Powershell

Working With Data
COMP2101 Fall 2017

Where-Object

- Collections of objects contain objects we are interested in and often also contain objects we are not interested in
- Where-Object is designed to act as a filter in a pipeline
- Objects passed to where-object are subjected to a test expression
- Matching objects are passed along in the pipe, non-matching ones have their handles discarded
- where is an alias for where-object
- ? is an alias for where-object

Where-Object Examples

- get-process | where-object processname -eq powershell
- get-process | where cpu -gt 10
- get-process | ? starttime -gt (get-date).addhours(-24)

Sort-Object

- Used to sort objects passing through a pipeline
- Default sort is defined by the object
- You can specify which properties to sort on
- Ascending is the default, you can specify -Descending
- You can specify -Unique, it eliminates duplicates based only on the sort property or properties
- sort is an alias for sort-object

Sort-Object Examples

- get-process
- get-process | sort-object
- get-process | sort-object cpu

- get-wmiobject -class win32_process |
  sort parentprocessid, processname |
  ft -autosize processid, parentprocessid, processname

- "red","green","blue","yellow" | sort

- "red","green","blue","yellow" | sort length

- (get-date),(get-date).adddays(-3),(get-date).addhours(-1) | sort

Export Cmdlets

- exporting provides a way of saving object properties in various formats in a file
- export-csv saves object properties in csv format suitable for later import to spreadsheets or similar programs
- export-clixml saves object properties in xml format suitable for later import to programs that understand xml like browsers or other powershell scripts
- import verb allows creating in-memory objects from those files

Export Verb Exercises

- get-process > procs.txt
  get-process | export-csv procs.csv
  get-process | export-clixml procs.xml

- Compare the contents and sizes of these three files of process object data

- import-csv procs.csv | format-table * -autosize
  import-csv procs.csv | get-member

- import-clixml procs.xml | get-member

Convert Cmdlets

- convertto verb allows conversion of object properties into csv, json, or html strings

- convertfrom verb allows creating in-memory objects from csv or json strings

Convert Cmdlets Examples

- get-date | convertto-csv
- get-date | convertto-json
- get-date | select * | convertto-json
- get-date | convertto-html
- get-date | convertto-csv | convertfrom-csv | get-member
- get-member | converto-json | convertfrom-json | get-member
- get-date | select * | converto-json | convertfrom-json | get-member

Selecting Objects

- Collections of objects may contain many objects, only some of which might be of interest to us
- The default property list presented by an object isn't always suitable, it may be too limited
- Objects passing through a pipeline can be rather large and piping collections of them can cause a great deal of memory and cpu usage, it can be helpful to trim them
- Some commands allow us to filter their output collections using parameters on their command line (e.g. get-process processname)

Select-Object Uses

- select-object can be used to extract the first or last objects in a collection, unique objects in a collection, or a specified number or set of objects in a collection
- select-object does not modify the objects when extracting objects
- select-object can be used to extract properties from objects and create new trivial objects containing only those properties
- select-object does not preserve methods or data typing from the original objects when it creates trivial objects, everything becomes a NoteProperty (think string)

Select-Object Exercises

- get-date | select-object year,month,day | format-table -autosize
- gwmi -class win32_processor | select -property name, numberofcores
- get-process | sort cpu | select -last 5
- get-process | select processname -unique
- "red","green","blue","yellow" | select -index 1,3
- Try get-member on each of these to see what is actually produced by these commands

Data Types

- Properties can be simple data types (int, bool, double, etc.) or they can be more complex objects (or collections of objects)
- There are hundreds of data types
- Simple data types include bool, int, long, single, double, char, string
- Putting quotes around text on the command line creates a string object
- Entering a number without quotes creates a numeric data type, units can be suffixed to numbers (e.g. 5kb, 11mb, 27gb, 6.3tb)
- Data types for created objects can be specified using casting (e.g. [double]12), causes data type conversion to be done if possible
- Collections are created using commas to specify objects, or the range operator .. (e.g. 1,2,3,4,5) (e.g. 1..5)

Data Types Examples

- 1kb
  26GB
  10mb
  8.9tb

- "12" | get-member
  ("12").gettype()

- [int]"12" | get-member
  ([int]"12").gettype()

- [bool]12
  [bool]0
  [bool]"12"
  [bool]"0"
  [bool]""

- [string]16
  [string]016

- [int]1.6gb
  [int]40gb
  [long]40gb

- [int]"red"
  [char]16
  [char]"16"

- [bool](get-date)
  [bool](cd /flooble)

- 6.7 -as [int]
  106 -as [char]
  123456789 / 1mb
  123456789 / 1mb -as [int]

Operators

- Operators provide ways to combine objects for various purposes
- Assignment operators (e.g. `=`, `+=`, `*=`, etc.) are used to assign values to variables
- Arithmetic operators (e.g. `+`, `-`, `*`, `/`, `%`) can be used to perform calculations on appropriate data types
- Comparison operators (e.g. `-eq`, `-ne`, `-lt`, `-gt`, `-band`, `-bor`, etc.) can be used to compare values and test conditions
- Logical operators (e.g. `-and`, `-or`, `!`, etc.) connect conditional expressions to create more complex expressions
- There are more types of operators, see `help about_operators` for details and lists of operator symbols

Operator Examples

- 6 + 9
  6 - 9
  6 * 9
  6 / 9
  6 % 9

- 6 -and 9
  6 -or 9
  6 -xor 9

- 6 -band 9
  6 -bor 9
  6 -bxor 9

- "red " + "green"
  "red " * 3

- 1..3
  1..3 + 4
  1..3 * 4

- 5 -lt 3
  "red" -eq "green"
  ! $?

- 1..3 | select {$_ * 5} | fl
  get-process|? cpu -gt 10

- gwmi -class win32_process|? {$_.getowner().user -eq "dsimpson"}|select processname

- gwmi -class win32_networkadapterconfiguration -filter ip_enabled=true |
  ? { $_.dnsdomain -ne $null -or $_.dnshostname -ne $null -or $_.dnsserversearchorder -ne $null } |
  select description, dnsserversearchorder, dnsdomain, dnshostname

Variables

- Variables are named storage for object handles
- Objects exist as long as at least one other thing in the system has a handle for them
- When the last handle is lost, so is the object
- Powershell displays objects produced by cmdlets, then discards the handles, releasing those objects
- To keep an object around, assign its handle to a variable (e.g. $mystring = "some string")

Variable Identification

- Variables always have a $ symbol preceding their name
- Variable names can contain numbers, letters, underscore and space (bad idea), use { } to clearly indicate what a variable name includes (e.g ${my variable number_3})
- Assigning to a variable is done using the assignment operators (e.g. $a = 3) (e.g. $d = get-date) (e.g. $b += 4)
- Setting a variable to only hold a particular data type is done using casting (e.g. [int]$mynumber = 37)

Variable Usage

- Dot notation can be used on objects referred to by variables (e.g. $d = get-date ; "Today is " + $d.dayofweek) to access properties and methods
- Parentheses can used to define order of execution of complex statements (e.g. $a = $b * ($g + 7))
- Variables can be used on a command line more or less wherever you might use an object

Variable Examples

- $a = 8 ; $a
  $b = 7 ; $b
  $c = $a + $b ; $c
  $a += 5 ; $a
  $b++ ; $b
  $c-- ; $c

- $d = get-date ; $d
  $d2 = $d.adddays(3.5) ; $d2
  $d3 = $d2.subtract($d) ; $d3

- $drives = gwmi -class win32_logicaldisk
  $filesystems = $drives | where-object size -gt 0

Variables and Types

- A variable is automatically created with a type suitable for what you store in it when you create it
- You can define the type of a variable and powershell will try to convert data you assign to that variable into the variable type

```
$a=5 ; $a.gettype()
$b="red"; $b.gettype()
$a="red"; $a.gettype()
$a=37.5; $a.gettype()
[int]$a=76
$a=8gb
[long]$a=20gb; $a
$a=6.5; $a
```

```
[double]$a=6.5; $a
$d=get-date; $d
$d=5; $d
[datetime]$d=5pb; $d
[bool]$f=0; $f
$f=1; $f
$f="no"; $f
$f=$true; $f
```

Array and Hash Variables

- Powershell supports arrays and hashes using objects as the indices
- The index can be a single object or a collection
- The index is indicated by putting it inside of [ ] (e.g. $myarray[2] $myarray["a","b","c"])
- Creating an array can be done by assigning a collection of objects to a variable (.e.g $mynums = 5..8) or running a cmdlet that creates a collection of objects (e.g. gwmi -class win32_printer)
- Indices will be numbers starting at zero (e.g. $mynums[0] would be 5, $mynums[1] would be 6, etc.), negative indices count backwards from the end of the array

Hash Variables

- Creating a hash can be done by assigning values to names inside @{ } (eg. $h = @{Number = 3 ; Shape = "Round" ; Colour = "Cyan"} ; $h)

- Elements can be added to a hash using normal assignment (e.g. $h = @{ } ; $h["blue"] = "lobster" ; $h["red"] = "cardinal" ; $h)

- Properties created by a select-object can be defined using hash syntax

- Hash notation is used to create hash elements having a name (or key) and a value

- Some cmdlets allow using hash notation when creating output (e.g. format-table, format-list, select-object, etc.)

Array and Hash Examples

- Generating a network interface report using custom objects

- ```
  $adapters = gwmi -Class win32_networkadapter
  $adapters
  $adapters[1,3,4]
  ```

- ```
  $filteredadapters = $adapters |
                      Where-Object adaptertype -Match ethernet
  $filteredadapters
  ```

- ```
  $filteredadapters | Select-Object Name,
                                    MACAddress,
  @{n="Speed(Mb)";e={$_.Speed/1000000 -as [int]}},
                                    Netenabled,
                       PowerManagementSupported |
                      Format-Table -AutoSize
  ```
  ```

Formatting Using Hash Syntax

- Multiple object properties can be created using hash syntax where it is useful
- A disk space report similar to the UNIX df

```
gwmi -class win32_logicaldisk |
where-object size -gt 0 |
format-table -autosize DeviceID,
    @{n="Size(GB)"; e={$_.size/1gb -as [int]}},
    @{n="Free(GB)"; e={$_.freespace/1gb -as [int]}},
    @{n="% Free"; e={100*$_.freespace/$_.size -as [int]}},
    ProviderName
```

Special Variables

- $_ is the current object in a pipeline
- $? contains the exit status of the last operation
- $Args contains an array of the undeclared command line parameters
- $Error contains the most recent errors
- $True, $False contain true and false for comparison and assignment purposes

Automatic Variables

- $MyInvocation contains information about the current script
- $profile contains the name of the profile file used for the current powershell invocation
- $home holds the path to the user's home directory
- $pwd contains the current directory path