Sistemas Operativos en Red

UD 04. Introducción a PowerShell - Parte 2

Autor: Sergi García, Gloria Muñoz

Actualizado Abril 2024

Licencia

Nomenclatura

A lo largo de este tema se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

📖 **Importante**

❕ **Atención**

💬 **Interesante**

ÍNDICE

Unidad 04. Introducción a PowerShell - Parte 2

1. Uso del bucle While en PowerShell

Un bucle while evalúa una condición antes de ejecutar el bloque de código que contiene. Si la condición es verdadera, el bloque de código se ejecuta. Después de cada ejecución del bloque, la condición se evalúa nuevamente. Esto continúa hasta que la condición se evalúa como falsa.

1.1 Ejemplos de Bucles While

Contador Simple

```
$contador = 1
while ($contador -le 5) {
    Write-Host "Contador es: $contador"
    $contador++
}
```

Explicación:

Este script inicializa un contador en 1 y lo incrementa en cada iteración del bucle while hasta que el contador supera 5. En cada iteración, imprime el valor actual del contador.

Simulación de ejecución:

Iteración 1: Contador es 1

Iteración 2: Contador es 2

Iteración 3: Contador es 3

Iteración 4: Contador es 4

Iteración 5: Contador es 5

Ejemplo 2: Espera hasta que un archivo exista

```
while (-not (Test-Path "C:\temp\miarchivo.txt")) {
    Write-Host "Esperando el archivo..."
    Start-Sleep -Seconds 2
}
Write-Host "Archivo encontrado."
```

Explicación:

Este bucle espera hasta que un archivo específico exista en la ubicación dada. Utiliza Test-Path para verificar la existencia del archivo. Si no existe, el script duerme durante dos segundos antes de verificar nuevamente.

Simulación de ejecución:

Iteración 1: Archivo no encontrado, espera 2 segundos.

Iteración 2: Archivo no encontrado, espera 2 segundos.

Iteración 3: Archivo encontrado, sale del bucle.

Menú de Opciones

```
do {
    Write-Host "1. Opción A"
    Write-Host "2. Opción B"
    Write-Host "3. Salir"
    $opción = Read-Host "Seleccione una opción"
} while ($opción -ne '3')
```

Explicación:

Este bucle muestra un menú de opciones y permite al usuario seleccionar una opción. El bucle continúa hasta que el usuario selecciona la opción para salir (3).

Simulación de ejecución:

Usuario selecciona 1, muestra menú de nuevo.

Usuario selecciona 2, muestra menú de nuevo.

Usuario selecciona 3, sale del bucle.

Validación de Entrada

```
do {
    $edad = Read-Host "Ingrese su edad (debe ser un número)"
} while ($edad -notmatch '^\d+$')
Write-Host "Edad válida: $edad años."
```

Explicación:

Solicita la edad del usuario y asegura que sea un número mediante una expresión regular. El bucle persiste hasta que se introduce una entrada válida.

Simulación de ejecución:

Usuario introduce "veinte", se solicita de nuevo.

Usuario introduce "30", bucle termina con "Edad válida: 30 años."

Loop de Intentos

```
$intentos = 0
while ($intentos -lt 3) {
    $usuario = Read-Host "Usuario"
    $contraseña = Read-Host "Contraseña"
    if ($usuario -eq "admin" -and $contraseña -eq "123") {
        Write-Host "Acceso concedido"
        break
    } else {
        Write-Host "Acceso denegado. Intente de nuevo."
    }
    $intentos++
}
```

Explicación:

Este script permite al usuario intentar iniciar sesión hasta tres veces. Si las credenciales son correctas, el bucle se interrumpe prematuramente.

Simulación de ejecución:

Intento 1: Usuario "user", Contraseña "pass", Acceso denegado.

Intento 2: Usuario "admin", Contraseña "123", Acceso concedido, bucle termina.

Suma de números

```
$suma = 0
$numero = 1
while ($numero -ne 0) {
    $numero = Read-Host "Introduce un número (0 para salir)"
    $suma += $numero
}
Write-Host "Suma total: $suma"
```

Explicación:

Permite al usuario introducir números continuamente, los cuales se suman a un total. El bucle termina cuando el usuario introduce 0.

Simulación de ejecución:

Usuario introduce 5, suma = 5.

Usuario introduce 15, suma = 20.

Usuario introduce 0, muestra "Suma total: 20", bucle termina.

Reintentar operación

```
$completado = $false
while (-not $completado) {
    try {
        # Intento de operación que puede fallar
        Write-Host "Intentando operación..."
        # Simula que la operación fue exitosa modificando la variable a
$true
        $completado = $true
    } catch {
        Write-Host "Error en la operación. Reintentando..."
        Start-Sleep -Seconds 1
    }
}
Write-Host "Operación completada con éxito."
```

Explicación:

Intenta realizar una operación que puede fallar, manejando errores con un bloque try-catch. Si la operación falla, reintentará después de una pausa de un segundo.

Simulación de ejecución:

Intento falla, reintentando después de 1 segundo.

Intento exitoso, "Operación completada con éxito."

Ejemplo 8: Leer y Procesar Datos hasta un Marcador

```
$linea = ""
while (($linea = Read-Host "Ingrese texto (FIN para terminar)") -ne
"FIN") {
    Write-Host "Procesando: $linea"
}
```

Explicación:

Permite al usuario ingresar líneas de texto hasta que escriba "FIN". Cada línea se procesa y se muestra de inmediato.

Simulación de ejecución:

- Usuario introduce "Hola", muestra "Procesando: Hola".
- Usuario introduce "Mundo", muestra "Procesando: Mundo".
- Usuario introduce "FIN", bucle termina.

Bucle infinito controlado

```
while ($true) {
    $entrada = Read-Host "Escriba 'salir' para terminar"
    if ($entrada -eq "salir") {
        break
    } else {
        Write-Host "Usted escribió: $entrada"
    }
}
```

Explicación:

Crea un bucle infinito que sólo termina cuando el usuario escribe "salir". De lo contrario, imprime lo que el usuario escribió.

Simulación de Ejecución:

Usuario escribe "hello", muestra "Usted escribió: hello".

Usuario escribe "salir", bucle termina.

Conteo Regresivo

```
$inicio = 10
while ($inicio -gt 0) {
    Write-Host $inicio
    Start-Sleep -Seconds 1
    $inicio--
}
```

```
Write-Host "¡Tiempo terminado!"
```

Explicación:

Realiza un conteo regresivo desde 10 hasta 1, mostrando cada número y esperando un segundo entre cada uno.

Simulación de ejecución:

Comienza en 10, cuenta regresivamente hasta 1.

Cada número se muestra cada segundo.

Muestra "¡Tiempo terminado!" al final.

2. Uso del bucle For en PowerShell

El bucle for en PowerShell se utiliza generalmente para iterar a través de secuencias o ejecutar un bloque de código de manera repetida durante un número definido de veces. La sintaxis básica de un bucle for es la siguiente:

```
for ($inicio; $condicion; $incremento) {
    # Código a ejecutar
}
```

Donde:

- **$inicio:** Inicialización de una variable, se ejecuta antes del primer ciclo.
- **$condicion:** Condición que debe ser verdadera para que el bucle continúe.
- **$incremento:** Incremento o actualización que se ejecuta al final de cada ciclo del bucle.

2.1 Ejemplos de Bucles For

Contar de 1 a 10

```
for ($i = 1; $i -le 10; $i++) {
    Write-Host "Número: $i"
}
```

Explicación:

Este script usa un bucle for para contar del 1 al 10. En cada iteración, incrementa el contador $i y muestra su valor.

Simulación de ejecución:

Número: 1

Número: 2

Número: 3

...

Número: 10

Mostrar los caracteres de una cadena

```
$cadena = "Hello"
for ($i = 0; $i -lt $cadena.Length; $i++) {
    Write-Host $cadena[$i]
}
```

Explicación:
Itera sobre cada carácter de la cadena "Hello" y los imprime uno por uno.
Simulación de ejecución:
H
e
l
l
o

Iterar sobre un Array

```
$colores = @("Rojo", "Verde", "Azul")
for ($i = 0; $i -lt $colores.Length; $i++) {
    Write-Host "Color $i: $($colores[$i])"
}
```

Explicación:
Recorre un array de colores e imprime cada color con su índice.
Simulación de ejecución:
Color 0: Rojo
Color 1: Verde
Color 2: Azul

Bucle For con decremento

```
for ($i = 10; $i -ge 1; $i--) {
    Write-Host "Cuenta regresiva: $i"
}
```

Explicación:
Realiza una cuenta regresiva desde 10 hasta 1, decrementando el valor de $i en cada iteración.
Simulación de ejecución:
Cuenta regresiva: 10
Cuenta regresiva: 9
...
Cuenta regresiva: 1

Tabla de multiplicar

```
$n = 5
for ($i = 1; $i -le 10; $i++) {
    $producto = $n * $i
    Write-Host "$n x $i = $producto"
}
```

Explicación:

Genera la tabla de multiplicar del 5, multiplicando el número 5 por los números del 1 al 10.

Simulación de ejecución:

5 x 1 = 5

5 x 2 = 10

...

5 x 10 = 50

Sumar números de un Array

```
$numeros = 1..10
$suma = 0
for ($i = 0; $i -lt $numeros.Length; $i++) {
    $suma += $numeros[$i]
}
Write-Host "Suma total: $suma"
```

Explicación:

Calcula la suma de los números del 1 al 10 almacenados en un array.

Simulación de Ejecución:

Suma total: 55

Bucle For con condiciones múltiples

```
for ($i = 1, $j = 10; $i -le 10 -and $j -ge 1; $i++, $j--) {
    Write-Host "i = $i, j = $j"
}
```

Explicación:

Utiliza dos variables en el bucle for. $i se incrementa de 1 a 10 y $j se decrementa de 10 a 1 simultáneamente.

Simulación de ejecución:

i = 1, j = 10

i = 2, j = 9

...

i = 10, j = 1

Ejecutar un Comando para cada elemento de un Array

```
$commands = @('date', 'whoami', 'pwd')
for ($i = 0; $i -lt $commands.Length; $i++) {
    Write-Host "Ejecutando: $($commands[$i])"
    Invoke-Expression $commands[$i]
}
```

Explicación:

Itera sobre un array de comandos de sistema y ejecuta cada uno de ellos, mostrando el comando que se está ejecutando.

Simulación de ejecución:

(Ejecuta los comandos y muestra la fecha actual, el usuario y el directorio de trabajo actual)

Bucle For sin bloque de inicialización

```
$i = 1
for (; $i -le 5; $i++) {
    Write-Host "Valor de i: $i"
}
```

Explicación:

Comienza con $i inicializado fuera del bucle y lo incrementa hasta que supera 5. Muestra el valor de $i en cada iteración.

Simulación de ejecución:

Valor de i: 1

Valor de i: 2

...

Valor de i: 5

Uso de For para procesar entrada de usuario

```
$continuar = $true
for (; $continuar; ) {
    $input = Read-Host "Escriba 'salir' para terminar, cualquier otra
cosa para continuar"
    $continuar = ($input -ne 'salir')
    if ($continuar) {
        Write-Host "Usted escribió: $input"
    }
}
```

Explicación:

Pide repetidamente al usuario que introduzca datos hasta que escriba 'salir'. Si el usuario no escribe 'salir', muestra lo que escribió.

Simulación de ejecución:

Usuario escribe "hola", muestra "Usted escribió: hola"

Usuario escribe "salir", termina el bucle

3. USO DEL BUCLE FOREACH EN POWERSHELL

El bucle foreach en PowerShell es extremadamente útil para iterar sobre colecciones de elementos, como arrays, listas, o resultados de comandos. Cada iteración del bucle toma un elemento de la colección y realiza operaciones con él. Aquí te presento 10 ejemplos prácticos de cómo usar el bucle foreach, incluyendo explicaciones detalladas y simulaciones de ejecución.

3.1 Ejemplo de uso del bucle Foreach

Impresión de elementos de un Array

```
$nombres = "Ana", "Beto", "Carlos"
foreach ($nombre in $nombres) {
    Write-Host $nombre
}
```

Explicación:

Este script recorre un array de nombres e imprime cada nombre en la consola.

Simulación de ejecución:

Ana

Beto

Carlos

Suma de números en un Array

```
$numeros = 1, 2, 3, 4, 5
$suma = 0
foreach ($numero in $numeros) {
    $suma += $numero
}
Write-Host "La suma es: $suma"
```

Explicación:

Este ejemplo suma todos los números en un array usando un bucle foreach.

Simulación de ejecución:

Suma total: 15

Ejemplo 3: Transformación de rexto

```
$textos = "hola", "mundo", "powershell"
foreach ($texto in $textos) {
    $textoMayuscula = $texto.ToUpper()
    Write-Host $textoMayuscula
}
```

Explicación:

Convierte cada elemento de un array de cadenas a mayúsculas y las imprime.

Simulación de ejecución:

HOLA

MUNDO

POWERSHELL

Filtrado de datos

```
$edades = 18, 22, 16, 29, 14
foreach ($edad in $edades) {
    if ($edad -ge 18) {
        Write-Host "Mayor de edad: $edad"
    }
}
```

Explicación:

Filtra y muestra solo las edades que son mayores o iguales a 18 años.

Simulación de ejecución:

Mayor de edad: 18

Mayor de edad: 22

Mayor de edad: 29

Procesamiento de objetos

```
$personas = @(
    @{Nombre="Ana"; Edad=20},
    @{Nombre="Beto"; Edad=17},
    @{Nombre="Carlos"; Edad=22}
)
foreach ($persona in $personas) {
    Write-Host "$($persona.Nombre) tiene $($persona.Edad) años"
}
```

Explicación:

Itera sobre una colección de diccionarios y muestra información sobre cada persona.

Simulación de ejecución:

Ana tiene 20 años

Beto tiene 17 años

Carlos tiene 22 años

Manipulación de archivos

```
$archivos = Get-ChildItem "C:\Archivos"
foreach ($archivo in $archivos) {
    Write-Host "Procesando archivo: $($archivo.Name)"
}
```

Explicación:

Recorre todos los archivos en un directorio específico e imprime el nombre de cada archivo.

Simulación de ejecución:

Procesando archivo: documento1.txt

Procesando archivo: imagen2.jpg

Procesando archivo: presentacion.pptx

Aplicación de Descuentos

```
$productos = @(100, 200, 150)
$descuento = 0.1  # 10% de descuento
foreach ($precio in $productos) {
    $precioConDescuento = $precio - ($precio * $descuento)
    Write-Host "Precio con descuento: $precioConDescuento"
}
```

Explicación:

Aplica un descuento del 10% a cada precio de producto en la lista.

Simulación de ejecución:

Precio con descuento: 90

Precio con descuento: 180

Precio con descuento: 135

Validación de entradas

```
$entradas = "email@example.com", "notanemail", "another@example.com"
foreach ($entrada in $entradas) {
    if ($entrada -match '^\S+@\S+\.\S+$') {
        Write-Host "$entrada es un email válido"
    } else {
        Write-Host "$entrada no es un email válido"
    }
}
```

Explicación:

Verifica si cada entrada en un array es un correo electrónico válido según una expresión regular.

Simulación de ejecución:

email@example.com es un email válido

notanemail no es un email válido

another@example.com es un email válido

Conversión de unidades

```
$kilometros = 1, 5, 10
foreach ($km in $kilometros) {
    $millas = $km * 0.621371
    Write-Host "$km km son equivalentes a $millas millas"
}
```

Explicación:

Convierte valores en kilómetros a millas utilizando una relación de conversión fija.

Simulación de ejecución:

1 km son equivalentes a 0.621371 millas

5 km son equivalentes a 3.106855 millas

10 km son equivalentes a 6.21371 millas

Creación de Directorios

```
$carpetas = "Proyectos", "Documentos", "Imágenes"
foreach ($carpeta in $carpetas) {
    New-Item -Path "C:\Usuarios\Usuario\$carpeta" -Type Directory
    Write-Host "Creada carpeta: $carpeta"
}
```

Explicación:

Crea directorios basados en los nombres especificados en un array de cadenas.

Simulación de ejecución:

Creada carpeta: Proyectos

Creada carpeta: Documentos

Creada carpeta: Imágenes

4. USO DE FUNCIONES EN POWERSHELL

Las funciones en PowerShell permiten agrupar código para realizar tareas específicas, lo cual facilita la reutilización y mejora la legibilidad del código. Se declaran con la palabra clave function seguida del nombre de la función y un bloque de código entre llaves {}. Pueden aceptar parámetros de entrada y devolver valores.

4.1 Ejemplos de Funciones

Función Simple

```
function Saludar {
    Write-Host "¡Hola, mundo!"
}
Saludar
```

Explicación:

Esta función no toma parámetros y simplemente imprime "¡Hola, mundo!" cada vez que se llama.

Simulación de ejecución:

Output: ¡Hola, mundo!

Función con Parámetros

```
function SaludarPersona {
    param($nombre)
    Write-Host "¡Hola, $nombre!"
}
SaludarPersona -nombre "Pedro"
```

Explicación:

Esta función toma un parámetro $nombre y utiliza este valor para personalizar un mensaje de

saludo.

Simulación de ejecución:

Output: ¡Hola, Pedro!

Función que devuelve valor

```
function Sumar {
    param($num1, $num2)
    return $num1 + $num2
}
$resultado = Sumar -num1 5 -num2 3
Write-Host "Resultado: $resultado"
```

Explicación:

Esta función toma dos números como parámetros y devuelve su suma. El resultado se almacena en una variable y se muestra.

Simulación de ejecución:

Output: Resultado: 8

Función con parámetros opcionales

```
function Saludar {
    param($nombre = "Invitado")
    Write-Host "¡Bienvenido, $nombre!"
}
Saludar
Saludar -nombre "Ana"
```

Explicación:

Esta función tiene un parámetro $nombre que es opcional y tiene un valor por defecto de "Invitado". Muestra un saludo que cambia según el nombre proporcionado.

Simulación de ejecución:

Primera llamada sin parámetro: ¡Bienvenido, Invitado!

Segunda llamada con parámetro: ¡Bienvenido, Ana!

Función con validación de parámetros

```
function Set-Edad {
    param([ValidateRange(1,150)][int]$edad)
    Write-Host "Edad configurada a $edad años."
}
Set-Edad -edad 25
```

Explicación:

Esta función valida que el valor de $edad esté entre 1 y 150. Si el valor está dentro del rango, se configura y muestra.

Simulación de ejecución:

Output: Edad configurada a 25 años.

Función que Itera sobre un Array

```
function MostrarColores {
    param($colores)
    foreach ($color in $colores) {
        Write-Host "Color: $color"
    }
}
$misColores = @("Rojo", "Verde", "Azul")
MostrarColores -colores $misColores
```

Explicación:

Esta función recibe un array de colores como parámetro y utiliza un bucle foreach para imprimir cada color.

Simulación de ejecución:

Output:

Color: Rojo

Color: Verde

Color: Azul

Función con múltiples bloques catch

```
function Dividir {
    param($num1, $num2)
    try {
        $resultado = $num1 / $num2
        Write-Host "Resultado: $resultado"
    } catch [System.DivideByZeroException] {
        Write-Host "Error: No se puede dividir por cero."
    } catch {
        Write-Host "Error desconocido."
    }
}
Dividir -num1 10 -num2 0
```

Explicación:

Esta función intenta dividir dos números y maneja posibles errores, incluyendo la división por cero.

Simulación de ejecución:

Output: Error: No se puede dividir por cero.

Función con confirmación

```
function EliminarArchivo {
    param($ruta)
    $confirm = Read-Host "¿Está seguro que desea eliminar el archivo? (s/n)"
    if ($confirm -eq 's') {
        Remove-Item $ruta -Force
        Write-Host "Archivo eliminado."
    } else {
        Write-Host "Operación cancelada."
    }
}
EliminarArchivo -ruta "C:\temp\miarchivo.txt"
```

Explicación:

Esta función solicita confirmación antes de eliminar un archivo. Si el usuario confirma, el archivo se elimina; de lo contrario, la operación se cancela.

Simulación de ejecución:

Input: s

Output: Archivo eliminado.

Función con procesamiento pipeline

```
function Get-NumerosPares {
    param([int[]]$numeros)
    process {
        foreach ($num in $numeros) {
            if ($num % 2 -eq 0) {
                Write-Output $num
            }
        }
    }
}
$array = 1..10
$pares = $array | Get-NumerosPares
Write-Host "Números pares: $pares"
```

Explicación:

Esta función filtra y devuelve solo los números pares de un array de números. Está diseñada para ser utilizada en un pipeline.

Simulación de ejecución:

Output: Números pares: 2 4 6 8 10

Función con parámetros de entrada por referencia

```
function DuplicarNumero {
    param([ref]$numero)
    $numero.Value *= 2
}
$miNumero = 10
DuplicarNumero ([ref]$miNumero)
Write-Host "Número duplicado: $miNumero"
```

Explicación:

Esta función toma un número como referencia y duplica su valor. El cambio afecta directamente a la variable original.

Simulación de ejecución:

Output: Número duplicado: 20

5. EJERCICIOS RESUELTOS

Ejercicio 1: Promedio de Notas

Desarrolla un script en PowerShell que permita al usuario ingresar las notas de un estudiante hasta que decida detenerse e imprima el promedio de estas notas.

Solución:

```
function CalcularPromedio {
    $notas = @()
    do {
        $nota = Read-Host "Introduce una nota (o 'fin' para terminar)"
        if ($nota -ne 'fin') {
            $notas += [double]$nota
        }
    } while ($nota -ne 'fin')

    $promedio = ($notas | Measure-Object -Average).Average
    Write-Host "El promedio de las notas es: $promedio"
}

CalcularPromedio
```

Explicación:

Este script utiliza una función CalcularPromedio que pide al usuario ingresar notas continuamente hasta que ingrese 'fin'. Las notas se acumulan en un array y luego se calcula el promedio utilizando Measure-Object. Finalmente, se muestra el promedio de las notas. Este ejercicio aplica un bucle do-while para la recogida de datos y Measure-Object para el cálculo del promedio.

Ejercicio 2: Conteo de elementos pares en un Array

Crea un script en PowerShell que defina un array de números y utilice un bucle foreach para contar cuántos números pares hay en el array, mostrando este conteo al final.

Solución:

```
function ContarPares {
    $numeros = 1..20
    $conteoPares = 0
    foreach ($numero in $numeros) {
        if ($numero % 2 -eq 0) {
            $conteoPares++
        }
    }
    Write-Host "Hay $conteoPares números pares en el array."
}

ContarPares
```

Explicación:

En este script, se define un array de números del 1 al 20. Utiliza un bucle foreach para iterar a través de cada número en el array y un condicional if para verificar si el número es par. Si es par, incrementa un contador. Al final del bucle, el script imprime el número total de elementos pares encontrados. Este ejercicio es útil para demostrar cómo trabajar con arrays y realizar conteos condicionales.

Ejercicio 3: Lista de Tareas Pendientes

Escribe un script en PowerShell que permita al usuario introducir una lista de tareas pendientes y luego iterar sobre la lista para imprimirlas numeradas.

Solución:

```
function ListarTareas {
    $tareas = @()
    do {
        $tarea = Read-Host "Introduce una tarea (o 'fin' para
terminar)"
        if ($tarea -ne 'fin') {
            $tareas += $tarea
        }
    } while ($tarea -ne 'fin')

    for ($i = 0; $i -lt $tareas.Length; $i++) {
        Write-Host "$($i + 1). $($tareas[$i])"
    }
}

ListarTareas
```

Explicación:

Este script recopila tareas en un array utilizando un bucle do-while hasta que el usuario escribe 'fin'. Luego, utiliza un bucle for para imprimir cada tarea con su número correspondiente. Este ejercicio muestra cómo combinar bucles do-while y for para la entrada y procesamiento de datos en una lista simple y numerada.

Ejercicio 4: Sistema de Gestión de Inventario

Desarrolla un sistema de gestión de inventario en PowerShell que permita al usuario añadir productos con sus cantidades y precios, actualizar cantidades de productos existentes, y mostrar un resumen del inventario actual. El sistema debe:

- Permitir al usuario ingresar productos hasta que decida terminar la entrada.
- Proporcionar opciones para añadir un nuevo producto o actualizar uno existente.
- Mostrar el inventario completo al final.

Solución:

```
function AñadirProducto($inventario) {
    $nombre = Read-Host "Introduce el nombre del producto"
    $cantidad = Read-Host "Introduce la cantidad del producto"
    $precio = Read-Host "Introduce el precio del producto"
    $inventario[$nombre] = @{Cantidad = $cantidad; Precio = $precio}
}

function ActualizarProducto($inventario) {
    $nombre = Read-Host "Introduce el nombre del producto para
actualizar"
    if ($inventario.ContainsKey($nombre)) {
        $cantidadNueva = Read-Host "Introduce la nueva cantidad del
producto"
        $inventario[$nombre].Cantidad = $cantidadNueva
    } else {
        Write-Host "Producto no encontrado"
    }
}

function MostrarInventario($inventario) {
    Write-Host "Inventario actual:"
    foreach ($item in $inventario.Keys) {
        $producto = $inventario[$item]
        Write-Host "Producto: $item, Cantidad: $($producto.Cantidad),
Precio: $($producto.Precio)"
    }
}

$inventario = @{}
do {
```

```
    Write-Host "1. Añadir producto"
    Write-Host "2. Actualizar producto"
    Write-Host "3. Salir"
    $opcion = Read-Host "Seleccione una opción"
    switch ($opcion) {
        '1' { AñadirProducto $inventario }
        '2' { ActualizarProducto $inventario }
        '3' { MostrarInventario $inventario; break }
        default { Write-Host "Opción no válida" }
    }
} while ($true)
```

Explicación:

Este script utiliza un bucle do-while para procesar las entradas del usuario hasta que decide salir. Las funciones AñadirProducto, ActualizarProducto, y MostrarInventario manejan las tareas específicas, mientras que un hashtable se utiliza para almacenar y gestionar el inventario.

Ejercicio 5: Procesador de Datos del Estudiante

Escribe un script en PowerShell que gestione una lista de estudiantes y sus calificaciones. El script debe permitir al usuario ingresar datos, calcular la calificación media de cada estudiante, y determinar quién tiene la mayor y menor calificación promedio. El sistema debe:

- Permitir la entrada de nombres y una lista de calificaciones para cada estudiante.
- Calcular y mostrar la calificación promedio para cada estudiante.
- Identificar y mostrar las calificaciones promedio más alta y más baja al final.

Solución:

```
function IngresarDatos() {
    $estudiantes = @{}
    $continuar = $true
    while ($continuar) {
        $nombre = Read-Host "Introduce el nombre del estudiante"
        $calificaciones = Read-Host "Introduce las calificaciones
separadas por comas"
        $calificacionesArray = $calificaciones -split ',' |
ForEach-Object { [int]$_ }
        $estudiantes[$nombre] = $calificacionesArray
        $continuar = Read-Host "¿Agregar otro estudiante? (s/n)" -eq
's'
    }
    return $estudiantes
}

function CalcularPromedios($estudiantes) {
    $promedios = @{}
    foreach ($estudiante in $estudiantes.Keys) {
```

```
        $promedio = ($estudiantes[$estudiante] | Measure-Object
-Average).Average
        $promedios[$estudiante] = $promedio
        Write-Host "$estudiante tiene un promedio de $promedio"
    }
    return $promedios
}

function MostrarExtremos($promedios) {
    $maxNombre = ($promedios.GetEnumerator() | Sort-Object Value
-Descending | Select-Object -First 1).Name
    $minNombre = ($promedios.GetEnumerator() | Sort-Object Value
-Ascending | Select-Object -First 1).Name
    Write-Host "La calificación más alta es de $maxNombre con
$($promedios[$maxNombre])"
    Write-Host "La calificación más baja es de $minNombre con
$($promedios[$minNombre])"
}

$estudiantes = IngresarDatos
$promedios = CalcularPromedios $estudiantes
MostrarExtremos $promedios
```

Explicación:

Este script administra las entradas de los estudiantes y sus calificaciones, calcula promedios utilizando bucles foreach, y utiliza funciones para separar la lógica de entrada de datos, cálculo de promedios y visualización de resultados. Utiliza Measure-Object para calcular el promedio de las calificaciones de cada estudiante y determina el estudiante con la calificación más alta y más baja.

Ejercicio 6: Simulador de Descuentos en Tienda

Desarrolla un simulador de descuentos para una tienda en PowerShell. El script debe permitir al usuario introducir múltiples precios de productos y un porcentaje de descuento único. Luego, debe calcular y mostrar el precio después del descuento para cada producto y el total ahorrado. El script debe:

- Solicitar al usuario que introduzca uno o más precios.
- Solicitar un porcentaje de descuento.
- Aplicar el descuento a cada precio y mostrar el nuevo precio.
- Calcular y mostrar el total ahorrado.

Solución:

```
function ObtenerPrecios() {
    $precios = @()
    do {
        $precio = Read-Host "Introduce el precio del producto"
        $precios += [double]$precio
```

```
    } while (Read-Host "¿Agregar otro producto? (s/n)" -eq 's')
    return $precios
}

function AplicarDescuento($precios, $descuento) {
    $totalAhorrado = 0
    foreach ($precio in $precios) {
        $precioDescuento = $precio - ($precio * $descuento / 100)
        Write-Host "Precio original: $precio, Precio con descuento:
$precioDescuento"
        $totalAhorrado += $precio - $precioDescuento
    }
    Write-Host "Total ahorrado: $totalAhorrado"
}

$precios = ObtenerPrecios
$descuento = Read-Host "Introduce el porcentaje de descuento"
AplicarDescuento $precios $descuento
```

Explicación:

Este script recoge múltiples precios de productos y un porcentaje de descuento. Calcula el nuevo precio después del descuento para cada producto y suma el total ahorrado, mostrando ambos en la consola. Utiliza bucles do-while y foreach para manejar múltiples entradas y procesamiento de cada precio.

6. BIBLIOGRAFÍA

https://somebooks.es/scripts-powershell-guia-principiantes/
https://learnxinyminutes.com/docs/es-es/powershell-es/
https://github.com/fleschutz/PowerShell
https://learn.microsoft.com/es-es/powershell/scripting/overview?view=powershell-7.4