Autor: Darío Hirschfeldt

PowerShell Scripting

Los archivos de nuestros scripts de PowerShell deberán tener como extensión **.ps1,** además podremos incorporar la ayuda del script de manera tal que se pueda invocar con el comando Get-Help. Se puede escribir con cualquier editor de texto aunque también se dispone del PowerShell ISE, que es un entorno de desarrollo quizá mas práctico y asistido.

Clásico Hola mundo

HolaMundo.ps1
```
#Clásico primer programa

Write-Host "Hola Mundo"
```

Además de la impresión por pantalla con el comando Write-Host, podemos ver que al igual que bash las líneas comentadas comienzan con el carácter #. Pero a diferencia de bash, Powershell permite comentarios multilínea encerrando al bloque de comentario entre <# #>, por ejemplo:

HolaMundo.ps1
```
<#
Clásico primer programa
#>

Write-Host "Hola Mundo"
```

Si lo ejecutamos:
```
PS C:\PowerShell> .\HolaMundo.ps1
Hola Mundo
```

Incorporación de la ayuda

Para incorporar la ayuda, PowerShell tiene un método de “Ayuda basada en comentarios”, en este apunte trataremos lo básico para incorporar la ayuda a los scripts, lo recomendable sería realizar la lectura del tópico conceptual **about_Comment_Based_Help** para complementar el tema.

HolaMundo.ps1
```
<#
.Synopsis
Muestra el mensaje Hola Mundo

.Example
C:\PS>HolaMundo.ps1
Hola Mundo
#>

Write-Host "Hola Mundo"
```

En el bloque de comentario vemos palabras clave, que son las que comienzan con punto, y debajo el contenido. Hay palabras clave que solo se pueden utilizar una vez como por ejemplo Synopsis (Breve descripción de lo que hace el script) y otras como Example que pueden aparecer mas de una vez.
Este bloque de ayuda puede escribirse:

a) Al comienzo del script (Como en el ejemplo) y solamente puede ser precedida por comentarios o líneas en blanco. Además si luego de la ayuda lo primero que aparece es la declaración de una función, debe haber una separación de al menos dos líneas en blanco entre el fin de la ayuda y la declaración de la función.

b) Al final del script.

Si ejecutamos con Get-help

```
PS C:\PowerShell> get-help .\HolaMundo.ps1

NOMBRE
    C:\PowerShell\HolaMundo.ps1

SINOPSIS
    Muestra el mensaje Hola Mundo
```

Acento grave

El acento grave Alt+96 ` es el carácter que permite escapear caracteres para mostrarlos literalmente, también permite escapear la línea, es decir si queremos partir un comando en varías líneas.
Otros caracteres especiales que utilizan el acento grave:

`b	Espacio
`n	Nueva línea
`r	Retorno de carro
`r`n	Retorno de carro + nueva línea
`t	Tabulación horizontal

Para profundizar sobre este tema, referirse a los tópicos conceptuales **about_escape_characters** y **about_special_characters**

Tipos de datos

Los tipos de datos mas comunes son: **Int, Int32, Int64, Double, Char, String, Boolean, Array, Object, PSObject, PSCustomObject**.

Variables

Las variables de PowerShell pueden ser utilizadas sin definir su tipo, en este caso la variable ajustará su tipo en función del tipo del valor asignado, lo que se conoce como tipado débil. Por ejemplo:

TipadoDébil.ps1

```powershell
#Tipado débil

$var = "0"
Write-Host `$var: $var $var.getType()

$var = 0
Write-Host `$var: $var $var.getType()
```

El nombre de la variable siempre debe comenzar con $, sea tanto para asignar valor como para referenciar su contenido. Si el nombre de la variable tiene caracteres especiales debe ir entre llaves, por ejemplo ${dd-mm-aaaa}.

En el ejemplo primero se asigna a la variable $var una cadena y luego un entero. Con el método getType obtenemos el Tipo.
Si ejecutamos el script:

```
PS C:\PowerShell> .\TipadoDébil.ps1
$var: 0 System.String
$var: 0 System.Int32
```

Si bien vemos que el valor mostrado por pantalla es el mismo, el tipo de dato cambia.

Ahora si intentamos operar, por ejemplo con una suma (+):

TipadoDébilSuma.ps1

```
1  #Tipado débil
2
3  $var = "0"
4  Write-Host Concatena: ($var + 1)
5
6  $var = 0
7  Write-Host Suma: ($var + 1)
```

La operación está entre paréntesis para que se pueda mostrar el resultado en el Write-Host. Esto da como salida:

```
PS C:\PowerShell> .\TipadoDébilSuma.ps1
Concatena: 01
Suma: 1
```

Dado que el operador + tiene distinto sentido si se trata de cadenas o de números, al encontrar una cadena como primer operando automáticamente va a concatenar lo que le siga, caso contrario cuando el primer operando es un entero.

Pero PowerShell también permite que definamos el tipo de una variable, sería un tipado fuerte:

TipadoFuerte.ps1

```
1  #Tipado fuerte
2
3  [Int]$var=0
4
5  Write-Host `$var: $var $var.getType()
6
7  $var="0"
8
9  Write-Host `$var: $var $var.getType()
```

Si ejecutamos:

```
PS C:\PowerShell> .\TipadoFuerte.ps1
$var: 0 System.Int32
$var: 0 System.Int32
```

Ahora vemos que el tipo no cambió, a pesar de que se asigno el string “0”. Probemos asignar una string con letras:

TipadoFuerte.ps1
```
#Tipado fuerte

[Int]$var=0

Write-Host `$var: $var $var.getType()

$var="Pepe"

Write-Host `$var: $var $var.getType()
```

Al ejecutar vemos que arroja un error:

```
PS C:\PowerShell> .\TipadoFuerte.ps1
$var: 0 System.Int32
No se puede convertir el valor "Pepe" al tipo "System.Int32".
En C:\PowerShell\TipadoFuerte.ps1: 7 Carácter: 5
+ $var <<<< ="Pepe"
    + CategoryInfo          : MetadataError: (:) [], Argument
    + FullyQualifiedErrorId : RuntimeException

$var: 0 System.Int32
```

Esto está bien porque es lo que buscamos al tipar la variable. También nos da una idea de que de ser posible, PowerShell tratará de convertir el tipo del valor asignado al tipo de la variable, caso contrario arroja el error de incompatibilidad de tipos.

Sentencias de control

Condicional

Condicional.ps1
```
Param(
    [Parameter(Mandatory=$true)][Int32]$valor_A,
    [Parameter(Mandatory=$true)][Int32]$valor_B
)

if( $valor_A -eq $valor_B )
{
    Write-Host "Los valores son iguales"
}
else
{
    Write-Host "Los valores son distintos"
}
```

En el ejemplo anterior vemos que el condicional if tiene una sintaxis muy conocida, solo que las llaves son obligatorias. Los comparadores son al estilo bash (-eq, -ne, -gt, etc), también se agregan algunos interesantes como (-like, -notlike, -match, -contains y otros mas), referirse al tópico conceptual **about_Comparison_Operators** para complementar el tema. En materia de conectores lógicos tenemos; (-and, -or, -not o ! y -xor).

También introducimos la función Param, para organizar los parámetros de entrada. En este ejemplo se manejan dos parámetros que se asociarán a las variables $valor_A y $valor_B. También agregamos el tipo de dato esperado en cada variable (Int32) y especificamos que es obligatorio el

envío de los mismos. Es recomendable profundizar sobre la función Param en el tópico conceptual **about_Functions_Advanced_Parameters.**

Si lo ejecutamos:

```
PS C:\PowerShell> .\Condicional.ps1 0 1
Los valores son distintos
PS C:\PowerShell> .\Condicional.ps1 0 0
Los valores son iguales
```

Nótese que si ejecutamos el script sin parámetros, se abre una especie de prompt para que el usuario ingrese el parámetro necesario, esto se debe a la incorporación de Mandatory=$true.

```
PS C:\PowerShell> .\Condicional.ps1

cmdlet Condicional.ps1 en la posición 1 de la canalización de comandos
Proporcione valores para los parámetros siguientes:
valor_A:
```

Ciclo foreach

Foreach.ps1

```powershell
<#
.Synopsis
Muestra el nombre de los procesos cuya
memoria virtual supera los 512 MB
#>

$conjunto = Get-Process

foreach ($elemento in $conjunto) {
    if( $elemento.VM/1MB -gt 512 ){
        Write-Host $elemento.Name
    }
}
```

Si lo ejecutamos:

```
PS C:\PowerShell> .\Foreach.ps1
AvastSvc
explorer
powershell
SMSvcHost
```

Ciclo for

For.ps1

```powershell
<#
.Synopsis
Muestra las líneas de un archivo que tienen mas de 30 caracteres
indicando el numero de línea.
#>

Param(
    [Parameter(Mandatory=$true)][ValidateNotNullOrEmpty()][String]$archivo
)

#Constante
New-Variable -Name MAX_CHARS -Value 30 -Option Constant

$lineas = Get-Content $archivo

for( $i=0; $i -lt $lineas.Length; $i++ ){
    if( $lineas[ $i ].Length -gt $MAX_CHARS ){
        Write-Host ($i+1) "-" $lineas[ $i ]
    }
}
```

En este ejemplo se utiliza un ciclo for para recorrer las líneas de un archivo. Aprovechando el ejemplo vemos la forma de declarar una **constante** (MAX_CHARS). También notaremos la aparición de **ValidateNotNullOrEmpty** para validar que el string (Nombre del archivo) no sea cadena vacía.

Si lo ejecutamos:

```
PS C:\PowerShell> .\For.ps1 .\Foreach.ps1
3 - Muestra el nombre de los procesos cuya
4 - memoria virtual supera los 512 MB
9 - foreach ($elemento in $conjunto) {
10 -     if( $elemento.VM/1MB -gt 512 ){
```

Nótese que si ejecutamos el script sin parámetros, e ingresamos solo ENTER cuando se solicita el parámetro, arroja un error, esto se debe a la validación de cadena no vacía:

```
PS C:\PowerShell> .\For.ps1

cmdlet For.ps1 en la posición 1 de la canalización de comandos
Proporcione valores para los parámetros siguientes:
archivo:
C:\PowerShell\For.ps1 : No se puede validar el argumento del parámetro 'archivo'.
ento que no sea NULL o no esté vacío e intente ejecutar el comando de nuevo.
En línea: 1 Carácter: 10
+ .\For.ps1 <<<<
    + CategoryInfo          : InvalidData: (:) [For.ps1], ParameterBindingValidati
    + FullyQualifiedErrorId : ParameterArgumentValidationError,For.ps1
```

Ciclos while – do while – do until

While_DoUntil_DoWhile.ps1

```powershell
#Cuenta de 1 a 5
Write-Host "While"
while( $val -lt 5 ){ $val++; Write-Host $val }

#Se repite hasta que el usuario ingrese un valor mayor a 0
Write-Host "Do Until"
do{
    $opcion = Read-Host "Ingrese un valor mayor a 0"
}until ( [Int32]$opcion -gt 0 )

#Idem anterior, pero con Do While
Write-Host "Do While"
do{
    $opcion = Read-Host "Ingrese un valor mayor a 0"
}while ( [Int32]$opcion -le 0 )
```

Si lo ejecutamos:

```
PS C:\PowerShell> .\While_DoUntil_DoWhile.ps1
While
1
2
3
4
5
Do Until
Ingrese un valor mayor a 0: -1
Ingrese un valor mayor a 0: 0
Ingrese un valor mayor a 0: 1
Do While
Ingrese un valor mayor a 0: -1
Ingrese un valor mayor a 0: 0
Ingrese un valor mayor a 0: 1
PS C:\PowerShell>
```

Nótese que se pueden escribir distintos comandos o instrucciones en la misma línea utilizando como fin de línea el punto y coma (;). También vemos que al no forzar el tipo de la variable $opción a entero, hay que hacer un cast a [Int32] para que la comparación sea numérica.

Switch

Switch.ps1

```powershell
#Selecciona una opción de 1 a 3
$opcion = Read-Host "Ingrese un numero de 1 a 3"
switch ( $opcion ){
    1 {"Uno"}
    2 {"Dos"}
    3 {"Tres"}
    default {"Otro"}
}
```

```
PS C:\PowerShell> .\Switch.ps1
Ingrese un numero de 1 a 3: 1
Uno
```

```powershell
#Selecciona en base a una cadena
$nombre = "Darío"
switch ( $nombre ){
    Darío { "Es Daro" }
    Jorge { "Es Jorgito" }
    Lucía { "Es Lu" }
}
```

```
PS C:\PowerShell> .\Switch.ps1
Es Daro
```

```powershell
#Selecciona el pasatiempo correspondiente a los
#integrantes presentes en el array
$familia = @("Padre","Hija")

switch ( $familia ) {
    "Padre" { "Fútbol" }
    "Madre" { "Netflix" }
    "Hijo"  { "LOL" }
    "Hija"  { "Teléfono" }
}
```

```
PS C:\PowerShell> .\Switch.ps1
Fútbol
Teléfono
```

```powershell
#Si no ponemos un break o continue entra en todos
switch ( "Pepe" ){
    PEPE { "Es Pepe" }
    pepe { "Es Pepe" }
    Pepe { "Es Pepe" }
}

switch ( "Pepe" ){
    PEPE { "Es Pepe"; break }
    pepe { "Es Pepe"; break }
    Pepe { "Es Pepe"; break }
}
```

```
PS C:\PowerShell> .\Switch.ps1
Es Pepe
Es Pepe
Es Pepe
Es Pepe
```

```powershell
#Existe la posibilidad de realizar comparaciones
#en el case
$edad = Read-Host "Ingrese su edad"
switch ( $edad ){
    { $edad -ge 18  }
    {
        Write-Host "Mayor"
    }
    default
    {
        Write-Host "Menor"
    }
}
```

```
PS C:\PowerShell> .\Switch.ps1
Ingrese su edad: 10
Menor
```

La sentencia switch es bastante completa pueda trabajar además con archivos y expresiones regulares entre otras cosas, para mayor información referirse al tópico conceptual **about_Switch**.

Arrays

Arrays.ps1

```powershell
$pares=2,4,6,8

#Mostrar todos los elementos
$pares

#Mostrar el primer elemento
$pares[0]

#Mostrar el último elemento
$pares[-1]

#Agregar un elemento
$pares+=10

#Cantidad de elementos
$pares.count
$pares.length

#Se puede utilizar el operador de rango .. para
#generar una secuencia
$digitos=0..9

#Los arrays pueden tener distintos tipos
$mezclado=1,2,3,"A","B","C",0.5
```

```
PS C:\PowerShell> .\Arrays.ps1
2
4
6
8
2
8
5
5
```

Para mayor información sobre Arrays referirse al tópico conceptual **about_Arrays.**

Arrays bidimensionales

En las matrices “Jagged” la cantidad de columnas no es fija para toda la matriz, sino que puede variar con cada fila, además que pueden incrementar sus elementos.

Matrices.ps1

```powershell
#Matrices Jagged (Serradas)

$matriz=(1,2,3),(4,5),(6,7,8,9)

for( $i=0; $i -lt $matriz.length; $i++ ){
    for( $j=0; $j -lt $matriz[ $i ].length; $j++ ){
        Write-Host $matriz[ $i ][ $j ]" " -NoNewLine
    }
    Write-Host
}
```

```
PS C:\PowerShell> .\Matrices.ps1
1  2  3
4  5
6  7  8  9
```

También existen las matrices clásicas con filas y columnas fijas.

MatricesClasicas.ps1

```powershell
#Matrices Clásicas (Estáticas)

$filas=4
$columnas=3

$matriz=New-Object 'object[,]' $filas, $columnas

for( $i=0; $i -lt $filas; $i++ ){
    for( $j=0; $j -lt $columnas; $j++ ){
        $matriz[ $i , $j ] = $i*$j
    }
}

for( $i=0; $i -lt $filas; $i++ ){
    for( $j=0; $j -lt $columnas; $j++ ){
        Write-Host $matriz[ $i , $j ]" " -NoNewLine
    }
    Write-Host
}
```

```
PS C:\PowerShell> .\MatricesClasicas.ps1
0  0  0
0  1  2
0  2  4
0  3  6
PS C:\PowerShell>
```

Nótese que además de la forma de creación, varía la manera de referenciar los elementos en la primera se utiliza [fila][columna] y en la segunda [fila, columna].

Arrays asociativos (Hash tables)

En un array asociativo almacenaremos pares (Clave, Valor), referenciando a los elementos utilizando como subíndice las claves.

HashTables.ps1

```
1  #Arrays Asociativos
2
3  $hashA=@{} #Array vacío
4
5  #Agregar un elemento con la función Add
6  $hashA.Add( "Lunes", "Sistemas Operativos" )
7
8  #Agregar un elemento con el operador +
9  $hashA = $hashA + @{"Martes" = "Diseño" }
10
11 $hashA
```

```
PS C:\PowerShell> .\HashTables.ps1

Name                           Value
----                           -----
Martes                         Diseño
Lunes                          Sistemas Operativos
```

La columna Name representa las claves y Value el valor. Si quisiéramos podríamos modificar la salida del Hash, bastará con cambiar la línea 11:

```
11 $hashA | Format-Table @{Label='Día';Expression={$_.Name}},@{Label='Materia';Expression={$_.Value}} -AutoSize
```

```
PS C:\PowerShell> .\HashTables.ps1

Día    Materia
---    -------
Martes Diseño
Lunes  Sistemas Operativos
```

HashTables.ps1

```
1  #Arrays Asociativos
2
3  $hashB=@{29383491="Hirschfeldt";30592655="Romero"} #Array inicializado
4
5  $hashB.keys #Listar todas las claves
6
7  $hashB.values #Listar todos los elementos
8
9  $hashB[ 29383491 ] = "Hirschfeldt Dario" #Modificar un elemento
10
11 $hashB[ 29383491 ] #Listar un elemento en particular
12
13 $hashB.remove(29383491) #Eliminar un elemento
```

```
PS C:\PowerShell> .\HashTables.ps1
30592655
29383491
Romero
Hirschfeldt
Hirschfeldt Dario
```

Ahora supongamos que queremos contar la cantidad de ocurrencias de cada letra en una entrada dada:

HashTables.ps1

```
1  #Arrays Asociativos
2
3  $hashA=@{}
4
5  foreach( $letra in "a","A","B","b","C","d" ){
6      $hashA[ $letra ]++;
7  }
8  $hashA
```

```
PS C:\PowerShell> .\HashTables.ps1

Name                           Value
----                           -----
a                              2
B                              2
d                              1
C                              1
```

Eso está bien, ahora si quisiéramos que distinga entre mayúsculas y minúsculas, es decir que sea case sensitive, tendremos que declarar el array de otra manera:

HashTables.ps1

```
1  #Arrays Asociativos
2
3  $hashC=New-Object System.Collections.HashTable #Array case sensitive
4
5  foreach( $letra in "a","A","B","b","C","d" ){
6      $hashC[ $letra ]++;
7  }
8  $hashC
```

```
PS C:\PowerShell> .\HashTables.ps1

Name                           Value
----                           -----
d                              1
b                              1
B                              1
C                              1
a                              1
A                              1
```

Si quisiéramos que la salida se muestre ordenada por ejemplo por la clave (Name) bastará con cambiar la línea 8 por:

```
8  $hashC.getenumerator() | sort -property Name
```

```
PS C:\PowerShell> .\HashTables.ps1

Name                           Value
----                           -----
a                              1
A                              1
B                              1
b                              1
C                              1
d                              1
```

Para mayor información sobre el tema, referirse al tópico conceptual **about_Hash_Tables**