

Powershell

Basierend auf dem "Powershell Tutorial": <http://www.powershellpro.com/powershell-tutorial-introduction/> von Jesse Hamrick.

(Note to self: qemu: Exit VM: Ctrl-Alt-G, Exit Fullscreen: Ctrl-Alt-F)

Ausführungsumgebung

Man kann diese Beispiele entweder in der Powershell oder im ise (Integrated Scripting Environment) ausführen.

Grundsätzliche Syntax

- Output sind Objekte und nicht Text (im Gegensatz zu Unix Shells!)
- Kommandos sind *CmdLets*

```
Get-Command
Get-Command -Verb Get # alle Kommandos mit "Get" Verb anzeigen
                        # andere z.B. Add, Clear, New, and Set
Get-Command -Type CmdLet | sort-object noun | format-table -group noun
```

Command ist typischerweise ein Nomen!

einfaches I/O

```
> echo "Bitte gib was ein"
> $ret = Read-Host
> Write-Host "Danke: $ret" # gleich wie echo
```

Mehrzeilige Kommandos

```
> echo `
>> "foo"
>>
foo
>
```

Docu

```
Get-Help Kommando # oder
man Kommando
man -full Kommando
man -full Kommando | format-list # zeigt Infos in Listen Form an
man -full Kommando | format-list | more
help about_* | more
```

Pipes

Auf der Tastatur gibt's zwei 'Pipe' Zeichen, die gleich aussehen (können). Eines davon ist falsch. Eines ist richtig. Hinweis: es ist nicht dasjenige, das man im Unix verwendet...

History

- Kommandozeilen Log: (Fn-)F7

Dienste

- Get-Command -Noun Service
- Get-Service
- Start-Service ...

Alias

- Get-Alias
- Set-Alias gs Get-Service

Windows Alias sind nur in der jeweiligen Shell Session gültig. Um sie zu exportieren:

- Export-Alias -Path MyAliases.txt

Profile Anlegen

Ein Profile ist ein Script, das beim Starten der PowerShell ausgeführt wird. Der Pfad des aktuellen Profils ist in \$Profile gespeichert und kann angezeigt werden mittels:

- Profile

Erstellen eines Profils:

- Set-ExecutionPolicy Unrestricted # sonst werden keine Start Scripte ausgeführt
- Test-Path \$Profile # wenn False:
- New-Item -Path \$Profile -ItemType File -Force
- Notepad \$Profile

Gemeinsame Parameter von Kommandos

Nicht *alle* CmdLets haben diese...

```
-WhatIf          # zeigt an, was passieren *würde*, wenn man
                  # das Kommando in echt ausgeführt hätte
-Confirm         # zeigt Prompt an
-Verbose         # detaillierte Ausgabe
-Debug           # debugging Infos
-ErrorAction     # welche Aktion soll im Fehlerfall ausgeführt
                  # werden (continue, stop, silently continue
                  # oder inquire)
-ErrorVariable   # Variable, welche Fehler-Info enthalten soll
                  # (normalerweise $error)
-OutVariable     # Variable, welche Ausgabe enthalten soll
```

```
-OutBuffer      # Für Zwischenspeicherung der Ausgabe in  
                # einer Pipe
```

Objekte

- Get-Service | Get-Member
- Get-Service | Get-Member -MemberType Method
- Get-ChildItem -Path C:\ -Recurse | Where-Object {\$_.LastWriteTime -gt "2015-04-18"}
 - Where-Object filtert Objekte heraus, bei denen die Bedingung zutrifft
 - siehe:
 - Get-ChildItem | Get-Member

Ausgabeformat

Ohne genauere Angabe übernimmt PowerShell die Formatierung der Ausgaben eines Kommandos.

Genauer kann man dies mit Format-* einstellen:

- Get-Command Format-*

```
Format-Custom  
Format-List  
Format-Table  
Format-Wide
```

- Get-ChildItem -Path C:\ | Format-Table -AutoSize
- Get-ChildItem -Path C:\ | Format-List -Property FullName,LastWriteTime
- Get-ChildItem -Path C:\ | ConvertTo-HTML | Out-File Procs.html && Invoke-Item Procs.html
- Get-ChildItem -Path C:\ | Export-CSV Procs.csv && Invoke-Item Procs.csv

Ausgabe sortieren und gruppieren

- Get-Process | Group-Object Company | Sort-Object Count -Descending

Datei-Manipulation

Cmdlet	Command/Alias
Get-Location	pwd
Set-Location	cd
Copy-Item	cp
Remove-Item	rm
Move-Item	mv
Rename-Item	ren

New-Item	ni
Clear-Item	cli
Set-Item	si
Mkdir	
Get-Content	cat
Set-Content	sc

Provider

Machen Datenquellen, z.B. die Registry, in Form von Dateisystemen verfügbar, was deren Manipulation mittels Datei-Operationen ermöglicht.

Die Provider können Standard Optionen erweitern, welche spezifisch für die bearbeiteten Daten sind.

Provider werden auch Snap-Ins (DLLs) genannt.

Siehe auch [The PowerShell Software Developers Kit](#) für Anleitung zum selber machen.

- Get-PSProvider

```
Alias
Environment
FileSystem
Function
Registry
Variable
Certificate
```

Wo sind die entsprechenden Dateisysteme verfügbar?

- Get-PSDrive

```
Name      Provider      Root
----      -
Alias      Alias
C          FileSystem     C:\
cert       Certificate    \
D          FileSystem     D:\
Env        Environment
Function   Function
HKCU       Registry       HKEY_CURRENT_USER
HKLM       Registry       HKEY_LOCAL_MACHINE
Variable   Variable
X          FileSystem     X:\
```

und wie kommt man da rein?

- Set-Location Alias:
- Get-ChildItem | Get-Member # Eigenschaften der Alias anzeigen -> sie haben einen Namen
- Get-ChildItem -Name R* # alle Aliase deren Namen mit 'R' anfangen anzeigen

oder alternativ:

- Get-ChildItem | Where-Object {\$_.Name -like "R*" }

Arbeiten mit der Registry

```
> Get-PSDrive
Name          Provider      Root
-----          -
...
HKLM           Registry      HKEY_LOCAL_MACHINE

> cd HKLM:
> cd Software\Microsoft\ .NetFramework\Policy\Upgrades
> Get-ItemProperty .
...
> New-Item ...
```

Arbeiten mit Variablen

```
> cd Variable:
> ls
...
PSHOME          C:\...
..
> echo $PSHOME
C:\...
> Get-Content PSHOME
C:\...
```

```
> $foo = "hallo"
> echo $foo
> $foo
> $bar = "welt"
> $foobar = $foo + " " + $bar
> echo "ich sage $foobar"
> echo 'ich sage $foobar'
```

Da wir es in der PowerShell mit Objekten zu tun haben:

```
> $foobar = $foobar -replace "welt", "fridolin"
```

Spezielle Variablen

<code>\$_</code>	jetziges Pipeline Objekt
<code>\$Args</code>	Argument an jetzige Methode
<code>\$Error</code>	letztes Fehlerobjekt
<code>\$Home</code>	Heimverzeichnis des aktuellen Benutzers
<code>\$PSHome</code>	Heimverzeichnis der PowerShell

Alle Spezialvariablen:

- `Get-Help about_automatic_variables`

Variablen Typen

[int]	32-bit
[long]	64-bit
[string]	Unicode...
[char]	"
[byte]	8-bit char
[bool]	
[decimal]	128-bit float
[single]	32-bit float
[double]	64-bit float
[xml]	
[array]	
[hashtable]	

- [int]\$zahl = 3

Operatoren

=, +, -, *, /, %, +=, -=, ..., ++, --

Klammern für Sub-Ausdrücke können verwendet werden

- \$foo = 1 + (2 / 3)

Arrays

```
> $sack = @( 1, 2, 3 )
> $sack
1
2
3
> $sack[0]
1
> $sack.Count
3
> $tasche = $sack
> $sack[0] = 77
> $tasche[0]
77
> $sack + $sack
77
2
3
77
```

```
2  
3
```

Wenn man eine mehrzeilige Text Datei einliest, dann wird diese automatisch als Array ausgegeben.

```
> $arrComputers = get-Content -Path "meine_computer_liste.txt"
```

Schlaufen

```
> foreach($i in $sack) { echo $i }
```

Ebenfalls:

- while () {}
- do {} while ()
- do {} until ()
- for (init; cond; incr) {}
- foreach (\$i in \$collection) {}

In den Schleifenkonstrukten können die Anweisungen 'break' und 'continue' verwendet werden.

Hash Tables

```
> $hash = @{"Name" = "Tomaso"; "Alter" = 42 }  
> $hash["Lieblingsfarbe"] = "goldig"  
> $hash.Remove("Alter")  
> $hash.Clear()           # alle Einträge löschen
```

Vergleiche

```
-eq, -lt, -gt, -ge, -le, -ne  
-not, !, -and, -or  
> "Tom" -eq "TOM"  
True  
> "Tom" -ieq "TOM"  
True  
> "Tom" -ceq "TOM"  
False
```

Logische Operatoren

```
-not, !, -and, -or
```

if Anweisung

```
> if(1) { echo "True" } elseif(0) { echo "False" } else { echo "Fallback" }
```

switch Anweisung

```
> switch ($foo + $bar){  
    ($baz + $buz) { echo "Hm, ja, gleich wie bazbuz" }  
    "Hallo Welt"   { echo "wie erwartet" }  
    default        { echo "dann halt nicht" }  
}
```

Funktionen

```
> Function Zeit { Get-Date }  
> Zeit  
...  
> Function Addiere($a,$b) { echo ($a + $b) }  
> Addiere 1 2  
3
```

Alternativ:

```
> Function Addiere2 { param ($a,$b); echo ($a + $b) }  
> Addiere2 1 2  
3
```

Oder:

```
> Function Anzeigen { echo "Die übergebenen Argumente sind: '$args'" }  
> Anzeige Foo 1 2 3  
Die übergebenen Argumente sind: 'Foo 1 2 3'
```

Die einzelnen Argumente sind via `$args[$i]` erreichbar.

Per default errät PowerShell den Typ der Argumente, dieser kann aber auch explizit deklariert werden:

```
> Function Addiere([int]$a, [int]$b) { echo ($a + $b) }
```

Funktionen können mit der Spezial-Variable `$input` arbeiten, welche den *vollständigen* Inhalt der aktuellen Pipeline enthalten.

Skripte aufrufen

Um Skripte aus Skripten aufzurufen, kann man folgende Notation verwenden:

```
.{./mein_anderes_Skript.ps1}  
# das folgende Skript wird im Standard Suchpfad, sprich in $PSHome gesucht  
.{foo_Skript.ps1}
```

Filter

Im Gegensatz zu Funktionen arbeiten *Filter* mit der Variable `$_`, welche als Stream, d.h. während der Produktion der Daten, verarbeitet werden kann.

Ausgabe Umleitung

```
> ls > list.txt
> ls | OutFile -FilePath list.txt # ist das gleiche
> ls >> list.txt
> ls | OutFile -FilePath list.txt -append # dito
```

WMI / Windows Management Instrumentation

```
> $printers = Get-WmiObject -Class win32_Printer -namespace "root\CIMV2" `
    -computerName $ComputerName
> echo $printers[0].Name
>
> Get-WmiObject -List -Namespace "root\CIMV2"
```

Die WMI Administrative Tools von Microsoft enthalten das "WMI CIM Studio", mittels welchem man die WMI Informationen in einem GUI durchforsten kann.

```
> $NICs = Get-WmiObject Win32_NetworkAdapterConfiguration `|
>>     Where {$_.IPEnabled -eq "TRUE"}
>
> foreach($NIC in $NICs) {`
>>     $NIC.EnableDHCP() `
>> }
```

Um alle Methoden von 'Win32_NetworkAdapterConfiguration' anzuzeigen:

```
> Get-WmiObject Win32_NetworkAdapterConfiguration `|
>>     Get-Member -MemberType Methods | Format-List
```

Ggf. TODO

- <http://www.powershellpro.com/> * PowerShell Scripting with WMI Part 2 * Managing Active Directory with Windows PowerShell