

PowerShell®

Apuntes para Profesionales

Chapter 3: Operators

An operator is a character that represents an action. It tells the compiler/interpreter to perform mathematical, relational or logical operation and produce final result. PowerShell interprets its categories accordingly like arithmetic operators perform operations primarily on numbers, bit strings and other data types. Along with the basic operators, PowerShell has a number of special and coding effort (eg. like, -match, -replace, etc).

Section 3.1: Comparison Operators

PowerShell comparison operators are comprised of a leading dash (-) followed by a name (greater, lt, etc...).

Names can be preceded by special characters to modify the behavior of the operator:

- 1 # Case-insensitive Explicit (-eq)
 - 2 # Case-sensitive Explicit (-ceq)
- Case-insensitive is the default if not specified. ("a" -eq "A") same as ("a" -ceq "A")

Simple comparison operators:

```
2 -eq 2 # Equal to (-eq)
2 -ne 4 # Not equal to (-ne)
5 -gt 2 # Greater-than (-gt)
5 -ge 5 # Greater-than or equal to (-ge)
5 -lt 10 # Less-than (-lt)
5 -le 5 # Less-than or equal to (-le)
```

String comparison operators:

```
"MyString" -like "string*" # Match using the wildcard
"mystring" -notlike "string*" # Does not match using the wildcard
"mystring" -match "string*" # Matches a string using regular expressions
"mystring" -notmatch "other*" # Does not match a string using regular expressions
```

Collection comparison operators:

```
"abc" -contains "def" # Returns true when the value is in the array (left)
"abc" -notcontains "def" # Returns true when the value is not in the array (left)
"def" -in "abc", "def" # Returns true when the value is in the array (right)
"def" -notin "abc", "def" # Returns true when the value is not in the array (right)
```

Section 3.2: Arithmetic Operators

```
1 + 2 # Addition
1 - 2 # Subtraction
1 * 2 # Multiplication
1 / 2 # Division
1 % 2 # Modulus
```

Chapter 12: PowerShell Functions

A function is basically a named block of code. When you call the function name, the script block within that function runs. It is a list of PowerShell statements that has a name that you assign. When you run a function, you type the function name. It is a method of saving time when tackling repetitive tasks. PowerShell formats in three parts: the keyword 'function', followed by a Name, finally, the payload containing the script block, which is enclosed by curly/parenthesis style brackets.

Section 12.1: Basic Parameters

A function can be defined with parameters using the param block:

```
function Write-Greeting {
    param (
        [Parameter(Mandatory=$true)]
        [string]$Name,
        [Parameter(Mandatory=$true)]
        [int]$Age
    )
    "Hello $Name, you are $Age years old."
}
```

Or using the simple function syntax:

```
function Write-Greeting ($Name, $Age) {
    "Hello $Name, you are $Age years old."
}
```

Note: Casing parameters is not required in either type of parameter definition.

Simple function syntax (SFS) has very limited capabilities in comparison to the param block. Though you can define parameters to be exposed within the function, you cannot specify Parameter Attributes, unlike Parameter.Attributes, include [SwitchAttribute], with SFS (and this is a non-exhaustive list).

Functions can be invoked with ordered or named parameters.

The order of the parameters on the invocation is matched to the order of the declaration in the function header (default), or can be specified using the Position Parameter Attribute (as shown in the advanced function example above).

```
$greeting = Write-Greeting -Name "John" -Age 42
```

Alternatively, this function can be invoked with named parameters:

```
$greeting = Write-Greeting -Name "Bob" -Age 42
```

Section 12.2: Advanced Function

This is a copy of the advanced function snippet from the PowerShell ISE. Basically this is a template for many things you can use with advanced functions in PowerShell. Key points to note:

- get-help integration - the beginning of the function contains a comment block that is set up to be get-help cmdlet. The function block may be located at the end, if desired.
- cmdletization - function will behave like a cmdlet.

PowerShell® Notes for Professionals

Chapter 23: Sending Email

Parameter	Details
Attachments<String>	Path and file names of files to be attached to the message. Paths and filenames can be piped to Send-MailMessage.
Bcc<String>	Email addresses that receive a copy of an email message but does not appear as a recipient in the message. Enter names (optional) and the email address (required), such as Name: someone@example.com or someone@example.com.
Body<String>	Content of the email message.
BodyAsHtml	Indicates that the content is in HTML format.
Cc<String>	Email addresses that receive a copy of an email message. Enter names (optional) and the email address (required), such as Name: someone@example.com or someone@example.com.
Credential	Specifies a user account that has permission to send message from specified email address. The default is the current user. Enter name (optional) and the email address (required), such as Name: someone@example.com or someone@example.com.
DeliveryNotificationOption	Specifies the delivery notification options for the email message. Multiple values can be specified. Acceptable values: None, OnSuccess, OnFailure, Delay, Never.
Encoding	Encoding for the body and subject. Acceptable values: ASCII, UTF8, UTF7, UTF32, Unicode, BigEndianUnicode. Default: UTF8.
From	Email address from which the mail is sent. Enter names (optional) and the email address (required), such as Name: someone@example.com or someone@example.com.
Port	Alternate port on the SMTP server. The default value is 25. Available from Windows PowerShell 5.0.
Priority	Priority of the email message. Acceptable values: Normal, High, Low.
SmtpServer	Name of the SMTP server that sends the email message. Default value is the value of the \$PSDefaultParameterValues.
Subject	Subject of the email message.
To	Email addresses to which the mail is sent. Enter names (optional) and the email address (required), such as Name: someone@example.com or someone@example.com.
UsesSsl	Uses the Secure Sockets Layer (SSL) protocol to establish a connection to the remote computer to send mail.

A useful technique for Exchange Server administrators is to be able to send email messages via SMTP from PowerShell. Depending on the version of PowerShell installed on your computer or server, there are multiple ways to send emails via PowerShell. There is a native cmdlet option that is simple and easy to use. It uses the cmdlet Send-MailMessage.

Section 23.1: Send-MailMessage with predefined parameters

```
Send-MailMessage -To "bob@contoso.com" -Subject "Email Subject" -Attachments "C:\123456789.txt" -Body "Hello Bob" -BodyAsHtml -SmtpServer "smtp.contoso.com" -Credential (Get-Credential) -Encoding - UTF8 -Port 25
```

PowerShell® Notes for Professionals

Traducido por:

rortegag

100+ páginas

de consejos y trucos profesionales

Contenidos

Acerca de	1
Capítulo 1: Introducción a PowerShell	2
Sección 1.1: Permitir que los scripts almacenados en su máquina se ejecuten sin firmar.....	2
Sección 1.2: Alias y funciones similares	2
Sección 1.3: La canalización: uso de la salida de un cmdlet de PowerShell.....	3
Sección 1.4: Llamada a métodos de bibliotecas .Net.....	4
Sección 1.5: Instalación o configuración.....	4
Sección 1.6: Comentarios.....	5
Sección 1.7: Creación de objetos.....	5
Capítulo 2: Variables en PowerShell	7
Sección 2.1: Variable simple.....	7
Sección 2.2: Arrays	7
Sección 2.3: Asignación de listas de variables múltiples.....	7
Sección 2.4: Ámbito.....	8
Sección 2.5: Eliminar una variable	8
Capítulo 3: Operadores.....	9
Sección 3.1: Operadores de comparación.....	9
Sección 3.2: Operadores aritméticos	9
Sección 3.3: Operadores de asignación.....	10
Sección 3.4: Operadores de redirección.....	10
Sección 3.5: Mezclando tipos de operandos, el tipo del operando de la izquierda dicta el comportamiento.....	11
Sección 3.6: Operadores lógicos.....	11
Sección 3.7: Operadores de manipulación de cadenas de caracteres.....	11
Capítulo 4: Operadores especiales	13
Sección 4.1: Operador de expresión de array	13
Sección 4.2: Operación de llamada.....	13
Sección 4.3: Operador de puntos	13
Capítulo 5: Operaciones básicas con conjuntos.....	14
Sección 5.1: Filtrado: Where-Object / where / ?.....	14
Sección 5.2: Ordenar: Sort-Object / sort.....	14
Sección 5.3: Agrupación: Group-Object / group.....	15
Sección 5.4: Proyectar: Select-Object / select	15
Capítulo 6: Lógica condicional	17
Sección 6.1: if, else y else if.....	17
Sección 6.2: Negación.....	17
Sección 6.3: Abreviatura condicional if	18
Capítulo 7: Bucles.....	19
Sección 7.1: foreach.....	19

Sección 7.2: for	19
Sección 7.3: Método ForEach()	19
Sección 7.4: ForEach-Object	20
Sección 7.5: Continue	21
Sección 7.6: Break	21
Sección 7.7: While	22
Sección 7.8: Do	22
Capítulo 8: Sentencia Switch	24
Sección 8.1: Simple Switch	24
Sección 8.2: Sentencia Switch con parámetro CaseSensitive	24
Sección 8.3: Sentencia Switch con parámetro comodín	24
Sección 8.4: Sentencia Switch con parámetro de archivo	25
Sección 8.5: Simple Switch con condición por defecto	25
Sección 8.6: Sentencia Switch con parámetro Regex	25
Sección 8.7: Simple Switch con Break	26
Sección 8.8: Sentencia Switch con parámetro exacto	26
Sección 8.9: Sentencia Switch con expresiones	27
Capítulo 9: Cadenas de texto	28
Sección 9.1: Cadena de texto multilínea	28
Sección 9.2: Here-string	28
Sección 9.3: Concatenar cadenas de texto	28
Sección 9.4: Caracteres especiales	29
Sección 9.5: Creación de una cadena de texto básica	29
Sección 9.6: Formato de cadena de texto	30
Capítulo 10: HashTables	31
Sección 10.1: Acceder a un valor de la tabla hash por clave	31
Sección 10.2: Creación de una tabla hash	31
Sección 10.3: Añadir un par clave-valor a una tabla hash existente	31
Sección 10.4: Eliminar un par clave-valor de una tabla hash existente	32
Sección 10.5: Enumeración mediante claves y pares clave-valor	32
Sección 10.6: Recorrer una tabla hash	33
Capítulo 11: Trabajar con objetos	34
Sección 11.1: Examinar un objeto	34
Sección 11.2: Actualizar objetos	34
Sección 11.3: Crear un nuevo objeto	35
Sección 11.4: Crear instancias de clases genéricas	36
Capítulo 12: Funciones PowerShell	38
Sección 12.1: Parámetros básicos	38
Sección 12.2: Función avanzada	38
Sección 12.3: Parámetros obligatorios	40
Sección 12.4: Validación de parámetros	40

Sección 12.5: Función simple sin parámetros.....	42
Capítulo 13: Clases PowerShell.....	43
Sección 13.1: Listado de constructores disponibles para una clase.....	43
Sección 13.2: Métodos y propiedades.....	44
Sección 13.3: Sobrecarga del constructor.....	44
Sección 13.4: Obtener todos los miembros de una instancia.....	44
Sección 13.5: Plantilla de clase básica.....	45
Sección 13.6: Herencia de clase padre a clase hija.....	46
Capítulo 14: Módulos PowerShell.....	47
Sección 14.1: Crear un manifiesto de módulo.....	47
Sección 14.2: Ejemplo de módulo sencillo.....	47
Sección 14.3: Exportación de una variable desde un módulo.....	48
Sección 14.4: Estructuración de módulos PowerShell.....	48
Sección 14.5: Localización de los módulos.....	48
Sección 14.6: Visibilidad de los miembros del módulo.....	48
Capítulo 15: Perfiles de PowerShell.....	49
Sección 15.1: Crear un perfil básico.....	49
Capítulo 16: Propiedades calculadas.....	50
Sección 16.1: Tamaño del fichero en KB - Propiedades calculadas.....	50
Capítulo 17: Utilizar clases estáticas existentes.....	51
Sección 17.1: Añadir tipos.....	51
Sección 17.2: Uso de la clase Math de .Net.....	51
Sección 17.3: Creación instantánea de un nuevo GUID.....	51
Capítulo 18: Variables incorporadas.....	52
Sección 18.1: \$PSScriptRoot.....	52
Sección 18.2: \$Args.....	52
Sección 18.3: \$PSItem.....	52
Sección 18.4: \$?.....	52
Sección 18.5: \$error.....	52
Capítulo 19: Variables automáticas.....	54
Sección 19.1: \$OFS.....	54
Sección 19.2: \$?.....	54
Sección 19.3: \$null.....	54
Sección 19.4: \$error.....	55
Sección 19.5: \$pid.....	55
Sección 19.6: Valores booleanos.....	55
Sección 19.7: \$_ / \$PSItem.....	56
Sección 19.8: \$PSVersionTable.....	56
Capítulo 20: Variables de entorno.....	57
Sección 20.1: Las variables de entorno de Windows son visibles como una unidad PS llamada Env:.....	57

Sección 20.2: Llamada instantánea de variables de entorno con \$env:.....	57
Capítulo 21: Splatting.....	58
Sección 21.1: Tuberías y Splatting	58
Sección 21.2: Pasar un parámetro Switch mediante splatting.....	58
Sección 21.3: Splatting de la función de nivel superior a una serie de funciones internas.....	59
Sección 21.4: Parámetros de splatting	59
Capítulo 22: “Streams” de PowerShell: depuración, detalle, advertencia, error, salida e Información.....	61
Sección 22.1: Write-Output.....	61
Sección 22.2: Write Preferences	61
Capítulo 23: Envío de correo electrónico.....	63
Sección 23.1: Send-MailMessage con parámetros predefinidos.....	64
Sección 23.2: Enviar mensaje de correo electrónico simple	64
Sección 23.3: SMTPClient - Correo con archivo .txt en el cuerpo del mensaje	64
Capítulo 24: Remotaciones PowerShell	65
Sección 24.1: Conexión a un servidor remoto mediante PowerShell.....	65
Sección 24.2: Ejecutar comandos en un ordenador remoto	65
Sección 24.3: Habilitar PowerShell Remoting.....	66
Sección 24.4: Una buena práctica para limpiar automáticamente las PSSessions.....	67
Capítulo 25: Trabajar con la tubería de PowerShell	69
Sección 25.1: Escribir funciones con el ciclo de vida avanzado.....	69
Sección 25.2: Soporte básico de tuberías en funciones	69
Sección 25.3: Concepto de funcionamiento de la tubería.....	70
Capítulo 26: Trabajos en segundo plano de PowerShell	71
Sección 26.1: Creación de empleo básico	71
Sección 26.2: Gestión básica del trabajo.....	71
Capítulo 27: Comportamiento de retorno en PowerShell	73
Sección 27.1: Salida anticipada	73
Sección 27.2: ¡Te pillé! Devolver en la tubería.....	73
Sección 27.3: Devolver con un valor	73
Sección 27.4: Cómo trabajar con funciones de retorno.....	73
Sección 27.5: ¡Te pillé! Ignorar la salida no deseada.....	75
Capítulo 28: Análisis de CSV.....	76
Sección 28.1: Uso básico de Import-Csv	76
Sección 28.2: Importar desde CSV y asignar las propiedades al tipo correcto.....	76
Capítulo 29: Trabajar con archivos XML.....	77
Sección 29.1: Acceso a un archivo XML	77
Sección 29.2: Creación de un documento XML mediante XmlWriter().....	79
Sección 29.3: Añadir fragmentos de XML al documento XML actual	80
Capítulo 30: Comunicación con las API RESTful.....	85

Sección 30.1: Enviar mensaje a hipChat.....	85
Sección 30.2: Uso de REST con objetos PowerShell para GET y POST de muchos elementos.....	85
Sección 30.3: Utilizar los Webhooks de entrada de Slack.com.....	85
Sección 30.4: Uso de REST con objetos PowerShell para obtener y colocar datos individuales.....	85
Sección 30.5: Uso de REST con PowerShell para eliminar elementos.....	86
Capítulo 31: Consultas SQL de PowerShell	87
Sección 31.1: SQLEjemplo	87
Sección 31.2: SQLQuery	87
Capítulo 32: Expresiones regulares	88
Sección 32.1: Coincidencia simple.....	88
Sección 32.2: Sustituir	90
Sección 32.3: Sustituir texto por un valor dinámico utilizando un MatchEvaluador	90
Sección 32.4: Escape de caracteres especiales.....	91
Sección 32.5: Varias coincidencias	92
Capítulo 33: Alias.....	94
Sección 33.1: Get-Alias.....	94
Sección 33.2: Set-Alias	94
Capítulo 34: Utilizar la barra de progreso	95
Sección 34.1: Uso sencillo de la barra de progreso	95
Sección 34.2: Uso de la barra de progreso interior.....	96
Capítulo 35: Línea de comandos PowerShell.exe	97
Sección 35.1: Ejecutar un comando	97
Sección 35.2: Ejecutar un archivo de script	98
Capítulo 36: Nombres de los cmdlets	99
Sección 36.1: Verbos	99
Sección 36.2: Sustantivos	99
Capítulo 37: Ejecución de ejecutables	100
Sección 37.1: Aplicaciones GUI.....	100
Sección 37.2: Consola de flujos	100
Sección 37.3: Códigos de salida	100
Capítulo 38: Cumplimiento de los requisitos previos de los scripts.....	101
Sección 38.1: Imponer una versión mínima del host de PowerShell.....	101
Sección 38.2: Forzar la ejecución del script como administrador.....	101
Capítulo 39: Utilizar el sistema de ayuda	102
Sección 39.1: Actualización del sistema de ayuda.....	102
Sección 39.2: Uso de Get-Help	102
Sección 39.3: Ver la versión en línea de un tema de ayuda.....	102
Sección 39.4: Ejemplos de visualización	102
Sección 39.5: Ver la página de ayuda completa	102
Sección 39.6: Ver la ayuda de un parámetro específico	103

Capítulo 40: Módulos, scripts y funciones	104
Sección 40.1: Función.....	104
Sección 40.2: Script.....	104
Sección 40.3: Módulo.....	105
Sección 40.4: Funciones avanzadas.....	106
Capítulo 41: Convenciones de denominación	109
Sección 41.1: Funciones.....	109
Capítulo 42: Parámetros comunes	110
Sección 42.1: Parámetro ErrorAction.....	110
Capítulo 43: Conjuntos de parámetros	112
Sección 43.1: Parámetro establecido para imponer el uso de un parámetro cuando se selecciona otro.....	112
Sección 43.2: Parámetro establecido para limitar la combinación de parámetros.....	112
Capítulo 44: Parámetros dinámicos de PowerShell	113
Sección 44.1: "Parámetro dinámico "simple".....	113
Capítulo 45: GUI en PowerShell	115
Sección 45.1: Interfaz gráfica de usuario WPF para el cmdlet Get-Service.....	115
Capítulo 46: Codificación/Decodificación de URL	117
Sección 46.1: Codificar cadena de caracteres de consulta con `[System.Web.HttpUtility]::UrlEncode()`.....	117
Sección 46.2: Inicio rápido: Codificación.....	117
Sección 46.3: Inicio rápido: Descodificación.....	118
Sección 46.4: Codifique la cadena de consulta con `[uri]::EscapeDataString()`.....	118
Sección 46.5: Descodificar URL con `[uri]::UnescapeDataString()`.....	119
Capítulo 47: Tratamiento de errores	121
Sección 47.1: Tipos de error.....	121
Capítulo 48: Gestión de paquetes	123
Sección 48.1: Crear el repositorio de módulos PowerShell por defecto.....	123
Sección 48.2: Buscar un módulo por su nombre.....	123
Sección 48.3: Instalar un módulo por nombre.....	123
Sección 48.4: Desinstalar un módulo mi nombre y versión.....	123
Sección 48.5: Actualizar un módulo por nombre.....	123
Sección 48.6: Buscar un módulo PowerShell mediante un patrón.....	123
Capítulo 49: Comunicación TCP con PowerShell	124
Sección 49.1: Receptor TCP.....	124
Sección 49.2: Emisor TCP.....	125
Capítulo 50: Flujos de trabajos de PowerShell	126
Sección 50.1: Flujos de trabajos con parámetros de entrada.....	126
Sección 50.2: Ejemplo de flujo de trabajo sencillo.....	126
Sección 50.3: Ejecutar el flujo de trabajo como trabajo en segundo plano.....	126
Sección 50.4: Añadir un bloque paralelo a un flujo de trabajo.....	126
Capítulo 51: Incrustación de código gestionado (C# VB)	128

Sección 51.1: Ejemplo en C#.....	128
Sección 51.2: Ejemplo en VB.NET	128
Capítulo 52: ¿Cómo descargar el último artefacto de Artifactory usando un script PowerShell (v2.0 o inferior)?	130
Sección 52.1: Script PowerShell para descargar el último artefacto.....	130
Capítulo 53: Ayuda basada en comentarios.....	131
Sección 53.1: Función de ayuda basada en comentarios.....	131
Sección 53.2: Script de ayuda basada en comentarios	134
Capítulo 54: Módulo Archive	136
Sección 54.1: Compress-Archive con comodín.....	136
Sección 54.2: Actualizar ZIP existente con Compress-Archive	136
Sección 54.3: Extraer un Zip con Expand-Archive.....	136
Capítulo 55: Automatización de infraestructuras.....	137
Sección 55.1: Script sencillo para la prueba de integración black-box de aplicaciones de consola.....	137
Capítulo 56: PSScriptAnalyzer - Analizador de scripts PowerShell.....	138
Sección 56.1: Análisis de guiones con las reglas preestablecidas integradas.....	138
Sección 56.2: Análisis de secuencias de comandos según todas las reglas integradas	138
Sección 56.3: Lista de todas las reglas incorporadas	138
Capítulo 57: Configuración de estado deseada	139
Sección 57.1: Ejemplo sencillo - Activar WindowsFeature	139
Sección 57.2: Iniciando DSC (mof) en máquina remota.....	139
Sección 57.3: Importación de psd1 (archivo de datos) a una variable local.....	139
Sección 57.4: Lista de recursos disponibles del DSC	139
Sección 57.5: Importación de recursos para su uso en DSC.....	140
Capítulo 58: Uso de ShouldProcess	141
Sección 58.1: Ejemplo de uso completo.....	141
Sección 58.2: Añadir soporte -WhatIf y -Confirm a su cmdlet.....	142
Sección 58.3: Uso de ShouldProcess() con un argumento	142
Capítulo 59: Módulo de tareas programadas.....	143
Sección 59.1: Ejecutar un script PowerShell en una tarea programada.....	143
Capítulo 60: Módulo ISE	144
Sección 60.1: Scripts de prueba.....	144
Capítulo 61: Creación de recursos basados en clases de DSC	145
Sección 61.1: Crear una clase esqueleto de recursos DSC.....	145
Sección 61.2: Esqueleto de recursos DSC con propiedad clave	145
Sección 61.3: Recurso DSC con propiedad obligatoria.....	145
Sección 61.4: Recurso DSC con métodos requeridos.....	146
Capítulo 62: WMI y CIM.....	147
Sección 62.1: Consulta de objetos	147
Sección 62.2: Clases y espacios de nombres.....	148

Capítulo 63: Módulo ActiveDirectory	151
Sección 63.1: Usuarios.....	151
Sección 63.2: Módulo	151
Sección 63.3: Grupos.....	151
Sección 63.4: Equipos.....	151
Sección 63.5: Objetos.....	152
Capítulo 64: Módulo SharePoint	153
Sección 64.1: Carga del complemento de SharePoint.....	153
Sección 64.2: Iterar sobre todas las listas de una colección de sitios.....	153
Sección 64.3: Obtener todas las funciones instaladas en una colección de sitios.....	153
Capítulo 65: Introducción a Psake	154
Sección 65.1: Esquema básico	154
Sección 65.2: Ejemplo de FormatTaskName	154
Sección 65.3: Ejecutar tarea condicionalmente.....	154
Sección 65.4: ContinueOnError	155
Capítulo 66: Introducción a Pester	156
Sección 66.1: Primeros pasos con Pester.....	156
Capítulo 67: Gestión de secretos y credenciales.....	157
Sección 67.1: Acceso a la contraseña en texto plano.....	157
Sección 67.2: Solicitud de credenciales	157
Sección 67.3: Trabajar con credenciales almacenadas.....	157
Sección 67.4: Almacenar las credenciales de forma encriptada y pasarlas como parámetro cuando sea necesario.....	158
Capítulo 68: Seguridad y criptografía.....	159
Sección 68.1: Cálculo de los códigos hash de una cadena de caracteres mediante Criptografía .Net.....	159
Capítulo 69: Scripts de firma.....	160
Sección 69.1: Firmar un script.....	160
Sección 69.2: Eludir la política de ejecución de un único script.....	160
Sección 69.3: Cambio de la política de ejecución mediante Set-ExecutionPolicy.....	161
Sección 69.4: Obtener la política de ejecución actual.....	161
Sección 69.5: Obtener la firma de un script firmado	161
Sección 69.6: Creación de un certificado de firma de código autofirmado para pruebas	161
Capítulo 70: Anonimizar IP (v4 y v6) en un archivo de texto con PowerShell.....	163
Sección 70.1: Anonimizar la dirección IP en un archivo de texto	163
Capítulo 71: Servicios web de Amazon (AWS) Rekognition	164
Sección 71.1: Detectar etiquetas de imágenes con AWS Rekognition	164
Sección 71.2: Comparar la similitud facial con AWS Rekognition	165
Capítulo 72: Servicio de almacenamiento simple (S3) de Amazon Web Services (AWS)	166
Sección 72.1: Crear un nuevo Bucket S3.....	166
Sección 72.2: Cargar un archivo local en un bucket de S3.....	166

Sección 72.3: Borrar un Bucket S3	166
Créditos	167

Acerca de

Este libro ha sido traducido por rortegag.com

Si desea descargar el libro original, puede descargarlo desde:

<https://goalkicker.com/PowerShellBook/>

Si desea contribuir con una donación, hazlo desde:

<https://www.buymeacoffee.com/GoalKickerBooks>

Por favor, siéntase libre de compartir este PDF con cualquier persona de forma gratuita, la última versión de este libro se puede descargar desde:

<https://goalkicker.com/PowerShellBook/>

Este libro PowerShell® Apuntes para Profesionales está compilado a partir de la [Documentación de Stack Overflow](#), el contenido está escrito por la hermosa gente de Stack Overflow. El contenido del texto está liberado bajo Creative Commons BY-SA, ver los créditos al final de este libro quién contribuyó a los distintos capítulos. Las imágenes pueden ser copyright de sus respectivos propietarios a menos que se especifique lo contrario.

Este es un libro no oficial gratuito creado con fines educativos y no está afiliado con los grupo(s) o empresa(s) oficiales de PowerShell® ni Stack Overflow. Todas las marcas comerciales y marcas registradas son propiedad de sus respectivos propietarios de la empresa.

No se garantiza que la información presentada en este libro sea correcta ni exacta. Utilícelo bajo su propia responsabilidad.

Envíe sus comentarios y correcciones a web@petercv.com

Capítulo 1: Introducción a PowerShell

Versión	Incluido con Windows	Fecha de lanzamiento
1.0	XP / Server 2008	01-11-2006
2.0	7 / Server 2008 R2	01-11-2009
3.0	8 / Server 2012	01-08-2012
4.0	8.1 / Server 2012 R2	01-11-2013
5.0	10 / Server 2016 Tech Preview	16-12-2015
5.1	10 Edición Aniversario / Server 2016	27-01-2017

Sección 1.1: Permitir que los scripts almacenados en su máquina se ejecuten sin firmar

Por razones de seguridad, PowerShell está configurado por defecto para permitir sólo la ejecución de scripts firmados. Ejecutar el siguiente comando le permitirá ejecutar scripts no firmados (debe ejecutar PowerShell como Administrador para hacer esto).

```
Set-ExecutionPolicy RemoteSigned
```

Otra forma de ejecutar scripts PowerShell es utilizar `Bypass` como `ExecutionPolicy`:

```
powershell.exe -ExecutionPolicy Bypass -File "c:\MyScript.ps1"
```

O desde su consola PowerShell existente o sesión ISE ejecutando:

```
Set-ExecutionPolicy Bypass Process
```

También se puede conseguir una solución temporal para la política de ejecución ejecutando el ejecutable PowerShell y pasando cualquier política válida como parámetro `-ExecutionPolicy`. La directiva solo está en efecto durante el tiempo de vida del proceso, por lo que no se necesita acceso administrativo al registro.

```
C:\>powershell -ExecutionPolicy RemoteSigned
```

Hay muchas otras políticas disponibles, y los sitios en línea a menudo le animan a utilizar `Set-ExecutionPolicy Unrestricted`. Esta política se mantiene hasta que se cambia, y baja la postura de seguridad del sistema. Esto no es aconsejable. Se recomienda el uso de `RemoteSigned` porque permite código almacenado y escrito localmente, y requiere que el código adquirido remotamente sea firmado con un certificado de una raíz de confianza.

Además, tenga en cuenta que la Política de Ejecución puede ser aplicada por la Política de Grupo, por lo que incluso si la política se cambia a `Unrestricted` en todo el sistema, la Política de Grupo puede revertir esa configuración en su próximo intervalo de aplicación (normalmente 15 minutos). Puede ver la política de ejecución establecida en los distintos ámbitos utilizando `Get-ExecutionPolicy -List`

Documentación de TechNet:

[Set-ExecutionPolicy](#)
[about_Execution_Policies](#)

Sección 1.2: Alias y funciones similares

En PowerShell, hay muchas maneras de lograr el mismo resultado. Esto se puede ilustrar muy bien con el simple y familiar ejemplo de `Hello World`:

Uso de `Write-Host`:

```
Write-Host "Hello World"
```

Uso de `Write-Output`:

```
Write-Output 'Hello world'
```

Vale la pena señalar que aunque `Write-Output` y `Write-Host` escriben a la pantalla hay una sutil diferencia. `Write-Host` sólo escribe en stdout (es decir, en la pantalla de la consola), mientras que `Write-Output` escribe tanto en stdout COMO en el flujo de salida [success], lo que permite la [redirección](#). La redirección (y los flujos en general) permiten que la salida de un comando se dirija como entrada a otro, incluyendo la asignación a una variable.

```
> $message = Write-Output "Hello World"
> $message
"Hello World"
```

Estas funciones similares no son alias, pero pueden producir los mismos resultados si se quiere evitar "contaminar" el flujo de éxito.

`Write-Output` es el alias de `Echo` o `Write`

```
Echo 'Hello world'
Write 'Hello world'
```

O simplemente escribiendo ¡'Hello world'!

```
'Hello world'
```

Todo ello dará como resultado la salida de consola esperada

```
Hello world
```

Otro ejemplo de alias en PowerShell es la asignación común de comandos de símbolo del sistema antiguos y comandos BASH a cmdlets de PowerShell. Todos los siguientes producen un listado del directorio actual.

```
C:\Windows> dir
C:\Windows> ls
C:\Windows> Get-ChildItem
```

Por último, puede crear su propio alias con el cmdlet `Set-Alias`. Como ejemplo vamos a alisar `Test-NetConnection`, que es esencialmente el equivalente PowerShell al comando ping del símbolo del sistema, a "ping".

```
Set-Alias -Name ping -Value Test-NetConnection
```

¡Ahora puedes usar ping en lugar de `Test-NetConnection`! Ten en cuenta que, si el alias ya está en uso, sobrescribirás la asociación.

El alias estará activo hasta que la sesión esté activa. Una vez que cierre la sesión e intente ejecutar el alias que ha creado en su última sesión, no funcionará. Para solucionar este problema, puede importar todos sus alias desde un Excel a su sesión una vez, antes de empezar a trabajar.

Sección 1.3: La canalización: uso de la salida de un cmdlet de PowerShell

Una de las primeras preguntas que se hace la gente cuando empieza a utilizar PowerShell para crear scripts es cómo manipular la salida de un cmdlet para realizar otra acción.

El símbolo de canalización | se utiliza al final de un cmdlet para tomar los datos que exporta y pasarlos al cmdlet siguiente. Un ejemplo sencillo es utilizar `Select-Object` para mostrar únicamente la propiedad `Name` de un archivo mostrado desde `Get-ChildItem`:

```
Get-ChildItem | Select-Object
Name # Esto puede acortarse a:
gci | Select Name
```

Un uso más avanzado de la canalización nos permite canalizar la salida de un cmdlet en un bucle `foreach`:

```
Get-ChildItem | ForEach-Object {  
    Copy-Item -Path $_.FullName -destination C:\NewDirectory\  
}
```

Esto puede acortarse a:

```
gci | % { Copy $_.FullName C:\NewDirectory\ }
```

Tenga en cuenta que el ejemplo anterior utiliza la variable automática `$_`. `$_` es el alias abreviado de `$PSItem` que es una variable automática que contiene el elemento actual en el pipeline.

Sección 1.4: Llamada a métodos de bibliotecas .Net

Los métodos estáticos de la biblioteca .Net se pueden llamar desde PowerShell encapsulando el nombre completo de la clase en un tercer corchete y llamando al método con `::`:

```
# llamar a Path.GetFileName()  
C:\> [System.IO.Path]::GetFileName('C:\Windows\explorer.exe')  
explorer.exe
```

Los métodos estáticos pueden llamarse desde la propia clase, pero para llamar a métodos no estáticos se necesita una instancia de la clase .Net (un objeto).

Por ejemplo, el método `AddHours` no puede invocarse desde la propia clase `System.DateTime`. Requiere una instancia de la clase:

```
C:\> [System.DateTime]::AddHours(15)  
La invocación del método ha fallado porque [System.DateTime] no contiene un método llamado  
'AddHours'.  
En la línea:1 char:1  
+ [System.DateTime]::AddHours(15)  
+ ~~~~~  
+ CategoryInfo          : InvalidOperation: (:) [], RuntimeException  
+ FullyQualifiedErrorId : MethodNotFound
```

En este caso, primero creamos un objeto, por ejemplo:

```
C:\> $Object = [System.DateTime]::Now
```

Entonces, podemos usar métodos de ese objeto, incluso métodos que no pueden ser llamados directamente desde la clase `System.DateTime`, como el método `AddHours`:

```
C:\> $Object.AddHours(15)  
Monday 12 September 2016 01:51:19
```

Sección 1.5: Instalación o configuración

Windows

PowerShell se incluye con Windows Management Framework. La instalación y configuración no son necesarias en las versiones modernas de Windows.

Las actualizaciones de PowerShell pueden realizarse instalando una versión más reciente de Windows Management Framework.

Otras plataformas

PowerShell 6 puede instalarse en otras plataformas. Los paquetes de instalación están disponibles [aquí](#).

Por ejemplo, PowerShell 6, para Ubuntu 16.04, se publica en repositorios de paquetes para facilitar la instalación (y las actualizaciones).

Para instalar ejecute lo siguiente:

```
# Importar las claves GPG del repositorio público
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -

# Registrar el repositorio de Microsoft Ubuntu
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee
/etc/apt/sources.list.d/microsoft.list

# Actualizar apt-get
sudo apt-get update

# Instalar PowerShell
sudo apt-get install -y powershell

# Iniciar PowerShell
powershell
```

Después de registrar el repositorio de Microsoft una vez como superusuario, a partir de entonces, sólo tienes que utilizar `sudo apt-get upgrade powershell` para actualizarlo. A continuación, sólo tiene que ejecutar `powershell`.

Sección 1.6: Comentarios

Para comentar los scripts de potencia, anteponga el símbolo `#` (almohadilla) a la línea.

```
# Este es un comentario en PowerShell
Get-ChildItem
```

También puede tener comentarios de varias líneas utilizando `<#` y `#>` al principio y al final del comentario respectivamente.

```
<#
Se trata de un
comentario
multilínea
#>
Get-ChildItem
```

Sección 1.7: Creación de objetos

El cmdlet `New-Object` se utiliza para crear un objeto.

```
# Crea un objeto DateTime y almacena el objeto en la variable "$var"
$var = New-Object System.DateTime

# llamar al constructor con parámetros
$sr = New-Object System.IO.StreamReader -ArgumentList "file path"
```

En muchos casos, se creará un nuevo objeto para exportar datos o pasarlos a otro commandlet. Esto se puede hacer así:

```
$newObject = New-Object -TypeName PSObject -Property @{
    ComputerName = "SERVER1"
    Role = "Interface"
    Environment = "Production"
}
```

Hay muchas formas de crear un objeto. El siguiente método es probablemente la forma más corta y rápida de crear un `PSCustomObject`:

```
$newObject = [PSCustomObject]@{  
    ComputerName = 'SERVER1'  
    Role = 'Interface'  
    Environment = 'Production'  
}
```

En caso de que ya tengas un objeto, pero sólo necesites una o dos propiedades extra, puedes simplemente añadir esa propiedad usando `Select-Object`:

```
Get-ChildItem | Select-Object FullName, Name,  
    @{Name='DateTime'; Expression={Get-Date}},  
    @{Name='PropertyName'; Expression={'CustomValue'}}
```

Todos los objetos pueden ser almacenados en variables o pasados a la tubería. También podrías añadir estos objetos a una colección y luego mostrar los resultados al final.

Las colecciones de objetos funcionan bien con `Export-CSV` (e `Import-CSV`). Cada línea del CSV es un objeto, cada columna una propiedad.

Los comandos de formato convierten los objetos en texto para su visualización. Evite utilizar los comandos `Format-*` hasta el paso final de cualquier procesamiento de datos, para mantener la usabilidad de los objetos.

Capítulo 2: Variables en PowerShell

Las variables se utilizan para almacenar valores. Sea el valor del tipo que sea, necesitamos almacenarlo en algún lugar para poder utilizarlo en toda la consola/script. Los nombres de variables en PowerShell comienzan con \$, como en `$Variable1`, y los valores se asignan usando =, como `$Variable1 = "Valor 1"`. PowerShell soporta un gran número de tipos de variables; como cadenas de texto, enteros, decimales, matrices, e incluso tipos avanzados como números de versión o direcciones IP.

Sección 2.1: Variable simple

Todas las variables en PowerShell comienzan con un signo de dólar (\$). El ejemplo más sencillo es:

```
$foo = "bar"
```

Esta sentencia asigna una variable llamada `foo` con un valor de cadena de caracteres `"bar"`.

Sección 2.2: Arrays

La declaración de arrays en Powershell es casi igual que la instanciación de cualquier otra variable, es decir, se utiliza la sintaxis `$name` =. Los elementos del array se declaran separándolos por comas(,):

```
$myArrayOfInts = 1,2,3,4  
$myArrayOfStrings = "1","2","3","4"
```

Añadir a un array

Añadir a un array es tan sencillo como utilizar el operador +:

```
$myArrayOfInts = $myArrayOfInts + 5  
# ahora contiene 1,2,3,4 & 5!
```

Combinar arrays

De nuevo, es tan sencillo como utilizar el operador +.

```
$myArrayOfInts = 1,2,3,4  
$myOtherArrayOfInts = 5,6,7  
$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts  
# ahora contiene 1,2,3,4,5,6,7
```

Sección 2.3: Asignación de listas de variables múltiples

Powershell permite la asignación múltiple de variables y trata casi todo como un array o una lista. Esto significa que en lugar de hacer algo como esto:

```
$input = "foo.bar.baz"  
$parts = $input.Split(".")  
$foo = $parts[0]  
$bar = $parts[1]  
$baz = $parts[2]
```

Simplemente puedes hacer esto:

```
$foo, $bar, $baz = $input.Split(".")
```

Dado que Powershell trata las asignaciones de esta forma como listas, si hay más valores en la lista que elementos en tu lista de variables a los que asignarlos, la última variable se convierte en un array de los valores restantes. Esto significa que también puedes hacer cosas como esta:

```
$foo, $leftover = $input.Split(".") # Establece $foo = "foo", $leftover = ["bar","baz"]
$bar = $leftover[0] # $bar = "bar"
$baz = $leftover[1] # $baz = "baz"
```

Sección 2.4: Ámbito

El [ámbito](#) por defecto de una variable es el contenedor que la contiene. Si está fuera de un script u otro contenedor, el ámbito es Global. Para especificar un [ámbito](#), se prefija al nombre de la variable

`$scope:varname` de la siguiente manera:

```
$foo = "Global Scope"
function myFunc {
    $foo = "Function (local) scope"
    Write-Host $global:foo
    Write-Host $local:foo
    Write-Host $foo
}
myFunc
Write-Host $local:foo
Write-Host $foo
```

Salida:

Ámbito global Ámbito de función (local) Ámbito de función (local) Ámbito global Ámbito global

Sección 2.5: Eliminar una variable

Para eliminar una variable de la memoria, se puede utilizar el cmdlet [Remove-Item](#). Nota: El nombre de la variable NO incluye el `$`.

```
Remove-Item Variable:\foo
```

[Variable](#) tiene un proveedor para permitir que la mayoría de los cmdlets `*-item` funcionen como sistemas de archivos.

Otro método para eliminar variables es utilizar el cmdlet [Remove-Variable](#) y su alias `rv`.

```
$var = "Some Variable" # Definir la variable 'var' que contiene la cadena 'Some Variable'
$var # Para probar y mostrar la cadena 'Some Variable' en la consola
```

```
Remove-Variable -Name var
$var
```

```
# también se puede utilizar el alias 'rv'
rv var
```

Capítulo 3: Operadores

Un operador es un carácter que representa una acción. Indica al compilador/intérprete que realice una operación matemática, relacional o lógica específica y produzca un resultado final. PowerShell interpreta de manera específica y categoriza en consecuencia como operadores aritméticos realizan operaciones principalmente en números, pero también afectan cadenas y otros tipos de datos. Junto con los operadores básicos, PowerShell tiene una serie de operadores que ahorran tiempo y esfuerzo de codificación (por ejemplo: `-like`, `-match`, `-replace`, etc).

Sección 3.1: Operadores de comparación

Los operadores de comparación de PowerShell constan de un guión inicial (-) seguido de un nombre (eq para igual, gt para mayor que, etc...).

Los nombres pueden ir precedidos de caracteres especiales para modificar el comportamiento del operador:

```
i # Case-Insensitive Explicit (-ieq)
c # Case-Sensitive Explicit (-ceq)
```

Si no se especifica, por defecto se distingue entre mayúsculas y minúsculas, ("`a`" -eq "`A`") igual que ("`a`" -ieq "`A`").

Operadores de comparación simples:

```
2 -eq 2      # Igual a (==)
2 -ne 4      # No es igual a (!=)
5 -gt 2      # Mayor que (>)
5 -ge 5      # Mayor o igual que (>=)
5 -lt 10     # Menos de (<)
5 -le 5      # Menor o igual que (<=)
```

Operadores de comparación de cadenas de caracteres:

```
"MyString" -like "*String"      # Coincidencia mediante el carácter comodín (*)
"MyString" -notlike "Other*"    # No coincide utilizando el carácter comodín (*)
"MyString" -match '^String$'    # Coincide con una cadena utilizando expresiones regulares
"MyString" -notmatch '^Other$'  # No coincide con una cadena utilizando expresiones regulares
```

Operadores de comparación de colecciones:

```
"abc", "def" -contains "def"    # Devuelve true cuando el valor (derecho) está presente
# en el array (izquierda)
"abc", "def" -notcontains "123" # Devuelve true cuando el valor (derecho) no está presente
# en el array (izquierda)
"def" -in "abc", "def"          # Devuelve true cuando el valor (izquierda) está presente
# en el array (derecha)
"123" -notin "abc", "def"       # Devuelve true cuando el valor (izquierda) no está presente
# en el array (derecha)
```

Sección 3.2: Operadores aritméticos

```
1 + 2      # Adición
1 - 2      # Resta
-1         # Establecer valor negativo
1 * 2      # Multiplicación
1 / 2      # División
1 % 2      # Módulo
100 -shl 2 # Bit a la izquierda
100 -shr 1 # Bit a la derecha
```

Sección 3.3: Operadores de asignación

Simple aritmética:

```
$var = 1      # Asignación. Establece el valor de una variable al valor especificado
$var += 2    # Suma. Aumenta el valor de una variable en el valor especificado.
$var -= 1    # Resta. Disminuye el valor de una variable en el valor especificado.
$var *= 2    # Multiplicación. Multiplica el valor de una variable por el valor especificado.
$var /= 2    # División. Divide el valor de una variable por el valor especificado.
$var %= 2    # Módulo. Divide el valor de una variable por el valor especificado y luego
             # asigna el resto (módulo) a la variable
```

Incremento y decremento:

```
$var++      # Aumenta el valor de una variable, propiedad asignable o elemento del array en 1
$var--      # Disminuye el valor de una variable, propiedad asignable o elemento del array en 1
```

Sección 3.4: Operadores de redirección

Flujo de salida correcto:

```
cmdlet > file # Enviar la salida de éxito a un archivo, sobrescribiendo el contenido existente
cmdlet >> file # Enviar el resultado a un archivo, añadiéndolo al contenido existente
cmdlet 1>&2    # Enviar la salida de éxito y error al flujo de error
```

Flujo de salida de errores:

```
cmdlet 2> file # Enviar la salida de error a un archivo, sobrescribiendo el contenido existente
cmdlet 2>> file # Enviar la salida de error a un archivo, añadiéndolo al contenido existente
cmdlet 2>&1    # Enviar la salida de éxito y error al flujo de salida de éxito
```

Advertencia de flujo de salida: (PowerShell 3.0+)

```
cmdlet 3> file # Enviar la salida de advertencia a un archivo, sobrescribiendo el contenido
existente
cmdlet 3>> file # Enviar el mensaje de advertencia a un archivo, añadiéndolo al contenido
existente
cmdlet 3>&1    # Enviar la salida de éxito y advertencia al flujo de salida de éxito
```

Flujo de salida detallado: (PowerShell 3.0+)

```
cmdlet 4> file # Enviar salida detallada a un archivo, sobrescribiendo el contenido existente
cmdlet 4>> file # Enviar salida detallada a un archivo, añadiéndola al contenido existente
cmdlet 4>&1    # Enviar la salida de éxito y verbose al flujo de salida de éxito
```

Flujo de salida de depuración: (PowerShell 3.0+)

```
cmdlet 5> file # Enviar la salida de depuración a un archivo, sobrescribiendo el contenido
existente
cmdlet 5>> file # Enviar la salida de depuración a un archivo, añadiéndola al contenido existente
cmdlet 5>&1    # Enviar la salida de éxito y depuración al flujo de salida de éxito
```

Flujo de salida de información: (PowerShell 5.0+)

```
cmdlet 6> file # Enviar la información de salida a un archivo, sobrescribiendo el contenido
existente
cmdlet 6>> file # Enviar la información de salida a un archivo, añadiéndola al contenido
existente
cmdlet 6>&1    # Enviar la salida de éxito e información al flujo de salida de éxito
```

Todos los flujos de salida:

```
cmdlet *> file # Enviar todos los flujos de salida a un archivo, sobrescribiendo el contenido existente
cmdlet *>> file # Enviar todos los flujos de salida a un archivo, añadiéndolos al contenido existente
cmdlet *>&1 # Enviar todos los flujos de salida al flujo de salida de éxito
```

Diferencias al operador de tubería (|)

Los operadores de redirección sólo redirigen flujos a archivos o flujos a flujos. El operador de tubería bombea un objeto por la tubería a un cmdlet o a la salida. Cómo funciona la tubería difiere en general de cómo funciona la redirección y se puede leer en Trabajar con la tubería de PowerShell.

Sección 3.5: Mezclando tipos de operandos, el tipo del operando de la izquierda dicta el comportamiento

Para añadir

```
"4" + 2 # Da "42"
4 + "2" # Da 6
1,2,3 + "Hello" # Da 1,2,3,"Hello"
"Hello" + 1,2,3 # Da "Hello1 2 3"
```

Para multiplicar

```
"3" * 2 # Da "33"
2 * "3" # Da 6
1,2,3 * 2 # Da 1,2,3,1,2,3
2 * 1,2,3 # Da un error op_Multiplicar no aparece
```

El impacto puede tener consecuencias ocultas para los operadores de comparación:

```
$a = Read-Host "Enter a number"
Enter a number : 33
$a -gt 5
False
```

Sección 3.6: Operadores lógicos

```
-and # Lógica y
-or # Lógica o
-xor # Exclusiva lógica o
-not # Lógico no
! # Lógico no
```

Sección 3.7: Operadores de manipulación de cadenas de caracteres

Reemplazar operador:

El operador `-replace` sustituye un patrón en un valor de entrada utilizando una expresión regular. Este operador utiliza dos argumentos (separados por una coma): un patrón de expresión regular y su valor de sustitución (que es opcional y una cadena de caracteres vacía por defecto).

```
"The rain in Seattle" -replace 'rain','hail' # Devuelve: The hail in Seattle
"kenmyer@contoso.com" -replace '^[\\w]+@(.+)', '$1' # Devuelve: contoso.com
```

Operadores de división y unión:

El operador `-split` divide una cadena de caracteres en un array de subcadenas de caracteres.

```
"A B C" -split " " # Devuelve un objeto de colección de cadenas que contiene A,B y C.
```

El operador `-join` une un array de cadenas de caracteres en una única cadena de caracteres.

```
"E", "F", "G" -join ":" # Devuelve una sola cadena de caracteres: E:F:G
```

Capítulo 4: Operadores especiales

Sección 4.1: Operador de expresión de array

Devuelve la expresión como un array.

```
@(Get-ChildItem $env:windir\System32\ntdll.dll)
```

Devolverá un array con un elemento

```
@(Get-ChildItem $env:windir\System32)
```

Devolverá un array con todos los elementos de la carpeta (lo que no supone un cambio de comportamiento respecto a la expresión interna).

Sección 4.2: Operación de llamada

```
$command = 'Get-ChildItem' & $Command
```

Ejecutará el comando `Get-ChildItem`

Sección 4.3: Operador de puntos

`..\myScript.ps1` ejecuta `.\myScript.ps1` en el ámbito actual haciendo que cualquier función, y variable esté disponible en el ámbito actual.

Capítulo 5: Operaciones básicas con conjuntos

Un conjunto es una colección de elementos que puede ser cualquier cosa. Cualquier operador que necesitemos para trabajar con estos conjuntos son los operadores de conjuntos y la operación también se conoce como operación de conjuntos. Las operaciones básicas de conjuntos incluyen la unión, la intersección, la suma, la resta, etc.

Sección 5.1: Filtrado: Where-Object / where / ?

Filtrar una enumeración mediante una expresión condicional

Sinónimos:

Where-Object
where
?

Ejemplo:

```
$names = @( "Aaron", "Albert", "Alphonse", "Bernie", "Charlie", "Danny", "Ernie", "Frank" )  
$names | Where-Object { $_ -like "A*" }  
$names | where { $_ -like "A*" }  
$names | ? { $_ -like "A*" }
```

Retorna:

```
Aaron  
Albert  
Alphonse
```

Sección 5.2: Ordenar: Sort-Object / sort

Ordenar una enumeración en orden ascendente o descendente.

Sinónimos:

Sort-Object
sort

Suponiendo:

```
$names = @( "Aaron", "Aaron", "Bernie", "Charlie", "Danny" )
```

La ordenación ascendente es la predeterminada:

```
$names | Sort-Object  
$names | sort
```

```
Aaron  
Aaron  
Bernie  
Charlie  
Danny
```

Para solicitar el orden descendente:

```
$names | Sort-Object -Descending  
$names | sort -Descending
```

```
Danny
Charlie
Bernie
Aaron
Aaron
```

Puedes ordenar utilizando una expresión.

```
$names | Sort-Object { $_.length }
```

```
Aaron
Aaron
Danny
Bernie
Charlie
```

Sección 5.3: Agrupación: Group-Object / group

Puede agrupar una enumeración basándose en una expresión.

Sinónimos:

```
Group-Object
group
```

Ejemplos:

```
$names = @( "Aaron", "Albert", "Alphonse", "Bernie", "Charlie", "Danny", "Ernie", "Frank" )
$names | Group-Object -Property Length
$names | group -Property Length
```

Respuesta:

Cuenta	Nombre	Grupo
4	5	{Aaron, Danny, Ernie, Frank}
2	6	{Albert, Bernie}
1	8	{Alphonse}
1	7	{Charlie}

Sección 5.4: Proyectar: Select-Object / select

Proyectar una enumeración permite extraer miembros específicos de cada objeto, extraer todos los detalles o calcular valores para cada objeto.

Sinónimos:

```
Select-Object
SELECT
```

Seleccionar un subconjunto de propiedades:

```
$dir = dir "C:\MyFolder"
$dir | Select-Object Name, FullName, Attributes
$dir | select Name, FullName, Attributes
```

Nombre	Nombre Completo	Atributos
Images	C:\MyFolder\Images	Directory
Data.txt	C:\MyFolder\data.txt	Archive
Source.c	C:\MyFolder\source.c	Archive

Seleccionar el primer elemento y mostrar todas sus propiedades:

```
$d | select -first 1 *
```

PSPath

PSParentPath

PSChildName

PSDrive

PSProvider

PSIsContainer

BaseName

Mode

Name

Parent

Exists

Root

FullName

Extension

CreationTime

CreationTimeUtc

LastAccessTime

LastAccessTimeUtc

LastWriteTime

LastWriteTimeUtc

Attributes

Capítulo 6: Lógica condicional

Sección 6.1: if, else y else if

Powershell admite operadores lógicos condicionales estándar, al igual que muchos lenguajes de programación. Estos permiten que ciertas funciones o comandos se ejecuten bajo circunstancias particulares.

Con un `if` los comandos dentro de los corchetes (`{}`) sólo se ejecutan si se cumplen las condiciones dentro del `if()`.

```
$test = "test"
if ($test -eq "test"){
    Write-Host "si se cumple la condición"
}
```

También se puede hacer un `else`. Aquí los comandos `else` se ejecutan si **no** se cumplen las condiciones `if`:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "si se cumple la condición"
}
else{
    Write-Host "si no se cumple la condición"
}
```

o un `elseif`. Un `elseif` ejecuta los comandos si no se cumplen las condiciones del `if` y si se cumplen las condiciones del `elseif`:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "si se cumple la condición"
}
elseif ($test -eq "test"){
    Write-Host "si se cumple la condición ifelse"
}
```

Tenga en cuenta el uso anterior `-eq`(igualdad) CmdLet y no `=` o `==` como muchos otros idiomas hacen para la igualdad.

Sección 6.2: Negación

Es posible que desee negar un valor booleano, es decir, introducir una sentencia `if` cuando una condición es falsa en lugar de verdadera. Para ello, utilice la opción CmdLet `-Not`

```
$test = "test"
if (-Not $test -eq "test2"){
    Write-Host "si no se cumple la condición"
}
```

También puedes utilizar `!`:

```
$test = "test"
if (!( $test -eq "test2")){
    Write-Host "si no se cumple la condición"
}
```

también existe el operador `-ne` (no igual):

```
$test = "test"
if ($test -ne "test2"){
    Write-Host "variable test no es igual a 'test2'"
}
```

Sección 6.3: Abreviatura condicional if

Si desea utilizar la taquigrafía puede hacer uso de la lógica condicional con la siguiente taquigrafía. Sólo la cadena de caracteres "false" se evaluará a verdadero (2.0).

```
# Hecho en Powershell 2.0
$boolean = $false;
$string = "false";
$emptyString = "";

If($boolean){
    # esto no se ejecuta porque $boolean es falso
    Write-Host "Las condiciones abreviadas If pueden estar bien, pero asegúrate de que siempre sean booleanas."
}

If($string){
    # Esto se ejecuta porque la cadena de caracteres tiene una longitud distinta de cero
    Write-Host "Si la variable no es estrictamente nula o booleana falsa, se evaluará a verdadero ya que es un objeto o cadena con longitud mayor que 0."
}

If($emptyString){
    # Esto no se ejecuta porque la cadena de caracteres es de longitud cero
    Write-Host "También puede ser útil comprobar las cadenas de caracteres vacías."
}

If($null){
    # No se ejecuta porque la condición es nula
    Write-Host "Al marcar Nulls no se imprimirá esta sentencia."
}
```

Capítulo 7: Bucles

Un bucle es una secuencia de instrucciones que se repite continuamente hasta que se alcanza una determinada condición. Poder hacer que tu programa ejecute repetidamente un bloque de código es una de las tareas más básicas pero útiles de la programación. Un bucle le permite escribir una sentencia muy simple para producir un resultado significativamente mayor simplemente por repetición. Si se ha alcanzado la condición, la siguiente instrucción "pasa" a la siguiente instrucción secuencial o se ramifica fuera del bucle.

Sección 7.1: foreach

`ForEach` tiene dos significados diferentes en PowerShell. Uno es una [palabra clave](#) y el otro es un alias para el cmdlet `ForEach-Object`. El primero se describe aquí.

Este ejemplo muestra cómo imprimir todos los elementos de un array en la consola del host:

```
$Names = @('Amy', 'Bob', 'Celine', 'David')
ForEach ($Name in $Names)
{
    Write-Host "Hi, my name is $Name!"
}
```

Este ejemplo muestra cómo capturar la salida de un bucle `ForEach`:

```
$Numbers = ForEach ($Number in 1..20) {
    $Number # Alternativamente, Write-Output $Number
}
```

Al igual que el ejemplo anterior, este ejemplo, en cambio, demuestra la creación de un array antes de almacenar el bucle:

```
$Numbers = @()
ForEach ($Number in 1..20)
{
    $Numbers += $Number
}
```

Sección 7.2: for

```
for($i = 0; $i -le 5; $i++){
    "$i"
}
```

Un uso típico del bucle `for` es operar sobre un subconjunto de los valores de un array. En la mayoría de los casos, si desea iterar todos los valores de un array, considere el uso de una sentencia [foreach](#).

Sección 7.3: Método ForEach()

Version > 4.0

En lugar del cmdlet `ForEach-Object`, también existe la posibilidad de utilizar un método `ForEach` directamente en arrays de objetos de la siguiente manera

```
(1..10).ForEach({$_ * $_})
```

o, si se desea, se pueden omitir los paréntesis alrededor del bloque de guión

```
(1..10).ForEach{$_ * $_}
```

En ambos casos, el resultado será el siguiente

```
149
16
25
36
49
64
81
100
```

Sección 7.4: ForEach-Object

El cmdlet `ForEach-Object` funciona de forma similar a la sentencia `foreach`, pero toma su entrada de la tubería.

Uso básico

```
$object | ForEach-Object {
    code_block
}
```

Ejemplo:

```
$names = @("Any", "Bob", "Celine", "David")
$names | ForEach-Object {
    "Hi, my name is $_!"
}
```

`ForEach-Object` tiene dos alias por defecto, `foreach` y `%` (sintaxis abreviada). El más común es `%` porque `foreach` puede confundirse con la sentencia `foreach`. Ejemplos:

```
$names | % {
    "Hi, my name is $_!"
}
$names | foreach {
    "Hi, my name is $_!"
}
```

Uso avanzado

`ForEach-Object` se distingue de las soluciones `foreach` alternativas porque es un cmdlet, lo que significa que está diseñado para utilizar la canalización. Debido a esto, tiene soporte para tres scriptblocks al igual que un cmdlet o función avanzada:

- **Inicio:** Se ejecuta una vez antes de recorrer los elementos que llegan de la tubería. Suele utilizarse para crear funciones que se utilizarán en el bucle, crear variables, abrir conexiones (base de datos, web +), etc.
- **Proceso:** Ejecutado una vez por cada elemento llegado de la tubería. "Normal" `foreach` codeblock. Este es el valor por defecto utilizado en los ejemplos anteriores cuando no se especifica el parámetro.
- **Fin:** Se ejecuta una vez después de procesar todos los elementos. Suele utilizarse para cerrar conexiones, generar un informe, etc.

Ejemplo:

```
"Any", "Bob", "Celine", "David" | ForEach-Object -Begin {
    $results = @()
} -Process {
    # Crear y almacenar mensaje
    $results += "Hi, my name is $_!"
} -End {
    # Contar mensajes y salida
    Write-Host "Total messages: $($results.Count)"
    $results
}
```

Sección 7.5: Continue

El operador `Continue` funciona en los bucles `For`, `ForEach`, `While` y `Do`. Se salta la iteración actual del bucle, saltando al principio del bucle más interno.

```
$i = 0
while ($i -lt 20) {
    $i++
    if ($i -eq 7) { continue }
    Write-Host $i
}
```

Lo anterior mostrará del 1 al 20 en la consola, pero omitirá el número 7.

Nota: Cuando se utiliza un bucle pipeline se debe utilizar `return` en lugar de `Continue`.

Sección 7.6: Break

El operador `break` sale inmediatamente de un bucle de programa. Puede utilizarse en los bucles `For`, `ForEach`, `While` y `Do` o en una sentencia `Switch`.

```
$i = 0
while ($i -lt 15) {
    $i++
    if ($i -eq 7) {break}
    Write-Host $i
}
```

Lo anterior contará hasta 15 pero se detendrá en cuanto llegue a 7.

Nota: Cuando se utiliza un bucle tubería, `break` se comportará como `continue`. Para simular `break` en el bucle tubería es necesario incorporar alguna lógica adicional, cmdlet, etc. Es más fácil seguir con bucles no pipeline si necesita utilizar `break`.

Etiquetas Break

Break también puede llamar a una etiqueta colocada delante de la instanciación de un bucle:

```
$i = 0
:mainLoop While ($i -lt 15) {
    Write-Host $i -ForegroundColor 'Cyan'
    $j = 0
    While ($j -lt 15) {
        Write-Host $j -ForegroundColor 'Magenta'
        $k = $i*$j
        Write-Host $k -ForegroundColor 'Green'
        if ($k -gt 100) {
            break mainLoop
        }
        $j++
    }
    $i++
}
```

```
    $i++  
}
```

Nota: Este código incrementará `$i` a 8 y `$j` a 13 lo que causará que `$k` sea igual a 104. Dado que `$k` es superior a 104, el código saldrá de ambos bucles.

Sección 7.7: While

Un bucle `while` evaluará una condición y si es verdadera realizará una acción. Mientras la condición sea verdadera, la acción continuará realizándose.

```
while(condition){  
    code_block  
}
```

El siguiente ejemplo crea un bucle que realiza una cuenta atrás de 10 a 0

```
$i = 10  
while($i -ge 0){  
    $i  
    $i--  
}
```

A diferencia del bucle `Do-While`, la condición se evalúa antes de la primera ejecución de la acción. La acción no se ejecutará si la condición inicial es falsa.

Nota: Al evaluar la condición, PowerShell tratará la existencia de un objeto de retorno como verdadero. Esto se puede utilizar de varias maneras, pero a continuación se muestra un ejemplo para supervisar un proceso. Este ejemplo generará un proceso de bloc de notas y luego dormirá el shell actual mientras ese proceso se esté ejecutando. Cuando cierre manualmente la instancia del bloc de notas, la condición `while` fallará y el bucle se romperá.

```
Start-Process notepad.exe  
while(Get-Process notepad -ErrorAction SilentlyContinue){  
    Start-Sleep -Milliseconds 500  
}
```

Sección 7.8: Do

Los bucles `Do` son útiles cuando siempre quieres ejecutar un bloque de código al menos una vez. Un bucle `Do` evaluará la condición después de ejecutar el codeblock, a diferencia de un bucle `while` que lo hace antes de ejecutar el codeblock.

Puede utilizar los bucles de dos maneras:

- Bucle `while` la condición es verdadera:

```
Do {  
    code_block  
} while (condition)
```

- Bucle `until` que la condición es verdadera, en otras palabras, bucle `while` la condición es falsa:

```
Do {  
    code_block  
} until (condition)
```

Ejemplos reales:

```
$i = 0  
Do {  
    $i++  
    "Number $i"  
} while ($i -ne 3)
```

```
Do {  
    $i++  
    "Number $i"  
} until ($i -eq 3)
```

Do-While y **Do-Until** son bucles antónimos. Si el código dentro del mismo, la condición se invertirá. El ejemplo anterior ilustra este comportamiento.

Capítulo 8: Sentencia Switch

Una sentencia `switch` permite comprobar la igualdad de una variable con una lista de valores. Cada valor se denomina `case`, y la variable que se activa se comprueba para cada caso de `switch`. Permite escribir un script que puede elegir entre una serie de opciones, pero sin necesidad de escribir una larga serie de sentencias `if`.

Sección 8.1: Simple Switch

Las sentencias `switch` comparan un único valor de prueba con múltiples condiciones, y realiza cualquier acción asociada para comparaciones exitosas. Puede dar lugar a múltiples comparaciones/acciones.

Dado el siguiente interruptor...

```
switch($myValue)
{
    'First Condition' { 'First Action' }
    'Second Condition' { 'Second Action' }
}
```

Se mostrará `'First Action'` si `$myValue` se establece como `'First Condition'`.

Si `$myValue` se establece como `'Second Condition'`, se mostrará `'Section Action'`.

No se mostrará nada si `$myValue` no coincide con ninguna de las condiciones.

Sección 8.2: Sentencia Switch con parámetro CaseSensitive

El parámetro `-CaseSensitive` obliga a las sentencias `switch` a realizar coincidencias exactas, sensibles a mayúsculas y minúsculas, con las condiciones.

Ejemplo:

```
switch -CaseSensitive ('Condition')
{
    'condition' {'First Action'}
    'Condition' {'Second Action'}
    'condition' {'Third Action'}
}
```

Salida:

Second Action

La segunda acción es la única que se ejecuta porque es la única condición que coincide exactamente con la cadena de texto `'Condition'` al tener en cuenta la distinción entre mayúsculas y minúsculas.

Sección 8.3: Sentencia Switch con parámetro comodín

El parámetro `-Wildcard` permite que las sentencias `switch` realicen coincidencias comodín con las condiciones.

Ejemplo:

```
switch -Wildcard ('Condition')
{
    'Condition' {'Normal match'}
    'Condit*' {'Zero or more wildcard chars.'}
    'C[ac]ndit[f-l]on' {'Range and set of chars.'}
    'C?ndition' {'Single char. wildcard'}
    'Test*' {'No match'}
}
```

Salida:

Normal match
Zero or more wildcard chars.
Range and set of chars.
Single char. wildcard

Sección 8.4: Sentencia Switch con parámetro de archivo

El parámetro `-file` permite que la sentencia `switch` reciba la entrada de un archivo. Cada línea del archivo es evaluada por la sentencia `switch`.

Ejemplo de archivo `input.txt`:

```
condition  
test
```

Ejemplo de sentencia switch:

```
switch -file input.txt  
{  
    'condition' {'First Action'}  
    'test' {'Second Action'}  
    'fail' {'Third Action'}  
}
```

Salida:

```
First Action  
Second Action
```

Sección 8.5: Simple Switch con condición por defecto

La palabra clave `Default` se utiliza para ejecutar una acción cuando ninguna otra condición coincide con el valor introducido.

Ejemplo:

```
switch('Condition')  
{  
    'Skip Condition'  
    {  
        'First Action'  
    }  
    'Skip This Condition Too'  
    {  
        'Second Action'  
    }  
    Default  
    {  
        'Default Action'  
    }  
}
```

Salida:

```
Default Action
```

Sección 8.6: Sentencia Switch con parámetro Regex

El parámetro `-Regex` permite que las sentencias `switch` realicen coincidencias de expresiones regulares con las condiciones.

Ejemplo:

```
switch -Regex ('Condition')
{
    'Con\D+ion' {'One or more non-digits'}
    'Conditio*$' {'Zero or more "o"'}
    'C.ndition' {'Any single char.'}
    '^C\w+ition$' {'Anchors and one or more word chars.'}
    'Test' {'No match'}
}
```

Salida:

```
One or more non-digits
Any single char.
Anchors and one or more word chars.
```

Sección 8.7: Simple Switch con Break

La palabra clave `break` puede utilizarse en sentencias `switch` para salir de la sentencia antes de evaluar todas las condiciones.

Ejemplo:

```
switch('Condition')
{
    'Condition'
    {
        'First Action'
    }
    'Condition'
    {
        'Second Action'
        break
    }
    'Condition'
    {
        'Third Action'
    }
}
```

Salida:

```
First Action
Second Action
```

Debido a la palabra clave `break` en la segunda acción, la tercera condición no se evalúa.

Sección 8.8: Sentencia Switch con parámetro exacto

El parámetro `-Exact` obliga a las sentencias `switch` a realizar coincidencias exactas e insensibles a mayúsculas y minúsculas con las condiciones de cadena de texto.

Ejemplo:

```
switch -Exact ('Condition')
{
    'condition' {'First Action'}
    'Condition' {'Second Action'}
    'conditioN' {'Third Action'}
    '^*ondition$' {'Fourth Action'}
    'Conditio*' {'Fifth Action'}
}
```

Salida:

First Action
Second Action
Third Action

Las acciones primeras a tercera se ejecutan porque sus condiciones asociadas coinciden con la entrada. Las cadenas de texto regex y comodín de las condiciones cuarta y quinta no coinciden.

Tenga en cuenta que la cuarta condición también coincidiría con la cadena de entrada si se estuviera realizando una coincidencia de expresiones regulares, pero se ha ignorado en este caso porque no es así.

Sección 8.9: Sentencia Switch con expresiones

Las condiciones también pueden ser expresiones:

```
$myInput = 0
switch($myInput) {
    # porque el resultado de la expresión, 4,
    # no es igual a nuestra entrada este bloque no debe ejecutarse.
    (2+2) { 'True. 2 +2 = 4' }
    # porque el resultado de la expresión, 0,
    # es igual a nuestra entrada, este bloque debe ejecutarse.
    (2-2) { 'True. 2-2 = 0' }
    # porque nuestra entrada es mayor que -1 y es menor que 1
    # la expresión se evalúa como verdadera y el bloque debe ejecutarse.
    { $_ -gt -1 -and $_ -lt 1 } { 'True. Value is 0' }
}
# Salida
True. 2-2 = 0
True. Value is 0
```

Capítulo 9: Cadenas de texto

Sección 9.1: Cadena de texto multilínea

Hay varias formas de crear una cadena multilínea en PowerShell:

Puede utilizar los caracteres especiales para retorno de carro y/o nueva línea manualmente o utilizar la variable de entorno `NewLine` para insertar el valor "newline" del sistema)

```
"Hello`r`nWorld"  
"Hello{0}World" -f [environment]::NewLine
```

Crear un salto de línea al definir una cadena de texto (antes de las comillas de cierre)

```
"Hello  
World"
```

Utilizar un here-string. *Es la técnica más habitual.*

```
@"  
Hello  
World  
"@
```

Sección 9.2: Here-string

Las here-string son muy útiles para crear cadenas de texto multilínea. Una de las mayores ventajas en comparación con otras cadenas de texto multilínea es que se pueden utilizar comillas sin tener que escaparlas con un punto y aparte.

Here-string

Aquí las cadenas de texto comienzan con `@"` y un salto de línea y terminan con `"@"` en su propia línea (**"@" deben ser los primeros caracteres de la línea, ni siquiera espacio en blanco/tabulación**).

```
@"  
Simple  
    Multiline string  
with "quotes"  
"@
```

Here-string literal

También puedes crear una here-string literal utilizando comillas simples, cuando no quieras que ninguna expresión se expanda como una cadena literal normal.

```
@'  
The following line won't be expanded  
$(Get-Date)  
because this is a literal here-string  
'@
```

Sección 9.3: Concatenar cadenas de texto

Utilizar variables en una cadena de texto

Puede concatenar cadenas de texto utilizando variables dentro de una cadena de texto entre comillas dobles. Esto no funciona con las propiedades.

```
$string1 = "Power"  
$string2 = "Shell"  
"Greetings from $string1$string2"
```

Utilizar el operador +

También puedes unir cadenas utilizando el operador +.

```
$string1 = "Greetings from"  
$string2 = "PowerShell"  
$string1 + " " + $string2
```

Esto también funciona con las propiedades de los objetos.

```
"The title of this console is '" + $host.Name + '"'
```

Uso de subexpresiones

La salida/resultado de una subexpresión `$()` puede utilizarse en una cadena de texto. Esto es útil cuando se accede a propiedades de un objeto o se realiza una expresión compleja. Las subexpresiones pueden contener varias expresiones separadas por punto y coma ;

```
"Tomorrow is $($((Get-Date).AddDays(1).DayOfWeek))"
```

Sección 9.4: Caracteres especiales

Cuando se utiliza dentro de una cadena entre comillas dobles, el carácter de escape (backtick ```) representa un carácter especial.

```
`0 # Null  
`a # Alerta/Pitido  
`b # Retroceso  
`f # Alimentación de formularios (se utiliza para la salida de impresora)  
`n # Nueva línea  
`r # Retorno de carro  
`t # Tabulación horizontal  
`v # Tabulación vertical (utilizada para la salida de impresora)
```

Por ejemplo:

```
> "This`uses`ttab`r`nThis is on a second line"  
This uses tab  
This is on a second line
```

También puede escapar caracteres especiales con significados especiales:

```
`# # Comment-operator  
`$ # Variable operator  
`` # Carácter de escape  
`` # Comilla única  
`" # Doble comilla
```

Sección 9.5: Creación de una cadena de texto básica

Cadena de texto

Las cadenas de texto se crean envolviendo el texto con comillas dobles. Las cadenas de texto entre comillas dobles pueden evaluar variables y caracteres especiales.

```
$myString = "Some basic text"  
$mySecondString = "String with a $variable"
```

Para utilizar una comilla doble dentro de una cadena de texto, debe escaparse utilizando el carácter de escape, la barra invertida (```). Las comillas simples pueden utilizarse dentro de una cadena de texto entre comillas dobles.

```
$myString = "A ``double quoted`` string which also has 'single quotes'."
```

Cadena de texto literal

Las cadenas de texto literales son cadenas de texto que no evalúan variables ni caracteres especiales. Se crean utilizando comillas simples.

```
$myLiteralString = 'Simple text including special characters (`n) and a $variable-reference'
```

Para utilizar comillas simples dentro de una cadena de texto literal, utilice comillas simples dobles o un here-string literal. Las comillas dobles pueden utilizarse con seguridad dentro de una cadena de texto literal

```
$myLiteralString = 'Simple string with ''single quotes'' and "double quotes".'
```

Sección 9.6: Formato de cadena de texto

```
$hash = @{ city = 'Berlin' }
```

```
$result = 'You should really visit {0}' -f $hash.city  
Write-Host $result #prints "You should really visit Berlin"
```

Las cadenas de texto con formato pueden utilizarse con el operador `-f` o con el método .NET estático `[String]::Format(string format, args)`.

Capítulo 10: HashTables

Una tabla hash es una estructura que asigna claves a valores. Véase [Tabla Hash](#) para más detalles.

Sección 10.1: Acceder a un valor de la tabla hash por clave

Un ejemplo de definición de una tabla hash y de acceso a un valor por la clave

```
$HashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$HashTable.Key1
# salida
Value1
```

Ejemplo de acceso a una clave con caracteres no válidos para el nombre de una propiedad:

```
$HashTable = @{
    'Key 1' = 'Value3'
    Key2 = 'Value4'
}
$HashTable.'Key 1'
# Salida
Value3
```

Sección 10.2: Creación de una tabla hash

Ejemplo de creación de un HashTable vacío:

```
$HashTable = @{}

```

Ejemplo de creación de un HashTable con datos:

```
$HashTable = @{
    Name1 = 'Value'
    Name2 = 'Value'
    Name3 = 'Value3'
}

```

Sección 10.3: Añadir un par clave-valor a una tabla hash existente

Un ejemplo, para añadir una clave "Key2" con un valor de "Value2" a la tabla hash, utilizando el operador de suma:

```
$HashTable = @{
    Key1 = 'Value1'
}
$HashTable += @{Key2 = 'Value2'}
$HashTable

# Salida

Name Value
----
Key1 Value1
Key2 Value2
```

Un ejemplo, para añadir una clave "Key2" con un valor de "Value2" a la tabla hash utilizando el método Add:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable.Add("Key2", "Value2")
$hashTable
```

Salida

```
Name Value
----
Key1 Value1
Key2 Value2
```

Sección 10.4: Eliminar un par clave-valor de una tabla hash existente

Un ejemplo, para eliminar una clave "Key2" con un valor de "Value2" de la tabla hash, utilizando el operador Remove:

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$hashTable.Remove("Key2", "Value2")
$hashTable
```

Salida

```
Name Value
----
Key1 Value1
```

Sección 10.5: Enumeración mediante claves y pares clave-valor

Enumeración mediante claves

```
foreach ($key in $var1.Keys) {
    $value = $var1[$key]
    # o
    $value = $var1.$key
}
```

Enumeración por pares clave-valor

```
foreach ($keyvaluepair in $var1.GetEnumerator()) {
    $key1 = $_.Key1
    $val1 = $_.Val1
}
```

Sección 10.6: Recorrer una tabla hash

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
foreach($key in $hashTable.Keys)
{
    $value = $hashTable.$key
    Write-Output "$key : $value"
}
# Salida
Key1 : Value1
Key2 : Value2
```

Capítulo 11: Trabajar con objetos

Sección 11.1: Examinar un objeto

Ahora que tienes un objeto, sería bueno averiguar qué es. Puede utilizar el cmdlet `Get-Member` para ver qué es un objeto y qué contiene:

```
Get-Item c:\windows | Get-Member
```

Esto produce:

```
TypeName: System.IO.DirectoryInfo
```

Seguido de una lista de propiedades y métodos que tiene el objeto.

Otra forma de obtener el tipo de un objeto es utilizar el método `GetType`, de la siguiente manera:

```
C:\> $Object = Get-Item C:\Windows
C:\> $Object.GetType()
```

IsPublic	IsSerial	Name	BaseType
True	True	DirectoryInfo	System.IO.FileSystemInfo

Para ver una lista de las propiedades que tiene el objeto, junto con sus valores, puede utilizar el cmdlet `Format-List` con su parámetro `-Property` establecido en: `*` (es decir, todas).

He aquí un ejemplo, con el resultado:

```
C:\> Get-Item C:\Windows | Format-List -Property *
```

PSPath : Microsoft.PowerShell.Core\FileSystem::C:\Windows
PSParentPath : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName : Windows
PSDrive : C
PSProvider : Microsoft.PowerShell.Core\FileSystem
PSIsContainer : True
Mode : d-----
BaseName : Windows
Target : {}
LinkType :
Name : Windows
Parent :
Exists : True
Root : C:\
FullName : C:\Windows
Extension :
CreationTime : 30/10/2015 06:28:30
CreationTimeUtc : 30/10/2015 06:28:30
LastAccessTime : 16/08/2016 17:32:04
LastAccessTimeUtc : 16/08/2016 16:32:04
LastWriteTime : 16/08/2016 17:32:04
LastWriteTimeUtc : 16/08/2016 16:32:04
Attributes : Directory

Sección 11.2: Actualizar objetos

Añadir propiedades

Si desea añadir propiedades a un objeto existente, puede utilizar el cmdlet `Add-Member`. Con `PSObjects`, los valores se guardan en un tipo de "Propiedades de nota".

```
$Object = New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}
```

```
Add-Member -InputObject $Object -Name "SomeNewProp" -Value "A value" -MemberType NoteProperty
```

```
# Devuelve
PS> $Object
Name      ID      Address  SomeNewProp
-----
nem       12      A        value
```

También puede añadir propiedades con el Cmdlet `Select-Object` (las llamadas propiedades calculadas):

```
$newObject = $Object | Select-Object *, @{label='SomeOtherProp'; expression={'Another value'}}
```

```
# Devuelve
PS> $newObject
Name      ID      Address  SomeNewProp  SomeOtherProp
-----
nem       12      A        value        Another value
```

El comando anterior puede abreviarse así:

```
$newObject = $Object | Select *,@{l='SomeOtherProp';e={'Another value'}}
```

Eliminar propiedades

Puede utilizar el Cmdlet `Select-Object` para eliminar propiedades de un objeto:

```
$Object = $newObject | Select-Object * -ExcludeProperty ID, Address
```

```
# Devuelve
PS> $Object
Name      SomeNewProp  SomeOtherProp
-----
nem       A value      Another value
```

Sección 11.3: Crear un nuevo objeto

PowerShell, a diferencia de otros lenguajes de scripting, envía objetos a través de la tubería. Lo que esto significa es que cuando se envían datos de un comando a otro, es esencial ser capaz de crear, modificar y recoger objetos.

Crear un objeto es sencillo. La mayoría de los objetos que crees serán objetos personalizados en PowerShell, y el tipo a utilizar para ello es `PSObject`. PowerShell también te permitirá crear cualquier objeto que podrías crear en .NET.

He aquí un ejemplo de creación de un nuevo objeto con algunas propiedades:

Opción 1: `New-Object`

```
$newObject = New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}
```

```
# Devuelve
PS> $newObject
Name      ID      Address
-----
nem       12
```

Puede almacenar el objeto en una variable anteponiendo al comando `$newObject =`

También puede ser necesario almacenar colecciones de objetos. Esto se puede hacer mediante la creación de una variable de colección vacía, y la adición de objetos a la colección, así:

```
$newCollection = @()
$newCollection += New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}
```

A continuación, es posible que desee iterar a través de esta colección objeto por objeto. Para ello, busque la sección Bucle en la documentación.

Opción 2: `Select-Object`

Una forma menos habitual de crear objetos que aún encontrarás en Internet es la siguiente:

```
$newObject = 'unuseddummy' | Select-Object -Property Name, ID, Address
$newObject.Name = $env:username
$newObject.ID = 12
```

```
# Devuelve
PS> $newObject
Name      ID      Address
-----
nem       12
```

Opción 3: acelerador de tipo `PSCustomObject` (requiere PSv3+)

El acelerador de tipo ordenado obliga a PowerShell a mantener nuestras propiedades en el orden en que las hemos definido. No necesitas el acelerador de tipo ordenado para usar `[PSCustomObject]`:

```
$newObject = [PSCustomObject][Ordered]@{
    Name = $env:Username
    ID = 12
    Address = $null
}
```

```
# Devuelve
PS> $newObject
Name      ID      Address
-----
nem       12
```

Sección 11.4: Crear instancias de clases genéricas

Nota: ejemplos escritos para PowerShell 5.1 Puede crear instancias de Clases Genéricas

```
# Nullable System.DateTime
[Nullable[datetime]]$nullableDate = Get-Date -Year 2012
$nullableDate
$nullableDate.GetType().FullName
$nullableDate = $null
$nullableDate

# Normal System.DateTime
[datetime]$aDate = Get-Date -Year 2013
$aDate
$aDate.GetType().FullName
$aDate = $null # Lanza una excepción cuando PowerShell intenta convertir null en
```

Da el resultado:

```
Saturday, 4 August 2012 08:53:02
```

```
System.DateTime
```

```
Sunday, 4 August 2013 08:53:02
```

```
System.DateTime
```

```
Cannot convert null to type "System.DateTime".
```

```
At line:14 char:1
```

```
+ $aDate = $null
```

```
+ ~~~~~
```

```
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
```

```
+ FullyQualifiedErrorId : RuntimeException
```

Las colecciones genéricas también son posibles

```
[System.Collections.Generic.SortedDictionary[int, String]]$dict =
```

```
[System.Collections.Generic.SortedDictionary[int, String]]::new()
```

```
$dict.GetType().FullName
```

```
$dict.Add(1, 'a')
```

```
$dict.Add(2, 'b')
```

```
$dict.Add(3, 'c')
```

```
$dict.Add('4', 'd') # powershell convierte automáticamente '4' en 4
```

```
$dict.Add('5.1', 'c') # powershell convierte automáticamente '5.1' a 5
```

```
$dict
```

```
$dict.Add('z', 'z')
```

```
# powershell no puede convertir 'z' a System.Int32 por lo que arroja un error
```

Da el resultado:

```
System.Collections.Generic.SortedDictionary`2[[System.Int32, mscorlib, Version=4.0.0.0,
```

```
Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.String, mscorlib, Version=4.0.0.0,
```

```
Culture=neutral, PublicKeyToken=b77a5c561934e089]]
```

```
Key Value
```

```
--- -----
```

```
1 a
```

```
2 b
```

```
3 c
```

```
4 d
```

```
5 c
```

```
No se puede convertir el argumento "key", con valor: "z", de "Add" al tipo "System.Int32": "No se puede convertir el argumento "z" al tipo "System.Int32". Error: " La cadena de texto de entrada no tenía el formato correcto."
```

```
At line:15 char:1
```

```
+ $dict.Add('z', 'z') # powershell no puede convertir 'z' a System.Int32 así que ...
```

```
+ ~~~~~
```

```
+ CategoryInfo          : NotSpecified: (:) [], MethodException
```

```
+ FullyQualifiedErrorId : MethodArgumentConversionInvalidCastArgument
```

Capítulo 12: Funciones PowerShell

Una función es básicamente un bloque de código con nombre. Cuando llamas al nombre de la función, el bloque de script dentro de esa función se ejecuta. Es una lista de sentencias PowerShell que tiene un nombre que tú asignas. Cuando ejecutas una función, escribes el nombre de la función. Es un método para ahorrar tiempo al abordar tareas repetitivas. Los formatos de PowerShell constan de tres partes: la palabra clave 'Función', seguida de un Nombre y, por último, la carga útil que contiene el bloque de script, que se encierra entre corchetes de estilo curly/parenthesis.

Sección 12.1: Parámetros básicos

Una función puede definirse con parámetros mediante el bloque param:

```
function Write-Greeting {
    param(
        [Parameter(Mandatory, Position=0)]
        [String]$name,
        [Parameter(Mandatory, Position=1)]
        [Int]$age
    )
    "Hello $name, you are $age years old."
}
```

O utilizando la sintaxis de función simple:

```
function Write-Greeting ($name, $age) {
    "Hello $name, you are $age years old."
}
```

Nota: No es necesario fundir los parámetros en ninguno de los dos tipos de definición de parámetros.

La sintaxis de función simple (SFS) tiene capacidades muy limitadas en comparación con el bloque `param`.

Aunque puede definir parámetros para exponerlos dentro de la función, no puede especificar atributos de parámetros, utilizar validación de parámetros, incluir `[CmdletBinding()]`, con SFS (y esta es una lista no exhaustiva).

Las funciones pueden invocarse con parámetros ordenados o con nombre.

El orden de los parámetros en la invocación coincide con el orden de la declaración en la cabecera de la función (por defecto), o puede especificarse mediante el atributo de parámetro `Position` (como se muestra en el ejemplo de función avanzada, más arriba).

```
$greeting = Write-Greeting "Jim" 82
```

Alternativamente, esta función puede invocarse con parámetros con nombre

```
$greeting = Write-Greeting -name "Bob" -age 82
```

Sección 12.2: Función avanzada

Esta es una copia del fragmento de función avanzada de Powershell ISE. Básicamente esta es una plantilla para muchas de las cosas que puedes usar con funciones avanzadas en Powershell. Puntos clave a tener en cuenta:

- integración de `Get-Help` - el principio de la función contiene un bloque de comentarios que está configurado para ser leído por el cmdlet `Get-Help`. El bloque de función puede situarse al final, si se desea.
- `cmdletbinding` - la función se comportará como un cmdlet
- parámetros
- conjuntos de parámetros

```

<#
.Synopsis
  Short description
.DESRIPTION
  Long description
.EXAMPLE
  Example of how to use this cmdlet
.EXAMPLE
  Another example of how to use this cmdlet
.INPUTS
  Inputs to this cmdlet (if any)
.OUTPUTS
  Output from this cmdlet (if any)
.NOTES
  General notes
.COMPONENT
  The component this cmdlet belongs to
.ROLE
  The role this cmdlet belongs to
.FUNCTIONALITY
  The functionality that best describes this cmdlet
#>
function Verb-Noun
{
  [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
    SupportsShouldProcess=$true,
    PositionalBinding=$false,
    HelpUri = 'http://www.microsoft.com/',
    ConfirmImpact='Medium')]

  [Alias()]
  [OutputType([String])]
  Param
  (
    # Param1 help description
    [Parameter(Mandatory=$true,
      ValueFromPipeline=$true,
      ValueFromPipelineByPropertyName=$true,
      ValueFromRemainingArguments=$false,
      Position=0,
      ParameterSetName='Parameter Set 1')]

    [ValidateNotNull()]
    [ValidateNotNullOrEmpty()]
    [ValidateCount(0,5)]
    [ValidateSet("sun", "moon", "earth")]
    [Alias("p1")]
    $Param1,

    # Param2 help description
    [Parameter(ParameterSetName='Parameter Set 1')]
    [AllowNull()]
    [AllowEmptyCollection()]
    [AllowEmptyString()]
    [ValidateScript({$true})]
    [ValidateRange(0,5)]
    [int]
    $Param2,

    # Param3 help description
    [Parameter(ParameterSetName='Another Parameter Set')]
    [ValidatePattern("[a-z]*")]
    [ValidateLength(0,15)]
    [String]
    $Param3
  )
}

```

```

Begin
{
}
Process
{
    if ($pscmdlet.ShouldProcess("Target", "Operation"))
    {
    }
}
End
{
}
}

```

Sección 12.3: Parámetros obligatorios

Los parámetros de una función pueden marcarse como obligatorios

```

function Get-Greeting{
    param
    (
        [Parameter(Mandatory=$true)]$name
    )
    "Hello World $name"
}

```

Si la función se invoca sin un valor, la línea de comandos solicitará el valor:

```
$greeting = Get-Greeting
```

```
cmdlet Get-Greeting at command pipeline position 1
Supply values for the following parameters:
name:
```

Sección 12.4: Validación de parámetros

Existen varias formas de validar la entrada de parámetros en PowerShell.

En lugar de escribir código dentro de las funciones o scripts para validar los valores de los parámetros, estos `ParameterAttributes` lanzarán si se pasan valores no válidos.

ValidateSet

A veces necesitamos restringir los posibles valores que puede aceptar un parámetro. Digamos que queremos permitir sólo rojo, verde y azul para el parámetro `$Color` en un script o función.

Podemos utilizar el atributo de parámetro `ValidateSet` para restringir esto. Tiene la ventaja adicional de permitir completar tabulaciones al establecer este argumento (en algunos entornos).

```

param(
    [ValidateSet('red', 'green', 'blue', IgnoreCase)]
    [string]$Color
)

```

También puede especificar `IgnoreCase` para desactivar la distinción entre mayúsculas y minúsculas.

ValidateRange

Este método de validación de parámetros toma un valor Int32 mínimo y máximo, y requiere que el parámetro esté dentro de ese rango.

```
param(  
    [ValidateRange(0, 120)]  
    [Int]$Age  
)
```

ValidatePattern

Este método de validación de parámetros acepta parámetros que coincidan con el patrón regex especificado.

```
param(  
    [ValidatePattern("\w{4-6}\d{2}")]  
    [string]$UserName  
)
```

ValidateLength

Este método de validación de parámetros comprueba la longitud de la cadena de texto introducida.

```
param(  
    [ValidateLength(0, 15)]  
    [String]$PhoneNumber  
)
```

ValidateCount

Este método de validación de parámetros comprueba la cantidad de argumentos pasados, por ejemplo, un array de cadenas de texto.

```
param(  
    [ValidateCount(1, 5)]  
    [String[]]$ComputerName  
)
```

ValidateScript

Finalmente, el método `ValidateScript` es extraordinariamente flexible, tomando un scriptblock y evaluándolo usando `$_` para representar el argumento pasado. A continuación, pasa el argumento si el resultado es `$true` (incluyendo cualquier salida como válida).

Puede utilizarse para comprobar la existencia de un archivo:

```
param(  
    [ValidateScript({Test-Path $_})]  
    [IO.FileInfo]$Path  
)
```

Para comprobar que un usuario existe en AD:

```
param(  
    [ValidateScript({Get-ADUser $_})]  
    [String]$UserName  
)
```

Y prácticamente cualquier otra cosa que puedas escribir (ya que no se limita a oneliners):

```
param(
    [ValidateScript({
        $AnHourAgo = (Get-Date).AddHours(-1)
        if ($_ -lt $AnHourAgo.AddMinutes(5) -and $_ -gt $AnHourAgo.AddMinutes(-5)) {
            $true
        } else {
            throw "That's not within five minutes. Try again."
        }
    })]
    [String]$TimeAboutAnHourAgo
)
```

Sección 12.5: Función simple sin parámetros

Este es un ejemplo de una función que devuelve una cadena. En el ejemplo, la función se llama en una sentencia que asigna un valor a una variable. El valor en este caso es el valor de retorno de la función.

```
function Get-Greeting{
    "Hello World"
}

# Invocar la función
$greeting = Get-Greeting

# demostrar el resultado
$greeting
Get-Greeting
```

`function` declara que el siguiente código es una función.

`Get-Greeting` es el nombre de la función. Cada vez que sea necesario utilizar esa función en el script, se puede llamar a la función mediante la invocación por su nombre.

`{ ... }` es el bloque de script que ejecuta la función.

Si el código anterior se ejecuta en el ISE, los resultados serían algo así:

```
Hello World
Hello World
```

Capítulo 13: Clases PowerShell

Una clase es una plantilla de código de programa extensible para crear objetos, proporcionando valores iniciales de estado (variables miembro) e implementaciones de comportamiento (funciones miembro o métodos). Se utiliza como modelo para definir la estructura de los objetos. Un objeto contiene datos a los que accedemos mediante propiedades y sobre los que podemos trabajar mediante métodos. PowerShell 5.0 añadió la posibilidad de crear tus propias clases.

Sección 13.1: Listado de constructores disponibles para una clase

Version \geq 5.0

En PowerShell 5.0+ puedes listar los constructores disponibles llamando al método estático `new` sin paréntesis.

```
PS> [DateTime]::new
OverloadDefinitions
-----
datetime new(long ticks)
datetime new(long ticks, System.DateTimeKind kind)
datetime new(int year, int month, int day)
datetime new(int year, int month, int day, System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second)
datetime new(int year, int month, int day, int hour, int minute, int second, System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar, System.DateTimeKind kind)
```

Esta es la misma técnica que puede utilizar para listar las definiciones de sobrecarga de cualquier método

```
> 'abc'.CompareTo
Overload      Definitions
-----
int           CompareTo(System.Object value)
int           CompareTo(string strB)
int           IComparable.CompareTo(System.Object obj)
int           IComparable[string].CompareTo(string other)
```

Para versiones anteriores puedes crear tu propia función para listar los constructores disponibles:

```
function Get-Constructor {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline=$true)]
        [type]$type
    )
    Process {
        $type.GetConstructors() |
        Format-Table -Wrap @{
            n="$($type.Name) Constructors"
            e="{ ($_.GetParameters() | % { $_.ToString() }) -Join ", " }"
        }
    }
}
```

Uso:

```
Get-Constructor System.DateTime
# 0 [datetime] | Get-Constructor
DateTime Constructors
-----
Int64 ticks
Int64 ticks, System.DateTimeKind kind
Int32 year, Int32 month, Int32 day
Int32 year, Int32 month, Int32 day, System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, System.DateTimeKind
kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
endar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
endar calendar, System.DateTimeKind kind
```

Sección 13.2: Métodos y propiedades

```
class Person {
    [string] $FirstName
    [string] $LastName
    [string] Greeting() {
        return "Greetings, {0} {1}!" -f $this.FirstName, $this.LastName
    }
}
$x = [Person]::new()
$x.FirstName = "Jane"
$x.LastName = "Doe"
$greeting = $x.Greeting() # "Greetings, Jane Doe!"
```

Sección 13.3: Sobrecarga del constructor

```
class Person {
    [string] $Name
    [int] $Age

    Person([string] $Name) {
        $this.Name = $Name
    }

    Person([string] $Name, [int]$Age) {
        $this.Name = $Name
        $this.Age = $Age
    }
}
```

Sección 13.4: Obtener todos los miembros de una instancia

```
PS > Get-Member -InputObject $anObjectInstance
```

Esto devolverá todos los miembros del tipo instancia. A continuación se muestra una parte de una salida de ejemplo para la instancia String

Name	MemberType	Definition
Clone	Method	System.Object Clone(), System.Object ICloneable.Clone()
CompareTo	Method	int CompareTo(System.Object value), int CompareTo(string strB), i...
Contains	Method	bool Contains(string value)
CopyTo	Method	void CopyTo(int sourceIndex, char[] destination, int destinationI...
EndsWith	Method	bool EndsWith(string value), bool EndsWith(string value, System.S...
Equals	Method	bool Equals(System.Object obj), bool Equals(string value), bool E...
GetEnumerator	Method	System.CharEnumerator GetEnumerator(), System.Collections.Generic...
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
...		

Sección 13.5: Plantilla de clase básica

Definir una clase

```
class TypeName
{
    # Propiedad con validate set
    [ValidateSet("val1", "Val2")]
    [string] $P1

    # Propiedad estática
    static [hashtable] $P2

    # La propiedad oculta no se muestra como resultado de Get-Member
    hidden [int] $P3

    # Constructor
    TypeName ([string] $s)
    {
        $this.P1 = $s
    }

    # Método estático
    static [void] MemberMethod1([hashtable] $h)
    {
        [TypeName]::P2 = $h
    }

    # Método de instancia
    [int] MemberMethod2([int] $i)
    {
        $this.P3 = $i
        return $this.P3
    }
}
```

Sección 13.6: Herencia de clase padre a clase hija

```
class ParentClass
{
    [string] $Message = "It's under the Parent Class"

    [string] GetMessage()
    {
        return ("Message: {0}" -f $this.Message)
    }
}
```

Bar extiende Foo y hereda sus miembros

```
class ChildClass : ParentClass
{
}
$Inherit = [ChildClass]::new()
```

Así, **\$Inherit.Message** le dará el

```
"It's under the Parent Class"
```

Capítulo 14: Módulos PowerShell

A partir de la versión 2.0 de PowerShell, los desarrolladores pueden crear módulos PowerShell. Los módulos PowerShell encapsulan un conjunto de funciones comunes. Por ejemplo, hay módulos PowerShell específicos de un proveedor que gestionan varios servicios en la nube. También hay módulos PowerShell genéricos que interactúan con servicios de redes sociales y realizan tareas de programación comunes, como la codificación Base64, el trabajo con Named Pipes, etc.

Los módulos pueden exponer alias de comandos, funciones, variables, clases y mucho más.

Sección 14.1: Crear un manifiesto de módulo

```
@{
    RootModule = 'MyCoolModule.psm1'
    ModuleVersion = '1.0'
    CompatiblePSEditions = @('Core')
    GUID = '6b42c995-67da-4139-be79-597a328056cc'
    Author = 'Bob Schmob'
    CompanyName = 'My Company'
    Copyright = '(c) 2017 Administrator. All rights reserved.'
    Description = 'It does cool stuff.'
    FunctionsToExport = @()
    CmdletsToExport = @()
    VariablesToExport = @()
    AliasesToExport = @()
    DscResourcesToExport = @()
}
```

Todo buen módulo PowerShell tiene un manifiesto de módulo. El manifiesto del módulo simplemente contiene metadatos sobre un módulo de PowerShell, y no define el contenido real del módulo.

El archivo de manifiesto es un archivo de script PowerShell, con extensión `.psd1`, que contiene una HashTable. La HashTable del manifiesto debe contener claves específicas para que PowerShell la interprete correctamente como un archivo de módulo de PowerShell.

El ejemplo anterior proporciona una lista de las claves principales de HashTable que componen un manifiesto de módulo, pero hay muchas otras. El comando `New-ModuleManifest` le ayuda a crear un nuevo esqueleto de manifiesto de módulo.

Sección 14.2: Ejemplo de módulo sencillo

```
function Add {
    [CmdletBinding()]
    param (
        [int] $x
        , [int] $y
    )

    return $x + $y
}
Export-ModuleMember -Function Add
```

Este es un ejemplo sencillo del aspecto que podría tener un archivo de módulo de script PowerShell. Este archivo se llamaría `MyCoolModule.psm1`, y está referenciado desde el archivo de manifiesto del módulo (`.psd1`). Observará que el comando `Export-ModuleMember` nos permite especificar qué funciones del módulo queremos “exportar”, o exponer, al usuario del módulo. Algunas funciones serán internas y no deberían ser expuestas, por lo que se omitirán de la llamada a `Export-ModuleMember`.

Sección 14.3: Exportación de una variable desde un módulo

```
$FirstName = 'Bob'  
Export-ModuleMember -Variable FirstName
```

Para exportar una variable de un módulo, se utiliza el comando `Export-ModuleMember`, con el parámetro `-Variable`. Recuerde, sin embargo, que si la variable tampoco se exporta explícitamente en el archivo de manifiesto del módulo (`.psd1`), entonces la variable no será visible para el consumidor del módulo. Piense en el manifiesto del módulo como un “guardián”. Si una función o variable no está permitida en el manifiesto del módulo, no será visible para el consumidor del módulo.

Nota: Exportar una variable es similar a hacer público un campo de una clase. No es aconsejable. Sería mejor exponer una función para obtener el campo y una función para establecer el campo.

Sección 14.4: Estructuración de módulos PowerShell

En lugar de definir todas las funciones en un único archivo de módulo de secuencia de comandos PowerShell `.psm1`, es posible que desee dividir la función en archivos individuales. A continuación, puede dot-source estos archivos de su archivo de módulo de secuencia de comandos, que, en esencia, los trata como si fueran parte del propio archivo `.psm1`.

Considere esta estructura de directorios de módulos:

```
\MyCoolModule  
  \Functions  
    Function1.ps1  
    Function2.ps1  
    Function3.ps1  
MyCoolModule.psd1  
MyCoolModule.psm1
```

Dentro de tu archivo `MyCoolModule.psm1`, podrías insertar el siguiente código:

```
Get-ChildItem -Path $PSScriptRoot\Functions |  
  ForEach-Object -Process { . $PSItem.FullName }
```

De este modo, los archivos de función individuales se incluirían en el archivo de módulo `.psm1`.

Sección 14.5: Localización de los módulos

PowerShell busca módulos en los directorios listados en `$Env:PSModulePath`.

Un módulo llamado `foo`, en una carpeta llamada `foo` se encontrará con `Import-Module foo`.

En esa carpeta, PowerShell buscará un manifiesto de módulo (`foo.psd1`), un archivo de módulo (`foo.psm1`), una DLL (`foo.dll`).

Sección 14.6: Visibilidad de los miembros del módulo

Por defecto, sólo las funciones definidas en un módulo son visibles fuera del módulo. En otras palabras, si defines variables y alias en un módulo, no estarán disponibles excepto en el código del módulo.

Para anular este comportamiento, puede utilizar el cmdlet `Export-ModuleMember`. Tiene parámetros denominados `-Function`, `-Variable` y `-Alias` que permiten especificar exactamente qué miembros se exportan.

Es importante tener en cuenta que si utiliza `Export-ModuleMember`, **sólo** serán visibles los elementos que especifique.

Capítulo 15: Perfiles de PowerShell

Sección 15.1: Crear un perfil básico

Un perfil PowerShell se utiliza para cargar automáticamente variables y funciones definidas por el usuario.

Los perfiles PowerShell no se crean automáticamente para los usuarios.

Para crear un perfil PowerShell `C:>New-Item -ItemType File $profile`.

Si estás en ISE puedes utilizar el editor incorporado `C:>psEdit $profile`.

Una manera fácil de empezar con su perfil personal para el host actual es guardar algún texto en la ruta almacenada en la variable `$profile`.

```
"#Current host, current user" > $profile
```

Se pueden realizar más modificaciones en el perfil utilizando PowerShell ISE, el bloc de notas, Visual Studio Code o cualquier otro editor.

La variable `$profile` devuelve por defecto el perfil de usuario actual para el host actual, pero puedes acceder a la ruta a la política de máquina (todos los usuarios) y/o al perfil para todos los hosts (consola, ISE, terceros) utilizando sus propiedades.

```
PS> $PROFILE | Format-List -Force
AllUsersAllHosts      : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost  :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts  : C:\Users\user\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost :
C:\Users\user\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Length : 75
```

```
PS> $PROFILE.AllUsersAllHosts
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
```

Capítulo 16: Propiedades calculadas

Las propiedades calculadas en PowerShell son propiedades derivadas personalizadas (calculadas). Permiten al usuario dar formato a una determinada propiedad de la forma que desee. El cálculo (expresión) puede ser muy posiblemente cualquier cosa.

Sección 16.1: Tamaño del fichero en KB - Propiedades calculadas

Analicemos el siguiente fragmento,

```
Get-ChildItem -Path C:\MyFolder | Select-Object Name, CreationTime, Length
```

Simplemente emite el contenido de la carpeta con las propiedades seleccionadas. Algo como,

```
Name           CreationTime      Length
----           -
AnotherFile.txt 1/26/2017 2:45:02 PM 546000
filetomove.txt  1/5/2017 2:36:01 PM   5
```

¿Y si quiero mostrar el tamaño del archivo en KB? Aquí es donde las propiedades calculadas son útiles.

```
Get-ChildItem C:\MyFolder | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length / 1Kb}}
```

Que produce,

```
Name           Size_In_KB
----           -
AnotherFile.txt 533.203125
Secondfile.txt 1066.4111328125
```

La `Expression` es lo que contiene el cálculo para la propiedad calculada. Y sí, ¡puede ser cualquier cosa!

Capítulo 17: Utilizar clases estáticas existentes

Estas clases son bibliotecas de referencia de métodos y propiedades que no cambian de estado, en una palabra, inmutables. No necesitas crearlas, simplemente las usas. Las clases y métodos como estos se llaman clases estáticas porque no se crean, destruyen o cambian. Puedes referirte a una clase estática rodeando el nombre de la clase con corchetes.

Sección 17.1: Añadir tipos

Por nombre de conjunto, añada la biblioteca

```
Add-Type -AssemblyName "System.Math"
```

o por ruta del archivo:

```
Add-Type -Path "D:\Libs\CustomMath.dll"
```

Para Usar tipo añadido:

```
[CustomMath.Namespace]::Method(param1, $variableParam, [int]castMeAsIntParam)
```

Sección 17.2: Uso de la clase Math de .Net

Puede utilizar la clase .Net `Math` para realizar cálculos (`[System.Math]`)

Si desea conocer los métodos disponibles, puede utilizar:

```
[System.Math] | Get-Member -Static -MemberType Methods
```

He aquí algunos ejemplos de cómo utilizar la clase `Math`:

```
PS C:\> [System.Math]::Floor(9.42)
```

```
9
```

```
PS C:\> [System.Math]::Ceiling(9.42)
```

```
10
```

```
PS C:\> [System.Math]::Pow(4,3)
```

```
64
```

```
PS C:\> [System.Math]::Sqrt(49)
```

```
7
```

Sección 17.3: Creación instantánea de un nuevo GUID

Utilice clases .NET existentes de forma instantánea con PowerShell mediante `[class]::Method(args)`:

```
PS C:\> [guid]::NewGuid()
```

```
Guid
```

```
----
```

```
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Del mismo modo, en PowerShell 5+ puede utilizar el cmdlet `New-Guid`:

```
PS C:\> New-Guid
```

```
Guid
```

```
----
```

```
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Para obtener el GUID sólo como `[String]`, haga referencia a la propiedad `.Guid`:

```
[guid]::NewGuid().Guid
```

Capítulo 18: Variables incorporadas

PowerShell ofrece una variedad de útiles variables “automáticas” (incorporadas). Algunas variables automáticas sólo se rellenan en circunstancias especiales, mientras que otras están disponibles de forma global.

Sección 18.1: \$PSScriptRoot

```
Get-ChildItem -Path $PSScriptRoot
```

Este ejemplo recupera la lista de elementos hijos (directorios y archivos) de la carpeta donde reside el archivo de script.

La variable automática `$PSScriptRoot` es `$null` si se utiliza desde fuera de un archivo de código PowerShell. Si se utiliza dentro de una secuencia de comandos de PowerShell, define automáticamente la ruta completa del sistema de archivos al directorio que contiene el archivo de secuencia de comandos.

En Windows PowerShell 2.0, esta variable sólo es válida en módulos de scripts (.psm1). A partir de Windows PowerShell 3.0, es válida en todos los scripts.

Sección 18.2: \$Args

`$Args`

Contiene un array de los parámetros no declarados y/o valores de parámetros que se pasan a una función, script o bloque de script. Al crear una función, puede declarar los parámetros utilizando la palabra clave `param` o añadiendo una lista de parámetros separados por comas entre paréntesis después del nombre de la función.

En una acción de evento, la variable `$Args` contiene objetos que representan los argumentos del evento que se está procesando. Esta variable sólo se rellena dentro del bloque Acción de un comando de registro de eventos. El valor de esta variable también se puede encontrar en la propiedad `SourceArgs` del objeto `PSEventArgs` (`System.Management.Automation.PSEventArgs`) que devuelve `Get-Process`.

Sección 18.3: \$PSItem

```
Get-Process | ForEach-Object -Process {  
    $PSItem.Name  
}
```

Igual que `$_`. Contiene el objeto actual en el objeto pipeline. Puede utilizar esta variable en comandos que realicen una acción en cada objeto o en objetos seleccionados de una canalización.

Sección 18.4: \$?

```
Get-Process -Name doesnotexist  
Write-Host -Object "Was the last operation successful? $?"
```

Contiene el estado de ejecución de la última operación. Contiene TRUE si la última operación tuvo éxito y FALSE si falló.

Sección 18.5: \$error

```
Get-Process -Name doesnotexist  
Write-Host -Object ('The last error that occurred was: {0}' -f $error[0].Exception.Message)
```

Contiene un array de objetos de error que representan los errores más recientes. El error más reciente es el primer objeto de error del array (`$error[0]`).

Para evitar que un error se añada al array `$error`, utilice el parámetro común `ErrorAction` con el valor `Ignore`.

Capítulo 19: Variables automáticas

Las Variables Automáticas son creadas y mantenidas por Windows PowerShell. Uno tiene la capacidad de llamar a una variable casi cualquier nombre en el libro; Las únicas excepciones a esto son las variables que ya están siendo gestionados por PowerShell. Estas variables, sin duda, serán los objetos más repetitivos que utilice en PowerShell junto a las funciones (como `$?` - indica el estado de Éxito/Fracaso de la última operación).

Sección 19.1: `$OFS`

La variable denominada Separador del campo de salida contiene un valor de cadena de texto que se utiliza al convertir un array en una cadena de texto. Por defecto `$OFS = " "` (*un espacio*), pero se puede cambiar:

```
PS C:\> $array = 1,2,3
PS C:\> "$array" # se utilizará OFS por defecto
1 2 3
PS C:\> $OFS = ",." # cambiamos OFS por coma y punto
PS C:\> "$array"
1, .2, .3
```

Sección 19.2: `$?`

Contiene el estado de la última operación. Cuando no hay error, se establece en `True`:

```
PS C:\> Write-Host "Hello"
Hello
PS C:\> $?
True
```

Si hay algún error, se establece en `False`:

```
PS C:\> wrt-host
wrt-host : The term 'wrt-host' is not recognized as the name of a cmdlet, function, script file,
or operable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and
try again.
At line:1 char:1
+ wrt-host
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (wrt-host:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\> $?
False
```

Sección 19.3: `$null`

`$null` se utiliza para representar un valor ausente o indefinido.

`$null` se puede utilizar como un marcador de posición vacío para el valor vacío en arrays:

```
PS C:\> $array = 1, "string", $null
PS C:\> $array.Count
3
```

Si utilizamos el mismo array como fuente para `ForEach-Object`, procesará los tres elementos (incluyendo `$null`):

```
PS C:\> $array | ForEach-Object {"Hello"}
Hello
Hello
Hello
```

Tenga cuidado. Esto significa que `ForEach-Object` **PROCESARÁ** incluso `$null` por sí mismo:

```
PS C:\> $null | ForEach-Object {"Hello"} # ¡¡¡ESTO HARA UNA ITERACION!!!
Hello
```

Lo cual es un resultado muy inesperado si lo comparas con el clásico bucle `foreach`:

```
PS C:\> foreach($i in $null) {"Hello"} # ESTO NO HARÁ NINGUNA ITERACIÓN
PS C:\>
```

Sección 19.4: `$error`

Array de los objetos de error más recientes. El primero del array es el más reciente:

```
PS C:\> throw "Error" # el resultado aparecerá en rojo
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~
+ CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
+ FullyQualifiedErrorId : Error
PS C:\> $error[0] # el resultado será una cadena de texti normal (no roja)
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~
+ CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
+ FullyQualifiedErrorId : Error
```

Sugerencias de uso: Cuando utilice la variable `$error` en un cmdlet de formato (por ejemplo, `format-list`), tenga en cuenta que debe utilizar el modificador `-Force`. De lo contrario, el formato del cmdlet mostrará los contenidos de `$error` de la forma indicada anteriormente.

Las entradas de error se pueden eliminar mediante, por ejemplo, `$error.Remove($error[0])`.

Sección 19.5: `$pid`

Contiene el ID del proceso de alojamiento actual.

```
PS C:\> $pid
26080
```

Sección 19.6: Valores booleanos

`$true` y `$false` son dos variables que representan lógicamente `TRUE` y `FALSE`.

Tenga en cuenta que tiene que especificar el signo del dólar como primer carácter (que es diferente de `C#`).

```
$boolExpr = "abc".Length -eq 3 # la longitud de "abc" es 3, por lo tanto $boolExpr será True
if($boolExpr -eq $true){
    "Length is 3"
}
# el resultado será "Length is 3"
$boolExpr -ne $true
# el resultado será False
```

Fíjate que cuando usas booleanos verdadero/falso en tu código escribes `$true` o `$false`, pero cuando Powershell devuelve un booleano, parece `True` o `False`.

Sección 19.7: `$_` / `$PSItem`

Contiene el objeto/elemento que está siendo procesado actualmente por la canalización.

```
PS C:\> 1..5 | % { Write-Host "The current item is $_" }
The current item is 1
The current item is 2
The current item is 3
The current item is 4
The current item is 5
```

`$PSItem` y `$_` son idénticos y pueden utilizarse indistintamente, pero `$_` es con diferencia el más utilizado.

Sección 19.8: `$PSVersionTable`

Contiene una tabla hash de sólo lectura (`Constant`, `AllScope`) que muestra detalles sobre la versión de PowerShell que se está ejecutando en la sesión actual.

```
$PSVersionTable # esta llamada resulta en esto:
Name                                     Value
----                                     -
PSVersion                               5.0.10586.117
PSCompatibleVersions                    {1.0, 2.0, 3.0, 4.0...}
BuildVersion                             10.0.10586.117
CLRVersion                               4.0.30319.42000
WSManStackVersion                       3.0
PSRemotingProtocolVersion               2.3
SerializationVersion                   1.1.0.1
```

La forma más rápida de poner en marcha una versión de PowerShell:

```
$PSVersionTable.PSVersion
# result :
Major      Minor      Build      Revision
-----
5          0          10586      117
```

Capítulo 20: Variables de entorno

Sección 20.1: Las variables de entorno de Windows son visibles como una unidad PS llamada Env:

Puede ver la lista con todas las variables de entorno con:

```
Get-Childitem env:
```

Sección 20.2: Llamada instantánea de variables de entorno con \$env:

```
$env:COMPUTERNAME
```

Capítulo 21: Splatting

Splatting es un método para pasar múltiples parámetros a un comando como una sola unidad. Esto se hace almacenando los parámetros y sus valores como pares clave-valor en una hashtable y splatting a un cmdlet utilizando el operador splatting @.

Splatting puede hacer que un comando sea más legible y le permite reutilizar parámetros en múltiples llamadas a comandos.

Sección 21.1: Tuberías y Splatting

Declarar el splat es útil para reutilizar conjuntos de parámetros varias veces o con ligeras variaciones:

```
$splat = @{
    Class = "Win32_SystemEnclosure"
    Property = "Manufacturer"
    ErrorAction = "Stop"
}

Get-WmiObject -ComputerName $env:COMPUTERNAME @splat
Get-WmiObject -ComputerName "Computer2" @splat
Get-WmiObject -ComputerName "Computer3" @splat
```

Sin embargo, si el splat no está indentado para su reutilización, puede que no desee declararlo. En su lugar, se puede canalizar:

```
@{
    ComputerName = $env:COMPUTERNAME
    Class = "Win32_SystemEnclosure"
    Property = "Manufacturer"
    ErrorAction = "Stop"
} | % { Get-WmiObject @_ }
```

Sección 21.2: Pasar un parámetro Switch mediante splatting

Para utilizar Splatting para llamar a `Get-Process` con el interruptor `-FileVersionInfo` similar a este:

```
Get-Process -FileVersionInfo
```

Esta es la llamada utilizando splatting:

```
$MyParameters = @{
    FileVersionInfo = $true
}
```

```
Get-Process @MyParameters
```

Nota: Esto es útil porque puede crear un conjunto de parámetros por defecto y realizar la llamada muchas veces de esta forma

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters -Name WmiPrvSE
Get-Process @MyParameters -Name explorer
```

Sección 21.3: Splatting de la función de nivel superior a una serie de funciones internas

Sin splatting es muy engorroso intentar pasar valores hacia abajo a través de la pila de llamadas. Pero si combinas splatting con el poder de **@PSBoundParameters** entonces puedes pasar la colección de parámetros de nivel superior hacia abajo a través de las capas.

```
Function Outer-Method
{
    Param
    (
        [string]
        $First,

        [string]
        $Second
    )

    Write-Host ($First) -NoNewLine

    Inner-Method @PSBoundParameters
}
Function Inner-Method
{
    Param
    (
        [string]
        $Second
    )

    Write-Host (" {0}!" -f $Second)
}

$parameters = @{
    First = "Hello"
    Second = "World"
}
```

```
Outer-Method @parameters
```

Sección 21.4: Parámetros de splatting

El splatting se realiza sustituyendo el signo \$ por el operador de splatting @ cuando se utiliza una variable que contiene una HashTable de parámetros y valores en una llamada de comando.

```
$MyParameters = @{
    Name = "iexplore"
    FileVersionInfo = $true
}
```

```
Get-Process @MyParameters
```

Sin splatting:

```
Get-Process -Name "iexplore" -FileVersionInfo
```

Puede combinar parámetros normales con parámetros splattings para añadir fácilmente parámetros comunes a sus llamadas.

```
$MyParameters = @{  
    ComputerName = "StackOverflow-PC"  
}
```

```
Get-Process -Name "iexplore" @MyParameters
```

```
Invoke-Command -ScriptBlock { "Something to execute remotely" } @MyParameters
```

Capítulo 22: “Streams” de PowerShell: depuración, detalle, advertencia, error, salida e Información

Sección 22.1: Write-Output

`Write-Output` genera una salida. Esta salida puede ir al siguiente comando después de la tubería o a la consola por lo que simplemente se muestra.

El Cmdlet envía objetos por el pipeline primario, también conocido como “flujo de salida” o “pipeline de éxito”. Para enviar objetos de error por el canal de error, utilice `Write-Error`.

1.) Salida al siguiente Cmdlet del pipeline

```
Write-Output 'My text' | Out-File -FilePath "$env:TEMP\Test.txt"
```

```
Write-Output 'Bob' | ForEach-Object {  
    "My name is $_"  
}
```

2.) Salida a la consola ya que `Write-Output` es el último comando del pipeline

```
Write-Output 'Hello world'
```

3.) Falta 'Write-Output' CmdLet, pero la salida se sigue considerando 'Write-Output'
'Hello world'

- El cmdlet `Write-Output` envía el objeto especificado al siguiente comando.
- Si el comando es el último del pipeline, el objeto se muestra en la consola.
- El intérprete de PowerShell trata esto como un `Write-Output` implícito.

Dado que el comportamiento por defecto de `Write-Output` es mostrar los objetos al final de un pipeline, generalmente no es necesario utilizar el Cmdlet. Por ejemplo, `Get-Process | Write-Output` es equivalente a `Get-Process`.

Sección 22.2: Write Preferences

Los mensajes se pueden escribir con;

```
Write-Verbose "Detailed Message"  
Write-Information "Information Message"  
Write-Debug "Debug Message"  
Write-Progress "Progress Message"  
Write-Warning "Warning Message"
```

Cada una de ellas tiene una variable de preferencia;

```
$VerbosePreference = "SilentlyContinue"  
$InformationPreference = "SilentlyContinue"  
$DebugPreference = "SilentlyContinue"  
$ProgressPreference = "Continue"  
$WarningPreference = "Continue"
```

La variable de preferencia controla cómo se gestionan el mensaje y la posterior ejecución del script;

```
$InformationPreference = "SilentlyContinue"  
Write-Information "This message will not be shown and execution continues"
```

```
$InformationPreference = "Continue"  
Write-Information "This message is shown and execution continues"
```

```
$InformationPreference = "Inquire"  
Write-Information "This message is shown and execution will optionally continue"
```

```
$InformationPreference = "Stop"  
Write-Information "This message is shown and execution terminates"
```

El color de los mensajes puede controlarse para `Write-Error` configurando;

```
$host.PrivateData.ErrorBackgroundColor = "Black"  
$host.PrivateData.ErrorForegroundColor = "Red"
```

Existen opciones similares para `Write-Verbose`, `Write-Debug` y `Write-Warning`.

Capítulo 23: Envío de correo electrónico

Parámetro

`Attachments<String[]>`

`Bcc<String[]>`

`Body<String_>`

`BodyAsHtml`

`Cc<String[]>`

`Credential`

`DeliveryNotificationOption`

`Encoding`

`From`

`Port`

`Priority`

`SmtpServer`

`Subject`

`To`

`UseSsl`

Detalles

Ruta y nombres de los archivos que se adjuntarán al mensaje. Las rutas y los nombres de archivo se pueden enviar a [Send-MailMessage](#).

Direcciones de correo electrónico que reciben una copia de un mensaje de correo electrónico, pero no aparecen como destinatarios en el mensaje. Introduzca los nombres (opcional) y la dirección de correo electrónico (obligatorio), como Name `someone@example.com` o `someone@example.com`.

Contenido del mensaje de correo electrónico.

Indica que el contenido está en formato HTML.

Direcciones de correo electrónico que reciben una copia de un mensaje de correo electrónico. Introduzca los nombres (opcional) y la dirección de correo electrónico (obligatorio), como Name `someone@example.com` o `someone@example.com`.

Especifica una cuenta de usuario que tiene permiso para enviar mensajes desde la dirección de correo electrónico especificada. El valor predeterminado es el usuario actual. Introduzca un nombre como `User` o `Domain\User`, o introduzca un objeto `PSCredential`.

Especifica las opciones de notificación de entrega para el mensaje de correo electrónico. Se pueden especificar varios valores. Las notificaciones de entrega se envían en el mensaje a la dirección especificada en el parámetro `To`. Valores aceptables: `None`, `OnSuccess`, `OnFailure`, `Delay`, `Never`.

Codificación para el cuerpo y el asunto. Valores aceptables: `ASCII`, `UTF8`, `UTF7`, `UTF32`, `Unicode`, `BigEndianUnicode`, `Default`, `OEM`.

Direcciones de correo electrónico desde las que se envía el correo. Introduzca los nombres (opcional) y la dirección de correo electrónico (obligatorio), como Name `someone@example.com` o `someone@example.com`.

Puerto alternativo del servidor SMTP. El valor por defecto es 25. Disponible a partir de Windows PowerShell 3.0.

Prioridad del mensaje de correo electrónico. Valores aceptables: `Normal`, `High`, `Low`.

Nombre del servidor SMTP que envía el mensaje de correo electrónico. El valor por defecto es el valor de la variable `$PSEmailServer`.

Asunto del mensaje de correo electrónico.

Direcciones de correo electrónico a las que se envía el correo. Introduzca los nombres (opcional) y la dirección de correo electrónico (obligatorio), como Name `someone@example.com` o `someone@example.com`.

Utiliza el protocolo Secure Sockets Layer (SSL) para establecer una conexión con el ordenador remoto para enviar correo.

Una técnica útil para los administradores de Exchange Server es poder enviar mensajes de correo electrónico a través de SMTP desde PowerShell. Dependiendo de la versión de PowerShell instalada en su ordenador o servidor, existen múltiples formas de enviar correos electrónicos a través de PowerShell. Hay una opción de cmdlet nativo que es simple y fácil de usar. Utiliza el cmdlet [Send-MailMessage](#).

Sección 23.1: Send-MailMessage con parámetros predefinidos

```
$parameters = @{
    From = 'from@bar.com'
    To = 'to@bar.com'
    Subject = 'Email Subject'
    Attachments = @( 'C:\files\samplefile1.txt', 'C:\files\samplefile2.txt' )
    BCC = 'bcc@bar.com'
    Body = 'Email body'
    BodyAsHTML = $False
    CC = 'cc@bar.com'
    Credential = Get-Credential
    DeliveryNotificationOption = 'onSuccess'
    Encoding = 'UTF8'
    Port = '25'
    Priority = 'High'
    SmtServer = 'smtp.com'
    UseSSL = $True
}
```

Aviso: Splatting requiere @ en lugar de \$ delante del nombre de la variable
Send-MailMessage @parameters

Sección 23.2: Enviar mensaje de correo electrónico simple

```
Send-MailMessage -From sender@bar.com -Subject "Email Subject" -To receiver@bar.com -SmtServer smtp.com
```

Sección 23.3: SMTPClient - Correo con archivo .txt en el cuerpo del mensaje

```
# Definir el txt que irá en el cuerpo del email
$txt_File = "c:\file.txt"

function Send_mail {
    # Definir la configuración del correo electrónico
    $EmailFrom = "source@domain.com"
    $EmailTo = "destination@domain.com"
    $Txt_Body = Get-Content $txt_File -RAW
    $Body = $Body_Custom + $Txt_Body
    $Subject = "Email Subject"
    $SMTPServer = "smtpserver.domain.com"
    $SMTPClient = New-Object Net.Mail.SmtpClient($SMTPServer, 25)
    $SMTPClient.EnableSsl = $false
    $SMTPClient.Send($EmailFrom, $EmailTo, $Subject, $Body)
}

$Body_Custom = "This is what contain file.txt : "

Send_mail
```

Capítulo 24: Remotaciones PowerShell

Sección 24.1: Conexión a un servidor remoto mediante PowerShell

Utilizando las credenciales de su ordenador local:

```
Enter-PSSession 192.168.1.1
```

Solicitud de credenciales en el ordenador remoto

```
Enter-PSSession 192.168.1.1 -Credential $(Get-Credential)
```

Sección 24.2: Ejecutar comandos en un ordenador remoto

Una vez habilitado Powershell remoting (Enable-PSRemoting) puede ejecutar comandos en el equipo remoto de la siguiente manera:

```
Invoke-Command -ComputerName "RemoteComputerName" -ScriptBlock {  
    Write-Host "Remote Computer Name: $ENV:ComputerName"  
}
```

El método anterior crea una sesión temporal y la cierra justo después de que el comando o bloque de script finaliza.

Para dejar la sesión abierta y ejecutar otros comandos en ella más tarde, es necesario crear primero una sesión remota:

```
$Session = New-PSSession -ComputerName "RemoteComputerName"
```

A continuación, puede utilizar esta sesión cada vez que invoque comandos en el equipo remoto:

```
Invoke-Command -Session $Session -ScriptBlock {  
    Write-Host "Remote Computer Name: $ENV:ComputerName"  
}
```

```
Invoke-Command -Session $Session -ScriptBlock {  
    Get-Date  
}
```

Si necesita utilizar credenciales diferentes, puede añadirlas con el parámetro -Credential:

```
$Cred = Get-Credential  
Invoke-Command -Session $Session -Credential $Cred -ScriptBlock {...}
```

Advertencia de serialización remota

Nota:

Es importante saber que remoting serializa los objetos PowerShell en el sistema remoto y los deserializa en tu extremo de la sesión remoting, es decir, se convierten a XML durante el transporte y pierden todos sus métodos.

```
$Output = Invoke-Command -Session $Session -ScriptBlock {  
    Get-WmiObject -Class win32_printer  
}
```

```
$Output | Get-Member -MemberType Method  
TypeName: Deserialized.System.Management.ManagementObject # root\cimv2\Win32_Printer  
Name      MemberType Definition  
-----  
GetType   Method      type GetType()  
ToString  Method      string ToString(), string ToString(string format, System.IFormatProvi...
```

Mientras que usted tiene los métodos en el objeto PS regular:

```
Get-WmiObject -Class win32_printer | Get-Member -MemberType Method
TypeName: System.Management.ManagementObject # root\cimv2\Win32_Printer
Name MemberType Definition
-----
CancelAllJobs Method System.Management.ManagementBaseObject CancelAllJobs()
GetSecurityDescriptor Method System.Management.ManagementBaseObject GetSecurityDescriptor()
Pause Method System.Management.ManagementBaseObject Pause()
PrintTestPage Method System.Management.ManagementBaseObject PrintTestPage()
RenamePrinter Method System.Management.ManagementBaseObject RenamePrinter(System.String
NewPrinterName)
Reset Method System.Management.ManagementBaseObject Reset()
Resume Method System.Management.ManagementBaseObject Resume()
SetDefaultPrinter Method System.Management.ManagementBaseObject SetDefaultPrinter()
SetPowerState Method System.Management.ManagementBaseObject SetPowerState(System.UInt16
PowerState, System.String Time)
SetSecurityDescriptor Method System.Management.ManagementBaseObject
SetSecurityDescriptor(System.Management.ManagementObject # Win32_SecurityDescriptor Descriptor)
```

Uso de argumentos

Para utilizar argumentos como parámetros para el bloque de scripting remoto, se puede utilizar el parámetro `ArgumentList` de `Invoke-Command`, o utilizar la sintaxis `$Using:`.

Uso de `ArgumentList` con parámetros sin nombre (es decir, en el orden en que se pasan al scriptblock):

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $args[0]
    Remove-Item -Path $args[1] -ErrorAction SilentlyContinue -Force
}
```

Uso de `ArgumentList` con parámetros con nombre:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Param($serviceToShowInRemoteSession,$fileToDelete)
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $serviceToShowInRemoteSession
    Remove-Item -Path $fileToDelete -ErrorAction SilentlyContinue -Force
}
```

Utilizando la sintaxis `$Using::`:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ScriptBlock {
    Get-Service $Using:servicesToShow
    Remove-Item -Path $fileName -ErrorAction SilentlyContinue -Force
}
```

Sección 24.3: Habilitar PowerShell Remoting

En primer lugar, PowerShell Remoting debe estar habilitado en el servidor al que desea conectarse de forma remota.

```
Enable-PSRemoting -Force
```

Este comando hace lo siguiente:

- Ejecuta el cmdlet `Set-WSManQuickConfig`, que realiza las siguientes tareas:
- Inicia el servicio WinRM.
- Establece el tipo de inicio del servicio WinRM en Automático.
- Crea una escucha para aceptar peticiones en cualquier dirección IP, si no existe ya ninguna.
- Habilita una excepción en el cortafuegos para las comunicaciones WS-Management.
- Registra las configuraciones de sesión `Microsoft.PowerShell` y `Microsoft.PowerShell.Workflow`, si no están ya registradas.
- Registra la configuración de la sesión `Microsoft.PowerShell32` en equipos de 64 bits, si no está ya registrada.
- Activa todas las configuraciones de sesión.
- Cambia el descriptor de seguridad de todas las configuraciones de sesión para permitir el acceso remoto.
- Reinicia el servicio WinRM para hacer efectivos los cambios anteriores.

Sólo para entornos sin dominio

Para servidores en un dominio AD la autenticación remota PS se realiza a través de Kerberos ('Por defecto'), o NTLM ('Negociar'). Si desea permitir el acceso remoto a un servidor no perteneciente a un dominio, tiene dos opciones.

O bien configura la comunicación WSMAN a través de HTTPS (que requiere la generación de certificados) o activa la autenticación básica que envía tus credenciales a través del cable codificadas en base64 (que es básicamente lo mismo que texto plano, así que ten cuidado con esto).

En cualquier caso, tendrás que añadir los sistemas remotos a tu lista de hosts de confianza de WSMAN.

Activación de la autenticación básica

```
Set-Item WSMan:\localhost\Service\AllowUnencrypted $true
```

A continuación, en el ordenador *desde* el que deseas conectarte, debes indicarle que confíe en el ordenador al que *te* estás conectando.

```
Set-Item WSMan:\localhost\Client\TrustedHosts '192.168.1.1,192.168.1.2'  
Set-Item WSMan:\localhost\Client\TrustedHosts *.contoso.com  
Set-Item WSMan:\localhost\Client\TrustedHosts *
```

Importante: Debes decirle a tu cliente que confíe en el ordenador direccionado de la forma en que quieres conectarte (por ejemplo, si te conectas por IP, debe confiar en la IP y no en el nombre de host).

Sección 24.4: Una buena práctica para limpiar automáticamente las PSSessions

Cuando se crea una sesión remota mediante el cmdlet `New-PSSession`, la `PSSession` persiste hasta que finaliza la sesión actual de PowerShell. Esto significa que, por defecto, la `PSSession` y todos los recursos asociados seguirán utilizándose hasta que finalice la sesión actual de PowerShell.

Múltiples `PSSessions` activas pueden convertirse en una carga para los recursos, especialmente en el caso de scripts de larga ejecución o interconectados que crean cientos de `PSSessions` en una única sesión de PowerShell.

Es una buena práctica eliminar explícitamente cada `PSSession` después de que haya terminado de usarse. [1]

La siguiente plantilla de código utiliza `try-catch-finally` para lograr lo anterior, combinando la gestión de errores con una forma segura de garantizar que todas las PSSessions creadas se eliminan cuando terminan de utilizarse:

```
try
{
    $session = New-PSSession -Computername "RemoteMachineName"
    Invoke-Command -Session $session -ScriptBlock {write-host "This is running on
$ENV:ComputerName"}
}
catch
{
    Write-Output "ERROR: $_"
}
finally
{
    if ($session)
    {
        Remove-PSSession $session
    }
}
```

Referencias: [1]

<https://learn.microsoft.com/es-es/powershell/module/microsoft.powershell.core/new-pssession>

Capítulo 25: Trabajar con la tubería de PowerShell

PowerShell introduce un modelo de canalización de objetos, que permite enviar objetos enteros a través de la canalización para consumir commandlets o (al menos) la salida. A diferencia de la canalización clásica basada en cadenas, la información de los objetos canalizados no tiene que estar en posiciones específicas. Los commandlets pueden declarar interactuar con objetos de la canalización como entrada, mientras que los valores de retorno se envían a la canalización automáticamente.

Sección 25.1: Escribir funciones con el ciclo de vida avanzado

Este ejemplo muestra cómo una función puede aceptar una entrada en cadena e iterar eficientemente.

Tenga en cuenta que las estructuras de `begin` y `end` de la función son opcionales cuando se utiliza `pipelining`, pero ese `process` es necesario cuando se utiliza `ValueFromPipeline` o `ValueFromPipelineByPropertyName`.

```
function Write-FromPipeline{
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline)]
        $myInput
    )
    begin {
        Write-Verbose -Message "Beginning Write-FromPipeline"
    }
    process {
        Write-Output -InputObject $myInput
    }
    end {
        Write-Verbose -Message "Ending Write-FromPipeline"
    }
}
```

```
$foo = 'hello', 'world', 1, 2, 3
```

```
$foo | Write-FromPipeline -Verbose
```

Salida:

```
VERBOSE: Beginning Write-FromPipeline
hello
world
123
VERBOSE: Ending Write-FromPipeline
```

Sección 25.2: Soporte básico de tuberías en funciones

Este es un ejemplo de una función con el soporte más simple posible para `pipelining`. Cualquier función con soporte para pipeline debe tener al menos un parámetro con el `ParameterAttribute ValueFromPipeline` o `ValueFromPipelineByPropertyName` establecido, como se muestra a continuación.

```
function Write-FromPipeline {
    param(
        [Parameter(ValueFromPipeline)] # Establece el atributo ParameterAttribute
        [String]$Input
    )
    Write-Host $Input
}
```

```
$foo = 'Hello World!'
```

```
$foo | Write-FromPipeline
```

Salida:

```
Hello World!
```

Nota: En PowerShell 3.0 y superior, se admiten Default Values para ParameterAttributes. En versiones anteriores, debe especificar ValueFromPipeline=`$true`.

Sección 25.3: Concepto de funcionamiento de la tubería

En una serie pipeline, cada función se ejecuta en paralelo a las demás, como hilos paralelos. El primer objeto procesado se transmite a la siguiente canalización y el siguiente procesamiento se ejecuta inmediatamente en otro hilo. Esto explica la gran ganancia de velocidad en comparación con el estándar `ForEach`.

```
@( bigFile_1, bigFile_2, ..., bigFile_n) | Copy-File | Encrypt-File | Get-Md5
```

1. paso - copiar el primer archivo (en el hilo `Copy-File`)
2. paso - copiar el segundo archivo (en el hilo `Copy-File`) y simultáneamente encriptar el primero (en `Encrypt-File`)
3. paso - copiar el tercer archivo (en el hilo `Copy-File`) y simultáneamente encriptar el segundo archivo (en `Encrypt-File`) y simultáneamente `Get-Md5` del primero (en `Get-Md5`)

Capítulo 26: Trabajos en segundo plano de PowerShell

Los Jobs se introdujeron en PowerShell 2.0 y ayudaron a resolver un problema inherente a las herramientas de línea de comandos. En pocas palabras, si se inicia una tarea de larga duración, el indicador no estará disponible hasta que la tarea finalice. Como ejemplo de una tarea de larga ejecución, piense en este simple comando de PowerShell:

```
Get-ChildItem -Path c:\ -Recurse
```

Tardará un rato en obtener la lista completa de directorios de la unidad C:. Si lo ejecutas como Job, la consola recuperará el control y podrás capturar el resultado más tarde.

Sección 26.1: Creación de empleo básico

Iniciar un Bloque de Script como trabajo en segundo plano:

```
$job = Start-Job -ScriptBlock {Get-Process}
```

Iniciar un script como trabajo en segundo plano:

```
$job = Start-Job -FilePath "C:\YourFolder\Script.ps1"
```

Inicie un trabajo utilizando `Invoke-Command` en una máquina remota:

```
$job = Invoke-Command -ComputerName "ComputerName" -ScriptBlock {Get-Service winrm} -JobName "WinRM" -ThrottleLimit 16 -AsJob
```

Iniciar el trabajo como un usuario diferente (Solicita la contraseña):

```
Start-Job -ScriptBlock {Get-Process} -Credential "Domain\Username"
```

O

```
Start-Job -ScriptBlock {Get-Process} -Credential (Get-Credential)
```

Inicie el trabajo como un usuario diferente (No prompt):

```
$username = "Domain\Username"  
$password = "password"  
$secPassword = ConvertTo-SecureString -String $password -AsPlainText -Force  
$credentials = New-Object System.Management.Automation.PSCredential -ArgumentList @($username,  
$secPassword)  
Start-Job -ScriptBlock {Get-Process} -Credential $credentials
```

Sección 26.2: Gestión básica del trabajo

Obtiene una lista de todos los trabajos de la sesión actual:

```
Get-Job
```

Esperar a que termine un trabajo para obtener el resultado:

```
$job | Wait-job | Receive-Job
```

Tiempo de espera de una tarea si se prolonga demasiado (10 segundos en este ejemplo)

```
$job | Wait-job -Timeout 10
```

Detener un trabajo (completa todas las tareas pendientes en esa cola de trabajo antes de finalizar):

```
$job | Stop-Job
```

Elimina el trabajo de la lista de trabajos en segundo plano de la sesión actual:

\$job | [Remove-Job](#)

Nota: Lo siguiente sólo funcionará en Workflow.

Suspender un Workflow (Pausa):

\$job | [Suspend-Job](#)

Reanudar un Workflow:

\$job | [Resume-Job](#)

Capítulo 27: Comportamiento de retorno en PowerShell

Se puede utilizar para Salir del ámbito actual, que puede ser una función, un script o un bloque de script. En PowerShell, el resultado de cada sentencia se devuelve como salida, incluso sin una palabra clave `Return` explícita o para indicar que se ha alcanzado el final del ámbito.

Sección 27.1: Salida anticipada

```
function earlyexit {  
    "Hello"  
    return  
    "World"  
}
```

"Hello" se colocará en el canal de salida, "World" no

Sección 27.2: ¡Te pillé! Devolver en la tubería

```
Get-ChildItem | ForEach-Object { if ($_.IsReadOnly) { return } }
```

Los cmdlets de Pipeline (ej: `ForEach-Object`, `Where-Object`, etc) operan sobre cierres. El retorno aquí sólo se moverá al siguiente elemento en la tubería, no salir de procesamiento. Puede utilizar **break** en lugar de **return** si desea salir del proceso.

```
Get-ChildItem | ForEach-Object { if ($_.IsReadOnly) { break } }
```

Sección 27.3: Devolver con un valor

(parafraseado de [about_return](#))

Los siguientes métodos tendrán los mismos valores en la tubería

```
function foo {  
    $a = "Hello"  
    return $a  
}
```

```
function bar {  
    $a = "Hello"  
    $a  
    return  
}
```

```
function quux {  
    $a = "Hello"  
    $a  
}
```

Sección 27.4: Cómo trabajar con funciones de retorno

Una función devuelve todo lo que no es capturado por otra cosa.

Si utiliza la palabra clave **return**, no se ejecutarán las sentencias posteriores a la línea **return**.

Así:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    "Start"
    if($ExceptionalReturn){Return "Damn, it didn't work!"}
    New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
    Return "Yes, it worked!"
}
```

Test-Function

Devolverá:

- Inicio
- La clave de registro recién creada (esto se debe a que hay algunas declaraciones que crean una salida que puede que no esté esperando).
- ¡Sí, funcionó!

Test-Function -ExceptionalReturn Devolverá:

- Inicio
- ¡Maldición, no funcionó!

Si lo haces así:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    . {
        "Start"
        if($ExceptionalReturn)
        {
            $Return = "Damn, it didn't work!"
            Return
        }
        New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
        $Return = "Yes, it worked!"
        Return
    } | Out-Null
    Return $Return
}
```

Test-Function

Devolverá:

- ¡Sí, funcionó!

Test-Function -ExceptionalReturn Devolverá:

- ¡Maldición, no funcionó!

Con este truco puede controlar la salida devuelta incluso si no está seguro de qué escupirá cada sentencia. Funciona así

```
.{<Statements>} | Out-Null
```

el `.` hace que el siguiente bloque de script se incluya en el código
el `{ }` marca el bloque de script

el | `Out-Null` canaliza cualquier salida inesperada a `Out-Null` (¡para que desaparezca!) Como el bloque de script está incluido, tiene el mismo ámbito que el resto de la función. Así que puedes acceder a variables que fueron hechas dentro del scriptblock.

Sección 27.5: ¡Te pillé! Ignorar la salida no deseada

Inspirado por

- [PowerShell: La función no tiene un valor de retorno adecuado](#)

```
function bar {  
    [System.Collections.ArrayList]$MyVariable = @()  
    $MyVariable.Add("a") | Out-Null  
    $MyVariable.Add("b") | Out-Null  
    $MyVariable  
}
```

El `Out-Null` es necesario porque el método `.NET ArrayList.Add` devuelve el número de elementos de la colección después de añadir. Si se omite, la tubería habría contenido `1, 2, "a", "b"`

Existen múltiples formas de omitir la salida no deseada:

```
function bar  
{  
    # El cmdlet New-Item devuelve información sobre el archivo/carpeta recién creado  
    New-Item "test1.txt" | out-null  
    New-Item "test2.txt" > $null  
    [void](New-Item "test3.txt")  
    $tmp = New-Item "test4.txt"  
}
```

Nota: para saber más sobre por qué preferir `> $null`, véase [tema aún no creado].

Capítulo 28: Análisis de CSV

Sección 28.1: Uso básico de Import-Csv

Dado el siguiente archivo CSV

```
String,DateTime,Integer
First,2016-12-01T12:00:00,30
Second,2015-12-01T12:00:00,20
Third,2015-12-01T12:00:00,20
```

Se pueden importar las filas CSV en objetos PowerShell utilizando el comando `Import-Csv`

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows
```

String	DateTime	Integer
First	2016-12-01T12:00:00	30
Second	2015-11-03T13:00:00	20
Third	2015-12-05T14:00:00	20

```
> Write-Host $row[0].String1
Third
```

Sección 28.2: Importar desde CSV y asignar las propiedades al tipo correcto

Por defecto, `Import-Csv` importa todos los valores como cadenas de texto, por lo que para obtener objetos `DateTime` e `Integer`, necesitamos convertirlos o parsearlos.

Uso de `Foreach-Object`:

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | ForEach-Object {
    # Propiedades de fundición
    $_.DateTime = [datetime]$_ .DateTime
    $_.Integer = [int]$_ .Integer

    # Objeto de salida
    $_
}
```

Utilización de propiedades calculadas:

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | Select-Object String,
    @{name="DateTime";expression={ [datetime]$_ .DateTime }},
    @{name="Integer";expression={ [int]$_ .Integer }}
```

Salida:

String	DateTime	Integer
First	01.12.2016 12:00:00	30
Second	03.11.2015 13:00:00	20
Third	05.12.2015 14:00:00	20

Capítulo 29: Trabajar con archivos XML

Sección 29.1: Acceso a un archivo XML

```
<!-- file.xml -->
<people>
  <person id="101">
    <name>Jon Lajoie</name>
    <age>22</age>
  </person>
  <person id="102">
    <name>Lord Gaben</name>
    <age>65</age>
  </person>
  <person id="103">
    <name>Gordon Freeman</name>
    <age>29</age>
  </person>
</people>
```

Cargar un archivo XML

Para cargar un archivo XML, puede utilizar cualquiera de estas opciones:

```
# Primer metodo
$xml = New-Object System.Xml.XmlDocument
$file = Resolve-Path(".\file.xml")
$xml.load($file)
```

```
# Segundo metodo
[xml] $xml = Get-Content ".\file.xml"
```

```
# Tercer metodo
$xml = [xml] (Get-Content ".\file.xml")
```

Acceso a XML como objetos

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml
```

```
PS C:\> $xml.people
```

```
person
```

```
-----
{Jon Lajoie, Lord Gaben, Gordon Freeman}
```

```
PS C:\> $xml.people.person
```

```
id      name          age
--      -
101     Jon Lajoie    22
102     Lord Gaben    65
103     Gordon Freeman 29
```

```
PS C:\> $xml.people.person[0].name
Jon Lajoie
```

```
PS C:\> $xml.people.person[1].age
65
```

```
PS C:\> $xml.people.person[2].id
103
```

Acceso a XML con XPath

```
PS C:\> $xml = [xml](Get-Content file.xml)
```

```
PS C:\> $xml
```

```
PS C:\> $xml.SelectNodes("//people")
```

```
person
```

```
{Jon Lajoie, Lord Gaben, Gordon Freeman}
```

```
PS C:\> $xml.SelectNodes("//people//person")
```

id	name	age
101	Jon Lajoie	22
102	Lord Gaben	65
103	Gordon Freeman	29

```
PS C:\> $xml.SelectSingleNode("people//person[1]//name")
```

```
Jon Lajoie
```

```
PS C:\> $xml.SelectSingleNode("people//person[2]//age")
```

```
65
```

```
PS C:\> $xml.SelectSingleNode("people//person[3]//@id")
```

```
103
```

Acceso a XML que contiene espacios de nombres con XPath

```
PS C:\> [xml]$xml = @"
```

```
<ns:people xmlns:ns="http://schemas.xmlsoap.org/soap/envelope/">
```

```
<ns:person id="101">
```

```
<ns:name>Jon Lajoie</ns:name>
```

```
</ns:person>
```

```
<ns:person id="102">
```

```
<ns:name>Lord Gaben</ns:name>
```

```
</ns:person>
```

```
<ns:person id="103">
```

```
<ns:name>Gordon Freeman</ns:name>
```

```
</ns:person>
```

```
</ns:people>
```

```
"@
```

```
PS C:\> $ns = new-object Xml.XmlNamespaceManager $xml.NameTable
```

```
PS C:\> $ns.AddNamespace("ns", $xml.DocumentElement.NamespaceURI)
```

```
PS C:\> $xml.SelectNodes("//ns:people/ns:person", $ns)
```

id	name
101	Jon Lajoie
102	Lord Gaben
103	Gordon Freeman

Sección 29.2: Creación de un documento XML mediante XmlWriter()

```
# Establecer el formato
$xmlsettings = New-Object System.Xml.XmlWriterSettings
$xmlsettings.Indent = $true
$xmlsettings.IndentChars = " "

# Establecer el Nombre del archivo Crear el documento
$xmlWriter = [System.XML.XmlWriter]::Create("C:\YourXML.xml", $xmlsettings)

# Escribir la declaración XML y establecer el XSL
$xmlWriter.WriteStartDocument()
$xmlWriter.WriteProcessingInstruction("xml-stylesheet", "type='text/xsl' href='style.xsl'")

# Iniciar el elemento raíz
$xmlWriter.WriteStartElement("Root")

$xmlWriter.WriteStartElement("Object") # <-- Start <Object>

$xmlWriter.WriteElementString("Property1", "Value 1")
$xmlWriter.WriteElementString("Property2", "Value 2")

$xmlWriter.WriteStartElement("SubObject") # <-- Start <SubObject>
$xmlWriter.WriteElementString("Property3", "Value 3")
$xmlWriter.WriteEndElement() # <-- End <SubObject>

$xmlWriter.WriteEndElement() # <-- End <Object>

$xmlWriter.WriteEndElement() # <-- End <Root>

# Finalizar y cerrar el documento XML
$xmlWriter.WriteEndDocument()
$xmlWriter.Flush()
$xmlWriter.Close()
```

Archivo XML de salida

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<Root>
  <Object>
    <Property1>Value 1</Property1>
    <Property2>Value 2</Property2>
    <SubObject>
      <Property3>Value 3</Property3>
    </SubObject>
  </Object>
</Root>
```

Sección 29.3: Añadir fragmentos de XML al documento XML actual

Muestra de datos

Documento XML

En primer lugar, vamos a definir un documento XML de ejemplo llamado **"books.xml"** en nuestro directorio actual:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
  <book>
    <title>Of Mice And Men</title>
    <author>John Steinbeck</author>
    <pageCount>187</pageCount>
    <publishers>
      <publisher>
        <isbn>978-88-58702-15-4</isbn>
        <name>Pascal Covici</name>
        <year>1937</year>
        <binding>Hardcover</binding>
        <first>>true</first>
      </publisher>
      <publisher>
        <isbn>978-05-82461-46-8</isbn>
        <name>Longman</name>
        <year>2009</year>
        <binding>Hardcover</binding>
      </publisher>
    </publishers>
    <characters>
      <character name="Lennie Small" />
      <character name="Curley's Wife" />
      <character name="George Milton" />
      <character name="Curley" />
    </characters>
    <film>True</film>
  </book>
  <book>
    <title>The Hunt for Red October</title>
    <author>Tom Clancy</author>
    <pageCount>387</pageCount>
    <publishers>
      <publisher>
        <isbn>978-08-70212-85-7</isbn>
        <name>Naval Institute Press</name>
        <year>1984</year>
        <binding>Hardcover</binding>
        <first>>true</first>
      </publisher>
      <publisher>
        <isbn>978-04-25083-83-3</isbn>
        <name>Berkley</name>
        <year>1986</year>
        <binding>Paperback</binding>
      </publisher>
      <publisher>
        <isbn>978-08-08587-35-4</isbn>
        <name>Penguin Putnam</name>
        <year>2010</year>
        <binding>Paperback</binding>
      </publisher>
    </publishers>
    <characters>
```

```

        <character name="Marko Alexadrovich Ramius" />
        <character name="Jack Ryan" />
        <character name="Admiral Greer" />
        <character name="Bart Mancuso" />
        <character name="Vasily Borodin" />
    </characters>
    <film>True</film>
</book>
</books>

```

Nuevos datos

Lo que queremos hacer es añadir algunos libros nuevos a este documento, digamos *Patriot Games* de Tom Clancy (sí, soy fan de las obras de Clancy ^_^) y un favorito de Ciencia Ficción: *The Hitchhiker's Guide to the Galaxy*, de Douglas Adams, sobre todo porque es divertido leer a Zaphod Beeblebrox.

De alguna manera hemos adquirido los datos de los nuevos libros y los hemos guardado como una lista de `PSCustomObjects`:

```

$newBooks = @(
    [PSCustomObject] @{
        "Title" = "Patriot Games";
        "Author" = "Tom Clancy";
        "PageCount" = 540;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-39-913241-4";
                "Year" = "1987";
                "First" = $True;
                "Name" = "Putnam";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Jack Ryan", "Prince of Wales", "Princess of Wales",
            "Robby Jackson", "Cathy Ryan", "Sean Patrick Miller"
        );
        "film" = $True;
    },
    [PSCustomObject] @{
        "Title" = "The Hitchhiker's Guide to the Galaxy";
        "Author" = "Douglas Adams";
        "PageCount" = 216;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-33-025864-7";
                "Year" = "1979";
                "First" = $True;
                "Name" = "Pan Books";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Arthur Dent", "Marvin", "Zaphod Beeblebrox", "Ford Prefect",
            "Trillian", "Slartibartfast", "Dirk Gently"
        );
        "film" = $True;
    }
);

```

Plantillas

Ahora tenemos que definir algunas estructuras XML para nuestros nuevos datos. Básicamente, queremos crear un esqueleto/plantilla para cada lista de datos. En nuestro ejemplo, eso significa que necesitamos una plantilla

para el libro, los personajes y los editores. También podemos utilizar esto para definir algunos valores por defecto, como el valor de la etiqueta `film`.

```
$t_book = [xml] '@'
<book>
  <title />
  <author />
  <pageCount />
  <publishers />
  <characters />
  <film>False</film>
</book>
'@;

$t_publisher = [xml] '@'
<publisher>
  <isbn/>
  <name/>
  <year/>
  <binding/>
  <first>>false</first>
</publisher>
'@;

$t_character = [xml] '@'
<character name="" />
'@;
```

Hemos terminado con el montaje.

Añadir los nuevos datos

Ahora que ya tenemos nuestros datos de muestra, vamos a añadir los objetos personalizados al objeto documento XML.

```
# Leer el documento xml
$xml = [xml] Get-Content .\books.xml;

# Vamos a mostrar una lista de títulos para ver lo que tenemos actualmente:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};

# Salidas:
# title          author          ISBN
# -----
# Of Mice And Men      John Steinbeck    978-88-58702-15-4
# The Hunt for Red October  Tom Clancy        978-08-70212-85-7

# Mostremos también nuestros nuevos libros:
$newBooks | Select Title, Author, @{N="ISBN";E={$_Publishers[0].ISBN}};

# Salidas:
# Title          Author          ISBN
# -----
# Patriot Games      Tom Clancy        978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy  Douglas Adams    978-0-33-025864-7

# Ahora a fusionar los dos:

ForEach ( $book in $newBooks ) {
  $root = $xml.SelectSingleNode("/books");

  # Añade la plantilla de un book como un nuevo nodo al elemento raíz
  [void]$root.AppendChild($xml.ImportNode($t_book.book, $true));
```

```

# Seleccione el nuevo elemento hijo
$newElement = $root.SelectSingleNode("book[last()]");

# Actualizar los parámetros de ese nuevo elemento para que coincidan con nuestros datos
actuales del nuevo book.
$newElement.title = [String]$book.Title;
$newElement.author = [String]$book.Author;
$newElement.pageCount = [String]$book.PageCount;
$newElement.film = [String]$book.Film;

# Iterar a través de las propiedades que son Hijos de nuestro nuevo Elemento:
ForEach ( $publisher in $book.Publishers ) {
    # Crear el nuevo elemento hijo publisher
    # Observe el uso de "SelectSingleNode" aquí, esto permite el uso del método
    "AppendChild" ya que devuelve
    # un objeto de tipo XmlElement en lugar de los datos $Null que se almacenan
    actualmente en esa hoja del archivo
    # Arbol de documentos XML

    [void]$newElement.SelectSingleNode("publishers").AppendChild($xml.ImportNode($t_publisher.publisher , $true));

    # Actualizar los valores de atributo y texto de nuestro nuevo elemento XML para que
    coincidan con nuestros nuevos datos.
    $newPublisherElement = $newElement.SelectSingleNode("publishers/publisher[last()]");
    $newPublisherElement.year = [String]$publisher.Year;
    $newPublisherElement.name = [String]$publisher.Name;
    $newPublisherElement.binding = [String]$publisher.Binding;
    $newPublisherElement.isbn = [String]$publisher.ISBN;
    If ( $publisher.first ) {
        $newPublisherElement.first = "True";
    }
}

ForEach ( $character in $book.Characters ) {
    # Seleccione el elemento xml character
    $charactersElement = $newElement.SelectSingleNode("characters");

    # Añadir un nuevo elemento hijo de character
    [void]$charactersElement.AppendChild($xml.ImportNode($t_character.character, $true));

    # Seleccione los nuevos character/elemento de characters
    $characterElement = $charactersElement.SelectSingleNode("character[last()]");

    # Actualiza los valores de atributo y texto para que coincidan con nuestros nuevos
    datos
    $characterElement.name = [String]$character;
}
}

# Echa un vistazo al nuevo XML:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
    $_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};

```

Salida:

# title	author	ISBN
# -----	-----	----
# Of Mice And Men	John Steinbeck	978-88-58702-15-4
# The Hunt for Red October	Tom Clancy	978-08-70212-85-7
# Patriot Games	Tom Clancy	978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy	Douglas Adams	978-0-33-025864-7

Ahora podemos escribir nuestro XML en el disco, o en la pantalla, o en la web, ¡o donde sea!

Beneficios

Aunque puede que este no sea el procedimiento para todo el mundo, he descubierto que ayuda a evitar un montón de

```
[void]$xml.SelectSingleNode("/complicated/xpath/goes[here]").AppendChild($xml.CreateElement("newElementName")) seguido de  
$xml.SelectSingleNode("/complicated/xpath/goes/here/newElementName") = $textValue
```

Creo que el método detallado en el ejemplo es más limpio y fácil de interpretar para los humanos normales.

Mejoras

Puede ser posible cambiar la plantilla para incluir elementos con hijos en lugar de dividir cada sección como una plantilla separada. Sólo tienes que tener cuidado de clonar el elemento anterior cuando hagas un bucle a través de la lista.

Capítulo 30: Comunicación con las API RESTful

REST son las siglas de Representational State Transfer (Transferencia de Estado Representacional). Se basa en un protocolo de comunicaciones sin estado, cliente-servidor y almacenable en caché, y se utiliza sobre todo el protocolo HTTP. Se utiliza principalmente para crear servicios web ligeros, mantenibles y escalables. Un servicio basado en REST se denomina servicio RESTful y las API que se utilizan para ello son API RESTful. En PowerShell, [Invoke-RestMethod](#) se utiliza para tratar con ellos.

Sección 30.1: Enviar mensaje a hipChat

```
$params = @{
    Uri = "https://your.hipchat.com/v2/room/934419/notification?auth_token=???"
    Method = "POST"
    Body = @{
        color = 'yellow'
        message = "This is a test message!"
        notify = $false
        message_format = "text"
    } | ConvertTo-Json
    ContentType = 'application/json'
}
```

```
Invoke-RestMethod @params
```

Sección 30.2: Uso de REST con objetos PowerShell para GET y POST de muchos elementos

Obtenga sus datos REST con GET y almacénelos en un objeto PowerShell:

```
$Users = Invoke-RestMethod -Uri http://jsonplaceholder.typicode.com/users
```

Modifica muchos elementos de tus datos:

```
$Users[0].name = "John Smith"
$Users[0].email = "John.Smith@example.com"
$Users[1].name = "Jane Smith"
$Users[1].email = Jane.Smith@example.com
```

Devuelve por POST todos los datos REST:

```
$Json = $Users | ConvertTo-Json
Invoke-RestMethod -Method Post -Uri "http://jsonplaceholder.typicode.com/users" -Body $Json -
ContentType 'application/json'
```

Sección 30.3: Utilizar los Webhooks de entrada de Slack.com

Defina el payload a enviar para posibles datos más complejos

```
$Payload = @{ text="test string"; username="testuser" }
```

Utilice el cmdlet [ConvertTo-Json](#) e [Invoke-RestMethod](#) para ejecutar la llamada

```
Invoke-RestMethod -Uri "https://hooks.slack.com/services/yourwebhookstring" -Method Post -Body
(ConvertTo-Json $Payload)
```

Sección 30.4: Uso de REST con objetos PowerShell para obtener y colocar datos individuales

Obtenga sus datos REST con GET y almacénelos en un objeto PowerShell:

```
$Post = Invoke-RestMethod -Uri http://jsonplaceholder.typicode.com/posts/1
```

Modifica tus datos:

```
$Post.title = "New Title"
```

Devolver los datos REST con POST

```
$Json = $Post | ConvertTo-Json  
Invoke-RestMethod -Method Put -Uri "http://jsonplaceholder.typicode.com/posts/1" -Body $Json -  
ContentType 'application/json'
```

Sección 30.5: Uso de REST con PowerShell para eliminar elementos

Identifique el elemento que desea eliminar y bórralo:

```
Invoke-RestMethod -Method Delete -Uri http://jsonplaceholder.typicode.com/posts/1
```

Capítulo 31: Consultas SQL de PowerShell

Item	Descripcion
\$ServerInstance	Aquí tenemos que mencionar la instancia en la que está presente la base de datos
\$Databas	Aquí tenemos que mencionar la base de datos en la que está presente la tabla
\$Query	Aquí tenemos la consulta que queremos ejecutar en SQL
\$Username & \$Password	Nombre de usuario y contraseña que tienen acceso a la base de datos

En este documento se explica cómo utilizar las consultas SQL con PowerShell.

Sección 31.1: SQLEjemplo

Para consultar todos los datos de la tabla *MachineName* podemos utilizar un comando como el que se muestra a continuación.

```
$Query="Select * from MachineName"
```

```
$Inst="ServerInstance"
```

```
$DbName="DatabaseName"
```

```
$UID="User ID"
```

```
$Password="Password"
```

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password $Password
```

Sección 31.2: SQLQuery

Para consultar todos los datos de la tabla *MachineName* podemos utilizar un comando como el que se muestra a continuación.

```
$Query="Select * from MachineName"
```

```
$Inst="ServerInstance"
```

```
$DbName="DatabaseName"
```

```
$UID="User ID"
```

```
$Password="Password"
```

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password $Password
```

Capítulo 32: Expresiones regulares

Sección 32.1: Coincidencia simple

Puede determinar rápidamente si un texto incluye un patrón específico utilizando Regex. Hay varias maneras de trabajar con Regex en PowerShell.

```
# Texto de ejemplo
$text = @"
This is (a) sample
text, this is
a (sample text)
"@
```

```
# Patrón de muestra: Contenido envuelto en ()
$pattern = '\(.*?\)'
```

Utilización del operador `-match`

Para determinar si una cadena coincide con un patrón utilizando el operador incorporado `-matches`, utilice la sintaxis `'input' -match 'pattern'`. Esto devolverá verdadero o falso dependiendo del resultado de la búsqueda. Si hubo coincidencia, puede ver la coincidencia y los grupos (si están definidos en el patrón) accediendo a la variable `$Matches`.

```
> $text -match $pattern
True
```

```
> $Matches
```

Name	Value
0	(a)

También puede utilizar `-match` para filtrar a través de un array de cadenas de texto y devolver sólo las cadenas de texto que contengan una coincidencia.

```
> $textarray = @"
This is (a) sample
text, this is
a (sample text)
"@ -split "`n"
```

```
> $textarray -match $pattern
This is (a) sample
a (sample text)
```

Version ≥ 2.0

Uso de `Select-String`

PowerShell 2.0 introdujo un nuevo cmdlet para buscar texto mediante regex. Devuelve un objeto `MatchInfo` por cada entrada de texto que contenga una coincidencia. Puede acceder a sus propiedades para encontrar grupos coincidentes, etc.

```
> $m = Select-String -InputObject $text -Pattern $pattern
```

```
> $m
This is (a) sample
text, this is
a (sample text)
```

```
> $m | Format-List *
```

```
IgnoreCase      : True
LineNumber      : 1
Line            : This is (a) sample
                  text, this is
                  a (sample text)
Filename        : InputStream
Path            : InputStream
Pattern         : \(..*?\)
Context         :
Matches         : {(a)}
```

Al igual que `-match`, `Select-String` también se puede utilizar para filtrar a través de un array de cadenas de texto mediante la canalización de un array a la misma. Crea un objeto `MatchInfo` por cada cadena de texto que incluya una coincidencia.

```
> $textarray | Select-String -Pattern $pattern
```

```
This is (a) sample
a (sample text)
```

También puedes acceder a los partidos, grupos, etc.

```
> $textarray | Select-String -Pattern $pattern | fl *
```

```
IgnoreCase      : True
LineNumber      : 1
Line            : This is (a) sample
Filename        : InputStream
Path            : InputStream
Pattern         : \(..*?\)
Context         :
Matches         : {(a)}
IgnoreCase      : True
LineNumber      : 3
Line            : a (sample text)
Filename        : InputStream
Path            : InputStream
Pattern         : \(..*?\)
Context         :
Matches         : {(sample text)}
```

`Select-String` también puede buscar utilizando un patrón de texto normal (sin regex) añadiendo el modificador `-SimpleMatch`.

Uso de `[regex]::Match()`

También puede utilizar el método estático `Match()` disponible en la clase .NET `[Regex]`.

```
> [regex]::Match($text, $pattern)
```

```
Groups      : {(a)}  
Success     : True  
Captures   : {(a)}  
Index       : 8  
Length      : 3  
Value       : (a)
```

```
> [regex]::Match($text, $pattern) | Select-Object -ExpandProperty Value  
(a)
```

Sección 32.2: Sustituir

Una tarea común para regex es reemplazar el texto que coincide con un patrón por un nuevo valor.

```
# Texto de ejemplo  
$text = @"  
This is (a) sample  
text, this is  
a (sample text)  
"@
```

```
# Patrón de muestra: Texto envuelto en ()  
$pattern = '\(.*?\)'
```

```
# Sustituya las coincidencia por:  
$newvalue = 'test'
```

Uso del operador `-replace`

El operador `-replace` de PowerShell se puede utilizar para sustituir texto que coincida con un patrón por un nuevo valor utilizando la sintaxis `'input' -replace 'pattern', 'newvalue'`.

```
> $text -replace $pattern, $newvalue  
This is test sample  
text, this is  
a test
```

Uso del método `[regex]::Replace()`

El reemplazo de coincidencias también puede realizarse mediante el método `Replace()` de la clase `[Regex]` .NET.

```
[regex]::Replace($text, $pattern, 'test')  
This is test sample  
text, this is  
a test
```

Sección 32.3: Sustituir texto por un valor dinámico utilizando un `MatchEvaluator`

A veces es necesario sustituir un valor que coincide con un patrón por un nuevo valor basado en esa coincidencia específica, lo que hace imposible predecir el nuevo valor. Para este tipo de escenarios, un `MatchEvaluator` puede ser muy útil.

En PowerShell, un `MatchEvaluator` es tan simple como un scriptblock con un único parámetro que contiene un objeto `Match` para la coincidencia actual. La salida de la acción será el nuevo valor para esa coincidencia específica. `MatchEvaluator` se puede utilizar con el método estático `[Regex]::Replace()`.

Por ejemplo: Sustitución del texto dentro de `()` por su longitud

```
# Texto de ejemplo
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

# Patrón de muestra: Contenido envuelto en ()
$pattern = '(?<=\().*?(?=\))'

$MatchEvaluator = {
    param($match)

    # Sustituir el contenido por la longitud del contenido
    $match.Value.Length
}
```

Salida:

```
> [regex]::Replace($text, $pattern, $MatchEvaluator)
This is 1 sample
text, this is
a 11
```

Ejemplo: Poner la muestra en mayúsculas

```
# Patrón de muestra: "Sample"
$pattern = 'sample'

$MatchEvaluator = {
    param($match)

    # Devuelve la coincidencia en mayúsculas
    $match.Value.ToUpper()
}
```

Salida:

```
> [regex]::Replace($text, $pattern, $MatchEvaluator)

This is (a) SAMPLE
text, this is
a (SAMPLE text)
```

Sección 32.4: Escape de caracteres especiales

Un patrón regex utiliza muchos caracteres especiales para describir un patrón. Por ejemplo, `.` significa "cualquier carácter", `+` es "uno o más", etc.

Para utilizar estos caracteres, como `.`, `+`, etc., en un patrón, debe escaparlos para eliminar su significado especial. Esto se hace utilizando el carácter de escape que es una barra invertida `\` en regex. Ejemplo: Para buscar `+`, utilice el patrón `\+`.

Puede ser difícil recordar todos los caracteres especiales en regex, así que para escapar cada carácter especial en una cadena que quieras buscar, puedes usar el método `[Regex]::Escape("input")`.

```
> [regex]::Escape("(foo)")
\(\foo\)

> [regex]::Escape("1+1.2=2.2")
1\+1\.2=2\.2
```

Sección 32.5: Varias coincidencias

Hay varias formas de encontrar todas las coincidencias de un patrón en un texto.

```
# Texto de ejemplo
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

# Patrón de muestra: Contenido envuelto en ()
$pattern = '\(.*?\)'
```

Uso de `Select-String`

Puede encontrar todas las coincidencias (coincidencia global) añadiendo el modificador `-AllMatches` a `Select-String`.

```
> $m = Select-String -InputObject $text -Pattern $pattern -AllMatches
```

```
> $m | Format-List *
```

```
IgnoreCase      : True
LineNumber      : 1
Line            : This is (a) sample
                  text, this is
                  a (sample text)
Filename        : InputStream
Path            : InputStream
Pattern         : \(.*?\)
Context         :
Matches         : {(a), (sample text)}
```

```
# Lista de todas las coincidencias
```

```
> $m.Matches
```

```
Groups          : {(a)}
Success         : True
Captures       : {(a)}
Index           : 8
Length          : 3
Value           : (a)

Groups          : {(sample text)}
Success         : True
Captures       : {(sample text)}
Index           : 37
Length          : 13
Value           : (sample text)
```

```
# Obtener texto coincidente
```

```
> $m.Matches | Select-Object -ExpandProperty Value
(a)
(sample text)
```

Uso de `[regex]::Matches()`

El método `Matches()` de la clase .NET `[regex]`- también se puede utilizar para realizar una búsqueda global de múltiples coincidencias.

```
> [regex]::Matches($text,$pattern)
```

```
Groups      : {(a)}  
Success     : True  
Captures   : {(a)}  
Index       : 8  
Length      : 3  
Value       : (a)  
Groups      : {(sample text)}  
Success     : True  
Captures   : {(sample text)}  
Index       : 37  
Length      : 13  
Value       : (sample text)
```

```
> [regex]::Matches($text,$pattern) | Select-Object -ExpandProperty Value
```

```
(a)  
(sample text)
```

Capítulo 33: Alias

Sección 33.1: Get-Alias

Para listar todos los alias y sus funciones:

Get-Alias

Para obtener todos los alias de un cmdlet específico:

```
PS C:\> get-alias -Definition Get-ChildItem
```

CommandType	Name	Version	Source
Alias	dir -> Get-ChildItem		
Alias	gci -> Get-ChildItem		
Alias	ls -> Get-ChildItem		

Para encontrar alias por coincidencia:

```
PS C:\> get-alias -Name p*
```

CommandType	Name	Version	Source
Alias	popd -> Pop-Location		
Alias	proc -> Get-Process		
Alias	ps -> Get-Process		
Alias	pushd -> Push-Location		
Alias	pwd -> Get-Location		

Sección 33.2: Set-Alias

Este cmdlet permite crear nuevos nombres alternativos para los cmdlets de salida

```
PS C:\> Set-Alias -Name proc -Value Get-Process
```

```
PS C:\> proc
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	SI	ProcessName
292	17	13052	20444	...19	7.94	620	1	ApplicationFrameHost
....								

Tenga en cuenta que cualquier alias que cree se mantendrá sólo en la sesión actual. Cuando inicies una nueva sesión necesitas crear tus alias de nuevo. Los Perfiles Powershell (ver [tema no creado todavía]) son geniales para estos propósitos.

Capítulo 34: Utilizar la barra de progreso

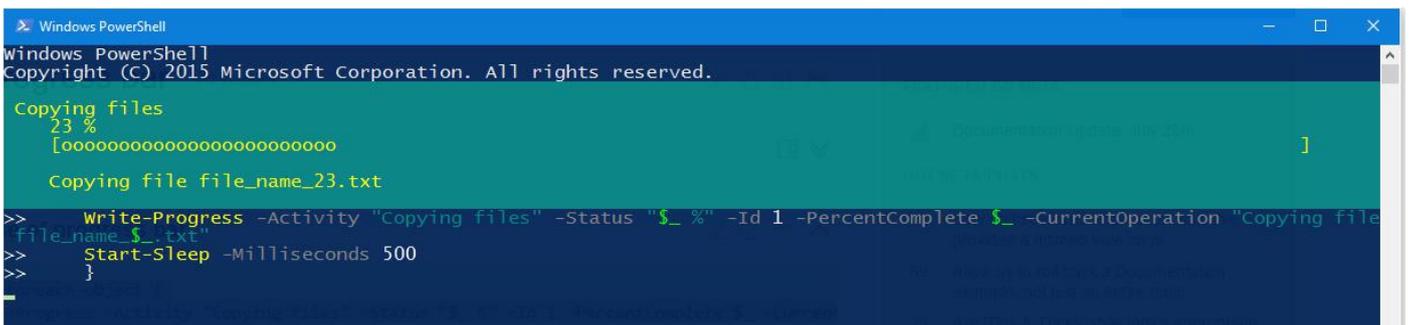
Una barra de progreso puede utilizarse para mostrar que algo está en proceso. Es una característica que ahorra tiempo y que uno debería tener. Las barras de progreso son increíblemente útiles mientras se depura para averiguar qué parte del script se está ejecutando, y son satisfactorias para las personas que ejecutan scripts para seguir lo que está sucediendo. Es común mostrar algún tipo de progreso cuando un script tarda mucho en completarse. Cuando un usuario lanza el script y no ocurre nada, uno empieza a preguntarse si el script se lanzó correctamente.

Sección 34.1: Uso sencillo de la barra de progreso

```
1..100 | ForEach-Object {  
    Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -PercentComplete $_ -  
    CurrentOperation "Copying file file_name_$_.txt"  
    Start-Sleep -Milliseconds 500  
    # sleep simula código de trabajo, sustituya esta línea por su código ejecutivo (es decir,  
    copia de archivos)  
}
```

Tenga en cuenta que, en aras de la brevedad, este ejemplo no contiene ningún código ejecutivo (simulado con `Start-Sleep`). Sin embargo, es posible ejecutarlo directamente tal cual y luego modificarlo y jugar con él.

Así es como se ve el resultado en la consola PS:



Así es como se ve el resultado en PS ISE:



Capítulo 35: Línea de comandos PowerShell.exe

Parámetro

`-Help | -? | /?`
`-File <FilePath> [<Args>]`
`-Command { - | <script-block> [-args <argarray>] | <string> [<CommandParameters>] }`
`-EncodedCommand <Base64EncodedCommand>`
`-ExecutionPolicy <ExecutionPolicy>`
`-InputFormat { Text | XML }`
`-Mta`
`-Sta`
`-NoExit`
`-NoLogo`
`-NonInteractive`
`-NoProfile`
`-OutputFormat { Text | XML }`
`-PSConsoleFile <FilePath>`
`-Version <Windows PowerShell version>`
`-Version <Windows PowerShell version>`

Descripción

Muestra la ayuda
Ruta al archivo de script que debe ejecutarse y argumentos (opcional)
Comandos a ejecutar seguidos de argumentos
Comandos codificados en Base64
Establece la política de ejecución sólo para este proceso
Establece el formato de entrada de los datos enviados al proceso. Texto (cadenas) o XML (CLIXML serializado)
PowerShell 3.0+: Ejecuta PowerShell en apartamento multihilo (STA es el predeterminado)
PowerShell 2.0: Ejecuta PowerShell en un departamento de un solo hilo (MTA es el predeterminado).
Deja la consola PowerShell en ejecución después de ejecutar el script/comando
Oculta el banner de derechos de autor en el lanzamiento
Oculta la consola al usuario
Evitar la carga de perfiles PowerShell para máquina o usuario
Establece el formato de salida de los datos devueltos por PowerShell. Texto (cadenas) o XML (CLIXML serializado)
Carga un archivo de consola creado previamente que configura el entorno (creado mediante [Export-Console](#)).
Especifica una versión de PowerShell para ejecutar. Se utiliza principalmente con **2.0**.
Especifica si se inicia el proceso PowerShell como una ventana `normal`, `hidden`, `minimized` o `maximized`.

Sección 35.1: Ejecutar un comando

El parámetro `-Command` se utiliza para especificar los comandos que se ejecutarán en el lanzamiento. Admite múltiples entradas de datos.

-Command <string>

Puede especificar los comandos a ejecutar en el lanzamiento como una cadena de caracteres. Se pueden ejecutar varias sentencias separadas por punto y coma `;`.

```
>PowerShell.exe -Command "(Get-Date).ToShortDateString()"
10.09.2016
```

```
>PowerShell.exe -Command "(Get-Date).ToShortDateString(); 'PowerShell is fun!'"
10.09.2016
PowerShell is fun!
```

-Command { scriptblock }

El parámetro `-Command` también admite una entrada de scriptblock (una o varias sentencias envueltas entre llaves `{ #code }`). Esto solo funciona cuando se llama a `PowerShell.exe` desde otra sesión de Windows PowerShell.

```
PS > powershell.exe -Command {
    "This can be useful, sometimes..."
    (Get-Date).ToShortDateString()
}
This can be useful, sometimes...
10.09.2016
```

-Command - (entrada estándar)

Puedes pasar comandos desde la entrada estándar utilizando `-Command -`. La entrada estándar puede proceder de `echo`, la lectura de un archivo, una aplicación de consola heredada, etc.

```
>echo "Hello World";"Greetings from PowerShell" | PowerShell.exe -NoProfile -Command -
Hello World
Greetings from PowerShell
```

Sección 35.2: Ejecutar un archivo de script

Puede especificar un archivo a un `ps1`-script para ejecutar su contenido en el lanzamiento utilizando el parámetro `-File`.

Script básico

MyScript.ps1

```
(Get-Date).ToShortDateString()
"Hello World"
```

Salida:

```
>PowerShell.exe -File Desktop\MyScript.ps1
10.09.2016
Hello World
```

Uso de parámetros y argumentos

Puede añadir parámetros y/o argumentos después de filepath para utilizarlos en el script. Los argumentos se utilizarán como valores para los parámetros de script no definidos/disponibles, el resto estará disponible en el array `$args`.

MyScript.ps1

```
param($Name)
```

```
"Hello $Name! Today's date it $((Get-Date).ToShortDateString())"
"First arg: $($args[0])"
```

Salida:

```
>PowerShell.exe -File Desktop\MyScript.ps1 -Name StackOverflow foo
Hello StackOverflow! Today's date it 10.09.2016
First arg: foo
```

Capítulo 36: Nombres de los cmdlets

Los CmdLets deben nombrarse utilizando un esquema de nomenclatura `<verb>-<noun>` para mejorar su localización.

Sección 36.1: Verbos

Los verbos utilizados para nombrar CmdLets deben nombrarse a partir de verbos de la lista proporcionada por [Get-Verb](#).

Encontrará más información sobre cómo utilizar los verbos en [Verbos aprobados para Windows PowerShell](#).

Sección 36.2: Sustantivos

Los sustantivos deben ir siempre en singular.

Sea coherente con los sustantivos. Por ejemplo, `Find-Package` necesita un proveedor, el sustantivo es `PackageProvider` y no `ProviderPackage`.

Capítulo 37: Ejecución de ejecutables

Sección 37.1: Aplicaciones GUI

```
PS> gui_app.exe (1)
PS> & gui_app.exe (2)
PS> & gui_app.exe | Out-Null (3)
PS> Start-Process gui_app.exe (4)
PS> Start-Process gui_app.exe -Wait (5)
```

Las aplicaciones GUI se inician en un proceso diferente y devuelven inmediatamente el control al host de PowerShell. A veces es necesario que la aplicación termine de procesarse antes de que se ejecute la siguiente sentencia PowerShell. Esto se puede conseguir canalizando la salida de la aplicación a `$null` (3) o utilizando `Start-Process` con el modificador `-Wait` (5).

Sección 37.2: Consola de flujos

```
PS> $ErrorActionPreference = "Continue" (1)
PS> & console_app.exe *>&1 | % { $_ } (2)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.ErrorRecord] } (3)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.WarningRecord] } (4)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.VerboseRecord] } (5)
PS> & console_app.exe *>&1 (6)
PS> & console_app.exe 2>&1 (7)
```

El flujo 2 contiene objetos `System.Management.Automation.ErrorRecord`. Tenga en cuenta que algunas aplicaciones como `git.exe` utilizan el "error stream" para fines informativos, que no son necesariamente errores en absoluto. En este caso es mejor mirar el código de salida para determinar si el "error stream" debe ser interpretado como errores.

PowerShell entiende estos flujos: `Output`, `Error`, `Warning`, `Verbose`, `Debug`, `Progress`. Las aplicaciones nativas suelen utilizar sólo estos flujos: `Output`, `Error`, `Warning`.

En PowerShell 5, todos los flujos se pueden redirigir al flujo de salida/éxito estándar (6).

En versiones anteriores de PowerShell, sólo se pueden redirigir flujos específicos al flujo de salida/éxito estándar (7). En este ejemplo, el "flujo de error" se redirigirá al flujo de salida.

Sección 37.3: Códigos de salida

```
PS> $LastExitCode
PS> $?
PS> $Error[0]
```

Estas son variables incorporadas en PowerShell que proporcionan información adicional sobre el error más reciente. `$LastExitCode` es el código de salida final de la última aplicación nativa que se ejecutó. `$?` y `$Error[0]` es el último registro de error generado por PowerShell.

Capítulo 38: Cumplimiento de los requisitos previos de los scripts

Sección 38.1: Imponer una versión mínima del host de PowerShell

```
#requires -version 4
```

Después de intentar ejecutar este script en una versión inferior, verá este mensaje de error

```
.\script.ps1: El script 'script.ps1' no se puede ejecutar porque contenía una sentencia "#requires" en la línea 1 para la versión 5.0 de Windows PowerShell. La versión requerida por el script no coincide con la versión actualmente en ejecución de Windows PowerShell versión 2.0.
```

Sección 38.2: Forzar la ejecución del script como administrador

```
Version ≥ 4.0  
#requires -RunAsAdministrator
```

Después de intentar ejecutar este script sin privilegios de administrador, verá este mensaje de error.

```
.\script.ps1: El script 'script.ps1' no se puede ejecutar porque contiene una sentencia "#requires" para ejecutarse como Administrador. La sesión actual de Windows PowerShell no se está ejecutando como Administrador. Inicie Windows PowerShell mediante la opción Ejecutar como administrador y, a continuación, intente ejecutar el script de nuevo.
```

Capítulo 39: Utilizar el sistema de ayuda

Sección 39.1: Actualización del sistema de ayuda

Version > 3.0

A partir de PowerShell 3.0, puede descargar y actualizar la documentación de ayuda sin conexión mediante un único cmdlet.

Update-Help

Para actualizar la ayuda en varios ordenadores (o en ordenadores no conectados a Internet).

Ejecute lo siguiente en un ordenador con los archivos de ayuda

```
Save-Help -DestinationPath \\Server01\Share\PSHelp -Credential $Cred
```

Para funcionar en muchos ordenadores a distancia

```
Invoke-Command -ComputerName (Get-Content Servers.txt) -ScriptBlock {Update-Help -SourcePath \\Server01\Share\Help -Credential $cred}
```

Sección 39.2: Uso de Get-Help

`Get-Help` se puede utilizar para ver la ayuda en PowerShell. Puede buscar cmdlets, funciones, proveedores u otros temas.

Para ver la documentación de ayuda sobre los trabajos, utilice:

```
Get-Help about_Jobs
```

Puede buscar temas utilizando comodines. Si desea obtener una lista de los temas de ayuda disponibles cuyo título empiece por `about_`, inténtelo:

```
Get-Help about_*
```

Si quisieras ayuda en `Select-Object`, usarías:

```
Get-Help Select-Object
```

También puedes utilizar los alias `help` o `man`.

Sección 39.3: Ver la versión en línea de un tema de ayuda

Puede acceder a la documentación de ayuda en línea utilizando:

```
Get-Help Get-Command -Online
```

Sección 39.4: Ejemplos de visualización

Mostrar ejemplos de uso de un cmdlet específico.

```
Get-Help Get-Command -Examples
```

Sección 39.5: Ver la página de ayuda completa

Consulte la documentación completa del tema.

```
Get-Help Get-Command -Full
```

Sección 39.6: Ver la ayuda de un parámetro específico

Puede ver la ayuda para un parámetro específico utilizando:

```
Get-Help Get-Content -Parameter Path
```

Capítulo 40: Módulos, scripts y funciones

Los *Módulos PowerShell* aportan capacidad de ampliación al administrador de sistemas, al DBA y al desarrollador. Ya sea simplemente como método para compartir funciones y scripts.

Las *Funciones PowerShell* son para evitar códigos repetitivos. Consulte [Funciones PS][1] [1]: Funciones PowerShell.

Los *Scripts PowerShell* se utilizan para automatizar tareas administrativas y consisten en un shell de línea de comandos y cmdlets asociados creados sobre .NET Framework.

Sección 40.1: Función

Una función es un bloque de código con nombre que se utiliza para definir código reutilizable que debe ser fácil de usar. Normalmente se incluye dentro de un script para ayudar a reutilizar código (para evitar código duplicado) o se distribuye como parte de un módulo para que sea útil para otros en múltiples scripts.

Escenarios en los que una función puede ser útil:

- Calcular la media de un grupo de números
- Generar un informe para los procesos en ejecución
- Escribir una función que compruebe si un ordenador está "sano" haciendo ping al ordenador y accediendo al c\$-share

Las funciones se crean utilizando la palabra clave `function`, seguida de un nombre de una sola palabra y un bloque de script que contiene el código que se ejecutará cuando se llame al nombre de la función.

```
function NameOfFunction {  
    Your code  
}
```

Demo

```
function HelloWorld {  
    Write-Host "Greetings from PowerShell!"  
}
```

Uso:

```
> HelloWorld  
Greetings from PowerShell!
```

Sección 40.2: Script

Un script es un archivo de texto con la extensión `.ps1` que contiene comandos PowerShell que se ejecutarán cuando se llame al script. Dado que los scripts son archivos guardados, son fáciles de transferir entre ordenadores.

Los guiones suelen escribirse para resolver un problema concreto, por ejemplo:

Ejecutar una tarea de mantenimiento semanal

Para instalar y configurar una solución/aplicación en un ordenador

Demo

MyFirstScript.ps1:

```
Write-Host "Hello World!"  
2+2
```

Puede ejecutar una secuencia de comandos introduciendo la ruta al archivo mediante un:

- Ruta absoluta, ej. `c:\MyFirstScript.ps1`
- Ruta relativa, ej. `.\MyFirstScript.ps1` si el directorio actual de su consola PowerShell es `C:\`

Uso:

```
> .\MyFirstScript.ps1
Hello World!
4
```

Un script también puede importar módulos, definir sus propias funciones, etc.

MySecondScript.ps1:

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}
```

```
HelloWorld
Write-Host "Let's get started!"
2+2
HelloWorld
```

Uso:

```
> .\MySecondScript.ps1
Greetings from PowerShell!
Let's get started!
4
Greetings from PowerShell!
```

Sección 40.3: Módulo

Un módulo es una colección de funciones reutilizables relacionadas (o cmdlets) que pueden distribuirse fácilmente a otros usuarios de PowerShell y utilizarse en varios scripts o directamente en la consola. Un módulo suele guardarse en su propio directorio y consta de:

- Uno o varios archivos de código con extensión `.psm1` que contengan funciones o conjuntos binarios (`.dll`) que contengan cmdlets.
- Un manifiesto de módulo `.psd1` que describa el nombre del módulo, la versión, el autor, la descripción, qué funciones/cmdlets proporciona, etc.
- Otros requisitos para que funcione, como dependencias, secuencias de comandos, etc.

Ejemplos de módulos:

- Un módulo que contiene funciones/cmdlets que realizan estadísticas sobre un conjunto de datos
- Un módulo para consultar y configurar bases de datos

Para facilitar que PowerShell encuentre e importe un módulo, a menudo se coloca en una de las ubicaciones conocidas de módulos de PowerShell definidas en `$env:PSModulePath`.

Demo

Enumera los módulos que están instalados en una de las ubicaciones de módulos conocidas:

```
Get-Module -ListAvailable
```

Importar un módulo, por ejemplo, el módulo `Hyper-V`:

```
Import-Module Hyper-V
```

Lista de comandos disponibles en un módulo, por ejemplo, el módulo `Microsoft.PowerShell.Archive`.

```
> Import-Module Microsoft.PowerShell.Archive
> Get-Command -Module Microsoft.PowerShell.Archive
```

CommandType	Name	Version	Source
Function	Compress-Archive	1.0.1.0	Microsoft.PowerShell.Archive
Function	Expand-Archive	1.0.1.0	Microsoft.PowerShell.Archive

Sección 40.4: Funciones avanzadas

Las funciones avanzadas se comportan de la misma manera que los cmdlets. PowerShell ISE incluye dos esqueletos de funciones avanzadas. Acceda a ellas a través del menú, editar, fragmentos de código o mediante Ctrl+J. (A partir de PS 3.0, las versiones posteriores pueden diferir).

Las principales funciones avanzadas son,

- ayuda integrada y personalizada para la función, accesible mediante `Get-Help`
- puede usar `[CmdletBinding()]` que hace que la función actúe como un cmdlet
- amplias opciones de parámetros

Versión sencilla:

```
<#
.Synopsis
    Descripción breve
.DESCRIPTION
    Descripción larga
.EXAMPLE
    Ejemplo de uso de este cmdlet
.EXAMPLE
    Otro ejemplo de cómo utilizar este cmdlet
#>
function Verb-Noun
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        # Descripción de la ayuda Param1
        [Parameter(Mandatory=$true,
            ValueFromPipelineByPropertyName=$true,
            Position=0)]
        $Param1,
        # Descripción de la ayuda de Param2
        [int]
        $Param2
    )

    Begin
    {
    }
    Process
    {
    }
    End
    {
    }
}
```

Versión completa:

```
<#
.Synopsis
    Descripción breve
.DESCRPTION
    Descripción larga
.EXAMPLE
    Ejemplo de uso de este cmdlet
.EXAMPLE
    Otro ejemplo de cómo utilizar este cmdlet
.INPUTS
    Entradas para este cmdlet (si las hay)
.OUTPUTS
    Salida de este cmdlet (si la hay)
.NOTES
    Notas generales
.COMPONENT
    Componente al que pertenece este cmdlet
.ROLE
    El rol al que pertenece este cmdlet
.FUNCTIONALITY
    La funcionalidad que mejor describe este cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
        SupportsShouldProcess=$true,
        PositionalBinding=$false,
        HelpUri = 'http://www.microsoft.com/',
        ConfirmImpact='Medium')]
    [OutputType([String])]
    Param
    (
        # Descripción de la ayuda Param1
        [Parameter(Mandatory=$true,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true,
            ValueFromRemainingArguments=$false,
            Position=0,
            ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,
        # Descripción de la ayuda de Param2
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,
        # Descripción de la ayuda de Param3
        [Parameter(ParameterSetName='Another Parameter Set')]
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )
}
```

```
Begin
{
}
Process
{
    if ($pscmdlet.ShouldProcess("Target", "Operation"))
    {
    }
}
End
{
}
}
```

Capítulo 41: Convenciones de denominación

Sección 41.1: Funciones

Get-User()

- Utilice el patrón *Verbo-Sustantivo* para nombrar una función.
- El verbo implica una acción, por ejemplo, `Get`, `Set`, `New`, `Read`, `Write` y muchos más. Consulte los [verbos aprobados](#).
- El sustantivo debe ser singular, aunque actúe sobre varios elementos. `Get-User()` puede devolver uno o varios usuarios.
- Utilice mayúsculas y minúsculas Pascal tanto para el verbo como para el sustantivo. Por ejemplo, `Get-UserLogin()`

Capítulo 42: Parámetros comunes

Sección 42.1: Parámetro ErrorAction

Los valores posibles son `Continue` | `Ignore` | `Inquire` | `SilentlyContinue` | `Stop` | `Suspend`.

El valor de este parámetro determinará cómo el cmdlet manejará los errores que no terminan (aquellos generados desde `Write-Host` por ejemplo; para aprender más sobre el manejo de errores vea [tema aún no creado]).

El valor por defecto (si se omite este parámetro) es `Continue`.

-ErrorAction Continue

Esta opción producirá un mensaje de error y continuará con la ejecución.

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
Write-Error "test" -ErrorAction Continue ; Write-Host "Second command" : test
+ CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException

Second command
```

-ErrorAction Ignore

Esta opción no producirá ningún mensaje de error y continuará con la ejecución. Tampoco se añadirá ningún error a la variable automática `$Error`. Esta opción se introdujo en la v3.

```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
Second command
```

-ErrorAction Inquire

Esta opción producirá un mensaje de error y pedirá al usuario que elija una acción a tomar.

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
Confirm
test
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): _
```

-ErrorAction SilentlyContinue

Esta opción no producirá un mensaje de error y continuará con la ejecución. Todos los errores se añadirán a la variable automática `$Error`.

```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
Second command
```

-ErrorAction Stop

Esta opción producirá un mensaje de error y no continuará con la ejecución.

```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
Write-Error "test" -ErrorAction Stop ; Write-Host "Second command" : test
At line:1 char:1
+ Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
```

-ErrorAction Suspend

Sólo disponible en flujos de trabajo Powershell. Cuando se utiliza, si el comando se encuentra con un error, el flujo de trabajo se suspende. Esto permite investigar dicho error y ofrece la posibilidad de reanudar el flujo de trabajo. Para obtener más información sobre el sistema de flujo de trabajo, consulta [tema aún no creado].

Capítulo 43: Conjuntos de parámetros

Los **conjuntos de parámetros** se utilizan para limitar la combinación posible de parámetros, o para imponer el uso de parámetros cuando se seleccionan 1 o más parámetros.

Los ejemplos explicarán el uso y la razón de un conjunto de parámetros.

Sección 43.1: Parámetro establecido para imponer el uso de un parámetro cuando se selecciona otro

Cuando desee, por ejemplo, imponer el uso del parámetro Contraseña si se proporciona el parámetro Usuario. (y viceversa).

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomethingToDo,
        [Parameter(ParameterSetName="Credentials", mandatory=$false)]
        [String]$Computername = "localhost",
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [String]$User,
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [SecureString]$Password
    )
    # Haz algo
}

# Esto no funcionará le pedirá usuario y contraseña
Do-Something -SomethingToDo 'get-help about_Functions_Advanced' -ComputerName

# Esto no funcionará, te pedirá la contraseña.
Do-Something -SomethingToDo 'get-help about_Functions_Advanced' -User
```

Sección 43.2: Parámetro establecido para limitar la combinación de parámetros

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomethingToDo,
        [Parameter(ParameterSetName="Silently", mandatory=$false)]
        [Switch]$Silently,
        [Parameter(ParameterSetName="Loudly", mandatory=$false)]
        [Switch]$Loudly
    )
    # Haz algo
}

# Esto no funcionará porque no se puede utilizar la combinación Silently y Loudly
Do-Something -SomethingToDo 'get-help about_Functions_Advanced' -Silently -Loudly
```

Capítulo 44: Parámetros dinámicos de PowerShell

Sección 44.1: "Parámetro dinámico "simple"

Este ejemplo añade un nuevo parámetro a `MyTestFunction` si `$SomeUsefulNumber` es mayor que 5.

```
function MyTestFunction
{
    [CmdletBinding(DefaultParameterSetName='DefaultConfiguration')]
    Param
    (
        [Parameter(Mandatory=$true)][int]$SomeUsefulNumber
    )

    DynamicParam
    {
        $paramDictionary = New-Object -Type
System.Management.Automation.RuntimeDefinedParameterDictionary
        $attributes = New-Object System.Management.Automation.ParameterAttribute
        $attributes.ParameterSetName = "__AllParameterSets"
        $attributes.Mandatory = $true
        $attributeCollection = New-Object -Type
System.Collections.ObjectModel.Collection[System.Attribute]
        $attributeCollection.Add($attributes)
        # Si "SomeUsefulNumber" es superior a 5, añade el parámetro "MandatoryParam1".
        if($SomeUsefulNumber -gt 5)
        {
            # Crear un parámetro de cadena obligatorio llamado "MandatoryParam1".
            $dynParam1 = New-Object -Type
System.Management.Automation.RuntimeDefinedParameter("MandatoryParam1", [String],
$attributeCollection)
            # Añade el nuevo parámetro al diccionario
            $paramDictionary.Add("MandatoryParam1", $dynParam1)
        }
        return $paramDictionary
    }

    process
    {
        Write-Host "SomeUsefulNumber = $SomeUsefulNumber"
        # Tenga en cuenta que los parámetros dinámicos necesitan una sintaxis específica
        Write-Host ("MandatoryParam1 = {0}" -f $PSBoundParameters.MandatoryParam1)
    }
}
```

Uso:

```
PS > MyTestFunction -SomeUsefulNumber 3
SomeUsefulNumber = 3
MandatoryParam1 =
```

```
PS > MyTestFunction -SomeUsefulNumber 6
cmdlet MyTestFunction at command pipeline position 1
Supply values for the following parameters:
MandatoryParam1:
```

```
PS > MyTestFunction -SomeUsefulNumber 6 -MandatoryParam1 test
SomeUsefulNumber = 6
MandatoryParam1 = test
```

En el segundo ejemplo de uso, se puede ver claramente que falta un parámetro.

Los parámetros dinámicos también se tienen en cuenta con el autocompletado. Esto es lo que ocurre si pulsa ctrl + espacio al final de la línea:

```
PS >MyTestFunction -SomeUsefulNumber 3 -<ctrl+space>
Verbose WarningAction WarningVariable OutBuffer
Debug InformationAction InformationVariable PipelineVariable
ErrorAction ErrorVariable OutVariable
```

```
PS >MyTestFunction -SomeUsefulNumber 6 -<ctrl+space>
MandatoryParam1 ErrorAction ErrorVariable OutVariable
Verbose WarningAction WarningVariable OutBuffer
Debug InformationAction InformationVariable PipelineVariable
```

Capítulo 45: GUI en PowerShell

Sección 45.1: Interfaz gráfica de usuario WPF para el cmdlet Get-Service

```
Add-Type -AssemblyName PresentationFramework
```

```
[xml]$XAMLWindow = '  
<Window  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    Height="Auto"  
    SizeToContent="WidthAndHeight"  
    Title="Get-Service">  
    <ScrollViewer Padding="10,10,10,0" ScrollViewer.VerticalScrollBarVisibility="Disabled">  
        <StackPanel>  
            <StackPanel Orientation="Horizontal">  
                <Label Margin="10,10,0,10">ComputerName:</Label>  
                <TextBox Name="Input" Margin="10" Width="250px"></TextBox>  
            </StackPanel>  
            <DockPanel>  
                <Button Name="ButtonGetService" Content="Get-Service" Margin="10"  
                    Width="150px" IsEnabled="false"/>  
                <Button Name="ButtonClose" Content="Close" HorizontalAlignment="Right"  
                    Margin="10" Width="50px"/>  
            </DockPanel>  
        </StackPanel>  
    </ScrollViewer >  
</Window>  
'  
  
# Crear el objeto Window  
$Reader=(New-Object System.Xml.XmlNodeReader $XAMLWindow)  
$Window=[Windows.Markup.XamlReader]::Load( $Reader )  
  
# Manejador de eventos TextChanged para entrada  
$TextboxInput = $Window.FindName("Input")  
$TextboxInput.add_TextChanged.Invoke({  
    $ComputerName = $TextboxInput.Text  
    $ButtonGetService.IsEnabled = $ComputerName -ne ''  
})  
  
# Manejador de Eventos Click para ButtonClose  
$ButtonClose = $Window.FindName("ButtonClose")  
$ButtonClose.add_Click.Invoke({  
    $Window.Close();  
})  
  
# Manejador de eventos de clic para ButtonGetService  
$ButtonGetService = $Window.FindName("ButtonGetService")  
$ButtonGetService.add_Click.Invoke({  
    $ComputerName = $TextboxInput.text.Trim()  
    try{  
        Get-Service -ComputerName $computerName | Out-GridView -Title "Get-Service on  
        $ComputerName"  
    } catch {  
        [System.Windows.MessageBox]::Show($_.exception.message, "Error", [System.Windows.Message  
        BoxButton]::OK, [System.Windows.MessageBoxImage]::Error)  
    }  
})  
  
# Abrir Window  
$Window.ShowDialog() | Out-Null
```

Esto crea una ventana de diálogo que permite al usuario seleccionar el nombre de un ordenador, a continuación, se mostrará una tabla de servicios y sus estados en ese equipo. Este ejemplo utiliza WPF en lugar de Windows Forms.

Capítulo 46: Codificación/Decodificación de URL

Sección 46.1: Codificar cadena de caracteres de consulta con `[System.Web.HttpUtility]::UrlEncode()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qqs_data = @{
    'foo1'='bar1';
    'foo2'='complex;/?:@&=+$, bar''''';
    'complex;/?:@&=+$, foo''''='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [System.Web.HttpUtility]::UrlEncode($qs.Name)
    $qs_value = [System.Web.HttpUtility]::UrlEncode($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

Con `[System.Web.HttpUtility]::UrlEncode()`, observará que los espacios se convierten en signos más (+) en lugar de `%20`:

```
https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&
complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1
```

Sección 46.2: Inicio rápido: Codificación

```
$url1 = [uri]::EscapeDataString("http://test.com?test=my value")
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value

$url2 = [uri]::EscapeUriString("http://test.com?test=my value")
# url2: http://test.com?test=my%20value

# HttpUtility requiere al menos .NET 1.1 para ser instalado.
$url3 = [System.Web.HttpUtility]::UrlEncode("http://test.com?test=my value")
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
```

Nota: [Más información sobre HTTPUtility.](#)

Sección 46.3: Inicio rápido: Decodificación

Nota: estos ejemplos utilizan las variables creadas en el *Inicio rápido: Codificación*.

```
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[uri]::UnescapeDataString($url1)
# Devuelve: http://test.com?test=my value

# url2: http://test.com?test=my%20value
[uri]::UnescapeDataString($url2)
# Devuelve: http://test.com?test=my value

# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[uri]::UnescapeDataString($url3)
# Devuelve: http://test.com?test=my+value
# Nota: No existe `[uri]::UnescapeUriString()`;
# lo que tiene sentido ya que el `[uri]::UnescapeDataString()`
# maneja todo lo que esta función manejaría y más.

# HttpUtility requiere al menos .NET 1.1 para ser instalado.
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[System.Web.HttpUtility]::UrlDecode($url1)
# Devuelve: http://test.com?test=my value

# HttpUtility requiere al menos .NET 1.1 para ser instalado.
# url2: http://test.com?test=my%20value
[System.Web.HttpUtility]::UrlDecode($url2)
# Devuelve: http://test.com?test=my value

# HttpUtility requiere al menos .NET 1.1 para ser instalado.
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[System.Web.HttpUtility]::UrlDecode($url3)
# Devuelve: http://test.com?test=my value
```

Nota: [Más información sobre HTTPUtility](#).

Sección 46.4: Codifique la cadena de consulta con `[uri]::EscapeDataString()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qqs_data = @{
    'foo1'='bar1';
    'foo2'='complex;/?:@&=+$, bar''''';
    'complex;/?:@&=+$, foo''''='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [uri]::EscapeDataString($qs.Name)
    $qs_value = [uri]::EscapeDataString($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

Con `[uri]::EscapeDataString()`, observará que el apóstrofo (') no se ha codificado:

```
https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&
complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1
```

Sección 46.5: Descodificar URL con `[uri]::UnescapeDataString()`

Codificado con `[uri]::EscapeDataString()`

En primer lugar, descodificaremos la URL y la cadena de consulta codificadas con `[uri]::EscapeDataString()` en el ejemplo anterior:

```
https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&
complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1
```

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&comple
x%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1'
$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*(\?([^#]*)?)(#.*)?)?'
# Ver Observaciones

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Esto le devuelve `[hashtable]$url_parts`; que es igual a (**Nota:** los *espacios* en las partes complejas son *espacios*):

```
PS > $url_parts

Name                Value
----                -
Scheme              https
Path                /foos
Server              example.vertigion.com
QueryString          foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40
                    %26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1
QueryStringParts    {foo2, complex;/?:@&=+$, foo', foo1}

PS > $url_parts.QueryStringParts

Name                Value
----                -
foo2                complex;/?:@&=+$, bar'
complex;/?:@&=+$, foo'
foo1                bar1
```

Codificado con [System.Web.HttpUtility]::UrlEncode()

Ahora, decodificaremos la URL y la Cadena de Consulta codificada con [System.Web.HttpUtility]::UrlEncode() en el ejemplo anterior:

```
https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&
complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1
```

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex
%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1'

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\[?([^\]]*)\]?)(#(?:.)*)?'
# Ver Observaciones

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Esto le devuelve [hashtable] \$url_parts; que es igual a (**Nota:** los espacios en las partes complejas son espacios):

```
PS > $url_parts
```

Name	Value
-----	-----
Scheme	https
Path	/foos
Server	example.vertigion.com
QueryString	foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1
QueryStringParts	{foo2, complex;/?:@&=+\$, foo'", foo1}

```
PS > $url_parts.QueryStringParts
```

Name	Value
-----	-----
foo2	complex;/?:@&=+\$, bar'"
complex;/?:@&=+\$, foo'"	bar2
foo1	bar1

Capítulo 47: Tratamiento de errores

Este tema trata sobre Tipos de Error y Manejo de Errores en PowerShell.

Sección 47.1: Tipos de error

Un error es un error, uno podría preguntarse cómo podría haber tipos en él. Bueno, con PowerShell el error cae ampliamente en dos criterios,

- Error de terminación
- Error no terminal

Como su nombre indica, los errores de terminación terminarán la ejecución y los errores de no terminación permitirán que la ejecución continúe con la siguiente sentencia.

Esto es cierto asumiendo que el valor de **\$ErrorActionPreference** es por defecto (Continue). **\$ErrorActionPreference** es una [Variable de Preferencia](#) que le dice a PowerShell qué hacer en caso de un error "Non-Terminating".

Error de terminación

Un error de terminación puede tratarse con una captura try típica, como la siguiente.

```
Try
{
    Write-Host "Attempting Divide By Zero"
    1/0
}
Catch
{
    Write-Host "A Terminating Error: Divide by Zero Caught!"
}
```

El fragmento anterior se ejecutará y el error será capturado a través del bloque catch.

Error no terminal

Por otro lado, un error no terminal no será capturado en el bloque catch por defecto. La razón es que un error no terminante no se considera un error crítico.

```
Try
{
    Stop-Process -Id 123456
}
Catch
{
    Write-Host "Non-Terminating Error: Invalid Process ID"
}
```

Si ejecuta la línea anterior no obtendrá la salida del bloque catch ya que el error no se considera crítico y la ejecución simplemente continuará a partir del siguiente comando. Sin embargo, el error se mostrará en la consola. Para manejar un error No-Terminante, simplemente tiene que cambiar la preferencia de error.

```
Try
{
    Stop-Process -Id 123456 -ErrorAction Stop
}
Catch
{
    "Non-Terminating Error: Invalid Process ID"
}
```

Ahora, con la preferencia de Error actualizada, este error se considerará un error de Terminación y se capturará en el bloque catch.

Invocación de errores de terminación y no terminación:

El cmdlet `Write-Error` simplemente escribe el error en el programa anfitrión que lo invoca. No detiene la ejecución. Mientras que `throw` le dará un error de terminación y detendrá la ejecución

```
Write-host "Going to try a non terminating Error "  
Write-Error "Non terminating"  
Write-host "Going to try a terminating Error "  
throw "Terminating Error "  
Write-host "This Line won't be displayed"
```

Capítulo 48: Gestión de paquetes

La gestión de paquetes PowerShell permite buscar, instalar, actualizar y desinstalar módulos PowerShell y otros paquetes.

PowerShellGallery.com es la fuente predeterminada de módulos PowerShell. También puede navegar por el sitio para los paquetes disponibles, comando y vista previa del código.

Sección 48.1: Crear el repositorio de módulos PowerShell por defecto

Si por alguna razón, el repositorio de módulos PowerShell por defecto PSGallery se elimina. Tendrá que crearlo. Este es el comando.

```
Register-PSRepository -Default
```

Sección 48.2: Buscar un módulo por su nombre

```
Find-Module -Name <Name>
```

Sección 48.3: Instalar un módulo por nombre

```
Install-Module -Name <name>
```

Sección 48.4: Desinstalar un módulo mi nombre y versión

```
Uninstall-Module -Name <Name> -RequiredVersion <Version>
```

Sección 48.5: Actualizar un módulo por nombre

```
Update-Module -Name <Name>
```

Sección 48.6: Buscar un módulo PowerShell mediante un patrón

Para encontrar un módulo que termine en DSC

```
Find-Module -Name *DSC
```

Capítulo 49: Comunicación TCP con PowerShell

Sección 49.1: Receptor TCP

```
Function Receive-TCPMessage {
    Param (
        [Parameter(Mandatory=$true, Position=0)]
        [ValidateNotNullOrEmpty()]
        [int] $Port
    )
    Process {
        Try {
            # Configurar el punto final y empezar a escuchar
            $endpoint = new-object System.Net.IPEndPoint([ipaddress]::any, $port)
            $listener = new-object System.Net.Sockets.TcpListener $EndPoint
            $listener.start()

            # Esperar una conexión entrante
            $data = $listener.AcceptTcpClient()

            # Configuración del flujo
            $stream = $data.GetStream()
            $bytes = New-Object System.Byte[] 1024

            # Leer datos del flujo y escribirlos en el host
            while (($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){
                $EncodedText = New-Object System.Text.ASCIIEncoding
                $data = $EncodedText.GetString($bytes, 0, $i)
                Write-Output $data
            }

            # Cerrar la conexión TCP y dejar de escuchar
            $stream.close()
            $listener.stop()
        }
        Catch {
            "Receive Message failed with: `n" + $Error[0]
        }
    }
}
```

Comienza a escuchar con lo siguiente y captura cualquier mensaje en la variable `$msg`:

```
$msg = Receive-TCPMessage -Port 29800
```

Sección 49.2: Emisor TCP

```
Function Send-TCPMessage {
    Param (
        [Parameter(Mandatory=$true, Position=0)]
        [ValidateNotNullOrEmpty()]
        [string]
        $EndPoint
    ,
        [Parameter(Mandatory=$true, Position=1)]
        [int]
        $Port
    ,
        [Parameter(Mandatory=$true, Position=2)]
        [string]
        $Message
    )
    Process {
        # Establecer conexión
        $IP = [System.Net.Dns]::GetHostAddresses($EndPoint)
        $Address = [System.Net.IPAddress]::Parse($IP)
        $Socket = New-Object System.Net.Sockets.TCPClient($Address, $Port)

        # Configurar el escritor de flujo
        $Stream = $Socket.GetStream()
        $Writer = New-Object System.IO.StreamWriter($Stream)

        # Escribir mensaje en el flujo
        $Message | % {
            $Writer.WriteLine($_)
            $Writer.Flush()
        }

        # Cerrar la conexión y el flujo
        $Stream.Close()
        $Socket.Close()
    }
}
```

Envía un mensaje con:

```
Send-TCPMessage -Port 29800 -Endpoint 192.168.0.1 -message "My first TCP message !"
```

Nota: Los mensajes TCP pueden ser bloqueados por el cortafuegos de su software o por cualquier cortafuegos externo que esté intentando atravesar. Asegúrese de que el puerto TCP que ha configurado en el comando anterior está abierto y de que ha configurado la escucha en el mismo puerto.

Capítulo 50: Flujos de trabajos de PowerShell

PowerShell Workflow es una función que se introdujo a partir de la versión 3.0 de PowerShell. Las definiciones de flujo de trabajo son muy similares a las definiciones de función de PowerShell, aunque se ejecutan en el entorno de Windows Workflow Foundation, en lugar de directamente en el motor de PowerShell.

El motor de flujo de trabajo incluye varias funciones exclusivas, entre las que destaca la persistencia de los trabajos.

Sección 50.1: Flujos de trabajos con parámetros de entrada

Al igual que las funciones PowerShell, los flujos de trabajo pueden aceptar parámetros de entrada. Los parámetros de entrada pueden vincularse opcionalmente a un tipo de datos específico, como una cadena, un entero, etc. Utilice la palabra clave `param` estándar para definir un bloque de parámetros de entrada, directamente después de la declaración del flujo de trabajo.

```
workflow DoSomeWork {
    param (
        [string[]] $ComputerName
    )
    Get-Process -ComputerName $ComputerName
}
```

```
DoSomeWork -ComputerName server01, server02, server03
```

Sección 50.2: Ejemplo de flujo de trabajo sencillo

```
workflow DoSomeWork {
    Get-Process -Name notepad | Stop-Process
}
```

Este es un ejemplo básico de definición de un flujo de trabajo PowerShell.

Sección 50.3: Ejecutar el flujo de trabajo como trabajo en segundo plano

Los flujos de trabajo PowerShell están equipados inherentemente con la capacidad de ejecutarse como un trabajo en segundo plano. Para llamar a un flujo de trabajo como un trabajo en segundo plano de PowerShell, utilice el parámetro `-AsJob` al invocar el flujo de trabajo.

```
workflow DoSomeWork {
    Get-Process -ComputerName server01
    Get-Process -ComputerName server02
    Get-Process -ComputerName server03
}
```

```
DoSomeWork -AsJob
```

Sección 50.4: Añadir un bloque paralelo a un flujo de trabajo

```
workflow DoSomeWork {
    parallel {
        Get-Process -ComputerName server01
        Get-Process -ComputerName server02
        Get-Process -ComputerName server03
    }
}
```

Una de las características únicas de PowerShell Workflow es la posibilidad de definir un bloque de actividades como paralelo. Para utilizar esta función, utilice la palabra clave `parallel` dentro de su flujo de trabajo.

Llamar a las actividades del flujo de trabajo en paralelo puede ayudar a mejorar el rendimiento de su flujo de trabajo.

Capítulo 51: Incrustación de código gestionado (C# | VB)

Parámetro

-TypeDefinition<String_>
-Language<String_>

Detalles

Acepta el código como cadena de caracteres
Especifica el lenguaje de Managed Code. Valores aceptados: CSharp, CSharpVersion3, CSharpVersion2, VisualBasic, JScript

Este tema describe brevemente cómo el código C# o VB .NET Managed puede ser programado y utilizado dentro de un script PowerShell. Este tema no explora todas las facetas del cmdlet `Add-Type`.

Para obtener más información sobre el cmdlet `Add-Type`, consulte la documentación de MSDN (para 5.1) aquí: <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/add-type>

Sección 51.1: Ejemplo en C#

Este ejemplo muestra cómo incrustar algo de C# básico en un script de PowerShell, añadirlo al espacio de ejecución/sesión y utilizar el código dentro de la sintaxis de PowerShell.

```
$code = "
using System;

namespace MyNameSpace
{
    public class Responder
    {
        public static void StaticRespond()
        {
            Console.WriteLine("Static Response");
        }

        public void Respond()
        {
            Console.WriteLine("Instance Respond");
        }
    }
}
"@

# Comprueba que el tipo no ha sido añadido previamente en la sesión, de lo contrario se lanza
una excepción
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language CSharp;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

Sección 51.2: Ejemplo en VB.NET

Este ejemplo muestra cómo incrustar algo de C# básico en un script de PowerShell, añadirlo al espacio de ejecución/sesión y utilizar el código dentro de la sintaxis de PowerShell.

```

$code = @"
Imports System

Namespace MyNameSpace
    Public Class Responder
        Public Shared Sub StaticRespond()
            Console.WriteLine("Static Response")
        End Sub

        Public Sub Respond()
            Console.WriteLine("Instance Respond")
        End Sub
    End Class
End Namespace
"@

# Comprueba que el tipo no ha sido añadido previamente en la sesión, de lo contrario se lanza
una excepción
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language VisualBasic;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();

```

Capítulo 52: ¿Cómo descargar el último artefacto de Artifactory usando un script PowerShell (v2.0 o inferior)?

Esta documentación explica y proporciona los pasos para descargar el último artefacto de un repositorio JFrog Artifactory utilizando PowerShell Script (v2.0 o inferior).

Sección 52.1: Script PowerShell para descargar el último artefacto

```
$username = 'user'
$password = 'password'
$DESTINATION = "D:\test\latest.tar.gz"
$client = New-Object System.Net.WebClient
$client.Credentials = new-object System.Net.NetworkCredential($username, $password)
$lastModifiedResponse =
$client.DownloadString('https://domain.org.com/artifactory/api/storage/FOLDER/repo/?lastModified
')
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestModifiedResponse = $serializer.DeserializeObject($lastModifiedResponse)
$downloadUriResponse = $getLatestModifiedResponse.uri
Write-Host $json.uri
$latestArtifactUrlResponse=$client.DownloadString($downloadUriResponse)
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestArtifact = $serializer.DeserializeObject($latestArtifactUrlResponse)
Write-Host $getLatestArtifact.downloadUri
$SOURCE=$getLatestArtifact.downloadUri
$client.DownloadFile($SOURCE,$DESTINATION)
```

Capítulo 53: Ayuda basada en comentarios

PowerShell dispone de un mecanismo de documentación denominado ayuda basada en comentarios. Permite documentar scripts y funciones con comentarios de código. La ayuda basada en comentarios se escribe la mayoría de las veces en bloques de comentarios que contienen varias palabras clave de ayuda. Las palabras clave de ayuda empiezan por puntos e identifican las secciones de ayuda que se mostrarán al ejecutar el cmdlet Get-Help.

Sección 53.1: Función de ayuda basada en comentarios

```
<#  
  
.SYNOPSIS  
    Obtiene el contenido de un fichero INI.  
  
.DESCRIPTION  
    Obtiene el contenido de un archivo INI y lo devuelve como hashtable.  
  
.INPUTS  
    System.String  
  
.OUTPUTS  
    System.Collections.Hashtable  
  
.PARAMETER FilePath  
    Especifica la ruta al archivo INI de entrada.  
  
.EXAMPLE  
    C:\PS>$IniContent = Get-IniContent -FilePath file.ini  
    C:\PS>$IniContent['Section1'].Key1  
    Obtiene el contenido de file.ini y accede a Key1 desde Section1.  
  
.LINK  
    Out-IniFile  
  
#>  
  
function Get-IniContent  
{  
    [CmdletBinding()]  
    Param  
    (  
        [Parameter(Mandatory=$true, ValueFromPipeline=$true)]  
        [ValidateNotNullOrEmpty()]  
        [ValidateScript({(Test-Path $_) -and ((Get-Item $_).Extension -eq ".ini")})]  
        [System.String]$FilePath  
    )  
  
    # Initialize output hash table.  
    $ini = @{}  
    switch -regex -file $FilePath  
    {  
        "^\[(.+)\]$" # Section  
        {  
            $section = $matches[1]  
            $ini[$section] = @{}  
            $CommentCount = 0  
        }  
    }  
}
```

```

"^(;.*)$" # Comment
{
    if( !($section) )
    {
        $section = "No-Section"
        $ini[$section] = @{}
    }
    $value = $matches[1]
    $CommentCount = $CommentCount + 1
    $name = "Comment" + $CommentCount
    $ini[$section][$name] = $value
}

"(.+?)\s*=\s*(.*)" # Key
{
    if( !($section) )
    {
        $section = "No-Section"
        $ini[$section] = @{}
    }
    $name,$value = $matches[1..2]
    $ini[$section][$name] = $value
}
}

return $ini
}

```

La documentación de la función anterior puede visualizarse ejecutando `Get-Help -Name Get-IniContent -Full`:

```

PS C:\Scripts> Get-Help -Name Get-IniContent -Full

NAME
    Get-IniContent

SYNOPSIS
    Gets the content of an INI file.

SYNTAX
    Get-IniContent [-FilePath] <String> [<CommonParameters>]

DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.

PARAMETERS
    -FilePath <String>
        Specifies the path to the input INI file.

        Required?                true
        Position?                 1
        Default value
        Accept pipeline input?    true (ByValue)
        Accept wildcard characters? false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    System.String

OUTPUTS
    System.Collections.Hashtable

----- EXAMPLE 1 -----

C:\PS>$IniContent = Get-IniContent -FilePath file.ini

C:\PS>$IniContent['Section1'].Key1
Gets the content of file.ini and access Key1 from Section1.

RELATED LINKS
    Out-IniFile

PS C:\Scripts>

```

Observe que las palabras clave basadas en comentarios que empiezan por `.` coinciden con las secciones de resultados de `Get-Help`.

Sección 53.2: Script de ayuda basada en comentarios

```
<#  
.SYNOPSIS  
    Lee un archivo CSV y lo filtra.  
.DESCRIPTION  
    El script ReadUsersCsv.ps1 lee un archivo CSV y lo filtra en la columna 'UserName'.  
.PARAMETER Path  
    Especifica la ruta del archivo de entrada CSV.  
.INPUTS  
    No. No puede enviar objetos a ReadUsersCsv.ps1.  
.OUTPUTS  
    Ninguno. ReadUsersCsv.ps1 no genera ninguna salida.  
.EXAMPLE  
    C:\PS> .\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe  
  
#>  
Param  
(  
    [Parameter(Mandatory=$true, ValueFromPipeline=$false)]  
    [System.String]  
    $Path,  
    [Parameter(Mandatory=$true, ValueFromPipeline=$false)]  
    [System.String]  
    $UserName  
)  
  
Import-Csv -Path $Path | Where-Object -FilterScript {$_.UserName -eq $UserName}
```

La documentación del script anterior puede visualizarse ejecutando `Get-Help -Name ReadUsersCsv.ps1 -Full`:

```
PS C:\Scripts> Get-Help -Name .\ReadUsersCsv.ps1 -Full
```

NAME

C:\Scripts\ReadUsersCsv.ps1

SYNOPSIS

Reads a CSV file and filters it.

SYNTAX

```
C:\Scripts\ReadUsersCsv.ps1 [-Path] <String> [-UserName] <String> [<CommonParameters>]
```

DESCRIPTION

The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.

PARAMETERS

-Path <String>

Specifies the path of the CSV input file.

Required?	true
Position?	1
Default value	
Accept pipeline input?	false
Accept wildcard characters?	false

-UserName <String>

Specifies the user name that will be used to filter the CSV file.

Required?	true
Position?	2
Default value	
Accept pipeline input?	false
Accept wildcard characters?	false

<CommonParameters>

This cmdlet supports the common parameters: Verbose, Debug, ErrorAction, ErrorVariable, WarningAction, WarningVariable, OutBuffer, PipelineVariable, and OutVariable. For more information, see about_CommonParameters (<http://go.microsoft.com/fwlink/?LinkID=113216>).

INPUTS

None. You cannot pipe objects to ReadUsersCsv.ps1.

OUTPUTS

None. ReadUsersCsv.ps1 does not generate any output.

----- EXAMPLE 1 -----

```
C:\PS>.\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe
```

RELATED LINKS

```
PS C:\Scripts>
```

Capítulo 54: Módulo Archive

Parámetro	Detalles
CompressionLevel	(Sólo Compress-Archive) Ajuste el nivel de compresión a <code>Fastest</code> , <code>Optimal</code> o <code>NoCompression</code> .
Confirm	Pide confirmación antes de ejecutar
Force	Fuerza la ejecución del comando sin confirmación
LiteralPath	Ruta que se utiliza literalmente, <i>no admite comodines</i> , utilice <code>LiteralPath</code> para especificar varias rutas.
Path	Ruta que puede contener comodines, utilice <code>Path</code> para especificar varias rutas
Update	(Sólo Compress-Archive) Actualizar archivo existente
WhatIf	Simular el comando

El módulo `Archive` `Microsoft.PowerShell.Archive` proporciona funciones para almacenar archivos en ficheros ZIP ([Compress-Archive](#)) y extraerlos ([Expand-Archive](#)). Este módulo está disponible en PowerShell 5.0 y superior.

En versiones anteriores de PowerShell se podían utilizar las [Community Extensions](#) o `.NET System.IO.Compression.FileSystem`.

Sección 54.1: Compress-Archive con comodín

```
Compress-Archive -Path C:\Documents\* -CompressionLevel Optimal -DestinationPath C:\Archives\Documents.zip
```

Este comando:

- Comprime todos los archivos de `C:\Documents`
- Utiliza la compresión `Optimal`
- Guarda el archivo resultante en `C:\Archives\Documents.zip`
 - `-DestinationPath` añadirá `.zip` si no está presente.
 - `-LiteralPath` puede usarse si requiere nombrarlo sin `.zip`.

Sección 54.2: Actualizar ZIP existente con Compress-Archive

```
Compress-Archive -Path C:\Documents\* -Update -DestinationPath C:\Archives\Documents.zip
```

- esto agregará o reemplazará todos los archivos `Documents.zip` con los nuevos de `C:\Documents`

Sección 54.3: Extraer un Zip con Expand-Archive

```
Expand-Archive -Path C:\Archives\Documents.zip -DestinationPath C:\Documents
```

- esto extraerá todos los archivos de `Documents.zip` en la carpeta `C:\Documents`

Capítulo 55: Automatización de infraestructuras

La automatización de los servicios de gestión de infraestructuras permite reducir los ETC y, de forma acumulativa, obtener un mejor rendimiento de la inversión mediante el uso de múltiples herramientas, orquestadores, motores de orquestación, scripts y una interfaz de usuario sencilla.

Sección 55.1: Script sencillo para la prueba de integración black-box de aplicaciones de consola

Este es un ejemplo sencillo de cómo automatizar pruebas para una aplicación de consola que interactúa con la entrada y salida estándar.

La aplicación probada lee y suma cada nueva línea y proporcionará el resultado después de una sola línea blanca. El power shell script escribe "pass" cuando la salida coincide.

```
$process = New-Object System.Diagnostics.Process
$process.StartInfo.FileName = ".\ConsoleApp1.exe"
$process.StartInfo.UseShellExecute = $false
$process.StartInfo.RedirectStandardOutput = $true
$process.StartInfo.RedirectStandardInput = $true
if ( $process.Start() ) {
    # input
    $process.StandardInput.WriteLine("1");
    $process.StandardInput.WriteLine("2");
    $process.StandardInput.WriteLine("3");
    $process.StandardInput.WriteLine();
    $process.StandardInput.WriteLine();
    # output check
    $output = $process.StandardOutput.ReadToEnd()
    if ( $output ) {
        if ( $output.Contains("sum 6") ) {
            Write "pass"
        }
        else {
            Write-Error $output
        }
    }
    $process.WaitForExit()
}
```

Capítulo 56: PSScriptAnalyzer - Analizador de scripts PowerShell

PSScriptAnalyzer, <https://github.com/PowerShell/PSScriptAnalyzer>, es un comprobador de código estático para módulos y scripts de Windows PowerShell. PSScriptAnalyzer comprueba la calidad del código de Windows PowerShell ejecutando un conjunto de reglas basadas en las mejores prácticas de PowerShell identificadas por el equipo de PowerShell y la comunidad. Genera `DiagnosticResults` (errores y advertencias) para informar a los usuarios sobre posibles defectos del código y sugiere posibles soluciones para mejorar.

```
PS> Install-Module -Name PSScriptAnalyzer
```

Sección 56.1: Análisis de guiones con las reglas preestablecidas integradas

ScriptAnalyzer incluye un conjunto de reglas preestablecidas que pueden utilizarse para analizar scripts. Entre ellas se incluyen: `PSGallery`, `DSC` y `CodeFormatting`. Pueden ejecutarse del siguiente modo:

Reglas de la Galería PowerShell

Para ejecutar las reglas de la Galería PowerShell utilice el siguiente comando:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

Reglas DSC

Para ejecutar las reglas `DSC` utilice el siguiente comando:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings DSC -Recurse
```

Reglas de formato del código

Para ejecutar las reglas de formato de código utilice el siguiente comando:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings CodeFormatting -Recurse
```

Sección 56.2: Análisis de secuencias de comandos según todas las reglas integradas

Para ejecutar el analizador de scripts contra un único archivo de script ejecute:

```
Invoke-ScriptAnalyzer -Path myscript.ps1
```

Esto analizará su script contra cada regla incorporada. Si su script es lo suficientemente grande, esto podría dar lugar a una gran cantidad de advertencias y/o errores.

Para ejecutar el analizador de scripts en un directorio completo, especifique la carpeta que contiene los archivos de script, módulo y DSC que desea analizar. Especifique el parámetro `Recurse` si también desea que se busquen archivos en subdirectorios para analizarlos.

```
Invoke-ScriptAnalyzer -Path . -Recurse
```

Sección 56.3: Lista de todas las reglas incorporadas

Para ver todas las reglas incorporadas ejecutar:

```
Get-ScriptAnalyzerRule
```

Capítulo 57: Configuración de estado deseada

Sección 57.1: Ejemplo sencillo - Activar WindowsFeature

```
configuration EnableIISFeature
{
    node localhost
    {
        WindowsFeature IIS
        {
            Ensure = "Present"
            Name = "Web-Server"
        }
    }
}
```

Si ejecuta esta configuración en Powershell (EnableIISFeature), producirá un archivo `localhost.mof`. Esta es la configuración "compilada" que puedes ejecutar en una máquina.

Para probar la configuración de DSC en su `localhost`, puede simplemente invocar lo siguiente:

```
Start-DscConfiguration -ComputerName localhost -Wait
```

Sección 57.2: Iniciando DSC (mof) en máquina remota

Iniciar un DSC en una máquina remota es casi igual de sencillo. Suponiendo que ya ha configurado Powershell remoto (o habilitado WSMAN).

```
$remoteComputer = "myserver.somedomain.com"
$cred = (Get-Credential)
Start-DSCConfiguration -ServerName $remoteComputer -Credential $cred -Verbose
```

Nb: Asumiendo que has compilado una configuración para tu nodo en tu máquina local (y que el archivo `myserver.somedomain.com.mof` está presente antes de iniciar la configuración)

Sección 57.3: Importación de psd1 (archivo de datos) a una variable local

A veces puede ser útil probar sus archivos de datos Powershell e iterar a través de los nodos y servidores.

Powershell 5 (WMF5) añadió esta pequeña característica para hacer esto llamado `Import-PowerShellDataFile`.

Ejemplo:

```
$data = Import-PowerShellDataFile -path .\MydataFile.psd1
$data.AllNodes
```

Sección 57.4: Lista de recursos disponibles del DSC

Para listar los recursos DSC disponibles en su nodo de autoría:

```
Get-DscResource
```

Esto listará todos los recursos para todos los módulos instalados (que están en su `PSModulePath`) en su nodo de autoría.

Para listar todos los recursos DSC disponibles que se pueden encontrar en las fuentes en línea (`PSGallery ++`) en WMF 5:

```
Find-DSCResource
```

Sección 57.5: Importación de recursos para su uso en DSC

Antes de poder utilizar un recurso en una configuración, debe importarlo explícitamente. El mero hecho de tenerlo instalado en su ordenador no le permitirá utilizar el recurso de forma implícita.

Importe un recurso utilizando `Import-DscResource`.

Ejemplo que muestra cómo importar el recurso `PSDesiredStateConfiguration` y el recurso `File`.

```
Configuration InstallPreReqs
{
    param(); # Los parámetros para DSC van aquí.

    Import-DscResource PSDesiredStateConfiguration

    File CheckForTmpFolder {
        Type = 'Directory'
        DestinationPath = 'C:\Tmp'
        Ensure = "Present"
    }
}
```

Nota: Para que los Recursos DSC funcionen, debe tener los módulos instalados en las máquinas de destino cuando ejecute la configuración. Si no los tiene instalados, la configuración fallará.

Capítulo 58: Uso de ShouldProcess

Parametro	Detalles
Target	El recurso que se modifica.
Action	La operación que se está realizando. Por defecto es el nombre del cmdlet.

Sección 58.1: Ejemplo de uso completo

Otros ejemplos no me explicaban claramente cómo activar la lógica condicional.

Este ejemplo también muestra que los comandos subyacentes también escucharán el indicador -Confirm.

```
<#
Restart-Win32Computer
#>

function Restart-Win32Computer
{
    [CmdletBinding(SupportsShouldProcess=$true, ConfirmImpact="High")]
    param (
        [parameter(Mandatory=$true, ValueFromPipeline=$true, ValueFromPipelineByPropertyName=$true)]
        [string[]]$computerName,
        [parameter(Mandatory=$true)]
        [string][ValidateSet("Restart", "LogOff", "Shutdown", "PowerOff")] $action,
        [boolean]$force = $false
    )
    BEGIN {
        # traducir la acción al valor numérico requerido por el método
        switch($action) {
            "Restart"
            {
                $_action = 2
                break
            }
            "LogOff"
            {
                $_action = 0
                break
            }
            "Shutdown"
            {
                $_action = 2
                break
            }
            "PowerOff"
            {
                $_action = 8
                break
            }
        }
        # para forzar, sume 4 al valor
        if($force)
        {
            $_action += 4
        }
        write-verbose "Action set to $action"
    }
    PROCESS {
        write-verbose "Attempting to connect to $computername"
        # así apoyamos -whatif y -confirm
        # que se activan mediante la opción SupportsShouldProcess
    }
}
```

```

# parámetro en el cmdlet bindnig
if($pscmdlet.ShouldProcess($computername)) {
    get-wmiobject win32_operatingsystem -computername $computername | invoke-wmimethod -
    name Win32Shutdown -argumentlist $_action
}
}
}
# Uso:
# Esto sólo mostrará una descripción de las acciones que este comando ejecutaría si se elimina -
WhatIf.
'localhost','server1'| Restart-Win32Computer -action LogOff -whatif

# Esto solicitará el permiso de la persona que llama para continuar con este tema.
# Atención: en este ejemplo obtendrá dos peticiones de confirmación porque todos los cmdlets
llamados por este cmdlet que también soporten ShouldProcess, pedirán sus propias
confirmaciones...
'localhost','server1'| Restart-Win32Computer -action LogOff -Confirm

```

Sección 58.2: Añadir soporte -WhatIf y -Confirm a su cmdlet

```

function Invoke-MyCmdlet {
    [CmdletBinding(SupportsShouldProcess = $true)]
    param()
    # ...
}

```

Sección 58.3: Uso de ShouldProcess() con un argumento

```

if ($PSCmdlet.ShouldProcess("Target of action")) {
    # Haz la cosa
}

```

Cuando se utiliza **-WhatIf**:

What if: Realizar la acción "Invoke-MyCmdlet" sobre el objetivo "Target of action"

Cuando se utiliza **-Confirm**:

¿Está seguro de que desea realizar esta acción? Realizando la operación "Invoke-MyCmdlet" en el objetivo "Target of action" [Y] Si [A] Si a Todo [N] No [L] No a Todo [S] Suspender [?] Ayuda (por defecto es «Y»):

Capítulo 59: Módulo de tareas programadas

Ejemplos de cómo utilizar el módulo de Tareas Programadas disponible en Windows 8/Server 2012 y en adelante.

Sección 59.1: Ejecutar un script PowerShell en una tarea programada

Crea una tarea programada que se ejecuta inmediatamente, luego en el arranque para ejecutar `C:\myscript.ps1` como `SYSTEM`

```
$ScheduledTaskPrincipal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType ServiceAccount
$ScheduledTaskTrigger1 = New-ScheduledTaskTrigger -AtStartup
$ScheduledTaskTrigger2 = New-ScheduledTaskTrigger -Once -At $(Get-Date) -RepetitionInterval
"00:01:00" -RepetitionDuration $([timeSpan] "24855.03:14:07")
$ScheduledTaskActionParams = @{
    Execute = "PowerShell.exe"
    Argument = '-executionpolicy Bypass -NonInteractive -c C:\myscript.ps1 -verbose >>
C:\output.log 2>&1"'
}
$ScheduledTaskAction = New-ScheduledTaskAction @ScheduledTaskActionParams
Register-ScheduledTask -Principal $ScheduledTaskPrincipal -Trigger
@($ScheduledTaskTrigger1,$ScheduledTaskTrigger2) -TaskName "Example Task" -Action
$ScheduledTaskAction
```

Capítulo 60: Módulo ISE

Windows PowerShell Integrated Scripting Environment (ISE) es una aplicación host que permite escribir, ejecutar y probar scripts y módulos en un entorno gráfico e intuitivo. Entre las principales características de Windows PowerShell ISE se incluyen el color de la sintaxis, la finalización de tabulaciones, Intellisense, la depuración visual, el cumplimiento de Unicode y la ayuda contextual, además de proporcionar una experiencia de scripting enriquecida.

Sección 60.1: Scripts de prueba

El uso sencillo pero potente del ISE es, por ejemplo, escribir código en la sección superior (con un coloreado intuitivo de la sintaxis) y ejecutar el código simplemente marcándolo y pulsando la tecla F8.

```
function Get-Sum
{
    foreach ($i in $Input)
    {$Sum += $i}
    $Sum
```

```
1..10 | Get-Sum
```

```
# salida
55
```

Capítulo 61: Creación de recursos basados en clases de DSC

A partir de la versión 5.0 de PowerShell, puede utilizar definiciones de clase de PowerShell para crear recursos de configuración de estado deseado (DSC).

Para ayudar en la construcción del Recurso DSC, hay un atributo `[DscResource()]` que se aplica a la definición de la clase, y un recurso `[DscProperty()]` para designar las propiedades como configurables por el usuario del Recurso DSC.

Sección 61.1: Crear una clase esqueleto de recursos DSC

```
[DscResource()]  
class File {  
}
```

Este ejemplo demuestra cómo construir la sección externa de una clase PowerShell, que declara un Recurso DSC. Todavía tiene que rellenar el contenido de la definición de la clase.

Sección 61.2: Esqueleto de recursos DSC con propiedad clave

```
[DscResource()]  
class Ticket {  
    [DscProperty(Key)]  
    [string] $TicketId  
}
```

Un Recurso DSC debe declarar al menos una propiedad clave. La propiedad clave es lo que identifica de forma única al recurso de otros recursos. Por ejemplo, supongamos que está creando un Recurso DSC que representa un ticket en un sistema de tickets. Cada ticket se representaría de forma única con un ID de ticket.

Cada propiedad que será expuesta al usuario del Recurso DSC debe ser decorada con el atributo `[DscProperty()]`. Este atributo acepta un parámetro clave, para indicar que la propiedad es un atributo clave para el Recurso DSC.

Sección 61.3: Recurso DSC con propiedad obligatoria

```
[DscResource()]  
class Ticket {  
    [DscProperty(Key)]  
    [string] $TicketId  
  
    [DscProperty(Mandatory)]  
    [string] $Subject  
}
```

Al crear un Recurso DSC, a menudo descubrirá que no todas las propiedades deben ser obligatorias. Sin embargo, hay algunas propiedades básicas que querrá asegurarse de que sean configuradas por el usuario del Recurso DSC. Utilice el parámetro `Mandatory` del atributo `[DscProperty()]` para declarar una propiedad como requerida por el usuario del Recurso DSC.

En el ejemplo anterior, hemos añadido una propiedad `Subject` a un recurso `Ticket`, que representa un ticket único en un sistema de tickets, y la hemos designado como propiedad `Mandatory`.

Sección 61.4: Recurso DSC con métodos requeridos

```
[DscResource()]
class Ticket {
    [DscProperty(Key)]
    [string] $TicketId

    # El asunto del ticket
    [DscProperty(Mandatory)]
    [string] $Subject

    # Obtener / Establecer si el ticket debe estar abierto o cerrado
    [DscProperty(Mandatory)]
    [string] $TicketState

    [void] Set() {
        # Crear o actualizar el recurso
    }

    [Ticket] Get() {
        # Devuelve el estado actual del recurso como un objeto
        $TicketState = [Ticket]::new()
        return $TicketState
    }

    [bool] Test() {
        # Devuelve $true si se cumple el estado deseado
        # Devuelve $false si no se cumple el estado deseado
        return $false
    }
}
```

Se trata de un recurso DSC completo que demuestra todos los requisitos básicos para crear un recurso válido. Las implementaciones de los métodos no están completas, pero se proporcionan con la intención de mostrar la estructura básica.

Capítulo 62: WMI y CIM

Sección 62.1: Consulta de objetos

CIM/WMI se utiliza normalmente para consultar información o la configuración de un dispositivo. A través de una clase que representa una configuración, proceso, usuario, etc. En PowerShell existen múltiples formas de acceder a estas clases e instancias, pero las más comunes son mediante los cmdlets `Get-CimInstance` (CIM) o `Get-WmiObject` (WMI).

Lista de todos los objetos de la clase CIM

Puede listar todas las instancias de una clase.

Version ≥ 3.0

CIM:

```
> Get-CimInstance -ClassName Win32_Process
```

ProcessId	Name	HandleCount	WorkingSetSize	VirtualSize
0	System Idle Process	0	4096	65536
4	System	1459	32768	3563520
480	Secure System	0	3731456	0
484	smss.exe	52	372736	2199029891072
....				
....				

WMI:

```
Get-WmiObject -Class Win32_Process
```

Utilizar un filtro

Puede aplicar un filtro para obtener sólo instancias específicas de una clase CIM/WMI. Los filtros se escriben utilizando WQL (por defecto) o CQL (añadir `-QueryDialect CQL`). `-Filter` utiliza la parte `WHERE` de una consulta WQL/CQL completa.

Version ≥ 3.0

CIM:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name = 'powershell.exe'"
```

ProcessId	Name	HandleCount	WorkingSetSize	VirtualSize
4800	powershell.exe	676	88305664	2199697199104

WMI:

```
Get-WmiObject -Class Win32_Process -Filter "Name = 'powershell.exe'"
```

```
...
Caption                : powershell.exe
CommandLine            : "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
CreationClassName      : Win32_Process
CreationDate           : 20160913184324.393887+120
CSCreationClassName    : Win32_ComputerSystem
CSName                 : STACKOVERFLOW-PC
Description            : powershell.exe
ExecutablePath         : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
ExecutionState         :
Handle                 : 4800
HandleCount            : 673
....
```

Mediante una consulta WQL:

También puede utilizar una consulta WQL/CQL para consultar y filtrar instancias.

Version ≥ 3.0

CIM:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE Name = 'powershell.exe'"
```

ProcessId	Name	HandleCount	WorkingSetSize	VirtualSize
4800	powershell.exe	673	88387584	2199696674816

Consulta de objetos en un espacio de nombres diferente:

Version ≥ 3.0

CIM:

```
> Get-CimInstance -Namespace "root/SecurityCenter2" -ClassName AntiVirusProduct
```

```
displayName           : Windows Defender
instanceGuid          : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState          : 397568
timestamp              : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName        :
```

WMI:

```
> Get-WmiObject -Namespace "root\SecurityCenter2" -Class AntiVirusProduct
```

```
__GENUS                : 2
__CLASS                 : AntiVirusProduct
__SUPERCLASS           :
__DYNASTY               : AntiVirusProduct
__RELPATH               :
AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
__PROPERTY_COUNT       : 6
__DERIVATION           : {}
__SERVER                : STACKOVERFLOW-PC
__NAMESPACE            : ROOT\SecurityCenter2
__PATH                 :
\\STACKOVERFLOWPC\ROOT\SecurityCenter2:AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
displayName             : Windows Defender
instanceGuid           : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState           : 397568
timestamp               : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName         : STACKOVERFLOW-PC
```

Sección 62.2: Clases y espacios de nombres

Hay muchas clases disponibles en CIM y WMI que están separadas en múltiples espacios de nombres. El espacio de nombres más común (y por defecto) en Windows es `root/cimv2`. Para encontrar la clase correcta, puede ser útil para listar todo o buscar.

Lista de clases disponibles

Puede listar todas las clases disponibles en el espacio de nombres por defecto (`root/cimv2`) en un ordenador.

Version \geq 3.0

CIM:

```
Get-CimClass
```

WMI:

```
Get-WmiObject -List
```

Buscar una clase

Puede buscar clases específicas utilizando comodines. Ej: Buscar clases que contengan la palabra `process`.

Version \geq 3.0

CIM:

```
> Get-CimClass -ClassName "*Process*"
```

```
NameSpace: ROOT/CIMV2
```

CimClassName	CimClassMethods	CimClassProperties
Win32_ProcessTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, ParentProcessID, ProcessID...}
Win32_ProcessStartTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, ParentProcessID, ProcessID...}
Win32_ProcessStopTrace	{}	{SECURITY_DESCRIPTOR, TIME_CREATED, ParentProcessID, ProcessID...}
CIM_Process	{}	{Caption, Description, InstallDate, Name...}
Win32_Process	{Create, Terminat...}	{Caption, Description, InstallDate, Name...}
CIM_Processor	{SetPowerState, R...}	{Caption, Description, InstallDate, Name...}
Win32_Processor	{SetPowerState, R...}	{Caption, Description, InstallDate, Name...}
...		

WMI:

```
Get-WmiObject -List -Class "*Process*"
```

Lista de clases en un espacio de nombres diferente

El espacio de nombres raíz se denomina simplemente `root`. Puede listar clases en otro espacio de nombres utilizando el parámetro `-NameSpace`.

Version \geq 3.0

CIM:

```
> Get-CimClass -Namespace "root/SecurityCenter2"
```

```
NameSpace: ROOT/SecurityCenter2
```

CimClassName	CimClassMethods	CimClassProperties
AntiSpywareProduct	{}	{displayName, instanceGuid, pathToSignedProductExe, pathToSignedReportingE...}
AntiVirusProduct	{}	{displayName, instanceGuid, pathToSignedProductExe, pathToSignedReportingE...}
FirewallProduct	{}	{displayName, instanceGuid, pathToSignedProductExe, pathToSignedReportingE...}

WMI:

```
Get-WmiObject -Class "__Namespace" -Namespace "root"
```

Lista de espacios de nombres disponibles

Para encontrar los espacios de nombres hijos disponibles de `root` (u otro espacio de nombres), consulte los objetos de la clase `__NAMESPACE` para ese espacio de nombres.

Version \geq 3.0

CIM:

```
> Get-CimInstance -Namespace "root" -ClassName "__Namespace"
```

Name	PSComputerName
----	-----
subscription	
DEFAULT	
CIMV2	
msdtc	
Cli	
SECURITY	
HyperVCluster	
SecurityCenter2	
RSOP	
PEH	
StandardCimv2	
WMI	
directory	
Policy	
virtualization	
Interop	
Hardware	
ServiceModel	
SecurityCenter	
Microsoft	
aspnet	
Appv	

WMI:

```
Get-WmiObject -List -Namespace "root"
```

Capítulo 63: Módulo ActiveDirectory

Este tema introducirá a algunos de los cmdlets básicos utilizados dentro del Módulo de Active Directory para PowerShell, para manipular Usuarios, Grupos, Equipos y Objetos.

Sección 63.1: Usuarios

Recuperar usuario de Active Directory

```
Get-ADUser -Identity JohnSmith
```

Recuperar todas las propiedades asociadas al usuario

```
Get-ADUser -Identity JohnSmith -Properties *
```

Recuperar propiedades seleccionadas para el usuario

```
Get-ADUser -Identity JohnSmith -Properties * | Select-Object -Property sAMAccountName, Name, Mail
```

Nuevo usuario AD

```
New-ADUser -Name "MarySmith" -GivenName "Mary" -Surname "Smith" -DisplayName "MarySmith" -Path "CN=Users,DC=Domain,DC=Local"
```

Sección 63.2: Módulo

```
# Añadir el módulo ActiveDirectory a la sesión PowerShell actual  
Import-Module ActiveDirectory
```

Sección 63.3: Grupos

Recuperar grupo de Active Directory

```
Get-ADGroup -Identity "My-First-Group" # Asegúrese de que si el nombre del grupo tiene espacio se utilizan comillas
```

Recuperar todas las propiedades asociadas a un grupo

```
Get-ADGroup -Identity "My-First-Group" -Properties *
```

Recuperar todos los miembros de un grupo

```
Get-ADGroupMember -Identity "My-First-Group" | Select-Object -Property sAMAccountName  
Get-ADgroup "MY-First-Group" -Properties Members | Select -ExpandProperty Members
```

Añadir un usuario AD a un grupo AD

```
Add-ADGroupMember -Identity "My-First-Group" -Members "JohnSmith"
```

Nuevo grupo AD

```
New-ADGroup -GroupScope Universal -Name "My-Second-Group"
```

Sección 63.4: Equipos

Recuperar equipos AD

```
Get-ADComputer -Identity "JohnLaptop"
```

Recuperar todas las propiedades asociadas al equipo

```
Get-ADComputer -Identity "JohnLaptop" -Properties *
```

Recuperar propiedades seleccionadas de un equipo

```
Get-ADComputer -Identity "JohnLaptop" -Properties * | Select-Object -Property Name, Enabled
```

Sección 63.5: Objetos

Recuperar un objeto de Active Directory

```
# La identidad puede ser ObjectGUID, Nombre Distinguido o muchos más
```

```
Get-ADObject -Identity "ObjectGUID07898"
```

Mover un objeto de Active Directory

```
Move-ADObject -Identity "CN=JohnSmith,OU=Users,DC=Domain,DC=Local" -TargetPath  
"OU=SuperUser,DC=Domain,DC=Local"
```

Modificar un objeto de Active Directory

```
Set-ADObject -Identity "CN=My-First-Group,OU=Groups,DC=Domain,DC=local" -Description "This is My  
First Object Modification"
```

Capítulo 64: Módulo SharePoint

Sección 64.1: Carga del complemento de SharePoint

La carga del Snapin de SharePoint puede realizarse de la siguiente manera:

```
Add-PSSnapin "Microsoft.SharePoint.PowerShell"
```

Esto sólo funciona en la versión de 64 bits de PowerShell. Si la ventana dice «Windows PowerShell (x86)» en el título que está utilizando la versión incorrecta.

Si el Snap-In ya está cargado, el código anterior provocará un error. Si se utiliza lo siguiente, se cargará sólo si es necesario, lo que puede utilizarse en Cmdlets/funciones:

```
if ((Get-PSSnapin "Microsoft.SharePoint.PowerShell" -ErrorAction SilentlyContinue) -eq $null)
{
    Add-PSSnapin "Microsoft.SharePoint.PowerShell"
}
```

Alternativamente, si inicia SharePoint Management Shell, incluirá automáticamente el Snap-In.

Para obtener una lista de todos los Cmdlets de SharePoint disponibles, ejecute lo siguiente:

```
Get-Command -Module Microsoft.SharePoint.PowerShell
```

Sección 64.2: Iterar sobre todas las listas de una colección de sitios

Imprime todos los nombres de las listas y el recuento de artículos.

```
$site = Get-SPSite -Identity https://mysharepointsite/sites/test
foreach ($web in $site.AllWebs)
{
    foreach ($list in $web.Lists)
    {
        # Imprime el título de la lista y el recuento de artículos
        Write-Output "$($list.Title), Items: $($list.ItemCount)"
    }
}
$site.Dispose()
```

Sección 64.3: Obtener todas las funciones instaladas en una colección de sitios

```
Get-SPFeature -Site https://mysharepointsite/sites/test
```

`Get-SPFeature` también puede ejecutarse en el ámbito web (`-Web <WebUrl>`), en el ámbito de granja (`-Farm`) y en el ámbito de aplicación web (`-WebApplication <WebAppUrl>`).

Obtener todas las funciones huérfanas de una colección de sitios

Otro uso de `Get-SPFeature` puede ser encontrar todas las funciones que no tienen ámbito:

```
Get-SPFeature -Site https://mysharepointsite/sites/test |? { $_.Scope -eq $null }
```

Capítulo 65: Introducción a Psake

Sección 65.1: Esquema básico

```
Task Rebuild -Depends Clean, Build {
    "Rebuild"
}
Task Build {
    "Build"
}
Task Clean {
    "Clean"
}
Task default -Depends Build
```

Sección 65.2: Ejemplo de FormatTaskName

```
# Will display task as:
# ----- Rebuild -----
# ----- Build -----
FormatTaskName "----- {0} -----"

# will display tasks in yellow colour:
# Running Rebuild
FormatTaskName {
    param($taskName)
    "Running $taskName" - foregroundcolor yellow
}

Task Rebuild -Depends Clean, Build {
    "Rebuild"
}
Task Build {
    "Build"
}
Task Clean {
    "Clean"
}
Task default -Depends Build
```

Sección 65.3: Ejecutar tarea condicionalmente

```
properties {
    $isOk = $false
}

# Por defecto, la tarea Build no se ejecutará, a menos que haya un parámetro $true
Task Build -precondition { return $isOk } {
    "Build"
}
Task Clean {
    "Clean"
}
Task default -Depends Build
```

Sección 65.4: ContinueOnError

```
Task Build -depends Clean {  
    "Build"  
}  
Task Clean -ContinueOnError {  
    "Clean"  
    throw "throw on purpose, but the task will continue to run"  
}  
Task default -Depends Build
```

Capítulo 66: Introducción a Pester

Sección 66.1: Primeros pasos con Pester

Para empezar con las pruebas unitarias de código PowerShell utilizando el módulo Pester, es necesario estar familiarizado con tres palabras clave/comandos:

- **Describe:** Define un grupo de pruebas. Todos los archivos de prueba Pester necesitan al menos un bloque Describe.
- **It:** Define una prueba individual. Dentro de un bloque Describe puede haber varios bloques It.
- **Should:** El comando de verificación/prueba. Se utiliza para definir el resultado que debe considerarse una prueba correcta.

Ejemplo:

```
Import-Module Pester

# Función de ejemplo para ejecutar pruebas
function Add-Numbers{
    param($a, $b)
    return [int]$a + [int]$b
}

# Grupo de pruebas
Describe "Validate Add-Numbers" {

    # Casos de prueba individuales
    It "Should add 2 + 2 to equal 4" {
        Add-Numbers 2 2 | Should Be 4
    }

    It "Should handle strings" {
        Add-Numbers "2" "2" | Should Be 4
    }

    It "Should return an integer"{
        Add-Numbers 2.3 2 | Should BeOfType Int32
    }
}
```

Salida:

```
Describing Validate Add-Numbers
[+] Should add 2 + 2 to equal 4 33ms
[+] Should handle strings 19ms
[+] Should return an integer 23ms
```

Capítulo 67: Gestión de secretos y credenciales

En Powershell, para evitar almacenar la contraseña en texto claro utilizamos diferentes métodos de encriptación y la almacenamos como cadena segura. Cuando no se especifica una clave o securekey, esto sólo funcionará para el mismo usuario en el mismo equipo será capaz de descifrar la cadena cifrada si no está utilizando Keys/SecureKeys. Cualquier proceso que se ejecute bajo esa misma cuenta de usuario será capaz de descifrar esa cadena cifrada en esa misma máquina.

Sección 67.1: Acceso a la contraseña en texto plano

La contraseña de un objeto credencial es una [SecureString] cifrada. Lo más sencillo es obtener una [NetworkCredential] que no almacene la contraseña cifrada:

```
$credential = Get-Credential
$plainPass = $credential.GetNetworkCredential().Password
```

El método de ayuda (.GetNetworkCredential()) sólo existe en objetos [PSCredential]. Para tratar directamente con una [SecureString], utilice los métodos .NET:

```
$bstr = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($secStr)
$plainPass = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
```

Sección 67.2: Solicitud de credenciales

Para solicitar credenciales, casi siempre debe utilizar el cmdlet [Get-Credential](#):

```
$credential = Get-Credential
```

Nombre de usuario pre-rellenado:

```
$credential = Get-Credential -UserName 'myUser'
```

Añade un mensaje personalizado:

```
$credential = Get-Credential -Message 'Please enter your company email address and password.'
```

Sección 67.3: Trabajar con credenciales almacenadas

Para almacenar y recuperar credenciales cifradas fácilmente, utilice la serialización XML incorporada de PowerShell (Clixml):

```
$credential = Get-Credential
$credential | Export-CliXml -Path 'C:\MyPath\cred.xml'
```

Para reimportar:

```
$credential = Import-CliXml -Path 'C:\MyPath\cred.xml'
```

Lo importante es recordar que, por defecto, esto utiliza la API de protección de datos de Windows, y la clave utilizada para cifrar la contraseña es específica tanto para el usuario como para la máquina en la que se está ejecutando el código.

Como resultado, la credencial encriptada no puede ser importada por un usuario diferente ni por el mismo usuario en un ordenador diferente.

Al cifrar varias versiones de la misma credencial con diferentes usuarios en ejecución y en diferentes ordenadores, puedes tener el mismo secreto disponible para varios usuarios.

Al poner el nombre del usuario y del ordenador en el nombre del archivo, puedes almacenar todos los secretos encriptados de forma que el mismo código pueda utilizarlos sin necesidad de codificar nada:

Cifrador

```
# ejecutar como cada usuario, y en cada ordenador
$credential = Get-Credential
$credential | Export-CliXml -Path "C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

El código que utiliza las credenciales almacenadas:

```
$credential = Import-CliXml -Path "C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

La versión correcta del archivo para el usuario en ejecución se cargará automáticamente (o fallará porque el archivo no existe).

Sección 67.4: Almacenar las credenciales de forma encriptada y pasarlas como parámetro cuando sea necesario.

```
$username = "user1@domain.com"
$pwdTxt = Get-Content "C:\temp\Stored_Password.txt"
$securePwd = $pwdTxt | ConvertTo-SecureString
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd
```

Ahora, \$credObject tiene las credenciales almacenadas y puedes pasarlo donde quieras.

Importar contraseña con AES

```
$username = "user1@domain.com"
$AESKey = Get-Content $AESKeyFilePath
$pwdTxt = Get-Content $SecurePwdFilePath
$securePwd = $pwdTxt | ConvertTo-SecureString -Key $AESKey
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd
```

Ahora, \$credObject tiene las credenciales almacenadas con clave AES y puedes pasarlo donde quieras.

Capítulo 68: Seguridad y criptografía

Sección 68.1: Cálculo de los códigos hash de una cadena de caracteres mediante Criptografía .Net

Utilizando el espacio de nombre `.Net System.Security.Cryptography.HashAlgorithm` para generar el código hash del mensaje con los algoritmos soportados.

```
$example="Nobody expects the Spanish Inquisition."
```

```
# calcular
```

```
$hash=[System.Security.Cryptography.HashAlgorithm]::Create("sha256").ComputeHash(  
[System.Text.Encoding]::UTF8.GetBytes($example))
```

```
# convertir a hexadecimal
```

```
[System.BitConverter]::ToString($hash)
```

```
#2E-DF-DA-DA-56-52-5B-12-90-FF-16-FB-17-44-CF-B4-82-DD-29-14-FF-BC-B6-49-79-0C-0E-58-9E-46-2D-3D
```

La parte `"sha256"` era el algoritmo hash utilizado.

el `-` puede suprimirse o cambiarse a minúsculas

```
# convertir a minúsculas hexadecimales sin '-'
```

```
[System.BitConverter]::ToString($hash).Replace("-", "").ToLower()
```

```
# 2edfdada56525b1290ff16fb1744cfb482dd2914ffbc649790c0e589e462d3d
```

Si se prefiere el formato base64, utilizar el convertidor base64 para la salida

```
# convertir a base64
```

```
[Convert]::ToBase64String($hash)
```

```
# Lt/a2lZSWxKQ/xb7F0TPtILdKRT/vLZJeQwOWJ5GLT0=
```

Capítulo 69: Scripts de firma

Sección 69.1: Firmar un script

La firma de un script se realiza utilizando el comando `Set-AuthenticodeSignature` y un certificado de firma de código.

```
# Obtener el primer certificado personal de firma de código disponible para el usuario conectado
$cert = @(Get-ChildItem -Path Cert:\CurrentUser\My -CodeSigningCert)[0]
# Firmar script con certificado
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1
```

También puede leer un certificado de un archivo `.pfx` utilizando:

```
$cert = Get-PfxCertificate -FilePath "C:\MyCodeSigningCert.pfx"
```

La secuencia de comandos será válida hasta que caduque el certificado. Si utiliza un servidor de fecha y hora durante la firma, la secuencia de comandos seguirá siendo válida después de que caduque el certificado. También es útil añadir la cadena de confianza del certificado (incluida la autoridad raíz) para ayudar a que la mayoría de los ordenadores confíen en el certificado utilizado para firmar el script.

```
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1 -IncludeChain All -
TimeStampServer "http://timestamp.verisign.com/scripts/timestamp.dll"
```

Se recomienda utilizar un servidor de marcas de tiempo de un proveedor de certificados de confianza, como Verisign, Comodo, Thawte, etc.

Sección 69.2: Eludir la política de ejecución de un único script

A menudo puede necesitar ejecutar un script sin firmar que no cumple con la política de ejecución actual. Una forma fácil de hacerlo es saltarse la política de ejecución para ese único proceso. Ejemplo:

```
powershell.exe -ExecutionPolicy Bypass -File C:\MyUnsignedScript.ps1
```

O puedes utilizar la taquigrafía:

```
powershell -ep Bypass C:\MyUnsignedScript.ps1
```

Otras políticas de ejecución:

Política	Descripción
AllSigned	Sólo se pueden ejecutar scripts firmados por un editor de confianza.
Bypass	Sin restricciones; se pueden ejecutar todos los scripts de Windows PowerShell.
Default	Normalmente RemoteSigned, pero se controla a través de ActiveDirectory
RemoteSigned	Los scripts descargados deben estar firmados por un editor de confianza antes de poder ejecutarse.
Restricted	No se pueden ejecutar scripts. Windows PowerShell sólo se puede utilizar en modo interactivo.
Undefined	NA
Unrestricted*	Similar al Bypass

Unrestricted* *Advertencia:* si ejecuta un script sin firmar descargado de Internet, se le pedirá permiso antes de que se ejecute.

Más información [aquí](#).

Sección 69.3: Cambio de la política de ejecución mediante Set-ExecutionPolicy

Para cambiar la política de ejecución para el ámbito por defecto (LocalMachine), utilice:

```
Set-ExecutionPolicy AllSigned
```

Para cambiar la política de un ámbito específico, utilice:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy AllSigned
```

Puede suprimir los avisos añadiendo el modificador `-Force`.

Sección 69.4: Obtener la política de ejecución actual

Obtener la política de ejecución efectiva para la sesión actual:

```
PS> Get-ExecutionPolicy
RemoteSigned
```

Lista todas las políticas de ejecución efectivas para la sesión actual:

```
PS> Get-ExecutionPolicy -List
```

Scope	ExecutionPolicy
MachinePolicy	Undefined
UserPolicy	Undefined
Process	Undefined
CurrentUser	Undefined
LocalMachine	RemoteSigned

Lista la política de ejecución para un ámbito específico, ej. proceso:

```
PS> Get-ExecutionPolicy -Scope Process
Undefined
```

Sección 69.5: Obtener la firma de un script firmado

Obtenga información sobre la firma Authenticode de un script firmado utilizando el cmdlet `Get-AuthenticodeSignature`:

```
Get-AuthenticodeSignature .\MyScript.ps1 | Format-List *
```

Sección 69.6: Creación de un certificado de firma de código autofirmado para pruebas

Cuando se firman scripts personales o cuando se prueba la firma de código, puede ser útil crear un certificado de firma de código autofirmado.

Version ≥ 5.0

A partir de PowerShell 5.0, puede generar un certificado de firma de código autofirmado utilizando el comando `New-SelfSignedCertificate`:

```
New-SelfSignedCertificate -FriendlyName "StackOverflow Example Code Signing" -CertStoreLocation Cert:\CurrentUser\My -Subject "SO User" -Type CodeSigningCert
```

En versiones anteriores, puede crear un certificado autofirmado utilizando la herramienta `makecert.exe` que se encuentra en el SDK de .NET Framework y en el SDK de Windows.

Un certificado autofirmado sólo será fiable para los ordenadores que lo tengan instalado. Para las secuencias de comandos que se van a compartir, se recomienda un certificado de una autoridad de certificación de confianza (interna o de terceros de confianza).

Capítulo 71: Servicios web de Amazon (AWS) Rekognition

Amazon Rekognition es un servicio que facilita la incorporación del análisis de imágenes a sus aplicaciones. Con Rekognition, puede detectar objetos, escenas y rostros en imágenes. También puede buscar y comparar rostros. La API de Rekognition le permite añadir rápidamente a sus aplicaciones sofisticadas búsquedas visuales y clasificaciones de imágenes basadas en aprendizaje profundo.

Sección 71.1: Detectar etiquetas de imágenes con AWS Rekognition

```
$BucketName = 'trevorrekognition'  
$FileName = 'kitchen.jpg'  
  
New-S3Bucket -BucketName $BucketName  
Write-S3Object -BucketName $BucketName -File $FileName  
$REKResult = Find-REKLabel -Region us-east-1 -ImageBucket $BucketName -ImageName $FileName  
  
$REKResult.Labels
```

Después de ejecutar el script anterior, debería tener resultados impresos en su host PowerShell con un aspecto similar al siguiente:

```
RESULTS:  
Confidence      Name  
-----  
86.87605        Indoors  
86.87605        Interior Design  
86.87605        Room  
77.4853         Kitchen  
77.25354        Housing  
77.25354        Loft  
66.77325        Appliance  
66.77325        Oven
```

Mediante el módulo AWS PowerShell junto con el servicio AWS Rekognition, puede detectar etiquetas en una imagen, como identificar objetos en una habitación, atributos sobre las fotos que tomó y el nivel de confianza correspondiente que AWS Rekognition tiene para cada uno de esos atributos.

El comando `Find-REKLabel` es el que permite invocar una búsqueda de estos atributos / etiquetas. Aunque puede proporcionar el contenido de la imagen como un array de bytes durante la llamada a la API, un método mejor es cargar sus archivos de imagen en un AWS S3 Bucket y, a continuación, apuntar el servicio Rekognition a los objetos S3 que desea analizar. El ejemplo anterior muestra cómo hacerlo.

Sección 71.2: Comparar la similitud facial con AWS Rekognition

```
$BucketName = 'trevorrekognition'  
  
### Crear un nuevo AWS S3 Bucket  
New-S3Bucket -BucketName $BucketName  
  
### Subir dos fotos diferentes de mí mismo a AWS S3 Bucket  
Write-S3Object -BucketName $BucketName -File myphoto1.jpg  
Write-S3Object -BucketName $BucketName -File myphoto2.jpg  
  
### Realiza una comparación facial entre las dos fotos con AWS Rekognition  
$Comparison = @{  
    SourceImageBucket = $BucketName  
    TargetImageBucket = $BucketName  
    SourceImageName = 'myphoto1.jpg'  
    TargetImageName = 'myphoto2.jpg'  
    Region = 'us-east-1'  
}  
$Result = Compare-REKFace @Comparison  
$Result.FaceMatches
```

El script de ejemplo proporcionado anteriormente debería darle resultados similares a los siguientes:

Face	Similarity
-----	-----
Amazon.Rekognition.Model.ComparedFace	90

El servicio AWS Rekognition permite realizar una comparación facial entre dos fotos. El uso de este servicio es bastante sencillo. Solo tiene que cargar dos archivos de imagen que desee comparar en un bucket de AWS S3. A continuación, invoque el comando `Compare-REKFace`, similar al ejemplo proporcionado anteriormente. Por supuesto, tendrá que proporcionar su propio nombre de bucket de S3 y nombres de archivo únicos a nivel mundial.

Capítulo 72: Servicio de almacenamiento simple (S3) de Amazon Web Services (AWS)

Parámetro	Detalles
<code>BucketName</code>	El nombre del bucket de AWS S3 sobre el que está operando.
<code>CannedACLName</code>	Nombre de la lista de control de acceso (ACL) integrada (predefinida) que se asociará al bucket de S3.
<code>File</code>	El nombre de un archivo del sistema de archivos local que se cargará en un bucket de AWS S3.

Esta sección de documentación se centra en el desarrollo contra el Servicio de Almacenamiento Simple (S3) de Amazon Web Services (AWS). S3 es realmente un servicio sencillo con el que interactuar. Creas "buckets" S3 que pueden contener cero o más "objetos". Una vez creado un bucket, puedes subir archivos o datos arbitrarios al bucket de S3 como un "objeto". Puedes hacer referencia a objetos S3, dentro de un bucket, por la "clave" (nombre) del objeto.

Sección 72.1: Crear un nuevo Bucket S3

```
New-S3Bucket -BucketName Trevor
```

El nombre del cubo de Simple Storage Service (S3) debe ser único a nivel mundial. Esto significa que, si alguien ya ha utilizado el nombre del bucket que desea utilizar, deberá decidir un nuevo nombre.

Sección 72.2: Cargar un archivo local en un bucket de S3

```
Set-Content -Path myfile.txt -Value 'PowerShell Rocks'  
Write-S3Object -BucketName powershell -File myfile.txt
```

Subir archivos desde tu sistema de archivos local a AWS S3 es fácil, utilizando el comando `Write-S3Object`. En su forma más básica, solo necesita especificar el parámetro `-BucketName`, para indicar en qué bucket de S3 desea cargar un archivo, y el parámetro `-File`, que indica la ruta relativa o absoluta al archivo local que desea cargar en el bucket de S3.

Sección 72.3: Borrar un Bucket S3

```
Get-S3Object -BucketName powershell | Remove-S3Object -Force  
Remove-S3Bucket -BucketName powershell -Force
```

Para eliminar un bucket de S3, primero debes eliminar todos los objetos de S3 que están almacenados dentro del bucket, siempre que tengas permiso para hacerlo. En el ejemplo anterior, estamos recuperando una lista de todos los objetos dentro de un cubo, y luego canalizarlos en el comando `Remove-S3Object` para eliminarlos. Una vez eliminados todos los objetos, podemos utilizar el comando `Remove-S3Bucket` para eliminar el bucket.

Créditos

Muchas gracias a todas las personas de Stack Overflow Documentation que ayudaron a proporcionar este contenido, más cambios pueden ser enviados a web@petercv.com para que el nuevo contenido sea publicado o actualizado.

Traductor al español

[rortegag](#)

Adam M.	Capítulo 23
ajb101	Capítulo 51
Alban	Capítulo 25
Andrei Epure	Capítulo 28
ANIL	Capítulo 52
Anthony Neace	Capítulos 3 y 8
AP.	Capítulo 69
Austin T French	Capítulo 17
autosvet	Capítulos 1, 2 y 20
Avshalom	Capítulo 24
Bert Levrau	Capítulos 12, 27 y 43
boeproxy	Capítulo 13
Brant Bobby	Capítulos 1, 13, 19 y 58
briantist	Capítulos 17 y 67
camilohe	Capítulo 27
Chris N	Capítulos 1 y 11
Christophe	Capítulo 53
Christopher G. Lewis	Capítulo 7
Clijsters	Capítulos 1, 3 y 26
CmdrTchort	Capítulos 7 y 57
DarkLite1	Capítulos 1 y 22
Dave Anderson	Capítulo 22
DAXaholic	Capítulos 1 y 7
Deptor	Capítulo 25
djwork	Capítulo 11
Eris	Capítulos 2, 7 y 27
Euro Micelli	Capítulo 5
Florian Meyer	Capítulos 10 y 60
FoxDeploy	Capítulo 1
Frode F.	Capítulos 7, 8, 9, 13, 15, 21, 28, 29, 32, 35, 38, 39, 40, 62, 66 y 69
Giorgio Gambino	Capítulo 29
Giulio Caccin	Capítulo 55
Gordon Bell	Capítulos 1 y 3
Greg Bray	Capítulo 1
HAL9256	Capítulo 30
It	Capítulo 1
James Ruskin	Capítulos 12, 25 y 54
Jaqueline Vanek	Capítulo 13
jimmyb	Capítulo 23
JNYRanger	Capítulo 1
JPBlanc	Capítulo 3
jumbo	Capítulos 7, 8, 19, 27, 33, 34, 38 y 42
Keith	Capítulo 25
Kolob Canyon	Capítulo 15
Lachie White	Capítulo 63

Liam	Capítulos 2 y 6
Lieven Keersmaekers	Capítulo 29
lloyd	Capítulo 6
Luke Ryan	Capítulo 12
Madniz	Capítulo 39
Mark Wragg	Capítulos 1, 3 y 56
Mathieu Buisson	Capítulos 1 y 11
mattnicola	Capítulos 26 y 39
megamorf	Capítulos 11, 22 y 24
Mert Gülsoy	Capítulo 13
Mike Shepard	Capítulo 14
miken32	Capítulo 6
Moerwald	Capítulos 19 y 24
motcke	Capítulo 17
Mrk	Capítulo 1
Nikhil Vartak	Capítulos 12 y 41
Nooj	Capítulos 23 y 70
Poorkenny	Capítulos 1, 43 y 44
Prageeth Saravanan	Capítulos 2, 16 y 47
Ranadip Dutta	Capítulos 10, 13, 21 y 67
RapidCoder	Capítulo 54
Raziel	Capítulo 64
restless1987	Capítulos 2 y 9
Richard	Capítulos 7, 12, 26, 29 y 49
Roman	Capítulos 7, 19 y 65
Rowshi	Capítulo 29
Sam Martin	Capítulos 1, 24, 30, 45, 59 y 66
Schwarzie2478	Capítulo 58
SeeuD1	Capítulo 7
ShaneC	Capítulo 24
StephenP	Capítulo 7
Steve K	Capítulo 2
TessellatingHeckler	Capítulos 7 y 12
th1rdey3	Capítulo 1
TheIncorrigible1	Capítulo 7
tjrobinson	Capítulo 1
TravisEz13	Capítulos 1, 4, 5, 6, 8, 10, 12, 14, 21, 26, 36 y 48
Trevor Sullivan	Capítulos 14, 18, 50, 61, 71 y 72
Venkatakishnan	Capítulo 31
VertigoRay	Capítulos 7 y 46
void	Capítulo 9
vonPryz	Capítulo 1
W1M0R	Capítulo 37
Xalorous	Capítulos 1, 12 y 40
Xenophane	Capítulo 17
xvorsx	Capítulo 13
xXhRQ8sD2L7Z	Capítulo 21
YChi Lu	Capítulos 30 y 68