PowerShell

> ⚠ **DOCUMENTO EN DESARROLLO** ⚠

Introducción

PowerShell es una solución de automatización de tareas multiplataforma formada por un **shell de línea de comandos**, un **lenguaje de scripting** y un **marco de administración de configuración**. PowerShell está diseñado por Microsoft y funciona en Windows, Linux y macOS.

Introducido por primera vez en 2006, PowerShell ha evolucionado hasta convertirse en una herramienta esencial para administradores de sistemas, desarrolladores y profesionales de IT.

A diferencia de las interfaces tradicionales de línea de comandos, PowerShell utiliza un enfoque basado en objetos, lo que significa que las salidas de los comandos son representadas como objetos estructurados que pueden ser manipulados y filtrados de manera intuitiva. Esto proporciona una gran flexibilidad y eficiencia al realizar tareas administrativas y permite una mayor automatización de procesos.

Algunas de las características de PowerShell son:

- **Cmdlets**: PowerShell utiliza "*cmdlets*" (comandos compilados pequeños) como unidades básicas de trabajo. Estos *cmdlets* realizan funciones específicas y se pueden encadenar para crear secuencias más complejas de comandos. Se pronuncia "command-let" y la convención de nomenclatura de los cmdlets sigue un singular formato ***Verb-Noun*** para que se puedan detectar fácilmente.

- **Interactividad y scripts**: Se puede utilizar PowerShell de manera interactiva en la línea de comandos o escribir scripts completos para automatizar tareas repetitivas. Los scripts permiten realizar acciones complejas con un solo comando.

- **Acceso a tecnologías Microsoft**: PowerShell ofrece acceso directo a las tecnologías de Microsoft, como el Active Directory, Exchange, SharePoint y otras, facilitando la administración de entornos empresariales.

- **Consola gráfica**: además de la interfaz de línea de comandos, PowerShell cuenta con una consola gráfica llamada PowerShell ISE (Integrated Scripting Environment) que facilita la creación, depuración y ejecución de scripts.

- **Soporte multiplataforma**: a partir de la versión PowerShell Core, la herramienta se ha vuelto multiplataforma, lo que significa que ahora está disponible para sistemas operativos como Linux y macOS, ampliando su utilidad más allá del ecosistema Windows.

- **Automatización más eficiente**: PowerShell simplifica tareas administrativas mediante la automatización de procesos, ahorrando tiempo y reduciendo posibles errores humanos.

- **Consistencia**: la estructura basada en objetos y la sintaxis coherente de PowerShell contribuyen a una administración consistente y predecible de sistemas.

- **Integración con .NET**: al estar construido sobre el marco .NET, PowerShell permite la integración con bibliotecas y servicios .NET, ampliando aún más sus capacidades.

En resumen, PowerShell es una herramienta poderosa que ofrece una forma eficaz de administrar y automatizar tareas en entornos Windows y más allá, convirtiéndose en una herramienta esencial para profesionales de IT y administradores de sistemas.

> ⚠ Introducción generada por **ChatGPT** ⚠

Windows PowerShell vs PowerShell Core

Windows PowerShell es la implementación original de PowerShell lanzada por Microsoft junto con Windows 7 en **2009**. Está diseñada específicamente para entornos Windows y utiliza el marco **.NET Framework**. A lo largo de los años, Windows PowerShell se convirtió en una herramienta esencial para la administración de sistemas Windows, proporcionando una interfaz de línea de comandos y un entorno de scripting robusto para la automatización de tareas.

Por otro lado, **PowerShell Core** es una versión posterior y más avanzada de PowerShell que fue lanzada en **2016**. La principal diferencia es que **PowerShell Core es multiplataforma**, lo que significa que puede ejecutarse no solo en sistemas operativos Windows, sino también en Linux y macOS. Además, PowerShell Core es de código abierto y se basa en el marco **.NET Core**.

- **Ecosistema común**: ambas versiones comparten un conjunto básico de conceptos y cmdlets, lo que facilita la transición entre ellas. Sin embargo, hay algunas diferencias en los cmdlets y características específicas de cada versión.
- **Compatibilidad**: aunque PowerShell Core es más versátil al ser multiplataforma, Windows PowerShell sigue siendo la opción predeterminada en entornos Windows tradicionales. Muchos scripts y comandos desarrollados en Windows PowerShell pueden ejecutarse en PowerShell Core, pero puede haber casos en los que se requieran ajustes debido a las diferencias de plataforma.
- **Desarrollo continuo**: Microsoft ha indicado que PowerShell Core es el camino a seguir y recibirá actualizaciones continuas, mientras que Windows PowerShell está en modo de mantenimiento. A medida que evoluciona el ecosistema, se alienta a los usuarios a adoptar PowerShell Core para aprovechar las nuevas características y mejoras.

PowerShell Core representa la evolución de Windows PowerShell, siendo más versátil y adaptable a diferentes plataformas, mientras que Windows PowerShell sigue siendo relevante en entornos Windows más tradicionales.

> ⚠️ Sección generada por **ChatGPT** ⚠️

Comandos básicos

Los comandos compilados en PowerShell se conocen como ***cmdlets***, que se pronuncian como "command-let", no "CMD-let". La convención de nomenclatura de los *cmdlets* sigue un singular formato **Verb-Noun** para que se puedan detectar fácilmente. Por ejemplo, `Get-Process` es el *cmdlet* para determinar qué procesos se ejecutan y `Get-Service` es el *cmdlet* para recuperar una lista de servicios.

Las **funciones**, también conocidas como *cmdlets* de script y **alias**, son otros tipos de comandos de PowerShell. El término "comando de PowerShell" describe cualquier comando de PowerShell, independientemente de si es un cmdlet, una función o un alias.

También se pueden ejecutar comandos nativos del sistema operativo desde PowerShell, como programas tradicionales de línea de comandos como `ping.exe` y `ipconfig.exe`.

Algunos de los verbos más comunes:

- **Get** ➔ Utilizado para recuperar información.
- **Set** ➔ Utilizado para configurar o cambiar configuraciones.
- **New** ➔ Utilizado para crear nuevas instancias de objetos.
- **Remove** ➔ Utilizado para eliminar o remover elementos.
- **Invoke** ➔ Utilizado para ejecutar una acción u operación específica.
- **Start** ➔ Utilizado para iniciar un proceso u operación.

- **Stop** ➔ Utilizado para detener o finalizar un proceso u operación.
- **Enable** ➔ Utilizado para activar o habilitar una función.
- **Disable** ➔ Utilizado para desactivar o inhabilitar una función.
- **Test** ➔ Utilizado para realizar pruebas o verificaciones.
- **Update** ➔ Utilizado para actualizar o refrescar datos o configuraciones.

```
# Muestra la lista completa de verbos
Get-Verb
```

Sistema de ayuda

```
# Muestra la ayuda para un determinado comando
Get-Help <cmd> # alias: help

# Muestra la ayuda online para un determinado comando
Get-Help <cmd> -Online

# Muestra ejemplos de uso para un determinado comando
Get-Help <cmd> -Examples

# Muestra la ayuda para 'Get-Process' con paginación, facilitando la lectura.
Get-Help Get-Process | More   # Alias: help

# Muestra todos los temas de ayuda disponibles que comienzan con "about_".
Get-Help about_*

# Muestra la ayuda de un parámetro
Get-Help -Name Get-Process -Parameter Id

# Actualiza la base de datos de ayuda de PowerShell para obtener la información más reciente.
# Debe ejecutarse en una consola elevada con privilegios de administrador.
Update-Help

# Actualiza la ayuda para un determinado módulo y con un determinado idioma
Update-Help Microsoft.PowerShell.Management -Force -UICulture en-US

# Lista todos los comandos que utilizan el verbo 'Add' en su nombre.
# Es útil cuando se buscan comandos específicos relacionados con una acción (como 'Add').
Get-Command -Verb Add

# Muestra el comando completo que corresponde al alias 'ps'.
# Los alias son versiones abreviadas de comandos, útiles para ahorrar tiempo al escribir.
# Sin embargo, es recomendable usar los nombres completos en scripts formales
Get-Alias ps

# Muestra el alias que corresponde al comando 'Get-Process'.
# Esto te permite descubrir si un comando tiene un alias más corto que puedas utilizar.
Get-Alias -Definition Get-Process

# Muestra los alias y su definición
Get-Alias | Select-Object Name, Definition

# Muestra los miembros (propiedades y métodos) del objeto que devuelve 'Get-Process'.
Get-Process | Get-Member

# Abre una interfaz gráfica para rellenar los parámetros del comando 'Get-WinEvent'.
Show-Command Get-WinEvent

# Muestra la lista de módulos
Get-Module
```

```powershell
# Muestra la lista de módulos disponibles
Get-Module -ListAvailable

# Muesta la lista de comandos de un módulo
Get-Command -Module PSReadLine
```

Uso de la finalización con tabulación

TODO

Obtención de ayuda dinámica

TODO

Uso de los controladores de teclas de PSReadLine

TODO

Variables automáticas

```powershell
# Muestra la versión actual de PowerShell que se está utilizando.
# Es importante conocer la versión para asegurar la compatibilidad de scripts y módulos.
$PSVersionTable

# Muestra el directorio del usuario
$HOME

# Muestra un objeto path que representa la ruta de acceso completa
$PWD

# Contiene el objeto actual en el objeto de canalización o _pipe.
$PSItem

# Igual que '$PSItem'
$_
```

Configuración y Políticas de Ejecución

```powershell
# Muestra la lista de políticas de ejecución actuales para cada nivel de alcance
# Esto permite ver cómo están configuradas las políticas en el entorno.
Get-ExecutionPolicy -List

# Establece la política de ejecución a 'AllSigned', lo que requiere que todos los scripts estén firmados por un editor d
# Esto proporciona una capa adicional de seguridad al evitar la ejecución de scripts no firmados.
Set-ExecutionPolicy AllSigned

# Aquí se enumeran las políticas de ejecución disponibles en PowerShell y su descripción:
# - Restricted: No se ejecutan scripts.
# - RemoteSigned: Los scripts descargados solo se ejecutan si están firmados por un editor de confianza.
# - AllSigned: Todos los scripts deben estar firmados por un editor de confianza.
# - Unrestricted: Permite la ejecución de todos los scripts, sin restricciones.
# Utiliza 'help about_Execution_Policies' para obtener información adicional y detalles completos sobre cada política.
help about_Execution_Policies
```

Invocación de Comandos y Gestión del Ámbito

```powershell
# Llamar a comandos externos, ejecutables,
# y funciones usando el operador de llamada (&).
# Las rutas de archivos ejecutables con argumentos pasados o que contienen espacios pueden causar problemas.

# Ejemplo de ruta con espacios
"C:\Program Files\dotnet\dotnet.exe"
# El término 'C:\Program' no se reconoce como el nombre de un cmdlet,
# función, archivo de script o programa ejecutable.
# Verifica la ortografía del nombre, o si se incluyó una ruta,
# asegúrate de que la ruta sea correcta y vuelve a intentarlo.

# Ejemplo de llamada correcta
&"C:\Program Files\dotnet\dotnet.exe --help"   # falla porque PowerShell considera '--help' como parte del comando.
&"C:\Program Files\dotnet\dotnet.exe" --help   # éxito # Aquí '--help' se pasa como argumento separado.

# Alternativamente, puedes usar dot-sourcing aquí
."C:\Program Files\dotnet\dotnet.exe" --help   # éxito

# El operador de llamada (&) es similar a Invoke-Expression (IEX),
# pero IEX se ejecuta en el alcance actual.
# Un uso de '&' sería invocar un bloque de script dentro de tu script.
# Nota que las variables están en diferentes ámbitos

$i = 2
$scriptBlock = { $i=5; Write-Output $i }
& $scriptBlock # => 5
$i # => 2

invoke-expression '$i=5; Write-Output $i' # => 5
$i # => 5

# Alternativamente, para preservar los cambios en variables públicas
# puedes usar "Dot-Sourcing". Esto se ejecutará en el alcance actual.
$x = 1
& {$x = 2}; $x # => 1

. {$x = 2}; $x # => 2
```

Pipes

Los cmdlets en PowerShell se pueden encadenar usando tuberías (`|`) para realizar operaciones de filtrado, ordenación, selección, agrupación, conteo, exportación, y más, permitiendo una manipulación y procesamiento efectivos de datos.

```powershell
# 1. FILTRAR datos
# Filtra los procesos que usan más de 100 unidades de CPU.
Get-Process | Where-Object { $_.CPU -gt 100 }

# Filtra los servicios que están en el modo 'Running'
Get-Service | Where-Object { $PSItem.Status -eq 'Running' }

# Equivalente
Get-Service | ? { $_.Status -eq 'Running' }  # ? alias de 'Where-Object'

# Busca la cadena "error" en el contenido del archivo file.txt.
Get-Content file.txt | Select-String "error"

# 2. ORDENAR datos
# Ordena los procesos por uso de CPU en orden descendente.
Get-Process | Sort-Object CPU -Descending

# 3. SELECCIONAR propiedades
# Selecciona y muestra solo las propiedades Name y CPU de los procesos.
Get-Process | Select-Object Name, CPU

# 4. FORMATEAR la salida
# Muestra los procesos en formato de tabla, ajustando el tamaño automáticamente.
```

```powershell
Get-Process | Format-Table Name, CPU -AutoSize

# Muestra los procesos en formato de lista.
Get-Process | Format-List Name, CPU

# El siguiente comando ps (alias de Get-Process) devuelve una tabla con 3 columnas
# La tercera columna es el valor de memoria virtual en MB y usando 2 dígitos decimales
# Las columnas calculadas pueden escribirse más extensamente como:
# `@{name='lbl';expression={$_}`
ps | Format-Table ID,Name,@{n='VM(MB)';e={'{0:n2}' -f ($_.VM / 1MB)}} -autoSize

# 5. AGRUPAR resultados
# Agrupa los procesos basados en la propiedad 'Name'.
Get-Process | Group-Object -Property Name

# 6. CONTAR objetos
# Cuenta el número total de procesos.
Get-Process | Measure-Object | Select-Object -ExpandProperty Count

# Realiza la suma
Get-Process | Measure-Object -Property WorkingSet -Sum

# 7. MANIPULAR texto
# Ejecuta un bloque de código para cada proceso y muestra el nombre.
Get-Process | ForEach-Object { $_.Name }

# Convierte la salida de los procesos a una cadena de texto.
Get-Process | Out-String

# 8. EXPORTAR datos
# Exporta la información de procesos a un archivo CSV.
Get-Process | Select-Object Name, CPU | Export-Csv -Path "processes.csv" -NoTypeInformation

# Redirige la salida de los procesos a un archivo de texto.
Get-Process | Out-File -FilePath "processes.txt"

# 9. REDIRIGIR Salida
# Envía la salida de los procesos a la pantalla.
Get-Process | Out-Host

# 10. Ver Resultados en una Vista Previa
# Muestra la salida de los procesos una página a la vez.
Get-Process | More

# 11. Uso de `$_` en Tuberías
# `$_` representa el objeto actual en el objeto de tubería
# Filtra archivos cuyo nombre contiene 'c' y exporta los resultados a CSV.
Get-ChildItem | Where-Object { $_.Name -match 'c' } | Export-Csv -Path "exportado.txt"

# Filtra archivos cuyo nombre contiene 'c', convierte a HTML y guarda en un archivo.
Get-ChildItem | Where-Object { $_.Name -match 'c' } | ConvertTo-HTML | Out-File -FilePath "exportado.html"

# 12. Revisar Propiedades y Métodos
# Revisa los métodos y propiedades de los objetos en la tubería.
Get-ChildItem | Get-Member

# Revisa los métodos y propiedades del objeto 'DateTime'.
Get-Date | Get-Member

# Acceder a los métodos y propiedades.
# 'Get-Date' devuelve un objeto de tipo 'DateTime'
(Get-Date).Year # imprime la propiedad 'Year'
(Get-Date).GetType() # es un método de 'DateTime' que imprime el tipo

# 13. Continuación de Línea
# Ejemplo de comando largo usando ` (continuación de línea).
Get-Process | Sort-Object ID -Descending | Select-Object -First 10 Name, ID, VM `
 | Stop-Process -WhatIf

# 14. Uso de Objetos Calculados
# Muestra procesos con una columna calculada para memoria virtual en MB.
```

```powershell
ps | Format-Table ID, Name, @{n='VM(MB)'; e={'{0:n2}' -f ($_.VM / 1MB)}} -AutoSize

# 15. Uso de `ForEach-Object` (abreviado como `%`)
# Procesa una colección y cuenta el número de elementos.
 # ` es el caracter de continuación de línea.
(a, b, c) | ForEach-Object `
 -Begin { "Iniciando"; $counter = 0 } `
 -Process { "Procesando $_"; $counter++ } `
 -End { "Terminando: $counter" }

# También se puede termine la línea con un |
Get-Process |
  Sort-Object ID -Descending |
  Select-Object -First 10 Name,ID,VM | Stop-Process -WhatIf

# Realiza una operación por cada uno de los objetos de la colección
Get-ChildItem | ForEach-Object { Rename-Item $_ -NewName "Prefix_$_" }
```

A partir de PowerShell 7, PowerShell implementa los operadores `&&` y `||` para encadenar canalizaciones condicionalmente.

El operador `&&` ejecuta la canalización derecha, si la canalización izquierda se ha realizado correctamente. Por el contario, el operador `||` ejecuta la canalización derecha, si la canalización izquierda no se ha realizado correctamente.

```powershell
# Output → First Second
Write-Output 'First' && Write-Output 'Second'

# Output → Write-Error: Bad
Write-Error 'Bad' && Write-Output 'Second'
```

FileSystem

```powershell
New-Item -path file.txt -type 'file' -value 'contents'
New-Item -path file.txt -type 'dir'
Copy-Item <src> -destination <dest>
Move-Item -path  <src> -destination <dest>
Remove-Item <file>
Test-Path <path>
Rename-Item -path <path> -newname <newname>

# using .NET Base Class Library
[System.IO.File]::WriteAllText('test.txt', '')
[System.IO.File]::Delete('test.txt')

Get-Content -Path "test.txt"
Get-Process | Out-File -FilePath "processes.txt"# Output to file
Get-Process | Export-Csv -Path "processes.csv"  # Output to csv
$data = Import-Csv -Path "data.csv"             # Import from csv

# Abre Windows Explorer en la ruta actual (usando el alias ii de Invoke-Item)
ii .

# Desbloquear un archivo después de descargarlo de Internet
Get-ChildItem -Recurse | Unblock-File

# Cambiar el directorio de trabajo sin tener que memorizar la ruta anterior
Push-Location c:\temp # se cambia el directorio de trabajo a c:\temp
Pop-Location # revierte el cambio y se devuelve a donde estaba al principio
# Los aliases son : pushd y popd

# Actualizar la ruta de ejecuciones (PATH)
$env:PATH = [System.Environment]::GetEnvironmentVariable("Path", "Machine") +
 ";" + [System.Environment]::GetEnvironmentVariable("Path", "User")

# Encontrar Python en el path
```

```powershell
$env:PATH.Split(";") | Where-Object { $_ -like "*python*"}

# Para crear un acceso directo
$WshShell = New-Object -comObject WScript.Shell
$Shortcut = $WshShell.CreateShortcut($link)
$Shortcut.TargetPath = $file
$Shortcut.WorkingDirectory = Split-Path $file
$Shortcut.Save()

# $Profile es la ruta completa para su `Microsoft.PowerShell_profile.ps1`
# Todo el código alojado allí será ejecutado cuando se ejecuta una nueva sesión de PS
if (-not (Test-Path $Profile)) {
 New-Item -Type file -Path $Profile -Force
 notepad $Profile
}
```

Scripting

```powershell
# Single line comments start with a number symbol.

<#
  Multi-line comments
  like so
#>


################################################
## 1. Primitive Datatypes and Operators
################################################

# Numbers
3 # => 3

# Math
1 + 1   # => 2
8 - 1   # => 7
10 * 2  # => 20
35 / 5  # => 7.0

# Powershell uses banker's rounding,
# meaning [int]1.5 would round to 2 but so would [int]2.5
# Division always returns a float.
# You must cast result to [int] to round.
[int]5 / [int]3       # => 1.66666666666667
[int]-5 / [int]3      # => -1.66666666666667
5.0 / 3.0   # => 1.66666666666667
-5.0 / 3.0  # => -1.66666666666667
[int]$result = 5 / 3
$result # => 2

# Modulo operation
7 % 3  # => 1

# Exponentiation requires longform or the built-in [Math] class.
[Math]::Pow(2,3)  # => 8

# Enforce order of operations with parentheses.
1 + 3 * 2  # => 7
(1 + 3) * 2  # => 8

# Boolean values are primitives (Note: the $)
$True  # => True
$False  # => False

# negate with !
!$True   # => False
!$False  # => True
```

```powershell
# Boolean Operators
# Note "-and" and "-or" usage
$True -and $False  # => False
$False -or $True   # => True

# True and False are actually 1 and 0 but only support limited arithmetic.
# However, casting the bool to int resolves this.
$True + $True # => 2
$True * 8    # => '[System.Boolean] * [System.Int32]' is undefined
[int]$True * 8 # => 8
$False - 5   # => -5

# Comparison operators look at the numerical value of True and False.
0 -eq $False  # => True
1 -eq $True   # => True
2 -eq $True   # => False
-5 -ne $False # => True

# Using boolean logical operators on ints casts to booleans for evaluation.
# but their non-cast value is returned
# Don't mix up with bool(ints) and bitwise -band/-bor
[bool](0)     # => False
[bool](4)     # => True
[bool](-6)    # => True
0 -band 2     # => 0
-5 -bor 0     # => -5

# Equality is -eq (equals)
1 -eq 1  # => True
2 -eq 1  # => False

# Inequality is -ne (notequals)
1 -ne 1  # => False
2 -ne 1  # => True

# More comparisons
1 -lt 10  # => True
1 -gt 10  # => False
2 -le 2  # => True
2 -ge 2  # => True

# Seeing whether a value is in a range
1 -lt 2 -and 2 -lt 3  # => True
2 -lt 3 -and 3 -lt 2  # => False

# (-is vs. -eq) -is checks if two objects are the same type.
# -eq checks if the objects have the same values, but sometimes doesn't work
# as expected.
# Note: we called '[Math]' from .NET previously without the preceeding
# namespaces. We can do the same with [Collections.ArrayList] if preferred.
[System.Collections.ArrayList]$a = @()  # Point a at a new list
$a = (1,2,3,4)
$b = $a                                 # => Point b at what a is pointing to
$b -is $a.GetType()                     # => True, a and b equal same type
$b -eq $a                               # => None! See below
[System.Collections.Hashtable]$b = @{}  # => Point a at a new hash table
$b = @{'one' = 1
       'two' = 2}
$b -is $a.GetType()                     # => False, a and b types not equal

# Strings are created with " or ' but " is required for string interpolation
"This is a string."
'This is also a string.'

# Strings can be added too! But try not to do this.
"Hello " + "world!"  # => "Hello world!"

# A string can be treated like a list of characters
"Hello world!"[0]  # => 'H'
```

```powershell
# You can find the length of a string
("This is a string").Length  # => 16

# You can also format using f-strings or formatted string literals.
$name = "Steve"
$age = 22
"He said his name is $name."
# => "He said his name is Steve"
"{0} said he is {1} years old." -f $name, $age
# => "Steve said he is 22 years old"
"$name's name is $($name.Length) characters long."
# => "Steve's name is 5 characters long."

# Strings can be compared with -eq, but are case insensitive. We can
# force with -ceq or -ieq.
"ab" -eq "ab"  # => True
"ab" -eq "AB"  # => True!
"ab" -ceq "AB"  # => False
"ab" -ieq "AB"  # => True

# Escape Characters in Powershell
# Many languages use the '\', but Windows uses this character for
# file paths. Powershell thus uses '`' to escape characters
# Take caution when working with files, as '`' is a
# valid character in NTFS filenames.
"Showing`nEscape Chars" # => new line between Showing and Escape
"Making`tTables`tWith`tTabs" # => Format things with tabs

# Negate pound sign to prevent comment
# Note that the function of '#' is removed, but '#' is still present
`#Get-Process # => Fail: not a recognized cmdlet

# $null is not an object
$null  # => None

# $null, 0, and empty strings and arrays all evaluate to False.
# All other values are True
function Test-Value ($value) {
  if ($value) {
    Write-Output 'True'
  }
  else {
    Write-Output 'False'
  }
}

Test-Value ($null) # => False
Test-Value (0)     # => False
Test-Value ("")    # => False
Test-Value []      # => True
# *[] calls .NET class; creates '[]' string when passed to function
Test-Value ({})    # => True
Test-Value @()     # => False


##################################################
## 2. Variables and Collections
##################################################

# Powershell uses the "Write-Output" function to print
Write-Output "I'm Posh. Nice to meet you!"  # => I'm Posh. Nice to meet you!

# Simple way to get input data from console
$userInput = Read-Host "Enter some data: " # Returns the data as a string

# There are no declarations, only assignments.
# Convention is to use camelCase or PascalCase, whatever your team uses.
$someVariable = 5
$someVariable  # => 5

# Accessing a previously unassigned variable does not throw exception.
```

```powershell
# The value is $null by default

# Ternary Operators exist in Powershell 7 and up
0 ? 'yes' : 'no'  # => no


# The default array object in Powershell is an fixed length array.
$defaultArray = "thing","thing2","thing3"
# you can add objects with '+=', but cannot remove objects.
$defaultArray.Add("thing4") # => Exception "Collection was of a fixed size."
# To have a more workable array, you'll want the .NET [ArrayList] class
# It is also worth noting that ArrayLists are significantly faster

# ArrayLists store sequences
[System.Collections.ArrayList]$array = @()
# You can start with a prefilled ArrayList
[System.Collections.ArrayList]$otherArray = @(5, 6, 7, 8)

# Add to the end of a list with 'Add' (Note: produces output, append to $null)
$array.Add(1) > $null    # $array is now [1]
$array.Add(2) > $null    # $array is now [1, 2]
$array.Add(4) > $null    # $array is now [1, 2, 4]
$array.Add(3) > $null    # $array is now [1, 2, 4, 3]
# Remove from end with index of count of objects-1; array index starts at 0
$array.RemoveAt($array.Count-1) # => 3 and array is now [1, 2, 4]
# Let's put it back
$array.Add(3) > $null   # array is now [1, 2, 4, 3] again.

# Access a list like you would any array
$array[0]   # => 1
# Look at the last element
$array[-1]  # => 3
# Looking out of bounds returns nothing
$array[4]  # blank line returned

# Remove elements from a array
$array.Remove($array[3])  # $array is now [1, 2, 4]

# Insert at index an element
$array.Insert(2, 3)  # $array is now [1, 2, 3, 4]

# Get the index of the first item found matching the argument
$array.IndexOf(2)  # => 1
$array.IndexOf(6)  # Returns -1 as "outside array"

# You can add arrays
# Note: values for $array and for $otherArray are not modified.
$array + $otherArray  # => [1, 2, 3, 4, 5, 6, 7, 8]

# Concatenate arrays with "AddRange()"
$array.AddRange($otherArray)  # Now $array is [1, 2, 3, 4, 5, 6, 7, 8]

# Check for existence in a array with "in"
1 -in $array  # => True

# Examine length with "Count" (Note: "Length" on arrayList = each items length)
$array.Count  # => 8

# You can look at ranges with slice syntax.
$array[1,3,5]     # Return selected index  => [2, 4, 6]
$array[1..3]      # Return from index 1 to 3 => [2, 3, 4]
$array[-3..-1]    # Return from last 3 to last 1 => [6, 7, 8]
$array[-1..-3]    # Return from last 1 to last 3 => [8, 7, 6]
$array[2..-1]     # Return from index 2 to last (NOT as most expect) => [3, 2, 1, 8]
$array[0,2+4..6]  # Return multiple ranges with the + => [1, 3, 5, 6, 7]

# -eq doesn't compare array but extract the matching elements
$array = 1,2,3,1,1
$array -eq 1          # => 1,1,1
($array -eq 1).Count  # => 3
```

```powershell
# Tuples are like arrays but are immutable.
# To use Tuples in powershell, you must use the .NET tuple class.
$tuple = [System.Tuple]::Create(1, 2, 3)
$tuple.Item(0)      # => 1
$tuple.Item(0) = 3  # Raises a TypeError

# You can do some of the array methods on tuples, but they are limited.
$tuple.Length       # => 3
$tuple + (4, 5, 6)  # => Exception
$tuple[0..2]        # => $null (in powershell 5)    => [1, 2, 3] (in powershell 7)
2 -in $tuple        # => False


# Hashtables store mappings from keys to values, similar to (but distinct from) Dictionaries.
# Hashtables do not hold entry order as arrays do.
$emptyHash = @{}
# Here is a prefilled hashtable
$filledHash = @{"one"= 1
                "two"= 2
                "three"= 3}

# Look up values with []
$filledHash["one"]  # => 1

# Get all keys as an iterable with ".Keys".
$filledHash.Keys  # => ["one", "two", "three"]

# Get all values as an iterable with ".Values".
$filledHash.Values  # => [1, 2, 3]

# Check for existence of keys or values in a hash with "-in"
"one" -in $filledHash.Keys  # => True
1 -in $filledHash.Values    # => False (in powershell 5)    => True (in powershell 7)

# Looking up a non-existing key returns $null
$filledHash["four"]  # $null

# Adding to a hashtable
$filledHash.Add("five",5)  # $filledHash["five"] is set to 5
$filledHash.Add("five",6)  # exception "Item with key "five" has already been added"
$filledHash["four"] = 4    # $filledHash["four"] is set to 4, running again does nothing

# Remove keys from a hashtable
$filledHash.Remove("one") # Removes the key "one" from filled hashtable


####################################################
## 3. Control Flow and Iterables
####################################################

# Let's just make a variable
$someVar = 5

# Here is an if statement.
# This prints "$someVar is smaller than 10"
if ($someVar -gt 10) {
    Write-Output "$someVar is bigger than 10."
}
elseif ($someVar -lt 10) {    # This elseif clause is optional.
    Write-Output "$someVar is smaller than 10."
}
else {                        # This is optional too.
    Write-Output "$someVar is indeed 10."
}


<#
Foreach loops iterate over arrays
prints:
    dog is a mammal
    cat is a mammal
```

```
    mouse is a mammal
#>
foreach ($animal in ("dog", "cat", "mouse")) {
    # You can use -f to interpolate formatted strings
    "{0} is a mammal" -f $animal
}

<#
For loops iterate over arrays and you can specify indices
prints:
   0 a
   1 b
   2 c
   3 d
   4 e
   5 f
   6 g
   7 h
#>
$letters = ('a','b','c','d','e','f','g','h')
for($i=0; $i -le $letters.Count-1; $i++){
    Write-Host $i, $letters[$i]
}

<#
While loops go until a condition is no longer met.
prints:
    0
    1
    2
    3
#>
$x = 0
while ($x -lt 4) {
    Write-Output $x
    $x += 1  # Shorthand for x = x + 1
}

# Switch statements are more powerful compared to most languages
$val = "20"
switch($val) {
  { $_ -eq 42 }           { "The answer equals 42"; break }
  '20'                    { "Exactly 20"; break }
  { $_ -like 's*' }       { "Case insensitive"; break }
  { $_ -clike 's*'}       { "clike, ceq, cne for case sensitive"; break }
  { $_ -notmatch '^.*$'}  { "Regex matching. cnotmatch, cnotlike, ..."; break }
  default                 { "Others" }
}

# Handle exceptions with a try/catch block
try {
    # Use "throw" to raise an error
    throw "This is an error"
}
catch {
    Write-Output $Error.ExceptionMessage
}
finally {
    Write-Output "We can clean up resources here"
}


# Writing to a file
$contents = @{"aa"= 12
              "bb"= 21}
$contents | Export-CSV "$env:HOMEDRIVE\file.csv" # writes to a file

$contents = "test string here"
$contents | Out-File "$env:HOMEDRIVE\file.txt" # writes to another file

# Read file contents and convert to json
```

```powershell
Get-Content "$env:HOMEDRIVE\file.csv" | ConvertTo-Json


####################################################
## 4. Functions
####################################################

# Use "function" to create new functions
# Keep the Verb-Noun naming convention for functions
function Add-Numbers {
 $args[0] + $args[1]
}

Add-Numbers 1 2 # => 3

# Calling functions with parameters
function Add-ParamNumbers {
 param( [int]$firstNumber, [int]$secondNumber )
 $firstNumber + $secondNumber
}

Add-ParamNumbers -FirstNumber 1 -SecondNumber 2 # => 3

# Functions with named parameters, parameter attributes, parsable documentation
<#
.SYNOPSIS
Setup a new website
.DESCRIPTION
Creates everything your new website needs for much win
.PARAMETER siteName
The name for the new website
.EXAMPLE
New-Website -Name FancySite -Po 5000
New-Website SiteWithDefaultPort
New-Website siteName 2000 # ERROR! Port argument could not be validated
('name1','name2') | New-Website -Verbose
#>
function New-Website() {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipeline=$true, Mandatory=$true)]
        [Alias('name')]
        [string]$siteName,
        [ValidateSet(3000,5000,8000)]
        [int]$port = 3000
    )
    BEGIN { Write-Output 'Creating new website(s)' }
    PROCESS { Write-Output "name: $siteName, port: $port" }
    END { Write-Output 'Website(s) created' }
}


####################################################
## 5. Modules
####################################################

# You can import modules and install modules
# The Install-Module is similar to pip or npm, pulls from Powershell Gallery
Install-Module dbaTools
Import-Module dbaTools

$query = "SELECT * FROM dbo.sometable"
$queryParams = @{
    SqlInstance = 'testInstance'
    Database    = 'testDatabase'
    Query       = $query
}
Invoke-DbaQuery @queryParams

# You can get specific functions from a module
Import-Module -Function Invoke-DbaQuery
```

```powershell
# Powershell modules are just ordinary Posh files. You
# can write your own, and import them. The name of the
# module is the same as the name of the file.

# You can find out which functions and attributes
# are defined in a module.
Get-Command -module dbaTools
Get-Help dbaTools -Full


#####################################################
## 6. Classes
#####################################################

# We use the "class" statement to create a class
class Instrument {
    [string]$Type
    [string]$Family
}

$instrument = [Instrument]::new()
$instrument.Type = "String Instrument"
$instrument.Family = "Plucked String"

$instrument

<# Output:
Type              Family
----              ------
String Instrument Plucked String
#>


#####################################################
## 6.1 Inheritance
#####################################################

# Inheritance allows new child classes to be defined that inherit
# methods and variables from their parent class.

class Guitar : Instrument
{
    [string]$Brand
    [string]$SubType
    [string]$ModelType
    [string]$ModelNumber
}

$myGuitar = [Guitar]::new()
$myGuitar.Brand       = "Taylor"
$myGuitar.SubType     = "Acoustic"
$myGuitar.ModelType   = "Presentation"
$myGuitar.ModelNumber = "PS14ce Blackwood"

$myGuitar.GetType()

<#
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Guitar                                   Instrument
#>


#####################################################
## 7. Advanced
#####################################################

# The powershell pipeline allows things like High-Order Functions.
```

```powershell
# Group-Object is a handy cmdlet that does incredible things.
# It works much like a GROUP BY in SQL.

<#
 The following will get all the running processes,
 group them by Name,
 and tell us how many instances of each process we have running.
 Tip: Chrome and svcHost are usually big numbers in this regard.
#>
Get-Process | Foreach-Object ProcessName | Group-Object

# Useful pipeline examples are iteration and filtering.
1..10 | ForEach-Object { "Loop number $PSITEM" }
1..10 | Where-Object { $PSITEM -gt 5 } | ConvertTo-Json

# A notable pitfall of the pipeline is its performance when
# compared with other options.
# Additionally, raw bytes are not passed through the pipeline,
# so passing an image causes some issues.
# See more on that in the link at the bottom.

<#
 Asynchronous functions exist in the form of jobs.
 Typically a procedural language,
 Powershell can operate non-blocking functions when invoked as Jobs.
#>

# This function is known to be non-optimized, and therefore slow.
$installedApps = Get-CimInstance -ClassName Win32_Product

# If we had a script, it would hang at this func for a period of time.
$scriptBlock = {Get-CimInstance -ClassName Win32_Product}
Start-Job -ScriptBlock $scriptBlock

# This will start a background job that runs the command.
# You can then obtain the status of jobs and their returned results.
$allJobs = Get-Job
$jobResponse = Get-Job | Receive-Job


# Math is built in to powershell and has many functions.
$r=2
$pi=[math]::pi
$r2=[math]::pow( $r, 2 )
$area = $pi*$r2
$area

# To see all possibilities, check the members.
[System.Math] | Get-Member -Static -MemberType All


<#
 This is a silly one:
 You may one day be asked to create a func that could take $start and $end
 and reverse anything in an array within the given range
 based on an arbitrary array without mutating the original array.
 Let's see one way to do that and introduce another data structure.
#>

$targetArray = 'a','b','c','d','e','f','g','h','i','j','k','l','m'

function Format-Range ($start, $end, $array) {
    [System.Collections.ArrayList]$firstSectionArray = @()
    [System.Collections.ArrayList]$secondSectionArray = @()
    [System.Collections.Stack]$stack = @()
    for ($index = 0; $index -lt $array.Count; $index++) {
        if ($index -lt $start) {
            $firstSectionArray.Add($array[$index]) > $null
        }
        elseif ($index -ge $start -and $index -le $end) {
            $stack.Push($array[$index])
```

```
        }
        else {
            $secondSectionArray.Add($array[$index]) > $null
        }
    }
    $finalArray = $firstSectionArray + $stack.ToArray() + $secondSectionArray
    return $finalArray
}

Format-Range 2 6 $targetArray
# => 'a','b','g','f','e','d','c','h','i','j','k','l','m'

# The previous method works, but uses extra memory by allocating new arrays.
# It's also kind of lengthy.
# Let's see how we can do this without allocating a new array.
# This is slightly faster as well.

function Format-Range ($start, $end) {
  while ($start -lt $end)
  {
      $temp = $targetArray[$start]
      $targetArray[$start] = $targetArray[$end]
      $targetArray[$end] = $temp
      $start++
      $end--
  }
  return $targetArray
}

Format-Range 2 6 # => 'a','b','g','f','e','d','c','h','i','j','k','l','m'
```

Referencias

- https://learn.microsoft.com/es-es/powershell/

Licencia