PowerShell Scripting

Contenido

- Habilitar ejecución de scripts
- Tipos y colecciones
- Operadores
- Operadores de asignación
- Argumentos de entrada
- Evaluación de condiciones
- Operadores lógicos
- Comparación de cadenas
- Estructuras condicionales con if
- Bucles con for
- Bucles con While
- Bucles con Until
- Bucles con Foreach
- Archivos
- Funciones
- Windows PowerShell ISE

Habilitar ejecución de scripts

```
Set-ExecutionPolicy Unrestricted
```

Tipos y colecciones

En PowerShell, no es necesario declarar explícitamente el tipo de una variable, infiere el tipo automáticamente al asignar un valor. No obtantes si necesitas definir explícitamente el tipo de una variable, puedes hacerlo usando la notación [tipo].

```
PS C:\> $number = 42
PS C:\> echo $number.GetType()

IsPublic IsSerial Name
-------- -------- ----
True     True     Int32

PS C:\> [int]$a = 4
PS C:\> $a.GetType().Name
Int32
```

Podemos convertir los diferentes tipos por ejemplo:

```
PS C:\> $a=4
PS C:\> $cadena=$a.ToString()
PS C:\> $cadena.GetType().Name
String

#TryParse nos devuelve si es posible
PS C:\> [int]$b
PS C:\> [int]::TryParse($cadena, [ref]$b)
True
PS C:\> echo $b
4
PS C:\> [int]::TryParse("2w", [ref]$b)
False
PS C:\> echo $b
0

#Podemos hacerlo directamente, pero si hay algún error se parará el script
PS C:\> $b = [int]::Parse($cadena)
PS C:\> echo $b
4
```

Los tipos de variables más comunes que puedes encontrar y utilizar en PowerShell son:

- Números enteros **(Integers)**. Un entero es un número sin parte decimal

  ```
  PS C:\> [int]$a = 4
  PS C:\> $a.GetType().Name
  Int32
  ```

- Números de coma flotante **(Floating-Point)**. Los números de coma flotante son números que incluyen una parte decimal

```
PS C:\> $number = 3.14
PS C:\> echo $number.GetType().Name
Double
```

- Cadenas de texto **(Strings)**. Una cadena es una secuencia de caracteres utilizada para representar texto.

```
PS C:\>$cadena = "Hola, Mundo!"
PS C:\> echo $cadena.GetType().Name
String
```

```
PS C:\> $a="cadena"
PS C:\> echo $a
cadena
PS C:\> echo $a.Replace("ca","CC")
CCdena
PS C:\> echo $a.split("a")
c
den

PS C:\> echo $a.Split("a")[0]
c
PS C:\> echo $a
cadena
PS C:\> echo $a.Substring(2,4)
dena
PS C:\> echo $a.Remove(2,4)
ca
PS C:\> echo $a.Contains("a")
True
PS C:\> echo $a.IndexOf("a")
1
PS C:\> echo $a.Replace("ca","Ca")
Cadena
PS C:\> echo $a.Equals("cadena")
True
PS C:\> echo $a.Length
6
PS C:\> echo $a.Contains("an")
False
PS C:\> echo $a.ToLower()
cadena
PS C:\> echo $a.ToUpper()
CADENA
PS C:\> $a.Replace("ca","  ").Replace("na","  ")
 de
#Trim() elimina espacios al final y al princio
PS C:\> $a.Replace("ca","  ").Replace("na","  ").Trim()
de
```

Format: Formatea una cadena usando marcadores de posición {n} que se reemplazan por los valores proporcionados.

```
PS C:\> $nombre = "Tutankamón"
PS C:\> $edad = 3394
PS C:\> $cadena = [string]::Format("Nombre: {0}, Edad: {1}", $nombre, $edad)
```

Redondeo a Dos Decimales

```
PS C:\> $pi = 3.141592
PS C:\> $out = "pi : {0:F2}" -f $pi
PS C:\> Write-Output $out
pi : 3,14
```

Formato de Moneda

```
PS C:\> $pi = 3.141592
PS C:\> $out = "pi : {0:C2}" -f $pi
PS C:\> Write-Output $out
pi : 3,14 €
```

Formato Científico

```
PS C:\> $pi = 3.141592
PS C:\> $out = "pi : {0:E2}" -f $pi
PS C:\> Write-Output $out
pi : 3,14E+000
```

Formato de Porcentaje

```
PS C:\> $pi = 3.141592
PS C:\> $out = "pi : {0:P2}" -f $pi
PS C:\> Write-Output $out
pi : 314,16 %
```

Formato de Longitud Fija (Relleno a la Izquierda)

```
PS C:\> $pi = 3.141592
PS C:\> $out = "pi : {0,10}" -f $pi
PS C:\> Write-Output $out
pi :   3,141592
```

Formato de Longitud Fija (Relleno a la Derecha)

```
PS C:\> $pi = 3.141592
PS C:\> $out = "pi : {0,-10}" -f $pi
PS C:\> Write-Output $out
pi : 3,141592
```

Formato con Cero Padding (Relleno con ceros)

```
PS C:\> $pi = 3.141592
PS C:\> $out = "pi : {0:000.000}" -f $pi
PS C:\> Write-Output $out
pi : 003,142

PS C:\> $out = "pi : {0:00}" -f $pi
PS C:\> Write-Output $out
pi : 03
```

- Booleanos (**Booleans**). Un valor booleano puede ser True o False.

```
PS C:\>$B=$true
PS C:\> echo $B.GetType().Name
Boolean
```

- Arreglos **(Arrays)**. Un arreglo es una colección de elementos, que pueden ser de cualquier tipo de dato. Los elementos están indexados comenzando desde cero.

```
PS C:> $A = @(0)*4
PS C:> echo $A
0
0
0
0

PS C:\> $numbers = @(1, 2, 3, 4, 5)
PS C:\> echo $numbers.GetType().Name
Object[]

PS C:\> echo $numbers[0]
1

PS C:\> echo $numbers[0].GetType().Name
Int32

PS C:> echo $numbers.Length
5
PS C:\>  echo $numbers.Count
5

echo $numbers.Contains(2)
True

#podemos añadir nuevos elementos
PS C:\>  $numbers+="hola"
PS C:\>  echo $numbers
1
2
3
4
5
hola
PS C:\> echo $numbers.Length
6
```

Arrays multidimimensionales

```
PS C:> $XY = @(@(1, 2), @(3, 4))
PS C:>  echo $XY.Length
2

PS C:>  echo $XY[0] 1
2

PS C:>  echo $XY[0].Length
2

PS C:>  echo $XY[0][0]
1
```

- Hash Tables **(Tablas hash)**. Una tabla hash es una colección de pares clave-valor. Las claves deben ser únicas dentro de la tabla.

```
PS C:\> $person = @{
 Name = "Tutankamón"
 Age = 3358
 City = "Egipto"
}

PS C:\> echo $person.Name
Tutankamón

#Agregar una nueva clave
PS C:\> $person.Estado = "momificado"
PS C:\>  $person

Name                           Value
----                           -----
Estado                         momificado
Name                           Tutankamón
Age                            3358
City                           Egipto

#eliminar la clave
PS C:\> $person.Remove("Age")
PS C:\> $person

Name                           Value
----                           -----
Estado                         momificado
Name                           Tutankamón
City                           Egipto
```

- **ScriptBlocks**. Un ScriptBlock es un bloque de código que se puede ejecutar más tarde o pasar como un parámetro.

```
PS C:\>  $script = { param($name) "Hola, $name!" }
PS C:\> & $script "Tutankamón"
Hola, Tutankamón!
```

Operadores

```
PS C:\> $a=11
PS C:\> $b=5
PS C:\> $c=$a-$b     # resta
PS C:\> echo $c
6
PS C:\> $c=$a+$b     # suma
PS C:\> echo $c
16
PS C:\> $c=$a*$b     # multiplicación
PS C:\> echo $c
55
PS C:\> $c=$a/$b     # división
PS C:\> echo $c
2,2
PS C:\> $c=$a%$b     # resto o modulo
PS C:\> echo $c
1
```

Operadores de asignación

```
PS C:\> $a=6
PS C:\> $a+=2 ; echo $a   # $a=$a+2
8
PS C:\> $a-=2 ; echo $a   # $a=$a-2
6
PS C:\> $a++  ; echo $a   # $a=$a+1
7
PS C:\> $a--  ; echo $a   # $a=$a-1
6
PS C:\> $a/=2 ; echo $a   # $a=$a/2
3
PS C:\> $a*=3 ; echo $a   # $a=$a*3
9
PS C:\> $a%=3 ; echo $a   # $a=$a%3
0
```

- **Números aleatorios**

```
Get-Random # numero aletorio
Get-Random -Minimum 1 -Maximum 10
```

- **Otras conversiones**

```
#pasar a binario
[Convert]::ToString($decimal, 2)

$a="$pwd"

#Podemos ejecutar un texto como si fuese un comando &
$a="notepad"
&$a
```

Argumentos de entrada

Argumentos de entrada Read-Host

```
PS C:\> cat read_host.ps1
echo "Dame un entero"
$Name=Read-Host
echo "Sin pasar a int ($Name*$Name) = "
echo $($Name*$Name)
echo ""
$a = [int] $Name
echo "Pasando a int ($Name*$Name) = "
echo $($a*$a)
echo ""

PS C:\> .\read_host.ps1
Dame un entero
2
Sin pasar a int (2*2) =
22

Pasando a int (2*2) =
4
```

Argumentos de entrada Read-Host argst.ps1

```
PS C:\> cat .\argst.ps1
echo "tenemos $args parámetros de entrada"
echo "El argumento 0: $($args[0])"
echo "El argumento 1: $($args[1])"

PS C:\> .\argst.ps1 1 2 3
tenemos 1 2 3 parámetros de entrada
El argumento 0: 1
El argumento 1: 2
```

```
PS C:\> cat .\args.ps1
for($i=0;$i -lt $args.Length;$i++)
{
  $salida = $args[$i]
  Write-Output "i = $salida"
}

PS C:\> .\args.ps1 1 dos tres
      i = 1
      i = dos
      i = tres
```

```
PS C:\> cat .\argst.ps1
foreach ($i in $args)
{
echo $i
}

PS C:\> .\argst.ps1 1 dos tres 1 dos tres
1
dos
tres
1
dos
tres
```

Argumentos de entrada Read-Host param.ps1

```
PS C:\> cat .\param.ps1
param (
    [string]$Nombre,
    [int]$Nacimiento
)

# Calcular la edad
$Edad = (Get-Date).Year - $Nacimiento

Write-Output "Hola $Nombre, Tienes $Edad"

PS C:\> .\param.ps1 -Nombre Tutankamón -Nacimiento -1334
Hola Tutankamón, Tienes 3358

PS C:\> .\param.ps1 -Nombre Tutankamón
Hola Tutankamón, Tienes 2024

PS C:\> .\param.ps1
Hola , Tienes 2024
```

En el caso de que queramos darle un valor por defecto cambiamos:

```
param (
    [string]$Nombre = "Tutankamón",
    [int]$Nacimiento = "-1334"
)
```

y cuando lo ejecutamos obtenemos:

```
PS C:\> .\param_def.ps1 -Name Nefertiti -Nacimiento -1370
Hola Tutankamón, Tienes 3394

PS C:\Users\Administrador\powershell> .\param_def.ps1 -Name Nefertiti
Hola Tutankamón, Tienes 3358

PS C:\> .\param_def.ps1
Hola Tutankamón, Tienes 3358
```

Podemos completar la declaracion de un parámetro en un script de PowerShell utilizando **[Parameter()]**, el argumento **Mandatory=$true** dentro de la directiva parameter indica que el parámetro es obligatorio. Si el usuario no proporciona un valor para este parámetro al ejecutar el script, PowerShell solicitará que se ingrese uno. Si se omite, PowerShell genera un error pidiendo al usuario que proporcione el valor. **HelpMessage** especifica un mensaje de ayuda que se mostrará al usuario si el parámetro obligatorio no se proporciona.

```
PS C:\> cat .\param.ps1
param (
    [Parameter(Mandatory=$true, HelpMessage="Cómo te llamas?")]
    [string]$Nombre,

    [Parameter(Mandatory=$true, HelpMessage="¿En qué año naciste?")]
    [int]$Nacimiento
)

# Calcular la edad
$Edad = (Get-Date).Year - $Nacimiento

Write-Output "Hola $Nombre, Tienes $Edad"

PS C:\> .\param.ps1 -Nombre Tutankamón -Nacimiento -1334
Hola Tutankamón, Tienes 3358

PS C:\> .\param.ps1 -Nombre Tutankamón
cmdlet param.ps1 en la posición 1 de la canalización de comandos
Proporcione valores para los parámetros siguientes:
(Escriba !? para obtener Ayuda).
Nacimiento: -1334
Hola Tutankamón, Tienes 3358
PS C:\> .\param.ps1

cmdlet param.ps1 en la posición 1 de la canalización de comandos
Proporcione valores para los parámetros siguientes:
(Escriba !? para obtener Ayuda).
Nombre: !?
Cómo te llamas?
Nombre: Tutankamón
Nacimiento: -1334
Hola Tutankamón, Tienes 3358
```

Evaluación de condiciones

```
7 -eq 7 #True
7 -eq 8 #False
3 -gt 2 #True
```

Operadores lógicos

```
(5 -gt 1) -And (5 -lt 10)  #True
(5 -gt 1) -And (5 -lt 10)  #True
(5 -gt 1) -Or (5 -lt 1)    #True
(5 -gt 1) -Xor (5 -lt 1)   #True
-Not (5 -lt 1)             #True
```

Comparación de cadenas

```
"hola" -eq "hola"    #True
"hola" -eq "hoa"     #False
"hola" -ne "hoa"     #True
"hola" -ne "hola"    #False
```

Estructuras condicionales con if

```
$numero = -10
If ($numero -gt 0) {
  echo "$numero es mayor que 0"
  }
If ($numero -gt 0) {
  echo "$numero es mayor que 0"
  }else{
  echo "$numero es negativo"
  }

 #-10 es negativo
```

Bucles con for

```
for ($i=0;$i -lt 5;$i++){
   Write-Host $i
  }

# 0
# 1
# 2
# 3
# 4
```

Bucles con While

```
$i = 0
While ($i -lt 5) {
echo $i
$i +=1
}

# 0
# 1
# 2
# 3
# 4

$i = 0
do {
echo $i
$i +=1
} While ($i -lt 5)

# 0
# 1
# 2
# 3
# 4
```

Bucles con Until

```
$i = 0;
do {
   Write-Host $i
$i +=1
}
until ($i -ge 5)

# 0
# 1
# 2
# 3
# 4
```

Bucles con Foreach

```
$items = 0..4
$items | ForEach-Object {
  $i="$_"
  Write-Host $i
}

 # 0
 # 1
 # 2
 # 3
 # 4



foreach ($num in 1,2,"hola",4,5) {
echo $num
}
```

Archivos

```
echo "usuario,grupo" > usuarios.csv
echo "user01,group01" >> usuarios.csv
echo "user02,group02" >> usuarios.csv
echo "user03,group03" >> usuarios.csv
echo "user04,group04" >> usuarios.csv
Test-Path usuarios.csv  #True ver si existe

#Leer el archivo linea a linea
foreach ($i in get-content usuarios.csv){
  echo $i
  }

#Podemos importarlo desde un csv
$A = Import-Csv -Path usuarios.csv
echo $A.usuario

#Podemos recorrer los valores
foreach ($i in  $A){
  $u=$i.usuario ; echo "usuario = $u"
  }
```

Funciones

```
function foo($a, $b, $c) {
  "a: $a; b: $b; c: $c"
  }
foo 1 3 5
# a: 1; b: 3; c: 5
```

Ejemplo de función con recurrencia:

```
function Get-Factorial {
   param (
       [int]$n
   )

   if ($n -le 1) {
       return 1
   } else {
       return $n * (Get-Factorial -n ($n - 1))
   }
}

$number = 5
$result = Get-Factorial -n 5
Write-Output "El factorial de $number es $result"

#El factorial de 5 es 120
```

Windows PowerShell ISE