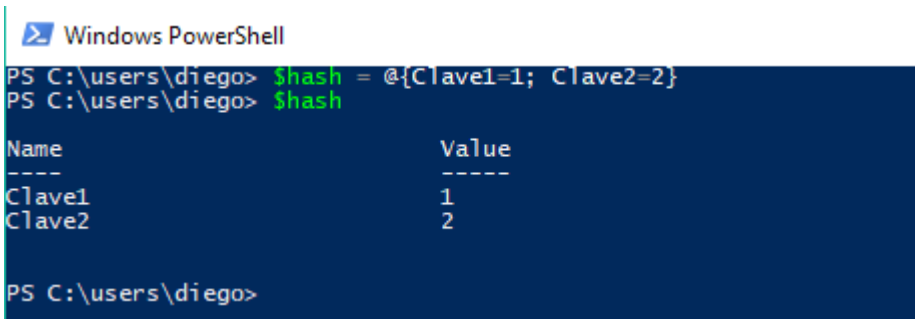


Hashtable

Es una colección de objetos que relaciona claves con valores de forma que podamos acceder al valor a través de la clave que tiene asociada.

Podemos crear una HashTable de la siguiente manera,

```
PS> $hash = @{Clave1=1; Clave2=2}
PS> $hash
```



Podemos acceder a cada uno de los elementos de la tabla indicando la clave:

```
PS> $hash.Clave1
```

Para añadir un nuevo elemento utilizamos el método Add(Clave, Valor) La clave tiene que ser única, por lo que antes de añadir el elemento es recomendable comprobar si existe la clave.

```
PS> $hash.Add("Clave4"="Valor4")
```

Para eliminar un elemento utilizamos el método Remove(Clave),

```
PS> $hash.Remove("Clave1")
```

Scripts y funciones

Cualquier comando que podemos ejecutar en una consola PowerShell puede ser escrito y guardado en un archivo **.ps1** para poder ser ejecutado de forma repetida.
Los scripts pueden aceptar parámetros de entrada para que serán procesados en el contenido del script.

Los objetos devueltos por la ejecución de un cmdlet pueden ser guardados en variables. Una variable es un texto precedido por el simbolo $. El siguiente código,

```
$contenido = get-childitem

write-host "Escribimos el contenido del objecto"
$contenido
```

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        08/10/2017      2:41                .VirtualBox
d-----        06/10/2017     15:58                .vscode
d-r---        08/10/2017      2:02                Contacts
d-r---        08/10/2017      2:44                Desktop
d-r---        08/10/2017      2:02                Documents
d-r---        08/10/2017      2:33                Downloads
d-r---        08/10/2017      2:02                Favorites
d-r---        08/10/2017      2:02                Links
d-r---        08/10/2017      2:02                Music
d-r---        08/10/2017      0:00                OneDrive
d-r---        08/10/2017      2:28                Pictures
d-r---        08/10/2017      2:02                Saved Games
d-r---        08/10/2017      2:02                Searches
d-r---        08/10/2017      2:02                Videos
-a----        10/10/2017      0:21             93 ejemVar.ps1
-a----        08/10/2017     18:38            139 ejer7.ps1
```

Tiene una variable llamada $contenido, donde se guarda la información devuelta por el cmdlet “Get-ChildItem”, y como dicha información se trata de un objecto. Se pueden acceder a los campos de dicha información como tal. Por ejemplo, si quisiéramos seleccionar solamente determinadas columnas del objeto, podríamos hacer esto,

```
$contenido = get-childitem

write-host "Escribimos el contenido del objecto"
$contenido | select-object -Property Mode, Name
```

```
Mode   Name
----   ----
d----- .VirtualBox
d----- .vscode
d-r--- Contacts
d-r--- Desktop
d-r--- Documents
d-r--- Downloads
d-r--- Favorites
d-r--- Links
d-r--- Music
d-r--- OneDrive
d-r--- Pictures
d-r--- Saved Games
d-r--- Searches
d-r--- Videos
-a---- ejemVar.ps1
-a---- ejer7.ps1
```

Comentarios

Los comentarios simples en PowerShell se realizan mediante un # al principio de la línea.

Estructura de los scripts

Un fichero de script de powershell sigue la siguiente estructura,

```
2
3  Param (<Parametro>,<Parametro>,...)
4
5  Comandos PowerShell
```

Las funciones son tareas que queremos utilizar de forma repetitiva. Están definidas en scripts y pueden utilizar parámetros de entrada. Las funciones que se crean necesitan ser “cargadas” en PowerShell para poder ser utilizadas.

Estructura de las funciones:

```
1  Function <NOMBRE> {
2    Param (<Parametro>,<Parametro>,...)
3
4    Comandos PowerShell
5
6  }
```

Script Básico

A continuación podemos ver un ejemplo básico, que tomando como entrada una ruta, una extensión de archivo y una fecha, muestra los archivos de este tipo modificados desde la fecha indicada en la ruta.

```
#Obtiene los archivos de un tipo, modificados a partir de una fecha dada
#Parametros de entrada
# -Ruta de la carpeta
# -Fecha

Param (
        [string] $ruta = ".",
        [string] $extension = "ps1",
        [datetime] $fecha
)

#Controlamos la entrada de parametros
if (!$fecha) {
        #Establecemos una fecha por defecto
        $fecha = (Get-Date).AddDays(-1)
}

Get-ChildItem Path $ruta -Include *.$extension -Recurse -File | Where-Object {$_.LastWriteTime -ge $fecha}
```

Opciones de los parámetros de entrada

Para definir los parámetros de entrada tenemos varias opciones, entre ellas:

- Establecer un parámetro como obligatorio: lo indicamos con [Parameter(Mandatory=$true)] antes del tipo de parámetro.
- Establecer el tipo de objeto de entrada: lo indicamos con el nombre del tipo de objeto entre corchetes, por ejemplo [string]
- Establecer un tipo booleano, si el parámetro está presente toma el valore $true: lo indicamos con [switch]

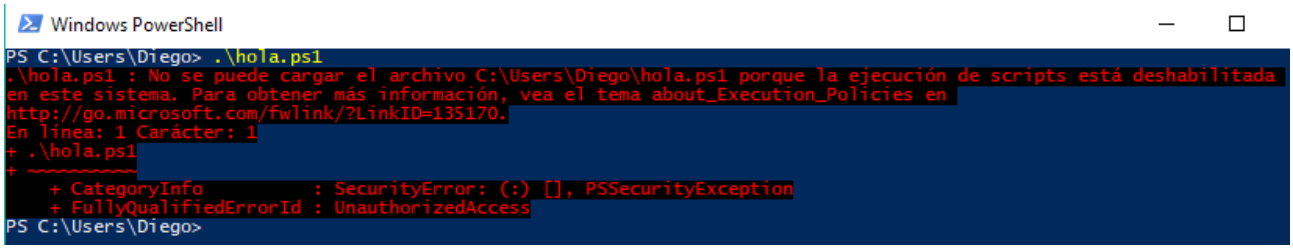
Ejecución de scripts

Tras salvar nuestro primer script. Como por ejemplo podría ser el siguiente,

```
Write-Host “Hola munfo!!”
```

Cuando realizamos la ejecución del script obtenemos un error,

```
PS> .\hola.ps1
```

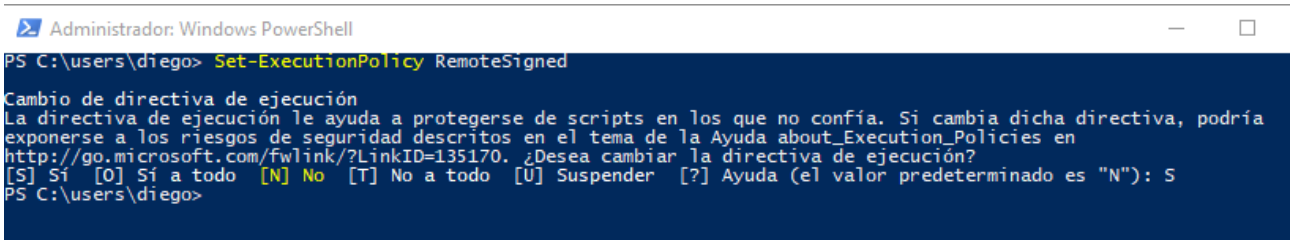


El error mostrado es debido a que por defecto los scripts de powershell están restringidos para evitar ataques maliciosos. Existen varios niveles de seguridad dependiendo del grado de protección que necesitemos:

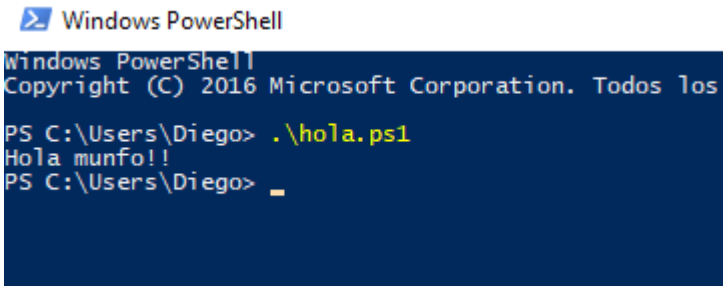
- **Restricted**: Los scripts no funcionan. (Configuación por defecto)
- **RemoteSigned**: Los scripts locales funcionarán. Pero los scripts que fueran creados en otra máquina no funcionarán, a no ser que sean firmados.
- **AllSigned**: Todos los scripts deben ser firmados.
- **Unrestricted**: Todos los scripts funcionan.

Para poder realizar nuestras pruebas, vamos a establecer el nivel de seguridad en **RemoteSigned**, y volver a realizar la ejecución:

```
PS> Set-ExecutionPolicy RemoteSigned
```




```
PS> .\hola.ps1
```



Estructuras condicionales

Una estructura condicional en powershell se realiza de manera muy similar a otros lenguajes de programación.

```
If (<condición 1>) {
    <código que se ejecuta si se cumple la condición 1>
} elseif (<condición 2>) {
    <código que se ejecuta si se cumple la condición 2>
} else {
    <código que se ejecuta si no se cumple niguna la condición>
}
```

Los operadores de comparación son:

- Igualdad “-eq”
- No igual “-ne”
- Mayor que “-gt”
- Menor que “-lt”
- Mayor que o igual a “-ge”
- Menor que o igual a “-le”

Bucles

En cuanto a los bucles disponibles en powershell, tenemos también los mayormente utilizados en otros lenguajes.

Ejemplo del bucle Foreach

```
# Example of PowerShell Loop
$NumArray = (1,2,3,4,5,6,7,8,9,10,11,12)
Foreach ($Item in $NumArray) {$Item * 13}
```

Ejemplo del bucle While

```
# PowerShell While Loop
$i =8
While ($i -le 96)
{
    $i
    $i +=8
}
```

Ejemplo del bucle Do..Until

```
$i = 7
do {
    $i; $i +=7
}
until ($i -gt 85)
```

Ejemplo del bucle Do..While

```
$i = 7
do {
    $i; $i +=7
}
while ($i -lt 85)
```