PowerShell 2.0

El lenguaje de administración de Windows

Ramiro Encinas Alarza - 2012

ramiro.encinas|at|gmail.com
https://sites.google.com/site/ramiroencinas/
http://lacapsulaverde.blogspot.com/

1 Introducción, requisitos e instalación

PowerShell 2.0 (en adelante PS2), es la segunda versión de un lenguaje de programación de tipo *script* creado por Microsoft, orientado a objetos, .NET y a tareas de administración. De momento, sólo funciona en los sistemas operativos Windows que se utilizan actualmente de forma mayoritaria.

Para saber si tu Windows viene con PS2, ve a Inicio, Ejecutar, pon **powershell** y acepta. Si aparece una ventana de línea de comandos cuyo prompt comienza con PS, PowerShell ya está instalado. En este caso, para saber la versión de PowerShell, teclea `$host` y pulsa ENTER.

Si tu Windows no tiene PS2 y quieres que lo tenga, instala Windows Management Framework Core (PS2 es un componente de este paquete de administración). Como requisito debes tener instalado como mínimo .NET Framework 3.5 SP1.

Adicionalmente, aconsejo instalar PowerGui que incorpora **PowerGui Script Editor**, muy útil para el desarrollo de programas con PS2.

2 Lo básico de PS2

Este apartado introduce nociones básicas sobre PS2. Primero hablaremos de los privilegios que utiliza PS2 para ejecutar scripts, y cómo se comporta con los errores.

También veremos lo que es un cmdlet y cómo ejecutar scripts, tipos de datos y el control de flujo mediante condiciones y bucles.

2.1 Privilegios de ejecución de scripts

Ve a Inicio, Ejecutar, escribe **powershell** y acepta. Es recomendable crear un acceso directo de este comando en el Escritorio para tener la shell de PS2 a mano. Si PS2 está bien instalado aparecerá una ventana de línea de comandos parecida al DOS cuyo prompt es algo parecido a:

```
PS C:\Documents and Settings\usuario>
```

Esto es, PS y después la ruta de la ubicación actual. En adelante utilizaremos

```
PS>
```

Para indicar que a continuación viene una línea de comando PS2.

Antes de ejecutar código en PS2 puede que necesites ajustar la política de ejecución de scripts de PS2. Para saber cual es la política de ejecución actual pon:

```
PS>get-executionpolicy
```

y pulsa ENTER.

Para que el usuario actual pueda ejecutar scripts sin restricciones, pon:

```
PS>set-executionpolicy unrestricted -scope currentuser
```

pulsa ENTER y confirma el cambio. Ahora el usuario actual ya puede ejecutar código PS2.

Para que cualquier usuario pueda ejecutar scripts sin restricciones, utiliza lo anterior sin el parámetro `-scope`, pero ten cuidado con los usuarios que puedan acceder a la máquina, ya sea en red o de forma local.

Además, existen tres tipos de restricciones más:

1. El que impide la ejecución de scripts en la máquina pero permite la ejecución en línea de comandos:

```
PS>set-executionpolicy restricted
```

2. El que sólo permite que se puedan ejecutar scripts firmados por alguien de confianza:

```
PS>set-executionpolicy allsigned
```

3. El que impide la ejecución de scripts descargados remotamente sino están firmados por alguien de confianza:

```
PS>set-executionpolicy remotesigned
```

2.2 Comportamiento de errores

También puedes ajustar el modo en que PS2 se comporta cuando se produce un error al ejecutar código o scripts PS2. Para ver el modo actual, pon:

```
PS>$ErrorActionPreference
```

Y devuelve uno de estos cuatro modos:

`Continue` : muestra errores y continúa con la ejecución.
`SilentlyContinue` : no muestra errores y continúa la ejecución.
`Inquire` : al producirse errores pregunta al usuario qué hacer.
`Stop` : detiene la ejecución.

El modo por defecto es `Continue`, y para cambiarlo, por ejemplo al modo `SilentlyContinue`, para que no muestre errores y los ignore, ponemos:

```
PS>$ErrorActionPreference = "SilentlyContinue"
```

2.3 Cmdlets: entrada/salida básica y ejecución de scripts

2.3.1 Cmdlets: entrada/salida básica

PS2 tiene una especie de comandos llamados **cmdlet** que son objetos y hacen de todo. También se utiliza mucho la tubería o redirección "|" para pasar datos de un **cmdlet** a otro (como *nix). Por ejemplo, si quieres saber la fecha/hora actual, pon:

```
PS>get-date
lunes, 07 de febrero de 2011 12:15:11
```

Si quieres ver todas sus propiedades en una lista, redirige su salida a `fl` (`Format-List`):

```
PS>get-date | fl

DisplayHint : DateTime
Date        : 07/02/2011 0:00:00
Day         : 7
DayOfWeek   : Monday
DayOfYear   : 38
```

```
Hour        : 21
Kind        : Local
Millisecond : 62
Minute      : 9
Month       : 2
Second      : 12
Ticks       : 634327097520625000
TimeOfDay   : 21:09:12.0625000
Year        : 2011
DateTime    : lunes, 07 de febrero de 2011 21:09:12
```

Y si quieres ver la salida en formato tabla, redirige la salida a `ft (Format-Table)`

```
PS>get-date | ft
```

```
Display Date      Day DayOfWe DayOfYe    Hour    Kind Millise  Minute   Month
   Hint                   ek      ar                    cond
------- ----      --- ------- -------    ----    ---- -------  ------   -----
...Time 07/0...     7  Monday      38      21   Local     562      20       2
```

Para obtener más funcionalidades de un cmdlet, puedes asignar su salida a una variable (precedida con $). Por ejemplo,

```
PS>$fecha = get-date
```

crea la variable `$fecha` que contiene lo mismo que `get-date`. De hecho, si escribes ahora

```
PS>$fecha
```

y pulsas ENTER, también aparece la fecha.

Puedes ver cualquier propiedad de `$fecha` indicándola después de un punto, por ejemplo, la propiedad año se ve así:

```
PS>$fecha.year
2011
```

También puedes ver todos los miembros (métodos y propiedades) de `$fecha` redirigiendo su salida al cmdlet `get-member` con una tubería:

```
PS>$fecha | get-member
```

```
   TypeName: System.DateTime

Name                MemberType     Definition
----                ----------     ----------
Add                 Method         System.DateTime Add(System.TimeSpan value)
AddDays             Method         System.DateTime AddDays(double value)
AddHours            Method         System.DateTime AddHours(double value)
AddMilliseconds     Method         System.DateTime AddMilliseconds(double
v...
…
```

que es lo mismo que

```
PS>get-date | get-member
```

Con el cmdlet `Write-Host` puedes mostrar por pantalla mensajes y más cosas:

```
PS>write-host hola, ¿que tal?
hola, ¿que tal?

PS>write-host $fecha
lunes, 07 de febrero de 2011 12:15:11
```

El cmdlet `Read-Host` pregunta por teclado e introduce la respuesta en una variable:

```
PS>$nombre = read-host "Introduce tu nombre"
Introduce tu nombre: Ramiro
PS>$nombre
Ramiro
```

En una misma línea puedes ejecutar varias expresiones separadas por punto y coma, así:

```
PS>$edad = read-host "Pon tu edad" ; write-host $edad ; "¡Qué viejo!"
Introduce tu edad: 35
35
¡Qué viejo!
```

Para ver una lista de todos los cmdlets disponibles utiliza

```
PS>get-command
```

Para acceder a la ayuda, utiliza

```
PS>get-help
```

PS2 incorpora muchos cmdlets por defecto, y además también hay disponibles muchos más que se pueden instalar para realizar operaciones en otras plataformas como Exchange, Directorio Activo, VMWare, etc.

2.3.2 Ejecución de scripts

PS2 ejecuta scripts con extensión `.ps1`, de forma que puedes escribir todo el código PS2 en un archivo de texto con la extensión indicada.

Para editar scripts sólo necesitas un editor de texto. Si quieres más funcionalidades como depuración y ver a tiempo real el valor de las variables, puedes utilizar PowerGui (indicado en la introducción de este documento).

Para ejecutar un script de PS2 tienes que indicar su ubicación de forma absoluta. Existen dos formas de hacerlo:

Desde la línea de comandos convencional:

```
powershell .\hola.ps1
```

En este caso, el archivo script `hola.ps1` debe estar ubicado en la carpeta actual. Tenemos que escribir toda la ruta si el script está en otra carpeta que no es la actual:

```
powershell c:\scripts\powershell\pruebas\hola.ps1
```

O desde la línea de comandos de PS2 (estamos en la misma carpeta que el script):

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

E indicando la ruta completa desde la línea de comandos de PS2:

```
PS>c:\scripts\powershell\pruebas\hola.ps1
```

2.4 Tipos de datos: variables, tablas, listas y hashes

En este apartado veremos cómo definir y trabajar con los tipos de datos más utilizados en PS2, desde la variable común que representa un sólo dato u objeto, hasta las distintas colecciones de datos (tablas, listas y hashes)

2.4.1 Variables

Las variables comienzan con $ y podemos meter en ella cualquier casi cualquier cosa, por ejemplo:

```
PS>$fecha = get-date
PS>$numero = 1
PS>$numero_decimal = 1.5
PS>$letra = 'y '
PS>$cadena = 'esto es una cadena'
```

Puedes asignar **el tipo** concreto de variable indicándolo entre corchetes. Estos son algunos:

```
[int]$numero_32bit       = 999999999
[long]$numero_64bit      = 999999999999999999
[string]$cadena          = "cadena en unicode"
[char]$caracter          = "a"
[byte]$numero_8bit       = 255
[bool]$verdad            = $true
[decimal]$numero_decimal = 467.45
```

Para mostrar el valor de una variable, simplemente escríbela y pulsa ENTER:

```
PS>$numero
1

PS>$cadena
esto es una cadena
```

Puedes sumar dos o más números:

```
PS>1 + 1.5
2.5
```

o sumar dos o más variables que contienen números:

```
PS>$numero + $numero_decimal
2.5
```

o incrementar directamente `$numero` en una unidad:

```
PS>$numero++
PS>$numero
2
```

Puedes unir dos o más cadenas:

```
PS>'y ' + 'esto es una cadena'
y esto es una cadena
```

Observa que los espacios en blanco se tienen en cuenta.

Para unir dos o más variables que contienen cadenas:

```
PS>$letra + $cadena
y esto es una cadena
```

PS2 tiene un tipo de variable especial llamada `$_` y después hablaremos de ella en el apartado 2.5.1. que trata sobre condiciones debido a su estrecha relación con éstas.

2.4.2 Tablas

Una tabla es una variable que puede contener cualquier número de elementos de cualquier tipo. El primer elemento tiene la posición 0. Los elementos se indican **separados por comas**:

```
PS>$tabla = $fecha,1,'esto es una cadena',$numero_decimal,$letra

PS>$tabla
domingo, 30 de enero de 2011 23:48:28
1
esto es una cadena
1,5
y
```

Como hemos dicho, los elementos de una tabla se enumeran desde cero, y puedes acceder a cualquier elemento indicando su posición entre corchetes **[ ]**:

```
PS>$tabla[2]
esto es una cadena
```

Puedes ver un rango de elementos indicando la posición de inicio y de fin. Por ejemplo, para ver los elementos de `$tabla` desde la posición 2 a la 4 sería

```
PS>$tabla[2..4]
esto es una cadena
1,5
y
```

La posición del último elemento de una tabla es `-1`:

```
PS>$tabla[-1]
y
```

También puede utilizarse `-1` con este mismo propósito en cualquier otra variable que al igual que las tablas, tengan colecciones de elementos.

Puedes ver el número de elementos de una tabla con el método `.count`

```
PS>$tabla.count
5
```

Puedes modificar el valor de un elemento indicando su posición e igualando al nuevo valor

```
PS>$tabla[-1] = 'esto es otra cadena'
PS>$tabla[-1]
esto es otra cadena
```

2.4.3 Listas

Una lista es un array dinámico en el que se pueden añadir y eliminar elementos fácilmente. Así se define la lista vacía `$lista`

```
PS>$lista = New-Object System.Collections.ArrayList
```

Como puedes ver, es un objeto **ArrayList** de .NET.

Puedes **añadir elementos** de cualquier tipo con el método `.add`

```
PS>$lista.add("cadena")
PS>$lista.add(45)
PS>$lista.add($fecha)
PS>$lista.add("otra cadena")
PS>$lista.add(450)
```

y visualizar su contenido:

```
PS>$lista
cadena
45
domingo, 30 de enero de 2011 23:48:28
otra cadena
450
```

Puedes **eliminar un elemento** con `.remove` indicando su valor:

```
PS>$lista.remove(45)
```

Y ver el resultado... el 45 ya no está

```
PS>$lista
cadena
domingo, 30 de enero de 2011 23:48:28
otra cadena
450
```

Puedes quitar **un rango de elementos** con `.removerange` indicando la posición donde quieres empezar a quitar y el número de posiciones que quieres quitar en total (al igual que las tablas, recuerda que el primer elemento tiene la posición 0). Por ejemplo, quiero quitar desde la posición 2, dos elementos hacia adelante (`"otra cadena"` y `450`).

```
PS>$lista.removerange(2,2)
```

Y vemos

```
PS>$lista
cadena
domingo, 30 de enero de 2011 23:48:28
```

También puedes **eliminar un elemento indicando su posición** con `.removeat`

```
PS>$lista.removeat(0)
```

Hemos quitado el primer elemento `"cadena"` en la posición `0`, y lo comprobamos:

```
PS>$lista
domingo, 30 de enero de 2011 23:48:28
```

Por último, puedes vaciar la lista con

```
PS>$lista.clear()
```

2.4.4 Hashes

Un hash es una lista donde cada elemento se compone de una clave única y su valor correspondiente (par clave - valor). Para crear un hash vacío:

```
PS>$hash = @{}
```

Puedes añadir clave - valor/es de cualquier tipo:

```
PS>$hash["primero"] = 1
PS>$hash["fecha"]   = $fecha
PS>$hash["tabla"]   = 1,"dos",3,4.5
PS>$hash[2]         = "dos"
```

Como siempre, para ver el contenido:

```
PS>$hash

Name                           Value
----                           -----
fecha                          31/01/2011 14:24:30
tabla                          {1, dos, 3, 4,5}
primero                        1
2                              dos
```

Para acceder al valor de un elemento (clave), indica el nombre del hash seguido de un punto y el nombre de la clave

```
PS>$hash.primero
1

PS>$hash.fecha
domingo, 30 de enero de 2011 23:48:28

PS>$hash.tabla
1
dos
3
4,5

PS>$hash.2
error
```

`$hash.2` produce un error porque la clave es un número. Esto se soluciona indicando el número entre corchetes **[ ]**. Las claves de tipo cadena dentro de comillas **" "** también pueden ir dentro de los corchetes:

```
PS>$hash[2]
dos

PS>$hash["fecha"]
domingo, 30 de enero de 2011 23:48:28

PS>$hash["primero"]
1

PS>$hash["tabla"]
1
dos
3
4,5
```

Es posible acceder sólo a las claves del hash:

```
PS>$hash.keys
fecha
tabla
primero
2
```

Y también sólo a los valores:

```
PS>$hash.values
31/01/2011 14:24:30
1
dos
3
4,5
1
dos
```

Uno de los elementos de $hash es la tabla $tabla. Podemos acceder a cualquiera de los valores de $tabla indicando su posición:

```
PS>$hash.tabla[0]
1
PS>$hash.tabla[1]
dos
PS>$hash.tabla[-1]
4,5
```

Eliminemos la fecha de $hash:

```
PS>$hash.remove("fecha")
$hash

Name                Value
----                -----
tabla               {1, dos, 3, 4,5}
primero             1
2                   dos
```

Y con

```
PS>$hash.clear()
```

vaciamos $hash.

2.5 Control de flujo: condiciones y bucles

2.5.1 Condiciones

`If`

Como en la mayoría de los lenguajes, `if` se encarga de evaluar y comparar casi todo. PS2 utiliza la estructura básica: `if (condición) {acción}`. También admite `else` y `elseif`. Por ejemplo, veamos este script:

```
$lenguaje = "español"
if ($lenguaje -eq "alemán") {"El lenguaje es alemán."}
```

Si no se cumple la condición y queremos evaluar otra condición, utilizamos `elseif`. Extendamos el ejemplo en varias líneas para verlo más claro:

```
$lenguaje = "español"

if ($lenguaje -eq "alemán")
{
      "El lenguaje es alemán."
}
elseif ($lenguaje -eq "francés")
{
      "El lenguaje es francés."
}
```

Si no se cumple ninguna condición, podemos ejecutar la acción que indique `else`

```
$lenguaje = "español"

if ($lenguaje -eq "alemán")
{
      "El lenguaje es alemán."
}
elseif ($lenguaje -eq "francés")
{
      "El lenguaje es francés."
}
else
{
      "El lenguaje no es alemán ni francés."
}
```

Esta estructura de evaluación de condiciones siempre comienza con `if`, y el orden de `elseif` y `else` da igual, dependiendo del camino que quieras seguir. En este ejemplo, como `$lenguaje` no es ni `"francés"` ni `"alemán"`, toma la acción de `else`

```
El lenguaje no es alemán ni francés
```

Los operadores de comparación más importantes:

`-eq`	igual que
`-gt`	mayor que
`-ge`	mayor o igual que
`-lt`	menor que
`-le`	menor o igual que
`-ne`	distinto que
`-like`	parecido a (admite * como comodín)
`-notlike`	no parecido a (admite * como comodín)

Pueden evaluarse dos y/o más condiciones con los operadores lógicos `-and` y `-or`

```
if ( ($lenguaje -ne "alemán") -and ($lenguaje -ne "fránces") )
```

En este ejemplo se tienen que cumplir las dos condiciones alrededor de `-and`, y todo ello debe ir dentro de paréntesis.

Switch

Si hay que evaluar muchas condiciones a la vez, es más cómodo utilizar `switch`

```
$lenguaje = "español"

switch ($lenguaje)
{
      "español" {"El lenguaje es español."}
      "francés" {"El lenguaje es francés."}
      "alemán"  {"El lenguaje es alemán."}
      "inglés"  {"El lenguaje es inglés."}
      "italiano"{"El lenguaje es italiano."}
      "danés"   {"El lenguaje es danés."}
      default   {"El lenguaje no está registrado."}
}
```

Where

Como hemos visto antes, podemos redirigir la salida de un cmdlet, objeto o colección de cosas por una tubería y acceder a sus propiedades. Muchas veces necesitamos encontrar alguna propiedad concreta con algún valor concreto, y aquí es donde utilizamos `Where`. Por ejemplo, ¿es 2011 el año actual?

```
PS>get-date | where {$_.year -eq 2011}
```

Como vemos aquí, aparece la variable especial `$_` de la hicimos referencia en el apartado anterior que hablaba de variables. `Where` toma la salida del cmdlet y evalúa lo que aparece a continuación entre llaves. `$_` realmente es la salida de `get-date`, y `$_.year` es la propiedad año de `get-date`. Después, el operador `-eq` indica si `$_.year` es igual a `2011`. Si se cumple la condición, devuelve la salida en sí como lo haría `get-date`, y en caso contrario no devuelve nada.

2.5.2 Bucles

PS2 implementa 5 tipos de bucles: `foreach, for, while, do-while` y `do-until`. Funcionan con tablas, listas, hashes y cualquier cosa que contenga una colección de elementos.

Foreach

es el más fácil de usar, pues hace un recorrido directo por todos los elementos que le des, por ejemplo de una tabla:

```
PS>$tabla = 1, 'dos', 3, 'cuatro', 5
PS>foreach ($elemento in $tabla) {$elemento}
1
dos
3
cuatro
5
```

`foreach` crea un `$elemento` por cada elemento de `$tabla` (`$elemento in $tabla`) y después, entre llaves, puedes hacer cualquier cosa con dicho elemento... en este caso lo muestra `{$elemento}`.

For

crea e inicializa una variable índice, evalúa una condición con esa variable y después la incrementa si se cumple la condición. Las tres cosas anteriores deben ir separadas por punto y coma y todo ello entre paréntesis. Si la condición se cumple, ejecuta lo que hay a continuación entre llaves:

```
PS>for ($i=1; $i -le 3; $i++) {$i}
1
2
3
```

Si `$i` es menor que `3`, `$i` se incrementa en uno (`$i++`) y ejecuta lo que hay a continuación entre llaves (en este caso muestra el valor de `$i`). En la última vuelta, `$i` vale `3`, lo muestra y se incrementa a `4`. Ahora `$i` vale `4`. En este momento `$i` no cumple la condición y el bucle termina.

While

es igual que un `For` y también evalúa la condición al principio del bucle, pero no tiene en cuenta ni la creación de la variable índice ni su incremento:

```
PS>$i = 1
PS>while ($i -le 3) {$i ; $i++}
1
2
3
```

La variable índice se crea antes de `while` y su incremento se produce dentro de la acción entre paréntesis (en una misma línea pueden ir dos o más acciones entre punto y coma como ya vimos). En este caso, si omitimos `$i++`, `$i` siempre vale `1` y el bucle sería infinito. La condición no tiene porque ser la evaluación de un número, también puede ser la evaluación de una cadena:

```
PS>$cadena = ""
PS>while ($cadena -ne "correcto") {$cadena = read-host "Contraseña"}
```

En este caso, mientras `$cadena` sea cualquier cosa menos `"correcto"` nos pide por teclado un valor nuevo para `$cadena`. Cuando tecleamos `"correcto"`, salimos del bucle, y en caso contrario, nos vuelve a pedir un valor nuevo para `$cadena`.

Do-While

Igual que `While` pero evalúa al final de cada bucle. Tomemos este script de ejemplo:

```
$cadena = "correcto"
do
{
      $cadena = read-host "Contraseña"
}
while ($cadena -ne "correcto")
```

La única diferencia al evaluar al final de una vuelta es que la primera vuelta siempre se produce. En este caso, aunque `$cadena` es igual a `"correcto"`, en la primera vuelta siempre nos pide un nuevo valor para `$cadena`.

Do-Until

Exactamente igual que `Do-While` pero cambia la lógica. `Do-While` da vueltas mientras se produzca la condición y `Do-Until` da vueltas hasta que se produzca la condición:

```
$cadena = "correcto"
do
{
      $cadena = read-host "Contraseña"
}
until ($cadena -eq "correcto")
```

En este ejemplo, la salida del bucle se produce cuando `$cadena` sea igual a `"correcto"`.

Bucles especiales

Se puede obtener una enumeración rápida del 1 al 10 así:

```
PS>1..10
1
2
3
4
5
6
7
8
9
10
```

Si redirigimos esa salida por una tubería, cada elemento de la enumeración sale de la tubería como la variable $_ y podemos hacer algo con ello:

```
PS>1..10 | %{$_ * 5}
5
10
15
20
25
30
35
40
45
50
```

y tenemos la tabla de multiplicar del 5. Cada $_ (cada elemento de la enumeración que sale de la tubería) es multiplicado por 5 y se visualiza el resultado.

3 Funciones

Existen varias formas de definir funciones en PS2, pero vamos a ver la más sencilla.

```
PS>function suma($x,$y) { $resultado = $x + $y; return $resultado }
```

Lo anterior define la función suma que toma como parámetros las variables $x e $y (entre paréntesis). A continuación, entre las llaves hace algo con ellas (las suma, coloca el resultado en $resultado y devuelve $resultado).

La llamada a la función es simple: se escribe el nombre de la función y a continuación se indican los parámetros:

```
PS>suma 1 2
3
```

La definición de la función anterior puede abreviarse más así:

```
PS>function suma($x,$y) { return $x + $y }
PS>suma 1 2
3
```

E incluso no tiene porqué devolver resultados con `return`, también los devuelve sin él:

```
PS>function suma($x,$y) { $x + $y }
PS>suma 1 2
3
```

Por último, podemos crear una variable con el resultado de la llamada a la función:

```
PS>function suma($x,$y) { $x + $y }
PS>$resultado = suma 1 2
PS>$resultado
3
```

4 Archivos

Todos los datos se guardan en archivos. Veamos algunas de las operaciones con archivos que puede realizar PS2.

4.1 Comprobación de archivos

¿Existe el archivo datos.txt en la carpeta actual?

```
PS>test-path .\datos.txt
```

Si devuelve `True`, existe. En caso contrario devuelve `False`.

4.2 Lectura rápida de archivos de texto

El cmdlet `get-content` puede leer directamente el contenido de un archivo de texto especificado y ubica su contenido en una variable nueva de tipo tabla. Cada línea leída del archivo de texto será un elemento de la tabla.

Supongamos el archivo de texto `datos.txt` con estas tres líneas:

```
primera
segunda
tercera
```

y en la misma ubicación que `datos.txt` ejecutamos:

```
PS>$contenido = get-content ./datos.txt
```

Ahora `$contenido` es una tabla que contiene las tres líneas leidas de `datos.txt`:

```
PS>$contenido
primera
segunda
tercera
```

Obviamente podemos indicar la ruta completa del archivo de texto a leer:

```
PS>$contenido = get-content "c:\datos curiosos\datos.txt"
```

4.3 Escritura rápida de archivos de texto

Para ello utilizamos una tubería hacia el cmdlet `out-file`.

Supongamos que necesitamos crear el archivo de texto `datos2.txt` con la variable `$contenido` que vimos antes:

```
PS>$contenido | out-file ./datos2.txt
```

Ahora el archivo `datos2.txt` tiene el mismo contenido que `datos.txt`.

Si necesitamos agregar una cuarta línea `datos2.txt`, lo hacemos así:

```
PS>"cuarta" | out-file ./datos2.txt -append
```

Con `out-file` podemos crear/agregar lo que queramos, y si implica varias líneas, también:

```
PS>get-date | format-list | out-file ./datos2.txt -append
```

4.4 Búsqueda de cadenas en archivos de texto

Tomemos de nuevo nuestro archivo de datos `datos.txt` para buscar cadenas en él. Busquemos la cadena `"segunda"`:

```
PS>select-string "segunda" ./datos.txt
datos.txt:2:segunda
```

Como vemos, el resultado de la búsqueda indica el nombre del archivo, el número de línea donde ha encontrado lo que buscábamos, y la cadena que estábamos buscando.

También podemos buscar en cualquier archivo con extensión `txt` que se encuentre en cualquier carpeta a partir de la actual:

```
PS>get-childitem -filter *.txt -recurse | select-string "segunda"
```

5 Expresiones Regulares

Las expresiones regulares es la forma más potente y efectiva para encontrar patrones de caracteres en cadenas de texto. PS2 utiliza el operador `-match` para este propósito.

Por ejemplo, necesitamos saber si la palabra **calabaza** se encuentra dentro de la variable `$cadena` :

```
PS>$cadena -match "calabaza"
```

Y devolverá `True` si lo encuentra o `False` en caso contrario.

Hay que tener en cuenta que `-match` no distingue entre minúsculas y mayúsculas en sus búsquedas. Si necesitamos hacer esa distinción, utilizaremos el operador `-cmatch`

5.1 Caracteres especiales

Las expresiones regulares utilizan ciertos caracteres especiales para definir patrones más interesantes. Vamos a verlos:

5.1.1 El carácter +

El carácter + equivale a **una o más veces el caracter de su izquierda**, de forma que

```
PS>$cadena -match "caf+é"
```

devuelve `True` en caso de que `$cadena` contenga cualquiera de estas cadenas:

```
café, caffé, cafffé, caffffé
```

5.1.2 Los caracteres [ ]

También podemos encontrar alternativas indicadas entre corchetes. Por ejemplo, queremos encontrar la palabra **casa** o **capa** dentro de la variable `$cadena` :

```
PS>$cadena -match "ca[sp]a"
```

como vemos, comenzamos con "ca", después viene la alternativa entre corchetes ("s" o "p") y sigue con "a" para encontrar tanto "casa" como "capa".

Podemos utilizar + junto con los corchetes para obtener ambas funcionalidades, así:

```
PS>$cadena -match "ca[sp]+a"
```

devuelve `True` con cualquiera de las siguientes cadenas dentro de `$cadena` :

```
casa, capa, cassa, cappa, capsa, capppssspsa
```

5.1.3 Los caracteres * ? { }

`*` equivale a **ninguna o más veces el caracter de su izquierda**, de forma que

```
PS>$cadena -match "caf*é"
```

devuelve `True` si `$cadena` es cualquiera de estos valores:

```
caé, café, caffé, caffffffffé
```

`?` equivale a **ninguna o una vez el caracter de su izquierda**, de forma que

```
PS>$cadena -match "caf?é"
```

devuelve `True` si `$cadena` es cualquiera de estos dos valores:

```
caé, café
```

Las llaves { } indican **el número de ocurrencias del caracter de su izquierda**. Podemos indicar el número mínimo y máximo de ocurrencias separados por una coma:

```
PS>$cadena -match "caf{1,5}é"
```

devuelve `True` si `$cadena` es cualquiera de estos cinco valores (entre una y cinco efes entre "ca" y "é"):

```
café, caffé, cafffé, caffffé, cafffffé
```

Si entre llaves sólo indicamos un número, sólo debe darse ese número de ocurrencias:

```
PS>$cadena -match "caf{4}é"
```

devuelve `True` si `$cadena` solamente es `caffffé`

Para encontrar un mínimo de ocurrencias sin un máximo, indicamos entre llaves el número mínimo y una coma:

```
PS>$cadena -match "caf{4,}é"
```

devuelve `True` si `$cadena` es `caffffé, cafffffffffé, etc.` (4 efes o más)

5.1.4 El carácter .

`.` equivale a **una vez cualquier carácter menos el retorno de carro**, de forma que

```
PS>$cadena -match "ca.é"
```

devuelve `True` si `$cadena` es cualquiera de estos valores:

```
"café", "calé", "cabé", "cazé"
```

El . suele utilizarse junto con * para indicar cualquier carácter (o no) un número de indefinido de veces. Por ejemplo:

```
PS>$cadena -match "ca.*é"
```

devuelve `True` si `$cadena` contiene cualquier cosa menos el retorno de carro entre "ca" y "é". Puede ser "caé", "café", y también puede ser "cachislamar salada porque no está él en lugar de otro"

5.1.5 El carácter |

Aquí la tubería también tiene su propósito: encontrar dos o más alternativas.

Por ejemplo:

```
PS>$cadena -match "café|té|colacao"
```

devuelve `True` si `$cadena` contiene café, té o colacao. También puede especificarse entre | cualquier combinación que hayamos visto antes, por ejemplo:

```
PS>$cadena -match "caf{1,5}é|caf*é|ca[sp]a"
```

5.2 Encontrar caracteres especiales

Hemos visto cómo encontrar multitud de patrones que incluyen caracteres "normales", pero ¿y sí queremos encontrar patrones que incluyan alguno de los caracteres especiales que hemos visto (+ [ ] * . |) o de los que nos quedan por ver?. Para ello hay que indicar el carácter especial precedido de la barra invertida o backslash (**\ )**

Por ejemplo, si queremos encontrar en la cadena dada $cadena una o más interrogantes juntas, ponemos:

```
$cadena -match "\?+"
```

Si queremos encontrar una barra invertida, hay que poner dos juntas:

```
$cadena -match "\\"
```

5.3 Anclajes

Se utilizan para encontrar patrones al principio y/o al final de una cadena de texto.

Si queremos encontrar la palabra "Hola" al principio de la cadena de texto que reside en `$cadena`, utilizamos el acento circunflejo a la izquierda de "Hola":

```
$cadena -match "^Hola"
```

De forma parecida, si queremos encontrar la palabra "adiós" al final de la cadena, utilizamos el símbolo dólar a la derecha de "adiós":

```
$cadena -match "adiós$"
```

Si utilizamos ambos anclajes nos aseguramos de que la cadena de texto dada sea exactamente la que buscamos. Por ejemplo, lo siguiente devuelve `True` si `$cadena` contiene exactamente "Hola y adiós":

```
$cadena -match "^Hola y adiós$"
```

5.4 Secuencias de escape

Las expresiones regulares agrupan ciertos rangos de caracteres en secuencias de escape.

Por ejemplo, si necesito encontrar un dígito del 0 al 9 dentro de `$cadena`, utilizamos la secuencia de escape `\d`, así:

```
$cadena -match "\d"
```

Esta es una lista de algunas secuencias de escape muy útiles:

Secuencia de escape	Equivalencia
\d	Cualquier dígito
\D	Cualquier carácter que no sea un dígito
\w	Cualquier letra, dígito y guión bajo
\W	Lo contrario que lo anterior
\s	Espacio en blanco, tabulador o retorno de carro
\S	Lo contrario de lo anterior

5.5 Extracción de patrones

Cuando `-match` o `-cmatch` encuentran un patrón en una variable dada, PS2 crea un hash nuevo llamado `$matches` para alojar lo encontrado:

```
PS>$cadena = "hola"
PS>$cadena -match "hola"
True
PS>$matches
Name                           Value
----                           -----
0                              hola
```

Quedando disponible el resultado en `$matches[0]`

```
PS>$matches[0]
hola
```

Es posible extraer varios patrones acotándolos entre paréntesis. Por ejemplo, extraigamos ahora dos patrones: los tres primeros dígitos y los tres últimos de `$cadena`:

```
PS>$cadena = "123hola456"
PS>$cadena -match "(^\d\d\d).*(\d\d\d$)"
True
PS>$matches
Name                           Value
----                           -----
2                              456
1                              123
0                              123hola456
```

Como vemos, el primer patrón encontrado (123) queda disponible en la posición 1 del hash y el segundo patrón encontrado queda disponible en la posición 2 del hash. La posición 0 del hash contiene la cadena completa.

Los valores de `$matches` cambiarán con los resultados del siguiente `-match` o `-cmatch`.

5.6 Sustitución

El operador `-replace`, PS2 sustituye unas cadenas por otras. Por ejemplo, dada la cadena `"Hola Manola"` sustituyamos `"Hola"` por `"Adiós"`:

```
PS>"Hola Manola" -replace "Hola", "Adiós"
Adiós Manola
```

Mediante expresiones regulares también podemos sustituir patrones dados entre paréntesis con los resultados obtenidos alojados en las variables temporales $1, $2, $3...:

```
PS>"Hola Manola" -replace '(.*) (.*)','$2 $1'
Manola Hola
```

6 Administración local

El punto fuerte de PS2 es la administración del sistema operativo, en especial los tres pilares fundamentales: memoria (procesos y servicios), sistema de archivos (archivos y carpetas) y red. También veremos qué puede hacer PS2 con el registro de Windows y el visor de sucesos. Por último veremos cómo PS2 trabaja junto con WMI (Windows Management Instrumentation) para así ampliar su alcance en la administración del sistema.

6.1 Procesos y Servicios

6.1.1 Procesos

La memoria de Windows, como cualquier otro sistema operativo sirve para alojar procesos que corren en el sistema. Con PS2 podemos verl todos los procesos que actualmente están corriendo en el sistema con:

```
PS>get-process
```

Si queremos ver la información relevante del proceso `powershell`, indicamos su nombre:

```
PS>get-process powershell

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    143       5    28600      27384   136     1,11   3184 powershell
```

Como ya hemos visto, podemos ver todas las propiedades y métodos de cualquier cmdlet. En este caso:

```
PS>get-process | get-member

      TypeName: System.Diagnostics.Process

Name                       MemberType     Definition
----                       ----------     ----------
Handles                    AliasProperty  Handles = Handlecount
Name                       AliasProperty  Name = ProcessName
NPM                        AliasProperty  NPM =
NonpagedSystemMemorySize
PM                         AliasProperty  PM = PagedMemorySize
VM                         AliasProperty  VM = VirtualMemorySize
WS                         AliasProperty  WS = WorkingSet
...
```

Aquí vemos que `WS` es el alias (la versión corta) de la propiedad `WorkingSet` que es el tamaño en bytes del proceso en cuestión:

```
PS>(get-process powershell).ws
29020160
```

Para verlo en megabytes, dividimos el resultado por un megabyte así:

```
PS>(get-process powershell).ws / 1mb
28,35546875
```

Otra propiedad interesante es la ruta donde está ubicado el archivo del proceso en cuestión. Por ejemplo, si tenemos el Block de notas abierto y queremos ver donde está su archivo en disco, ponemos:

```
PS>(get-process notepad).path
C:\WINDOWS\system32\notepad.exe
```

Para mostrar varias propiedades de un proceso, utilizamos la tubería dirigida por ejemplo a `Format-Table` con su argumento `-property` e indicamos todas las propiedades que necesitemos entre comas:

```
PS>get-process powershell | ft -property name, company -auto

Name       Company
----       -------
powershell Microsoft Corporation
```

El argumento `-auto` ajusta el ancho de las columnas al ancho de los datos visualizados. `Format-Table (ft)` visualiza bien pocas propiedades. Si indicamos muchas propiedades, utilizamos `Format-List (fl)`.

También podemos ver todos los procesos que tengan una propiedad común. Por ejemplo, queremos ver todos los procesos cuya compañía contenga la palabra "Microsoft":

```
PS>get-process | where-object {$_.company -like '*Microsoft*'}

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    105       5     1160        244    33     0,09   1236 alg
    111       4      892       1412    38     0,38   1260 ctfmon
    443      12    13644      13172   105    12,28    956 explorer
    309       6     2264        928    37     1,28   1732 lsass
     47       2     1048       3584    30     0,34   3928 notepad
   1068       6    38752      34708   140     8,48   2164 powershell
    296       7     1788       1360    22     3,09   1720 services
     19       1      168         48     4     0,05   1536 smss
```

También podemos ver en una lista los procesos agrupados y ordenados por la compañía a la que pertenecen:

```
PS>get-process | group-object company | sort-object name | fl
```

Por último, podemos finalizar un proceso de dos formas:

```
PS>get-process notepad | stop-process

PS>(get-process notepad).kill()
```

6.1.2 Servicios

Un servicio es un proceso que se inicia en el sistema antes de que el usuario inicie su sesión. Son típicos los servicios de Microsoft como la cola de impresión, el cliente DHCP, el cliente DNS, compartición de carpetas, audio, actualizaciones de Windows, conexiones de red, etc. Otros fabricantes también aportan servicios al sistema cuando sus productos son instalados: antivirus, controladores, servidores, etc.

Al igual que los procesos, podemos ver los servicios iniciados (no deshabilitados) con:

```
PS>get-service
```

y vemos el estado, el nombre y una descripción de cada uno.

El estado de un servicio puede ser `Running` (en ejecución) o `Stopped` (parado). Los servicios parados suelen depender de otros servicios para iniciarse. Para ver los servicios y sus dependencias utilizamos el argumento `servicesdependedon`:

```
PS>get-service | ft -property name, servicesdependedon -auto
```

Y aparecerán a la izquierda los nombres de los servicios y a la derecha los correspondientes servicios de los que dependen.

De modo parecido, podemos ver los servicios que dependen de cada servicio indicando el argumento `dependentservices`:

```
PS>get-service | ft -property name, dependentservices -auto
```

Podemos parar un servicio, como la cola de impresión indicando su nombre con:

```
PS>stop-service spooler
```

E iniciarlo con:

```
PS>start-service spooler
```

6.2 Archivos y carpetas

Los archivos son los contenedores básicos de información y las carpetas son grupos de archivos y carpetas.

6.2.1 Ubicación actual

Para ver la ruta donde estamos actualmente ubicados en la estructura de carpetas utilizamos la variable `$pwd`:

```
PS>$pwd

Path
----
C:\workspace
```

La ruta es una propiedad, y ya sabemos obtenerla directamente:

```
PS>$pwd.path
C:\workspace
```

Conseguimos el mismo resultado con el cmdlet `get-location`.

Para cambiar la ubicación actual a c:\, utilizamos:

```
PS>set-location c:\
```

Y vemos:

```
PS>$pwd

Path
----
C:\
```

6.2.2 Localización de archivos y carpetas

Podemos obtener un listado de todos los archivos y carpetas de la ubicación actual con `get-childitem`, `ls` o `dir`. En este caso utilizaremos el primero:

```
PS>get-childitem -force
```

El argumento `-force` visualiza todos los archivos, incluidos los archivos ocultos y de sistema. Como dato interesante, aparece en la columna de la izquierda los 5 modos que puede tener cada archivo mostrado. Estos modos son:

d (directory): carpeta.
a (archive): archivo.
r (read only): sólo lectura.
h (hidden): oculto.
s (system): archivo de sistema.

Para obtener toda la estructura de archivos y carpetas de la ubicación actual utilizamos el argumento `-recurse`:

```
PS>get-childitem -force -recurse
```

Para obtener toda la estructura de carpetas de la ubicación actual, utilizamos la propiedad `PsIsContainer`:

```
PS>get-childitem -force -recurse | where-object {$_.psiscontainer}
```

Todo lo anterior también funciona indicando una ruta concreta:

```
PS>get-childitem c:\workspace\powershell -force
```

Podemos utilizar comodines utilizando el argumento `-filter`. Por ejemplo: necesito todos los archivos PDF que se encuentren en la carpeta actual y todas las subcarpetas:

```
PS>get-childitem -filter *.PDF -recurse
```

Lo anterior también admite el uso de `?` al igual que en la línea de comandos convencional.

6.2.3 Búsqueda de cadenas de texto en archivos

También es útil buscar cadenas de texto dentro de archivos. De la siguiente forma, buscamos la palabra "hola" que se encuentre en cualquier archivo de la carpeta actual:

```
PS>get-childitem | where { $_ | select-string "hola" -quiet }
```

6.3 Registro de Windows

Toda la configuración de Windows y de una buena parte de los programas y aplicaciones que se pueden instalar en él guardan sus configuraciones en el registro de Windows.

Físicamente, el registro de Windows se compone de unos pocos archivos, y de forma lógica se divide en las siguientes zonas:

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
HKEY_USERS
HKEY_CURRENT_CONFIG

La más utilizada para tareas de administración suele ser HKEY_LOCAL_MACHINE donde guarda la mayor parte de la configuración del sistema operativo, servicios y más.

PS tiene una forma de acceder al registro muy natural, como si se tratara de letras de unidad, y hace uso de las abreviaciones de las zonas; por ejemplo, la forma abreviada de HKEY_LOCAL_MACHINE es HKLM y para entrar en HKLM utilizamos:

```
PS>set-location HKLM:
PS>pwd
Path
----
HKLM:\
```

Y con pwd, vemos que estamos ubicados en la raíz de HKLM.

6.3.1 Navegando por el registro

Como si de una estructura de archivos se tratara, podemos listar el contenido de la ubicación actual con `get-childitem`, `ls` o `dir`. Ahora utilizaremos `ls`:

```
PS>ls
```

Y aparecerán las carpetas principales de `HKLM`. De las cinco que aparecen, `SECURITY` da un error de acceso, pero es normal. Veamos cómo llegar a la lista completa de servicios del sistema desde aquí:

```
PS>cd SYSTEM\CurrentControlSet\Services
PS>ls

    Hive: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services


SKC  VC Name                           Property
---  -- ----                           --------
  0   5 Abiosdsk                      {ErrorControl, Group, Start, Tag...}
  1   5 abp480n5                      {ErrorControl, Group, Start, Tag...}
  1   7 ACPI                          {ErrorControl, Group, Start, Tag...}
  0   5 ACPIEC                        {ErrorControl, Group, Start, Tag...}
  1   5 adpu160m                      {ErrorControl, Group, Start, Tag...}
  1   0 adsi                          {}
  1   5 Aha154x                       {ErrorControl, Group, Start, Tag...}
  1   5 aic78u2                       {ErrorControl, Group, Start, Tag...}
  1   5 aic78xx                       {ErrorControl, Group, Start, Tag...}
  ...
```

Vamos a ver lo que tiene el servicio `Cdrom`:

```
PS>cd Cdrom
```

Veamos las propiedades que tiene actualmente este servicio:

```
PS>get-itemproperty .

PSPath               :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHI...
PSParentPath         :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHI...
PSChildName          : Cdrom
PSDrive              : HKLM
PSProvider           : Microsoft.PowerShell.Core\Registry
DependOnGroup        : {SCSI miniport}
ErrorControl         : 1
Group                : SCSI CDROM Class
Start                : 1
Tag                  : 2
Type                 : 1
DisplayName          : Controlador de CD-ROM
ImagePath            : system32\DRIVERS\cdrom.sys
AutoRun              : 1
```

6.3.2 Modificando el registro

Supongamos que queremos desactivar la ejecución automática de la unidad de CD/DVD al introducir un disco. Para ello cambiamos el valor de la propiedad `Autorun`, de 1 a 0:

```
PS>set-itemproperty . AutoRun 0
```

Veamos el resultado:

```
PS>(get-itemproperty .).AutoRun
0
```

Para eliminar la propiedad `Autorun`, utilizamos:

```
PS>remove-itemproperty . Autorun
```

Y para crear una propiedad nueva llamada `pruebas` con un valor de tipo binario `5`, utilizamos:

```
PS>new-itemproperty . -name pruebas -value 5 -type Binary
```

El argumento `-type` puede indicar 6 tipos de valores para una propiedad:

Omitendo `-type` es alfanumérico (REG_SZ o String)
`MultiString`: tabla de strings (REG_MULTI_SZ)
`ExpandString`: cadena expandible (REG_EXPAND_SZ)
`Binary`: binario (REG_BINARY)
`DWord`: doble palabra (REG_DWORD)
`QWord`: cuádruple palabra (REG_QWORD)

También podemos crear aquí una clave de registro nueva llamada `test` (una carpeta):

```
PS>new-item test
```

Ahora podemos entrar en `test` como hicimos con `Autorun` y crear propiedades como ya hemos visto.

Por último, para eliminar la clave de registro `test` y todas sus propiedades, nos ubicamos encima de ella y ponemos:

```
PS>remove-item test
```