

Arrays

"a","b","c" = array of strings
 1,2,3 = array of integers
 @() = empty array
 @(2) = array of 1 element
 1,(2,3),4 = array within array
 ,"hi" = Array of one element
 \$a[5] = 6th element of array*
 \$a[2][3] = 4th or 3rd element of array
 \$a[2..20] = Return elements 3 thru 21

OPERATIONS

Does this array have a 3 in it
 1,2,3,5,3,2 -contains 3
 Return all elements equal to 3:
 1,2,3,5,3,2 -eq 3
 Return all elements less than 3:
 1,2,3,5,3,2 -lt 3
 Other operators: -gt, -le, -ge, -ne

ASSOCIATIVE ARRAYS (HASHTABLES)

\$hash = @{} -- Create empty hashtable
 \$h = @{"foo=1;bar=2"} -- Create and initialize a hashtable
 \$hash.key1 = 1 -- Assign 1 to key "key1"
 \$hash.key1 -- Returns value of key1
 \$hash["key1"] -- Returns value of key1

Strings

CONSTANTS: "this is a string, this \$variable is expanded as is \$(2+2)"

'this is a string, this \$variable is not expanded' @"

This is a "here string" which can contain anything including carriage returns and quotes. Expressions \$(2+2) are evaluated
 "@
 @'

"here string" with single quotes do not evaluate expressions. '@

+ = Concatenate two strings
 * = Repeat a string some number of times
 -f = Format a string (.NET format specifiers)
 -replace = Replace operator ("abcd" -replace "bc", "TEST" gives aTESTd)
 -match = Regular expression match
 -like = Wildcard matching

ESCAPE SEQUENCES: Backwards apostrophe '. e.g. `o = (null), `a = (alert), `b = (backspace), `f = (form feed), `n = (new line), `r = (carriage return), `t = (tab), `v = (vertical quote)

Scripting

BREAK: exits a loop. Optional LABEL to break to

Example: while (1){\$a = something
 if (\$a -eq 1) break;}

CONTINUE: continues the next iteration of a loop without breaking out of it.

Example: while (1){ \$a = something
 if (\$a -eq 1) (continue)
 # This line is not reached unless \$a == 1
 # This line is never reached.

FOR: [:label] for ([initializer]; [condition]; [iterator]) {}

Example: for (\$i = 0; \$i -lt 5; \$i++) {Write-Object \$i}

FOREACH: [:label]

foreach (identifier in collection) {}

Expression | foreach {}

Expression | foreach {BEGIN{} PROCESS{} END{}}

Examples: \$i = 1,2,3
 foreach (\$z in \$i) {Write-Object \$z}
 Get-Process |foreach {BEGIN{\$x=1}
 PROCESS{\$x++}
 END{"\$X Processes"}}

RETURN: exits current script | function and returns a value. **Example:** function foo {return 1}

UNTIL: do
 {...} until (condition)

TYPE OPERATIONS: -is IS type (e.g. \$a -is [int])
 -as convert to type e.g. 1 -as [string] treats 1 as string

BLOCKS: Commands and expressions can be stored in a script block object and executed later.

Example: \$block = {Get-Process; \$a=1}
 &\$block

SWITCH: The variable \$_ is available in the script. \$_ represents current value being evaluated. If array is used in switch, each element of array is tested.

Example: \$var = "word1","word2","word3"
 switch -regex (\$var) {
 "word1" {"Multi-match Exact " + \$_}
 "word2" {"Multi-match Exact " + \$_}
 "w.*2" {"Pattern match Exact " + \$_}
 default {"Multi-match Default " + \$_}}

Output: Multi-match Exact word1
 Multi-match Exact word2
 Pattern match Exact word2
 Multi-match Default word3

FILTERS: shorthand of writing function with PROCESS script block. **e.g.** filter MyFilter {\$_ .name}

If/elseif/else if (condition) {...}
 elseif (condition) {...} else {...} (On the command line, the closing brace must be on the same line as elseif and else. This restriction does not apply to scripts)

FUNCTIONS: function MyFunction { write-object

\$args[0]
 function test ([string]\$label="default
 label",[int]\$start=0)
 { BEGIN {\$x=\$start} PROCESS {"\$label: \$_"; \$x++}
 END{"\$x total"}}

WHILE: [:label] while (condition)

{ ... }
 do
 { ... } while (condition)

SCOPES: Variables and other data elements may be instantiated in different scopes:

- **global** scope visible scopes.
- **script** scope visible all scopes within script file.
- **local** scope visible in the current scope its children.
- **private** scope visible only to that current scope.

A scope is created in the body of a shell function

Example: \$global:a = 4, \$script:a = 5

VARIABLES: \${scope:]name or \${anyname} or \${any path}. **Examples:** \$a = 1

\$(!@%\$%^&*())=3
 \$global:a = 1 # Visible everywhere
 \$local:a = 1 # defined in this scope and visible to children
 \$private:a=1 # same as local but invisible to child scopes
 \$script:a=1 # visible to everything in this script
 \$env:path = "d:\windows"
 \${C:\TEMP\testfile.txt}="This writes to a file"
 Get-Variable -scope 1 a #Gets value from parent scope
 Get-Variable -scope 2 a # grandparent

DOT-SOURCING: Dot sourcing allows running functions, script blocks, and scripts in the current scope rather than a local one. **Example:** . MyFunction
 If MyFunction sets a variable, it is set in the current scope rather than the function's local scope.

\$a = {\$x = Get-Process | Select -First 2}
 . \$a #Evaluates the script block in the current scope

THROW: provides the same functionality for scripts as the ThrowTerminatingError API does for cmdlets.

throw "Danger, Danger"
 Danger, Danger
 At line:1 char:6
 + throw <<<< "Danger, Danger"

Throw takes a string, exception, or ErrorRecord

Well Known Variables (Not Exhaustive)

\$\$ = Last token of previous command line
 \$? = Boolean status of last command
 \$^ = First token of previous command line
 \$_ = Current pipeline object
 \$Args = Arguments to script or function
 \$Error = Array of errors from previous commands
 \$Foreach = Reference to enumerator in a foreach loop
 \$Home = The user's home directory;
 \$Host = Reference to application hosting the POWERSHELL
 \$Input = Enumerator of objects piped to a script
 \$LastExitCode = Exit code of last program or script
 \$Matches = Hash table of matches found with the -match operator
 \$PSHome = The installation location of Windows PowerShell
 \$profile = The standard profile (may not be present)
 \$StackTrace = Last exception caught by Windows PowerShell
 \$Switch = Enumerator in a switch statement

Operators

INVOKE: The & operator can be used to invoke a script block or the name of a command or function.

Example: \$a = "Get-Process", &\$a

ASSIGNMENT: =, +=, -=, *=, /=, %=
LOGICAL: !, -not, -and, -or

COMPARISON: -eq, -ne, -gt -ge, -lt -le

COMMAND EXPANSION: \$() = Returns null

\$(1,2,3) = Returns an array containing 1,2,3.

\$(Get-Alias a*) = Returns evaluation of the expression

@(Get-Alias;Get-Process) = Executes the two commands and returns the results in an array

STATIC: :: (Double colon)

OBJECT PROPERTIES: An object's properties can be referenced directly with the "." operator. e.g. \$a = Get-Date, \$a.Date

METHOD CALLS: Methods can be called on objects. Examples:

\$a = "This is a string", \$a.ToUpper()

PRECEDENCE: () {}, @ \$, !, [], ., &, ++ --, Unary + -, * / %, Binary + -, Comparison Operators, -and -or, |, > >>, =

Traps

```
Trap [ExceptionType] {
    if (...)
    { continue
    # continue at the script statement after the one that cased the
    # trap; $? is updated but no error record is generated
    } else (...)
    { Break
    # rethrow the exception
    }
    # Doing nothing will do what is specified in the
    # $ErrorActionPreference setting
}
```