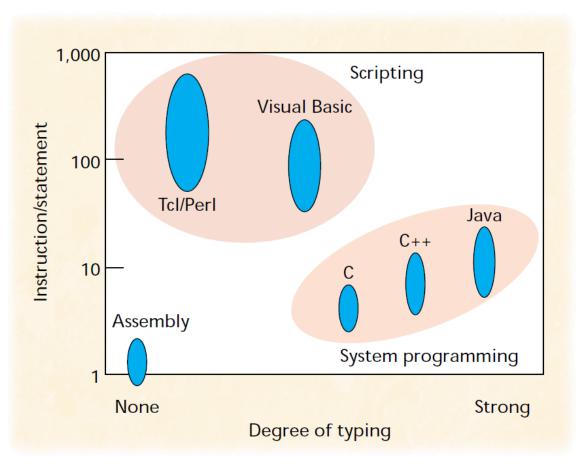
PowerShell Tutorial

Eigil Obrestad and Erik Hjelmås August 18, 2015



(OUSTERHOUT, J., "Scripting: Higher-Level Programming for the 21st Century", IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30.)

From Ousterhout, 1998:

While programming languages like C/C++ are designed for low-level construction of data structures and algorithms, scripting languages are designed for high-level "gluing" of existing components. Components are created with low-level languages and glued together with scripting languages.

WARNING!

The following presentation is NOT meant to be a comprehensive/complete tour of the PowerShell language.

The purpose is to get you started with some basic program constructions which you will recognize based on some-sort-of-programming-background.

At the end of the presentation (Credits section) you will find pointers to more comprehensive material (reference material).

Practice

You need a Windows host running on a physical or virtual machine with working access to the internet, and with PowerShell v2.0 installed.

Log in and open a terminal window, download the examples as we go along from http://www.ansatt.hig.no/erikh/tutorial-powershell/FILENAME

(or download all at once with filename powershell-examples.zip but remember to unblock before unzip)

We assume that you are using PowerShell 2.0 (as shipped with Windows 7 and Windows Server 2008R2) and have installed the PowerShell Community Extensions from http://pscx.codeplex.com/ and the GnuWin32 utilities http://sourceforge.net/projects/getgnuwin32/files/ (where you will find wget etc).

To allow for execution of scripts in powershell you need to set the correct execution policy:

```
# check what is current policy
Get-ExecutionPolicy
# change to only require signature on remote scripts
Set-ExecutionPolicy RemoteSigned
# you probably need to "run as administrator" to do this
```

To install PowerShell Community Extensions

```
# download Pscx-2.x.x.zip using a webbrowser
# windows explorer and browse to where it is
# right click on Pscx-2.x.x.zip, choose properties
# click unblock, ok
# right click, extract all to $PSHOME\Modules dir
# $PSHOME is probably
# C:\Windows\System32\Windows\PowerShell\v1.0
Import-Module Pscx
# place this command in $profile so it is run every time
# you start PowerShell, or do it globally with
# "run as administrator" and
New-Item $pshome\profile.ps1 -type file
notepad $pshome\profile.ps1
```

To install GnuWin32

```
# Run setup program from
# http://sourceforge.net/projects/getgnuwin32/files/
# cd to the directory where it was downloaded
download.bat # answer yes to a couple of questions
# run powershell as administrator
```

```
install.bat 'C:\Program files\GnuWin32'
notepad $pshome\profile.ps1
# add the following to include the gnuwin32 tools in PATH
# $env:path += ";C:/Program Files/GnuWin32/bin"
```

Hello World

```
# hello.ps1
Write-Host "hello world!"
```

execute as long as filename ends with .ps1:

```
.\hello.ps1
or direct from command line cmd (DOSPROMPT)
powershell -command "Write-Host \"hello world!\""
or direct from command line powershell
```

```
Write-Host "hello world!"
```

PowerShell commands are called *cmdlets* (pronounced "commandlets") and have the syntax verb-noun, e.g. Write-Host. Fortunately most of the cmdlets have aliases corresponding to the commands you might know from DOS (cmd.exe) or Unix/Linux. In addition there is also a short PowerShell alias to most cmdlets. To find the cmdlet to a command you know from before you can use the cmdlet Get-Alias:

To get help with the cmdlets, use the cmdlet Get-Help, e.g. Get-Help Write-Host | more. A nice feature is that you can view the help page in your browser (on the internet) by adding the parameter -online, e.g. Get-Help Write-Host -online.

Note that you can use TAB-completion on both commands and parameters.

1 Variables

Single Variables

All variables are prefixed with \$

We need to use `between \$firstname and? to avoid? being "part of" the variable name.

A single variable (sometimes called a *scalar*) is typed, but PowerShell chooses the type automatically for us by "guessing". Typing can be forced by prefixing the variable with e.g. [int]. What is important to know is that variables are instances of .NET objects, and these objects are also what is being passed through the pipe of piped commands (as opposed to just piping byte streams in other shells).

PowerShell uses namespaces, e.g. you can write \$fullname or \$variable:fullname. You can list all current variables with Get-Variable \$variable:*

Scope for a variable can be defined with Set-Variable -Scope. PowerShell can also *dot-source* script files to make a script's variables accessible from the command line.

```
. single-var.ps1  # dot-source it
$firstname.GetType()  # what kind of object is it?
$firstname | Get-Member # Which methods and properties are available?
```

PowerShell in itself, like much of Windows, is case-insensitive, however it preserves case when used.

Btw, `is the protection character (and line continuation character) in PowerShell (same as \ in bash). PowerShell does this differently from Unix/Linux scripts since \ (in addition to /) is used as a directory separator on Windows, see also

```
Get-Help about_escape_characters
```

Exercise

```
$name="Mysil"
```

Use the properties and methods of this object to

- ⇒ find out how many characters the string contains
- \Rightarrow print the string in upper case

Single and Double Quotes

```
# quotes.ps1

$name="Mysil"
Write-Host Hello $name
Write-Host "Hello $name"
Write-Host 'Hello $name'
```

Variables are expanded/interpolated inside double quotes, but not inside single quotes.

1.1 Arrays

Arrays

One-dimensional arrays:

```
# array.ps1

$os=@("linux", "windows")
$os+=@("mac")
Write-Host $os[1]  # print windows
Write-Host $os  # print array values
Write-Host $os.Count # length of array
```

Arrays are created with @(...)

Note how we display the length of the array by viewing a property (Count) of the object. Btw, Count is just a reference to the Length property

```
. ./array.ps1
$os.PSExtended | Get-Member
```

If you want to access an array element within an interpolated string, you have to place the array element in parentheses like this:

Write-Host "My operating system is \$(\$os[1])"

Associative Arrays

```
# assoc-array.ps1

$user=Q{
        "frodeh" = "Frode Haug";
        "ivarm" = "Ivar Moe"
      }

$user+=Q{"lailas"="Laila Skiaker"}

Write-Host $user["ivarm"] # print Ivar Moe
Write-Host Quser # print array values
Write-Host $user.Keys # print array keys
Write-Host $user.Count # print length of array
```

Associative arrays are created with $Q\{\ldots\}$ and are called Hashtables in PowerShell.

1.2 Structures/Classes

Structures/Classes

A simple object used as a struct:

Of course, since PowerShell is based on the object-oriented framework .NET, creating and manipulating objects is a world by it self, there are a plethora of ways of doing these things.

See what kind of object this is by running the commands on the command line and doing

```
$myhost
$myhost.GetType()
$myhost | Get-Member
```

Note also that we don't need the line continuation character $\dot{}$ when inside a block ($\{...\}$).

1.3 Command-line args

Command-Line Arguments

All command-line arguments in the array \$args

Scriptname retrieved from the object \$MyInvocation

2 INPUT 8

\$MyInvocation is one of PowerShell's builtin variables. Again, check what kind of object this is with

```
$MyInvocation.GetType()
$MyInvocation | Get-Member
# or check what a typical PowerShell command returns
Get-Process | Get-Member
(Get-Process).GetType()
# contrast this with a traditional cmd command
ipconfig | Get-Member
(ipconfig).GetType()
```

For all special variables in PowerShell, a good resource is http://www.neolisk.com/techblog/powershell-specialcharactersandtokens

Exercise

Rewrite the previous script to only have one string (just one set of double quotes (")), one at the beginning and one at the end, do not use single quotes either

2 Input

2.1 Input

Input From User

```
# input-user.ps1

$something=Read-Host "Say something here"
Write-Host "you said" $something
```

Input From the Pipeline

```
# input-pipe.ps1

$something="$input"
Write-Host "you said" $something
```

can be executed as

```
Write-Output "hey hey!" | .\input-pipe.ps1
```

\$input (another one of PowerShell's builtin variables) is a special variable which enumerates the incoming objects in the pipeline.

3 CONDITIONS 9

Input From Files

```
# input-file.ps1

$file=Get-Content hello.ps1
Write-Host @file -Separator "`n"
```

You can assign the entire output of a command directly to a variable.

2.2 System commands

Input from System Commands

Using \$(expr) inside a string will treat it as an *ad-hoc variable* evaluating the expression expr and inserting the output into the string.

3 Conditions

3.1 if/else

if/else

3.2 Operators

Comparison

3 CONDITIONS 10

Operator	Meaning
-lt	Less than
-gt	Greater than
-le	Less than or equal to
-ge	Greater than or equal to
-eq	Equal to
-ne	Not equal to

Note that many other test operators (e.g. file tests) are used as methods in the objects instead of separate operators.

Boolean

Operator	Meaning
-not	Not
!	Not
-and	And
-or	Or

There are not separate comparison operators for numbers and strings. Be careful when comparing objects with different types. Behaviour might be a bit strange (see page 209 of "Mastering PowerShell" by Weltner):

```
$ 123 -lt "123.4"
   False
$ 123 -lt "123.5"
   True
```

A set of *file test operators* is not available since this functionality is covered through cmdlets (e.g. Test-Path) and methods (e.g. PSIsContainer).

3 CONDITIONS 11

Boolean example

```
# if-bool.ps1

if ((1 -eq 2) -and (1 -eq 1) -or (1 -eq 1)) {
    Write-Host "And has precedence"
} else {
    Write-Host "Or has precedence"
}

# force OR precedence:

if ((1 -eq 2) -and ((1 -eq 1) -or (1 -eq 1))) {
    Write-Host "And has precedence"
} else {
    Write-Host "Or has precedence"
}
```

AND is always (as known from mathematics courses) evaluated before OR (binds more tightly). Write it down in logic (truth table) if you are unsure.

3.3 Switch/case

Switch/Case

```
# switch.ps1

$short = @{ yes="y"; nope="n" }
$ans = Read-Host
switch ($ans) {
   yes { Write-Host "yes" }
   nope { Write-Host "nope"; break }
   {$short.ContainsKey("$ans")} `
        { Write-Host $short[$ans] }
   default {Write-Host "$ans`???"}
}
```

Run example and see the difference between inputting yes, nope and nei.

In the example above {\$short.ContainsKey("\$ans")} checks if the content of \$ans has an entry (matches a key) in the associative array \$short. Switch in PowerShell continues testing each case unless it reads a break.

3.4 Where

Where/Where-Object

```
# where.ps1
Get-ChildItem | Where-Object {$_.Length -gt 1KB}
```

4 ITERATION 12

\$_ represents the current object in the pipeline.

In a pipeline we use Where-Object and ForEach-Object, but when processing a collection/array in a script we would use Where and ForEach (in other words: without the -object).

We can use KB, MB and GB and PowerShell understands what we mean.

Exercise

Use Get-Process and Where-Object to

- \Rightarrow list all powershell processes
- \Rightarrow store the process table in an array \$procs
- \Rightarrow list all processes with a working set greater than 10MB

4 Iteration

4.1 For

For loop

```
# for.ps1
for ($i=1;$i-le3;$i++) {
    Write-Host "$i"
}

# something more useful:

$file=Get-ChildItem
for ($i=0;$i-lt$file.Count;$i++) {
    if (!(Get-Item $file[$i]).PSIsContainer) {
        Write-Host $file[$i].Name "is a file"
    } else {
        Write-Host $file[$i].Name "is a directory"
    }
}
```

Normally you would use ForEach instead of for since you can simplify the first loop above like this:

```
ForEach ($i in 1..3) {
   Write-Host "$i"
}
```

4 ITERATION 13

4.2 While

While

```
# while.ps1
while ($i -le 3) {
    Write-Host $i
    $i++
}

# something more useful:

$file=Get-ChildItem
$i=0
while ($i -lt $file.Count) {
    if (!(Get-Item $file[$i]).PSIsContainer) {
        Write-Host $file[$i].Name "is a file"
    } else {
        Write-Host $file[$i].Name "is a directory"
    }
    $i++
}
```

The for example converted to while.

4.3 Foreach

Foreach loop

```
# foreach.ps1
foreach ($i in Get-ChildItem) {
    Write-Host $i.Name
}

# with associative arrays

$user=@{
    "frodeh" = "Frode Haug";
    "monicas" = "Monica Strand";
    "ivarm" = "Ivar Moe"
    }

foreach ($key in $user.Keys) {
    Write-Host $user[$key]
}
```

In a pipeline we would use ForEach-Object.

ForEach

If we want to read from the pipeline and do stuff object by object:

5 MATH 14

```
# foreach-pipe.ps1

foreach ($i in $input) {
    $foo += @($i)
}
Write-Host "size of foo is" $foo.Count
```

or

```
# foreach-object-pipe.ps1

$input | ForEach-Object {
    $foo += @($_)
}
Write-Host "size of foo is" $foo.Count
```

```
$ Get-ChildItem | ./foreach-object-pipe.ps1
size of foo is 20
```

\$input represents the pipeline and \$_ the current object in the pipeline.

5 Math

Operators

Operator	Meaning
+	Add
_	Subtract
*	Multiply
/	Divide
%	Modulus

```
# math.ps1
Write-Host "3+5 is" (3+5)
```

```
Write-Host "3+5 is" 3+5
Write-Host "3+5 is" (3+5)
Write-Host "3+5 is" $(3+5)
Write-Host "3+5 is (3+5)"
Write-Host "3+5 is $(3+5)"
```

6 FUNCTIONS 15

6 Functions

Functions

```
# func.ps1

# declare:
function add($a, $b) {
    Write-Host "$a+$b is" ($a+$b)
}
# use:
add 5.12 2.56
```

7 RegExp

Regular expressions intro 1/5

Special/Meta-characters:

```
\ | ( ) [ ] { } ^ $ * + ? .
```

These have to be protected with \, e.g. http://www\.hig\.no

To match c:\temp, you need to use the regex c:\\temp. As a string in C++ source code, this regex becomes "c:\\\\temp". Four backslashes to match a single one indeed.

(from http://www.regular-expressions.info/characters.html):

There are many different regular expression engines, which differs mostly in features and speed. In this tutorial we will try to stick with simple examples which will the same in most engines (perl, pcre, extended posix, .NET, ...).

Regular expressions intro 2/5

Describing characters:

Operator	Meaning
	Any single character
[abcd]	One of these characters
[^abcd]	Any one but these characters
[a-zA-Z0-9]	A character in these ranges

7 REGEXP 16

Regular expressions intro 3/5

Grouping:

Operator	Meaning
()	Group
1	OR

Anchoring:

Operator	Meaning
^	Beginning of line
\$	End of line

Regular expressions intro 4/5

Repetition operators/Modifiers/Quantifiers:

Operator	Meaning
?	0 or 1 time
*	0 or more times
+	1 or more times
{N}	N times
{N,}	At least N times
{N,M}	At least N but not more than M

Demo: four step example with

cat a.html | ForEach-Object {if(\$_ -match REGEXP)` {Write-Host \$matches[0]}}

Regular expressions intro 5/5

Finding URLs in HTML: (mailto|http)://[^"]*

Each line should be an email address: ^[A-Za-z0-9._-]+@[A-Za-z0-9.-]+\$

Remember that regexp engines are most often greedy, they try to match as much as possible, so using e.g. .* might match more than you were planning for.

7.1 PowerShell example

PowerShell example

8 POWERSHELL ONLY

When we use regular expressions inside scripts, it is very useful to be able to extract parts of the match. We can do this by specifying the part with (part) and refer to it later using \$matches[1], \$matches[2], etc. \$matches[0] matches the entire expression.

http://www.regular-expressions.info/powershell.html

8 PowerShell only

Advanced stuff

See the complete Mastering PowerShell book at http://powershell.com/cs/blogs/ebook/ for much more of what you can do with PowerShell

9 Credits

Credits

http://refcardz.dzone.com/refcardz/windows-powershell http://powershell.com/cs/blogs/ebook/ http://technet.microsoft.

com/en-us/library/ee692948.aspx http://www.techotopia.com/index.php/Windows_PowerShell_1.0_String_Quoting_and_Escape_

Sequences http://dmitrysotnikov.wordpress.com/2008/11/26/input-gotchas/ http://stackoverflow.com/questions/59819/how-do-i-create-a-custom-ty
http://www.powershellpro.com/powershell-tutorial-introduction/ http://en.wikipedia.org/wiki/Windows_PowerShell http:
//www.johndcook.com/powershell.html http://www.regular-expressions.info/ OUSTERHOUT, J., "Scripting: Higher-Level Programming for the 21st Century", IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30.)