U.S. DEPARTMENT OF ENERGY'S CYBERFORCE® PROGRAM

CyberForce® 101

PowerShell

October 2023

cyberforcecompetition@anl.gov

Intro to PowerShell

Overview

Windows PowerShell combines scripting and the command line. It's typically used by IT professionals and system administrators to control and automate administrative tasks.

The PowerShell **cmdlet** ("command let") is a lightweight command used in the base PowerShell environment. It's different from commands because they are .NET Framework class objects that cannot be executed separately. Cmdlets process works on objects and are record-based (processes a single object at a time). A cmdlet is a series of commands stored in a text file with a `.ps1` extension.

A cmdlet always consists of a verb and a noun, separated with a hyphen. Some of the verbs are explained below:

- Get - To get something
- Start - To run something
- Out - To output something
- Stop - To stop something that is running
- Set - To define something
- New - To create something

PowerShell Concepts

- **Cmdlets** - build-command written in .`net` languages like VB or C#. It allows developers to extend the set of cmdlets by loading and write PowerShell snap-ins.
- **Functions** - commands written in PowerShell. It can be developed without using other IDEs like Visual Studio
- **Scripts** - text files on disk with a `.ps1` extension
- **Applications** - existing windows programs
- **What if** - tells the cmdlet not to execute, but tells you what would happen if it were to run
- **Confirm** - instruct the cmdlet to prompt before executing the command
- **Verbose** - gives a higher level of detail
- **Debug** - instructs the cmdlet to provide debugging information
- **ErrorAction** - instructs the cmdlet to perform a specific action when an error occurs; allowed actions continue, stop, silently-continue, and inquire
- **ErrorVariable** - specifies the variable which holds error information

- **OutVariable** - tells the cmdlet to use a specific variable to hold the output information
- **OutBuffer** - instructs the cmdlet to hold the specific number of objects before calling the next cmdlet in the pipeline

Important Commands

- `Get-Help` - help about PowerShell commands and topics
- `Get-Command` - get information about anything that can be invoked
- `Get-Service` - finds all cmdlets with the word 'service' in it
- `Get-Member` - show what can be done with an object
- `Get-Module` - shows packages of commands
- `Get-Content` - takes a file and processes its contents and does something with it

Here is an example of the PowerShell syntax with these types of commands. Below we create a new folder.

```
New-Item -Path 'X:\cyberforce' -ItemType Directory
```

Advanced Cmdlets

The `Get-Unique` cmdlet can be used to get the unique objects from a sorted list of objects. So if you want to print a list and there are elements within that list that repeat, it will only print the repeated items once.

The `Measure-Object` cmdlet can be used to get properties of the passed output like min, max, size, count, line, etc. An example of syntax can be seen below. The output of the command will show how many lines, words, and characters there are in the `test.txt` document.

```
get-content D:\temp\test.txt | measure-object -character -line -word
```

The `Compare-Object` cmdlet can be used to compare two objects. The output will be a comparison operator that shows the difference between the reference object and the difference object. Syntax can be seen below.

```
Compare-Object -ReferenceObject $(Get-Content D:\temp\test.txt) -DifferenceObject
$(Get-Content D:\temp\test2.txt)
```

The `Format-List` cmdlet can be used to format the output as a list of properties where a property appears on a new line. The output of the command below will show the name of the file and properties like creation time. You can also use this command in combination with other cmdlets like `Get-Service` to see the properties of all the different services.

```
Format-List -InputObject $(file.txt)
```

The `Format-Wide` cmdlet is used to format the output as a table with one property per object. It has the same syntax as `Format-List` but the output looks different.

The `Where-Object` cmdlet can be used to select objects having particular property values from the collection of objects that are passed to it. This can be used with the other cmdlets `Get-Service` and `Get-Process` to see the services and processes that match a given description. Syntax can be seen below.

```
Get-Service | Where-Object {$_.Status -eq "Stopped"}
# or
Get-Process | Where-Object {$_.ProcessName -Match "^p.*"}
```

The cmdlet `Get-ChildItem` can be used to get the items or child items in one or more specific locations. For instance if we want to get the items of the current directory, we can use the command `Get-ChildItem -Name`. We can also do this with other directories by specifying the path.

The `ForEach-Object` cmdlet is sued to perform operations on each object of a collection of objects. This can be arithmetic operations or other operations like Split. An example can be seen below.

```
"CyberForce.Reign.Adventurer", "Department.of.Energy" | ForEach-Object
{$_.Split(".")}
# Output
CyberForce
Reign
Adventurer
Department
of
Energy
```

The `Start-Sleep` cmdlet suspends activity in a script or session for a particular period of time. The `-s` flag stops it for a certain amount of seconds while the `-m` flag stops it for a certain amount of minutes. The command would look like this: `Start-Sleep -s 15`.

The `Read-Host` cmdlet reads from the console. Below the code asks the user to pass an input and reads it into a variable that will save the value entered.

```
$user_input = Read-Host "Please input a value"
```

The `Select-Object` cmdlet selects an object or its properties based on different flags. For instance if want only the unique values in a list we can use the command `$List | Select-Object -Unique`. If we want to select the last five processes running we can use the command below.

```
Get-Process | Select-Object -Property ProcessName, Id, WS -Last 5
```

The `Sort-Object` cmdlet sorts objects by their properties. It can sort them by specific flags or just in alphabetical or numerical order. For instance if we want to sort a list, we would use the command `$List | Sort-Object`.

The `Write-Warning` cmdlet writes warning messages. Syntax can be seen below.

```
Write-Warning "Test Warning"
# Output
WARNING: Test Warning
```

Similarly, the `Write-Host` cmdlet writes customized messages. Let's say we have a list of even numbers and we want to show the sequence between each number. The command would be as shown below.

```
Write-Host (2,4,6,8,10,12) -Separator " -> "
# Output
2 -> 4 -> 6 -> 8 -> 10 -> 12
```

The `Invoke-Item` cmdlet performs a default action on a specified item. For instance if we wanted to open a file, we could use the command like below.

```
Invoke-Item "D:\temp\test.txt"
```

`Invoke-Expression` is similar but used for a command or expression on a local computer. For example, if we store a command in a variable and then use the `Invoke-Expression` cmdlet on it, it will perform that command.

```
> $Command = 'Get-Process'
> Invoke-Expression $Command
```

The `Measure-Command` cmdlet measures the time taken by a script or command. It will show metrics like days, hours, minutes, ticks, total days, total seconds, etc. Syntax can be seen below.

```
Measure-Command { Get-EventLog "Windows PowerShell"}
```

The `Invoke-History` cmdlet runs the last command from the current session that has already run. The `Add-History` cmdlet adds commands in current history. The `Get-History` cmdlet gets the command run in the current session. The `Get-Culture` cmdlet gets the current culture set in Windows.

The `New-Alias` cmdlet creates a new alias. The `Get-Alias` cmdlet gets all the alias present in the current session of powershell.

```
New-Alias -Name help -Value Get-Help
```

Variables

PowerShell variables are named objects. Variable names start with `$` and can contain alphanumeric characters and underscores in their names. A variable can be created by typing a valid variable name.

Special Variables

Special Variable	Description
`$$`	Represents the last token in the last line received by the session
`$?`	Represents the execution status of the last operator; contains TRUE if the last operation succeeded and FALSE if it failed
`$^`	Represents the first token in the last line received by the session
`$_`	Same as `$PSItem`; contains the current object in the pipeline object; use this in commands that perform an action on every object or on selected objects in a pipeline
`$ARGS`	Represents an array of the undeclared parameters and/or parameter values that are passed to a function, script, or script block
`$CONSOLEFILENAME`	Represents the path of the console file (`.psc1`) that was most recently used in the session
`$Error`	An array of error objects
`$EVENT`	Represents a PSEventArgs object that represents the event that is being processed
`$EVENTARGS`	Represents an object that represents the first even argument that derives from EventArgs of the event that is being processed
`$EVENTSUBSCRIBER`	Represents a PSEventSubscriber object that represents the event subscriber of the event that is being processed

Special Variable	Description
`$EXECUTIONCONTEXT`	Represents an EngineIntrinsics object that represents the execution context of the PowerShell host
`$HOME`	Represents the full path of the user's home directory
`$HOST`	Display the name of the current hosting application
`$PROFILE`	Stores entire path of a user profile for the default shell
`$PID`	Stores the process identifier
`$NULL`	Contains empty or NULL value
`$FALSE`	Contains FALSE value
`$TRUE`	Contains TRUE value
`$FOREACH`	Represents the enumerator (not the resulting values) of a ForEach loop; can use the properties and methods of enumerators on the value of the $ForEach variable
`$INPUT`	Represents an enumerator that enumerates all input that is passed to a function
`$LASTEXITCODE`	Represents the exit code of the last Windows-based program that was run
`$MATCHES`	The $Matches variables works with the -match and -notmatch operators
`$MYINVOCATION`	$MyInvocation is populated only for scripts, function, and script blocks; PSScriptRoot and PSCommandPath properties of the $MyInvocation automatic variable contain information about the invoker or calling script, not the current script
`$NESTEDPROMPTLEVEL`	Represents the current prompt level
`$PSCMDLET`	Represents an object that represents the cmdlet or advanced function that is being run
`$PSCOMMANDPATH`	Represents the full path and file name of the script that is being run
`$PSCULTURE`	Represents the name of the culture currently in use in the OS
`$PSDEBUGCONTEXT`	While debugging, this contains information about the debugging environment; otherwise it is NULL
`$PSHOME`	Represents the full path of the installation directory for PowerShell
`$PSITEM`	Same as $_ ; contains the current object in the pipeline object

Special Variable	Description
`$PSSCRIPTROOT`	Represents the directory from which a script is being run
`$PSSENDERINFO`	Represents information about the user who started the PSSession, including the user identity and the time zone of the originating computer
`$PSUICulture`	Holds the name of the current UI culture
`$PSVERSIONTABLE`	Represents a read-only hash table that displays details about the version of PowerShell that is running in the current session
`$SENDER`	Represents the object that generated this event
`$SHELLID`	Represents the identifier of the current shell
`$STACKTRACE`	Represents a stack trace for the most recent error
`$THIS`	In a script block that defines a script property or script method, the $This variable refers to the object that is being extended

Operators

Arithmetic operators (`+, -, *, /, %`) work the same way in PowerShell as they do in mathematical expressions. Assignment operators (`=, +=, -=`) also work the same way.

Comparison operators have slightly different syntax in PowerShell.

Comparison Operator	Description
`-eq`	equals
`-ne`	not equals
`-gt`	greater than
`-ge`	greater than or equal to
`-lt`	less than
`-le`	less than or equal to

Logical operators work the same but proceed a `-` like so: `-AND, -OR,` and `-NOT`.

The redirectional operator `>` assigns output to be printed into the redirected file/output device.

Looping

Loops allow us to execute a statement or group of statements multiple times. PowerShell has four types of loops: for loop, forEach loop, while loop, and do...while loop.

for loops execute a sequence of statements multiple times. The syntax can be seen below.

```
> $array = @("var1", "var2", "var3")

> for($i = 0; $i -lt $array.length; $i++) {$array[$i]}

# Output
var1
var2
var3
```

forEach loops are enhanced for loops that are mainly used to traverse a collection of elements like arrays. An example can be seen below.

```
> $array = @("var1", "var2", "var3")

> foreach ($element in $array) {$element}

# Output
var1
var2
var3

# or can do this
> $array | foreach { $_ }

# Output
var1
var2
var3
```

while loops repeat a statement or group of statements while a given condition is true by testing the condition before executing the loop body. Syntax can be seen below.

```
> $array = @("var1", "var2", "var3")
$counter = 0;

while($counter -lt $array.length){
        $array[$counter]
        $counter += 1
}

# Output
var1
```

```
var2
var3
```

A do...while loop is like a while loop but tests the condition at the end of the loop body. An example is shown below.

```
> $array = @("var1", "var2", "var3")
$counter = 0;

do {
        $array[$counter]
        $counter += 1
} while ($counter -lt $array.length)

# Output
var1
var2
var3
```

Conditions

PowerShell has the following types of decision making statements: if statements, if...else statements, nested if statements, and switch statements.

An if statement consists of a boolean expression followed by one or more statements. An example is shown below.

```
$x = 10

if($x -le 20) {
        write-host("This is an if statement")
}

# Output
This is an if statement
```

An if statement can be followed by an else statement, which executes when the boolean expression is false. The syntax can be seen below.

```
$x = 30

if($x -le 20){
   write-host("If statement executed")
}else {
   write-host("Else statement executed")
}

# Output
Else statement executed.
```

A nested if statement is where you use one if or an elseif statement inside another if or elseif statement. An example is shown below.

```
$x = 30
$y = 10

if($x -eq 30){
   if($y -eq 10) {
      write-host("x = 30 and y = 10")
   }
}

# Output
x = 30 and y = 10
```

A switch statement allows a variable to be tested for equality against a list of values. An example is provided below.

```
switch(3){
   1 {"One"}
   2 {"Two"}
   3 {"Three"; break }
   4 {"Four"}
   3 {"Three Again"}
}

# Output
Three
```

Arrays

Declaring an array in PowerShell is similar to how you declare arrays in other languages. The syntax can be seen below.

```
$A = 1, 2, 3
# or
$A = 1..3
```

The `getType()` method returns the type of the array. Elements can be accessed using the index of the array with brackets (`$myList[1]` or `$myList[$i]`).

Hashtables

A hashtable stores key/value pairs in a hash table. You specify an object that is used as a key and the value that you want linked to it. Typically, we use Strings or numbers as keys.

To declare a hashtable in a program, we have to declare a variable to reference the hashtable like below.

```
$hash = @{ ID = 1; Color = "Blue"}
# or
$hash = @{}
```

Hashtable values are accessed through keys. Dot notation can also be used to access hashtable keys or values.

Regular Expressions

Regular expressions are special sequences of characters that help you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They are used to search, edit, and manipulate text and data.

Subexpression	Matches
`^`	Matches the beginning of the line
`$`	Matches the end of the line
`.`	Matches any single character except newline; using **m** option allows it to match the newline too
`[...]`	Matches any single character in brackets
`[^...]`	Matches any single character not in brackets
`\A`	Beginning of the entire string
`\z`	End of the entire string
`\Z`	End of the entire string except allowable final line terminator
`re*`	Matches 0 or more occurrences of the preceding expression
`re+`	Matches 1 or more of the previous thing
`re?`	Matches 0 or 1 occurrence of the preceding expression
`re{ n,}`	Matches n or more occurrences of the preceding expression

Subexpression	Matches
`re { n, m}`	Matches at least n and at most m occurrences of the preceding expression
`a| b`	Matches either a or b
`(re)`	Groups regular expressions and remembers the matched text
`(?: re)`	Groups regular expressions without remembering the matched text
`(?> re)`	Matches the independent pattern without backtracking
`\w`	Matches the word characters
`\W`	Matches the nonword characters
`\s`	Matches the whitespace; equivalent to [`\t\n\r\f`]
`\S`	Matches the nonwhitespace
`\d`	Matches the digits; equivalent to [`0-9`]
`\D`	Matches the nondigits
`\A`	Matches the beginning of the string
`\Z`	Matches the end of the string; if a newline exists, it matches just before newline
`\z`	Matches the end of the string
`\G`	Matches the point where the last match finished
`\n`	Back-reference to capture group number "n"
`\b`	Matches the word boundaries when outside the brackets; matches the backspace (0x08) when inside the brackets
`\B`	Matches the nonword boundaries
`\n, \t, etc`	Matches newlines, carriage returns, tabs, etc.
`\Q`	Escape (quote) all characters up to `\E`
`\E`	Ends quoting begun with `\Q`

Backtick

The backtick (` ) operator is also called the word-wrap operator, and it allows a command to be written in multiple lines. It can be used for new line ( `n) or tab (`t) in sentences too.

Scripts

PowerShell scripts are stored in `.ps1` files. To execute a script, you need to right-click it and click "Run with PowerShell." You will get output that either says it's restricted, AllSigned, RemoteSigned, or unrestricted.

- **Restricted** - no scripts are allowed (default)
- **AllSigned** - scripts signed by a trusted developer can run; script will ask for confirmation that you want it to run before executing
- **RemoteSigned** - your scripts or scripts signed by a trusted developer can run
- **Unrestricted** - any script can run

Change Execution Policy

First, open an elevated PowerShell prompt by right clicking on PowerShell and selecting "Run as Administrator." Next, enter the sequence of commands below:

- `Get-ExecutionPolicy`
- `Set-executionpolicy unrestricted` (or to the setting of your choice)
- Enter `Y` in prompt
- `Get-ExecutionPolicy` to confirm that it has changed

PowerShell vs Command Prompt

PowerShell	Command Prompt
Integrated with the Windows OS; offers interactive command line interface and scripting language	Default command line provided by Microsoft; simple win32 application that interacts and talks with any win32 objects
Uses cmdlets which can be invoked either in the runtime environment or automation scripts	No such features
Considers them objects; output can be passed as input to other cmdlets through the pipeline	Output from cmdlet is not just a stream of text but collection of objects
Very advanced features, capabilities, and inner functioning	Very basic

PowerShell Uses

IT administrators use PowerShell for management in large corporate networks. It's easier to implement a new security solution that's dependent on running a new service on a multitude of servers that you're managing. It makes these kinds of tasks much quicker and easier.

Sources

1. Powershell Tutorial for Beginners: Learn Powershell Scripting
2. Powershell Tutorial