PowerShell Basic

@iLabAfrica, Strathmore University
John Ombagi

Aliases & Similar Functions

- In PowerShell, there are many ways to achieve the same result. This can be illustrated nicely with the simple and familiar Hello World example:
- Using Write-Host:
  - Write-Host "Hello World"
- Using Write-Output:
  - Write-Output 'Hello world'

What is the difference?

- Although Write-Output & Write-Host both write to the screen there is a subtle difference.
- Write-Host writes only to stdout (i.e. the console screen), whereas Write-Output writes to both stdout AND to the output [success] stream allowing for redirection.
- Redirection (and streams in general) allow for the output of one command to be directed as input to another including assignment to a variable.
- $message = Write-Output "Hello World"
- $message

Alias

- Write-Output is aliased to Echo or Write
  - Echo 'Hello world'
  - Write 'Hello world'
- Or, by simply typing 'Hello world'!
  - 'Hello world'
- Another example of aliases in PowerShell is the common mapping of both older command prompt commands and
- BASH commands to PowerShell cmdlets.
  - All of the following produce a directory listing of the current directory.
    - C:\Windows> dir
    - C:\Windows> ls
    - C:\Windows> Get-ChildItem
- You can create your own alias with the Set-Alias cmdlet!
- As an example let's alisas Test-NetConnection, which is essentially the PowerShell equivalent to the command prompt's ping command, to "ping".
  - Set-Alias -Name ping -Value Test-NetConnection
- Now you can use ping instead of Test-NetConnection! Be aware that if the alias is already in use, you'll overwrite the association.
- The Alias will be alive, till the session is active.
- To overcome this issue, you can import all your aliases from an excel into your session once, before starting your work.

The Pipeline

- Cmdlet - The pipeline symbol | is used at the end of a cmdlet to take the data it exports and feed it to the next cmdlet.
  - Get-ChildItem | Select-Object Name
- This may be shortened to: gci | Select Name
- More advanced usage of the pipeline allows us to pipe the output of a cmdlet into a foreach loop:

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

- This may be shortened to:
  - gci | % { Copy $_.FullName C:\NewDirectory\ }
- Note that the example above uses the $_ automatic variable. $_ is the short alias of $PSItem which is an automatic variable which contains the current item in the pipeline.

Calling .Net Library Methods

- Static .Net library methods can be called from PowerShell by encapsulating the full class name in third bracket and then calling the method using ::
  - E.g. calling Path.GetFileName()
    - `C:\> [System.IO.Path]::GetFileName('C:\Windows\explorer.exe')`
- Static methods can be called from the class itself, but calling non-static methods requires an instance of the .Net class (an object).
- Let’s look at an example…

Example – Calling non-static Methods

- For example, the AddHours method cannot be called from the System.DateTime class itself. It requires an instance of the class:
- C:\> [System.DateTime]::AddHours(15)
  - We will get an err.
- In this case, we first create an object, for example:
  - C:\> $Object = [System.DateTime]::Now
- Then, we can use methods of that object, even methods which cannot be called directly from the System.DateTime class, like the AddHours method:
  - C:\> $Object.AddHours(15)

Commenting

- To comment on power scripts by prepending the line using the # (hash) symbol

```
# This is a comment in PowerShell
Get-ChildItem
```

- You can also have multi-line comments using <# and #> at the beginning and end of the comment respectively.

```
<#
This is a
multi-line
comment
#>
Get-ChildItem
```

Variables in PowerShell

- All variables in PowerShell begin with a US dollar sign ($)
  - $foo = "bar"
- This statement allocates a variable called foo with a string value of "bar".

Arrays

- Array declaration in Powershell is almost the same as instantiating any other variable.
  - `$myArrayOfInts = 1,2,3,4`
  - `$myArrayOfStrings = "1","2","3","4"`
- Adding to an array is as simple as using the + operator:
  - `$myArrayOfInts = $myArrayOfInts + 5`
- Combining arrays together
  - `$myArrayOfInts = 1,2,3,4`
  - `$myOtherArrayOfInts = 5,6,7`
  - `$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts`

List Assignment of Multiple Variables

- Powershell allows multiple assignment of variables and treats almost everything like an array or list.

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

- You can simply do this:
  - `$foo, $bar, $baz = $input.Split(".")`
- You can also do:
  - `$foo, $leftover = $input.Split(".")`
  - `$bar = $leftover[0]`
  - `$baz = $leftover[1]`

Scope

- The default scope for a variable is the enclosing container. If outside a script, or other container then the scope is Global.
- To specify a scope, it is prefixed to the variable name `$scope:varname` like so:

```
$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
```

- Output:
    - Global Scope Function (local) scope Function (local) scope Global Scope Global Scope

Removing a variable

- To remove a variable from memory, one can use the Remove-Item cmdlet. The variable name does NOT include the $.
  - Remove-Item Variable:\foo
- Another method to remove variable is to use Remove-Variable cmdlet and its alias rv
  - $var = "Some Variable"
  - $var
  - Remove-Variable -Name var
  - $var
- Also can use alias 'rv'
  - rv var

Comparison Operators

- PowerShell comparison operators are comprised of a leading dash (-) followed by a name (eq for equal, gt for greater than, etc...)
- Names can be preceded by special characters to modify the behavior of the operator:
  - i - Case-Insensitive Explicit (-ieq)
  - c - Case-Sensitive Explicit (-ceq)
- Case-Insensitive is the default if not specified, ("a" -eq "A") same as ("a" -ieq "A").

Simple comparison operators

- Equal to (==): 2 -eq 2
- Not equal to (!=): 2 -ne 4
- Greater-than (>): 5 -gt 2
- Greater-than or equal to (>=): 5 -ge 5
- Less-than (<): 5 -lt 10
- Less-than or equal to (<=): 5 -le 5

String comparison operators

- "MyString" -like "*String"
- "MyString" -notlike "Other*"
- "MyString" -match '^String$'
- "MyString" -notmatch '^Other$'

Collection comparison operators

- "abc", "def" -contains "def"
- "abc", "def" -notcontains "123"
- "def" -in "abc", "def"
- "123" -notin "abc", "def"

Arithmetic Operators

- Addition 1 + 2
- Subtraction 3 - 2
- Set negative value -1
- Multiplication 1* 2
- Division 4 / 2
- Modulus 1 % 2
- Bitwise Shift-left 100 -shl 2
- Bitwise Shift-right 100 -shr 1

Assignment Operators

- Assignment. Sets the value of a variable to the specified value
  - `$var = 1`
- Addition. Increases the value of a variable by the specified value
  - `$var += 2`
- Subtraction. Decreases the value of a variable by the specified value
  - `$var -= 1`
- Multiplication. Multiplies the value of a variable by the specified value
  - `$var *= 2`
- Division. Divides the value of a variable by the specified value
  - `$var /= 2`
- Modulus. Divides the value of a variable by the specified value and then assigns the remainder (modulus) to the variable
  - `$var %= 2`
- Increment and decrement:
  - `$var++`
  - `$var--`

Redirection Operators

Success output stream:

- Send success output to file, overwriting existing content
  - cmdlet > file
- Send success output to file, appending to existing content
  - cmdlet >> file
- Send success and error output to error stream
  - cmdlet 1>&2

Error output stream

- Send error output to file, overwriting existing content
  - cmdlet 2> file
- Send error output to file, appending to existing content
  - cmdlet 2>> file
- Send success and error output to success output stream
  - cmdlet 2>&1

Warning output stream: (PowerShell 3.0+)

- Send warning output to file, overwriting existing content
    - cmdlet 3> file
- Send warning output to file, appending to existing content
    - cmdlet 3>> file
- Send success and warning output to success output stream
    - cmdlet 3>&1

All output streams:

- Send all output streams to file, overwriting existing content
  - `cmdlet *> file`
- Send all output streams to file, appending to existing content
  - `cmdlet *>> file`
- Send all output streams to success output stream
  - `cmdlet *>&1`

Mixing operand types

- The type of the left operand dictates the behavior
- Gives "42": "4" + 2
- Gives 6: 4 + "2"
- Gives 1,2,3,"Hello": 1,2,3 + "Hello"
- "Hello1 2 3": "Hello" + 1,2,3

For Multiplication

- Gives "33": "3" * 2
- Gives 6: 2 * "3"
- Gives 1,2,3,1,2,3: 1,2,3 * 2
- Gives an error op_Multiply is missing: 2 * 1,2,3

String Manipulation Operators

- Returns: The hail in Seattle
  - "The rain in Seattle" -replace 'rain','hail'
- The -split operator splits a string into an array of sub-strings. Returns an array string collection object containing A,B and C.
  - "A B C" -split " "
- The -join operator joins an array of strings into a single string. Returns a single string: E:F:G
  - "E","F","G" -join ":"

Creating Objects

- The New-Object cmdlet is used to create an object.
- Create a DateTime object and stores the object in variable "$var"
  - `$var = New-Object System.DateTime`
- Calling constructor with parameters
  - `$sr = New-Object System.IO.StreamReader -ArgumentList "file path"`
- In many instances, a new object will be created in order to export data or pass it to another commandlet. This can be done like so:

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

…Creating Objects

- There are many ways of creating an object.
- The following method is probably the shortest and fastest way to create a PSCustomObject:

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

…Creating Objects

- In case you already have an object, but you only need one or two extra properties, you can simply add that property by using Select-Object:

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

- All objects can be stored in variables or passed into the pipeline. You could also add these objects to a collection and then show the results at the end.
- Collections of objects work well with Export-CSV (and Import-CSV).
  - Each line of the CSV is an object, each column a property.
-

Get-Member

- Get-Member helps you discover what objects, properties, and methods are available for commands.
- Any command that produces object based output can be piped to Get-Member. A property is a characteristic about an item.
- Properties
  - Get-Service -Name w32time
  - Get-Service -Name w32time | Get-Member
- Once you know what type of object a command produces, you'll be able to use this information to find commands which accept that type of object as input.
- Get-Command -ParameterType ServiceController

EOF