# PowerShell Kurz-Referenz

## Argumente

Die Argumente zu einer Funktion sind in der $args Variablen. Man kann auf diese Argumente in einer ähnlichen Schleife durchlaufen wie diese:

foreach ($i in $args) {$i}

Der Zugriff auf jedes einzelne Argument erfolgt über den Index, beginnend bei 0.

$args[0]

Das letzte Element wir über den Index -1 zugegriffen.

$args[-1]

## Bunter Text

Um eine Textausgabe bunt zu gestalten, kann man die Vordergrundfarbe im Kommando **Write-Host** ändern.

**Write-Host** "test" *-foregroundcolor* "green"  
**Write-Host** "test" *-backgroundcolor* "red"

## Zeilenumbruch

Das Zeichen `n bewirkt einen Zeilenumbruch.

**Write-Host** "Zeile 1.`nZeile 2."

## Inverse Schrift

Das Kommando **Write-Warning** schreibt eine Nachricht invers.

**Write-Warning** "Es ist ein Fehler aufgetreten."

## Kommentare

Eine Raute am Beginn eines Textes ist ein Kommentar.

# Das ist ein Kommentar und wird nicht ausgeführt.

<# Das ist ein  
mehrzeiliger Kommentar #>

## Texteingabe von der Konsole

Für das Einlesen von Benutzereingaben wird **Read-Host** verwendet.

$a = **Read-Host** "Bitte einen Namen eingeben"

## Umfalten einer Skriptzeile

Mit dem ‚Backtick‘ kann man eine Zeile in der nächsten Zeile fortsetzen.

**Write-Host** **`**

"Das ist eine Fortsetzung der Zeile"

Auch beim Verkettungsoperator kann eine Zeile umgebrochen werden (sofern das Kommando eine Verkettung verwendet).

**Get-ChildItem** C:\Scripts |

**Sort-Object** Length *–Descending*

## Mehrere Kommandos in einer Zeile

Mehrere Kommandos in einer Zeile trennt man mit einem Strichpunkt.

$a = 1,2,3,4,5; $b = $a[2]; **Write-Host** $b

## Zuweisungen

Weist einer Variablen einen oder mehrere Werte zu.

= += -= \*= /= %= ++ --

## Bitwise Operators

-band Bitwise AND

-bor Bitwise OR (inclusive)

-bxor Bitwise OR (exclusive)

-bnot Bitwise NOT

-shl, -shr Bitwise shift operators. Bit shift left, bit shift right (arithmetic for signed, logical for unsigned values)

## Andere Operatoren

Split Einen String in ein Array von Strings aufteilen  
“abcdefghi” -split “de”

-join Mehrere Strings verbinden  
“abc”,”def”,”ghi” -join “;”

.. Range operator1..10 | foreach {$\_ \* 5}

-is, -isnot Prüft, ob ein Objekt mit einem gegebenen framework-Objekt übnereinstimmt.  
42 –is [int]

-as Typen-Umwandlung. Versucht, ein Objekt in den angegebenen Framework-Typ umzuwandeln.   
$a = 42 –as [String]

-f Formatiert String und Benutzt dabei die Formatier-Methoden für String-Objekte  
1..10 | **foreach** { "{0:N2}" -f $\_ }

[ ] Cast operator. Konvertiert Objekte in den eingeklammerten Typ  
[datetime]$birthday = "1/10/66"

, Konstruiert ein Array

. Führt ein Skript im aktuellen Geltungsbereich aus.   
.c:\scripts\sample.ps1

$( ) Subexpression operator

@( ) Array subexpression operator

& Aufruf-Operator. Damit kann man Kommandos aufrufen, die als String-Variable gespeichert sind.

$a = "Get-Process"

& $a

$sb = { **Get-Process** | **Select** *–First* 2 }

& $sb

## Logische Operatoren

-and -or -xor -not ! erlauben die Verbindung mehrerer Bedingungen.

## Redirection Operators

>, >> Sendet Ausgaben zu verschiedenen Ausgabe-Streams (success, error, warning, verbose, und debug) sowie zu Dateien.

\* Alle Ausgaben

1 Success

2 Fehler

3 Warnhinweis

4 detaillierte Ausgabe

5 Diagnose-Ausgabe

#Warnhinweis in Datei warning.txt  
Do-Something 3> warning.txt

# Textausgaben an die Datei verbose.txt anhängen  
Do-Something 4>> verbose.txt

# Etwas in den Ausgaben-Stream schreiben   
Do-Something 5>&1

# Alle Streams in die Datei out.txt umlenken  
Do-Something \*> out.txt

## Vergleiche

Kommandos, die Vergleiche enthalten (z.B. **Where-Object**) benutzen besondere Vergleichoperatoren, die mit einem Bindestrich eingeleitet werden.

Ein c unmittelbar nach dem Bindestrich macht den Vergleich von der Schreibweise (groß-klein) abhängig. Beispielsweise ist -ceq die groß-klein-Variante von -eq.

-lt kleiner als (less than)

-le kleiner oder gleich (less than or equal to)

-gt größer als (greater than)

-ge größer oder gleich (greater than or equal to)

-eq gleich (equal to)

-ne ungleich (not equal to)

-like -notlike wie (like), erlaubt Wildcards

-match -nomatch Findet einen regex-Ausdruck

-contains -notcontains Prüft, ob ein Array einen Variablenwert enthält  
1,2,3,4,5 -contains 3-

-in -notin Prüft, ob eine Variable exakt einer de Referenzvariablen entspricht  
“Windows“ –in “Windows“, “PowerShell“

## Arrays

Der Anfangsindex aller Array ist 0

"a", "b", "c" Array von Strings

1,2,3 Array ganzer Zahlen

@() Leeres Array

@(2) Array mit einem Element

1,(2,3),4 Array innerhalb eines Array

,"hi" Array mit einem Element

$arr[5] Sechstes Element in einem Array

$arr[2..20] 3. bis 21. Element

$arr[-1] Letztes Element

$arr[-3..-1] Die letzten drei Elemente

$arr[1,4+6..9] 2., 5.,7.-9. Element

@(Get-Process) Erzwingt die Ausgabe in einem Array

$arr=1..10  
$arr[($arr.length-1)..0] Kehrt ein Array um

$arr[1] += 200 Zählt 200 zum 2. Element

$b = $arr[0,1 + 3..6] Erzeugt ein neues Array mit den Elementen 1, 2 und 4-7

$z = $arr + $b Kombiniert zwei Array zu einem einzigen Array

## Assoziative Arrays

$hash = @{} Erzeugt eine leere Hash-Table

@{foo=1; bar='value2'} Erzeugt und initialisiert eine Has Table

[ordered]@{a=1; b=2; c=3} Erzeugt ein sortiertes Vokabelheft

$hash.key1 = 1 Zuweisung 1 zum Schlüssel key1

$hash.key1 Wert des Schlüssels key1

$hash["key1"] Wert des Schlüssels key1

$hash.GetEnumerator | sort Key Sortiert eine Has-Table nach der Eigenschaft Key

[pscustomobject]@{x=1; y=2} Erzeugt ein Objekt

## Lesen einer Textdatei

Das Kommando **Get-Content** liest den Text aus einer Datei in eine Variable.

$a = **Get-Content** C:\Scripts\Test.txt

Jede Zeile ist ein Element im Array $a. Jede Zeile kann über einen Index erreicht werden. Erste Zeile im Array $a:

$a[0]

Letzte Zeile im Array $a

$a[-1]

Die Anzahl der Zeilen, Wörter und Zeichen in einer Textdatei bestimmen.

**Get-Content** c:\scripts\test.txt |   
**Measure-Object** *-line* *-word* *-character*

## Schreiben einer Textdatei

Um eine Variable in eine Textdatei zu schreiben, benutzt man das **Out-File**–Kommando.

**Get-Process** | **Out-File** C:\Scripts\Test.txt

Um einen Text an eine bestehende Datei anzuhängen, muss der Parameter *–append* verwendet werden.

**Get-Process** | **Out-File** C:\Test.txt *–append*

Man kann auch die aus DOS bekannten Parameter zur Umlenkung verwenden (> schreiben, >> anhängen).

**Get-Process** > C:\Scripts\Test.txt

Eine andere Option erlaubt die Speicherung als CSV-Datei (Comma Separated Value).

**Get-Process** | **Export-CSV** C:\Test.csv

## Ausdruck

Die Ausgabe an den Drucker erfolgt mit dem Kommando **Out-Printer**:

**Get-Process** | **Out-Printer**

## Bedingte Anweisungen

Eine if -Anweisung schaut etwa so aus:

$a = "weiß"

if ($a -eq "red")

{" Die Farbe ist rot."}

elseif ($a -eq "weiß")

{" Die Farbe ist weiß."}

else

{" Die Farbe ist blau."}

Mit switch vereinfacht sich eine Mehrfachentscheidung:

$a = 2

switch ($a)

{

1 {"Die Farbe ist rot."}

2 {"Die Farbe ist blau."}

3 {"Die Farbe ist grün."}

4 {"Die Farbe ist gelb."}

default {"Es ist eine andere Farbe."}

}

## For- und For Each-Schleifen

Um Vorgänge zu wiederholen, benutzt man for oder foreach-Schleifen.

for ($a = 1; $a -le 10; $a++) {$a}

foreach ($i in **get-childitem** c:\scripts) {$i.extension}

## Do-Loop-Schleifen

Schleifen mit Abfrage am Ende:

$a = 1  
do {$a; $a++}   
while ($a -lt 10)  
  
$a = 1  
do {$a; $a++}   
until ($a –gt 10)

## COM-Objekt erzeugen

Die folgenden Zeilen öffnen Excel und schalten es ein.

$a = **New-Object** *-comobject* **`** "Excel.Application"  
$a.Visible = $True

## .NET-Objekt erzeugen

Eine Instanz des DotNet-Objekts system.Net.DNS erzeugen und die gewünschte Methode resolve aufrufen.

[system.Net.DNS]::resolve("207.46.198.30")

Eine Objekt-Referenz eines DotNet-Framework-Objekts erzeugen.

$a = **new-object** **`***-type* system.diagnostics.eventlog **`***-argumentlist* system

## Eigenschaften auswählen

Um mit bestimmten Eigenschaften einer Sammlung zu arbeiten oder diese anzuzeigen, leitet man das Objekt an das Kommando **Select-Object** weiter:

**Get-Process** | **Select-Object** Name, Company

## Daten sortieren

Sortieren nach der Eigenschaft ID:

**Get-Process** | **Sort-Object** ID

Änderung der Sortierrichtung

**Get-Process** | **Sort-Object** ID *–descending*

Nach mehreren Eigenschaften sortieren

**Get-Process** | **Sort-Object** ProcessName, ID

## WMI

Ein WMI-Objekt öffnen, um Informationen über den Computer zu bekommen

**Get-WMIObject** Win32\_BIOS

Wenn die gewünschte Klasse nicht im Namensraum cimv2 ist, muss der Parameter *-namespace* verwendet werden:

**Get-WMIObject** SystemRestore **`** *-namespace* root\default

Wenn es sich um einen anderen Computer handelt, muss der Parameter *–computername* verwendet werden:

**Get-WMIObject** Win32\_BIOS **`** *–computername* atl-ws-01

Mit dem Parameter *-query* kann die Ausgabe eingeschränkt werden.

**Get-WMIObject** *-query* **`** "Select \* From Win32\_Service `  
 Where State = 'Stopped'"

## Active Directory

Um das Programm an einen Active Directory Account zu binden, muss der LDAP-Provder benutzt werden.

$a = [adsi] "LDAP://cn=kenmyer, `  
 ou=Finance, dc=fabrikam, dc=com"

Alle Objekte in einer OU aufzulisten, ist etwas komplizierter. Man bindet das Programm an die OU und benutzt dann die Methode **PSBase\_GetChildren()**.

$objOU = [ADSI]**`**"LDAP://ou=Finance,dc=fabrikam,dc=com"  
$users = $objOU.PSBase.Get\_Children()  
$users | **Select-Object** displayName

## Lokale User

Um das Programm an einen lokalen User zu binden, benutzt man den WinNT provider:

$a = [adsi] "WinNT://atl-ws-01/kenmyer"  
$a.FullName

## Hilfe

Für jedes Kommando kann man mit **Get-Help** und dem Parameter *–full* eine ausführliche Information erhalten.

**Get-Help** Get-Process *–full*

Anzeige von Beispielen

**Get-Help** Get-Process *–examples*

Alle Kommandos erfährt man mit **Get-Command**.

**Get-Command**

Alle Alias-Namen erfährt man mit dem Kommendo **Get-Alias**:

**Get-Alias**

Lädt die neuesten Help-Dateien

**Update-Help**

Listet die importierten Module.

**Get-Module**

## Sicherheits-Einstellungen

Um Skripts aus der PowerShell ausführen zu können, müssen die Sicherheitseinstellungen geändert werden, weil PowerShell in der Anfangseinstellung nur signierte Skripts ausführt. Um auch lokal verfasste Skripts ausführen zu können, die signiert oder auch unsigniert sein können, benutzt man folgendes Kommando:

**Set-ExecutionPolicy** RemoteSigned

## Strings

**"Das ist ein String, diese $variable wird als Wert dargestellt, ebenso $(2+2)"**

**‘Das ist ein String, diese $variable wird nicht als Wertdargestellt‘**

**@"**

**Dieser String kann alles enthalten, auch Zeilenumbrüche**

**Und Anführungszeichen. Ausdrücke werden ausgewertet:**

**$(2+2\*5).**

**Der Endmarker des Strings muss wieder auf einer neuen Zeile sein**

**"@**

**@'**

**Mit einem einfaches Anführungszeichen wird ein Ausdruck nicht ausgewertet:**

**$(2+2\*5)**

**'@**

## Variable

Format: **$[scope:]name** Oder **${anyname}** Oder **${any path}**

**$path = "C:\Windows\System32"**

**Get-ChildItem ${env:ProgramFiles(x86)}**

**$processes = Get-Process**

**$global:a =**1 # überall sichtbar

**$local:a = 1 # sichtbar für Kinds-Objekte**

**$private:a = 1 # unsichtbar für Kinds-Objekte**

**$script:a = 1 # sichtbar in diesem Skript**

**# Using scope indicates a local variable in remote commands**

**and with Start-Job**

**$localVar = Read-Host "Bitte Verzeichnis eingeben"**

**Invoke-Command -ComputerName localhost -ScriptBlock {**

**dir $using:localVar }**

**Start-Job { dir $using:localVar -Recurse}**

**$env:Path += ";D:\Scripts"**

**Get-Command -Noun Variable # the Variable Cmdlets**

**Get-ChildItem var: # Listet alle Werte der Varaiablen var**

**# Streng typisierte Variablen (nur ganzzahlige Variablen erlaubt)**

**[int]$number=8**

**# Attribute für Variablen**

**[ValidateRange(1,10)][int]$number = 1**

**$number = 11 #liefert einen Fehler**

**# Variablen tauschen**

**$a=1;$b=2**

**$a,$b = $b,$a**

**# Mehrfach-Zuweisungen**

**$a,$b,$c = 0**

**$a,$b,$c = 'a','b','c'**

**$a,$b,$c = 'a b c'.split()**

**# Veriable ohne Schreibzugriff (read only) (können mit –Force überschrieben werden)**

**Set-Variable -Name ReadOnlyVar -Value 3 –Option ReadOnly**

**# Konstanten (können nicht überschrieben werden)**

**Set-Variable -Name Pi -Value 3.14 -Option Constant**

## Automatische Variablen

**$$** Letztes Token der vorigen Zeile

**$?** Boolescher Status des letzten Kommandos

**$^** Erstes Token der vorigen Zeile

**$\_, $PSItem** Aktuelles Pipeline-Objekt

**$Args** Argumente eines Skripts oder einer Funktion

**$Error** Fehler-Array vorige Kommandos

**$ForEach** Referenz zum Iterator der ForEach-Schleife

**$Home** Das Home-Verzeichnis des Benutzers

**$Host** Reference zur Anwendung, die PowerShell ausführt (zum Beispiel Power Gui Script Editor)

**$Input** Iterator der Objekte, die zum Skript weiter geleitet werden

**$LastExitCode** Exit-Kode des letzten Programms oder Skripts

**$Matches** Exit-Kode des letzten Programms oder Skripts

**$MyInvocation** Objekt mit Informationen über den aktuellen Befehl

**$PSHome** Der Speicherort von PowerShell

**$profile** Das Standard-Profil von PowerShell (muss nicht vorhanden sein)

**$Switch** Iterator in einer switch-Anweisung

**$True** Boolscher Wert für WAHR

**$False** Boolscher Wert für FALSCH

**$PSCulture** Aktuelle Kultur

**$PSUICulture** Aktuelle Kultur des UserInterface

**$PsVersionTable** Details zur verwendeten Version von PowerShell

**$Pwd** Der volle Pfad des aktuellen Verzeichnisses

## Variable für Voreinstellungen

**$ConfirmPreference** PowerShell fordert Bestätigung vor Ausführung eines Cmdlets oder einer Funktion

**$DebugPreference** Legt fest, wie PowerShell Determines sich beim Debuggen verhält

**$ErrorActionPreference** Legt fest, wie PowerShell auf einen nicht abbrechenden Fehler reagiert

**$FormatEnumerationLimit** Anzahl der angezeigte Werte

**$MaximumHistoryCount** Anzahl der gespeicherten Kommandos in der Kommando-History

**$OFS** Output Field Separator. Trennzeichen zum Verbinden von Array zu Strings. Anfangswert: Zwischenraum

**$OutputEncoding** Zeichenkodierung zum Senden von Text zu anderen Anwednungen

**$PSDefaultParameterValues** Anfangswerte der Parameter von Comandlets.

**$PSEmailServer** Der Mail-Server zum Senden von E-Mails

**$PSModuleAutoLoadingPreference** Aktiviert oder deaktiviert das automatische Laden von Modulen. Anfangswert: „All“

**$PSSessionApplicationName** Anwendungsname für ein Remote-Kommando das WS-Management technology benutzt

**$PSSessionConfigurationName** Session-Konfiguration für PowerShell-Sessions, die in der aktuellen Session angelegt warden.

**$PSSessionOption** Anlegen der Anfangswerte der “Advanced User Options” in einer Remote Session

**$VerbosePreference** Verhalten auf ausführliche Nachrichten von Skripts, Cmdlets und Provider.

**$WarningPreference** Verhalten auf Warnungen von Skripts, Cmdlets und Provider.

**$WhatIfPreference** Legt fest, ob WhatIf bei Befehlen, die das unterstützen, automatisch aktiviert ist

## Objekt untersuchen

Um Informationen über ein Objekt zu erhalten, legt man zuerst eine Instanz des Objekts an und leitet das Objekt an das Kommando **Get-Member** weiter. Beispielsweise erhält man mit dem folgenden Kommando alle Eigenschaften und Methoden der aktuellen Arbeitsumgebung.

**Get-Process** | **Get-Member**

Die Eigenschaften eines Objekts können mit dem Punkt-Operator bestimmt werden.

**$a = Get-Date**

**$a | Get-Member –MemberType Property**

**$a.Date**

**$a.TimeOfDay.Hours**

**$a | Get-Member -MemberType Property –Static**

Statische Eigenschaften, also Eigenschaften der Klasse können mit dem "::" Operator bestimmt werden.

**[DateTime]::Now**

Beispiele für das Aufrufen von Methoden eines Objekts

**$a = "This is a string"**

**$a | Get-Member –MemberType Method**

**$a.ToUpper()**

**$a.Substring(0,3)**

**$a | Get-Member -MemberType Method -Static**

Statische Methoden, also Methoden der Klasse können mit dem "::" Operator aufgerufen werden.

**[DateTime]::IsLeapYear(2012)**

## Konsolenfenster löschen

Um den Inhalt des Kommandofensters zu löschen, benutzt man das Kommando **Clear-Host** oder dessen Alias **cls**.

## Copy & Paste

Mit Windows-X-> PowerShell öffnen.

Das Eigenschaftsfenster durch einen Mausklick auf das Symbol links oben öffnen.

In der Dialog-Box **Options** den **QuickEdit Mode** auswählen und auf OK klicken.

Um einen Text zu kopieren, diesen Im Konsolenfenster markieren und auf Enter klicken. Um einen Text in der Zwischenablage einzufügen, auf die rechte Maustaste klicken.

## Skript ausführen

Um ein Skript auszuführen gibt man den vollen Pfad ein:

**C:\Scripts\Test.ps1**

Wenn der Pfad Spaces enthält, muss die Zeile mit Anführungszeichen eingeschlossen werden.

**&"C:\Scripts\My Scripts\test.ps1"**

Von außerhalb der Windows PowerSell, etwas aus einer DisolgBox cmd.exe, ruft man PowerShell auf und übergibt den Pfad als Parameter:

**powershell.exe –noexit C:\Scripts\Test.ps1**

Mit dem Parameter **-noexit** wird sichergestellt, dass dasa Fenster auch nach Ausführung desSkripts offen bleibt.

## Mehr Informationen

### Microsoft

*http://technet.microsoft.com/en-us/scriptcenter/dd742419.aspx.*[*http://technet.microsoft.com/de-de/library/cc196356.aspx*](http://technet.microsoft.com/de-de/library/cc196356.aspx)

*Microsoft Windows PowerShell*

*http://www.microsoft.com/powershell*

*Windows PowerShell Team Blog*

*http://blogs.msdn.com/PowerShell*

*MS TechNet Script Center*

*http://www.microsoft.com/technet/scriptcenter/hubs/msh.mspx*

*PowerShell Forum*

*http://social.technet.microsoft.com/Forums/en-US/winserverpowershell/*

*Hey, Scripting Guy! Blog*

*http://blogs.technet.com/b/heyscriptingguy/*

*Windows PowerShell Survival Guide*

[*http://social.technet.microsoft.com/wiki/contents/articles/183.windows-powershellsurvival-guide-en-us.aspx*](http://social.technet.microsoft.com/wiki/contents/articles/183.windows-powershellsurvival-guide-en-us.aspx)

### Community

#### PowerShell Community

*http://powershellcommunity.org*

#### PowerShell Code Repository

*http://poshcode.org*

#### PowerShell.com Community

*http://powershell.com*

#### PowerGUI.org Community

*http://powergui.org*

#### PowerShell Community Groups

*http://powershellgroup.org*

#### PowerShell Magazine

*http://powershellmagazine.com*

#### The PowerShell Community Toolbar

*http://powershell.ourtoolbar.com/*

*irc.freenode.net #PowerShell*

## Bücher (kostenlos)

#### Mastering PowerShell, Second Edition - Dr. Tobias Weltner

*http://powershell.com/cs/blogs/ebookv2/default.aspx*

#### Secrets of PowerShell Remoting - Don Jones and Dr. Tobias Weltner

*http://powershellbooks.com*

#### Administrator's Guide to Windows PowerShell Remoting Dr. Tobias Weltner, Aleksandar Nikolic, Richard Giles

*http://powershell.com/cs/media/p/4908.aspx*

#### Layman's Guide to PowerShell 2.0 Remoting - Ravikanth Chaganti

*http://www.ravichaganti.com/blog/?page\_id=1301*

#### WMI Query Language via PowerShell - Ravikanth Chaganti

*http://www.ravichaganti.com/blog/?page\_id=2134*

#### PowerShell 2.0 One Cmdlet at a Time - Jonathan Medd

*http://www.jonathanmedd.net/2010/09/powershell-2-0-one-cmdlet-at-a-time-availableas-pdf-download.html*

#### Effective Windows PowerShell - Keith Hill

*http://rkeithhill.wordpress.com/2009/03/08/effective-windows-powershell-the-freeebook/*