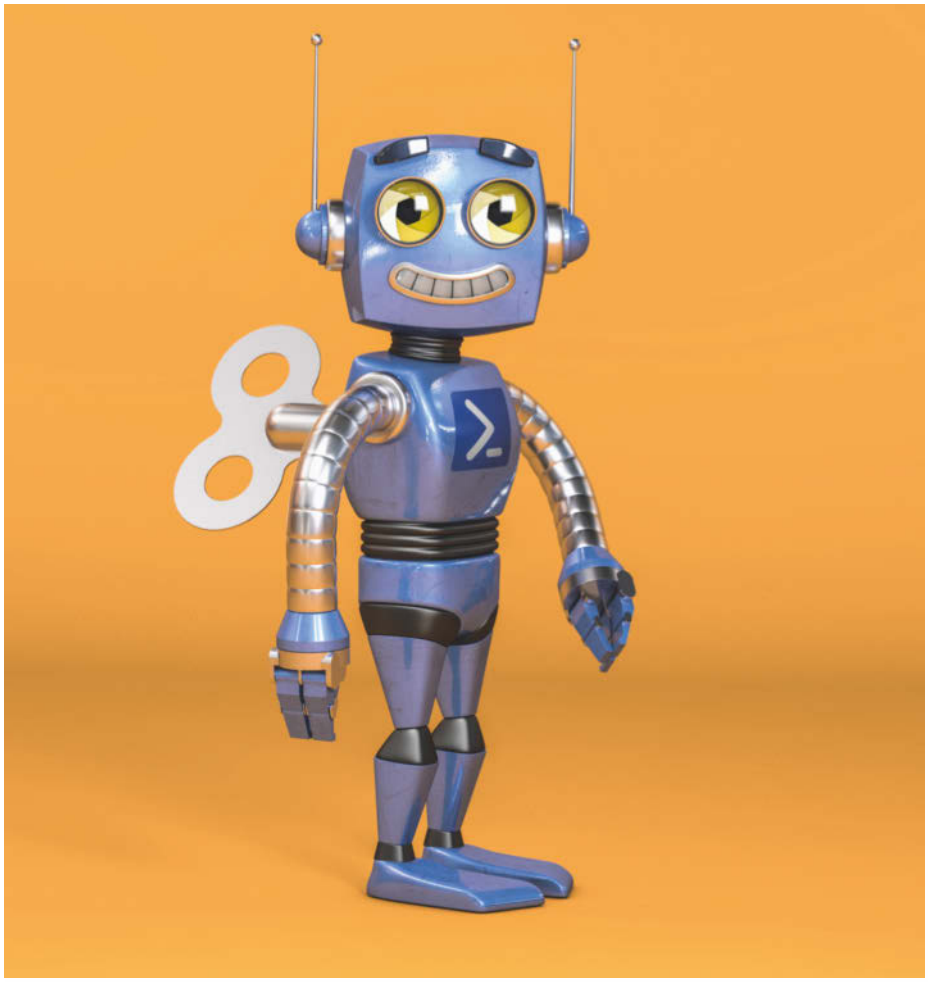


Power-Automatik

Loslegen mit der PowerShell, Teil 2

Schon als Textkonsole für den interaktiven Betrieb ist Microsofts PowerShell ein mächtiges Werkzeug. Ihre wahren Stärken entfaltet sie aber erst, wenn man sie als Basis für eigene Skripte verwendet. Deren Syntax steht der moderner Programmiersprachen in kaum etwas nach.

Von Hajo Schulz

Die Meinungen über die PowerShell gehen stark auseinander: Ihre Fans schätzen den großen Befehlsumfang und die Hilfen, die sie dem Benutzer bei der Eingabe von Befehlen gibt. Auf der anderen Seite stehen Anwender, die zwar gern in einer Konsole arbeiten, aber lieber bei der Eingabeaufforderung bleiben, weil sie deren Befehle schon kennen und ihnen die Syntax der meisten PowerShell-Kommandos viel zu geschwätzig ist. Weitgehende Einigkeit herrscht aber, wenn es darum geht, Befehlsabläufe mit Skripten zu automatisieren: Spätestens wenn die Ausgaben von Befehlen auseinanderzunehmen und in einer Schleife weiterzuverarbeiten sind, kann die antiquierte Batch-Syntax der Eingabeaufforderung nicht mit der Skriptsprache der PowerShell mithalten.

Die absoluten Basics zum Scripting mit der PowerShell haben wir bereits im ersten Teil dieses Artikels geklärt [1]. In Kurzform: PowerShell-Skripte sind Textdateien mit der Endung .ps1. Um sie abzuarbeiten, liest die PowerShell sie Zeile für Zeile ein und führt die enthaltenen Befehle aus, beinahe so, als hätte sie der Anwender gerade in der Konsole eingegeben.

Bevor die PowerShell überhaupt Skripte ausführt, müssen Sie mit Administratorrechten einmal den Befehl

```
Set-ExecutionPolicy RemoteSigned
```

ausführen, um die sogenannte Ausführungsrichtlinie zu lockern: Per Voreinstellung unterbindet Microsoft die Skript-Ausführung aus Sicherheitsgründen komplett; der Befehl erlaubt es der PowerShell, lokal erstellte oder vertrauenswürdig signierte Skripte abzuarbeiten.

Schleifen

Der Grund dafür, überhaupt ein Skript zu schreiben, besteht häufig darin, dass man einen Satz von Befehlen mehrfach ausführen will. Je nach Anwendungsfall sollen sich diese Befehle bei jedem Durchgang auf ein anderes Objekt beziehen, etwa eine andere Datei in einem bestimmten Ordner oder die nächste Zeile einer Textdatei.

Eine Programm- oder Skript-Struktur, die Befehle mehrfach nacheinander ausführt, nennen Programmierer eine Schleife. Die Skriptsprache der PowerShell stellt mehrere verschiedene Konstruktionen bereit, mit denen man solche Schleifen programmieren kann. Die einfachste ist die `while`-Schleife:

```
while(<Bedingung>)
{
  <Befehle>
}
```

Hier prüft die PowerShell zunächst die Bedingung, führt im Erfolgsfall die Befehle zwischen den geschweiften Klammern aus, prüft die Bedingung erneut und so weiter. Wenn die Bedingung nicht oder nicht mehr erfüllt ist, geht es hinter der schließenden geschweiften Klammer weiter. Ähnlich funktioniert die Konstruktion

```
do
{
  <Befehle>
} while(<Bedingung>)
```

Der Unterschied ist, dass die PowerShell die Befehle immer mindestens einmal ausführt und erst dann die Bedingung prüft. Schließlich führt

```
do
{
  <Befehle>
} until(<Bedingung>)
```

einen Satz von Befehlen genau so lange aus, *bis* die Bedingung erfüllt ist.

Eine beliebte und kompakte Schleifenkonstruktion ist die `for`-Schleife. Sie kommt häufig dann zum Einsatz, wenn eine bestimmte Anzahl von Wiederholungen gefragt ist:

```
for($i = 0; $i -lt 20; $i++)
{
  <Befehle>
}
```

Die Klammer hinter dem `for` enthält dabei drei durch Semikolons voneinander getrennte Elemente: eine Initialisierung, eine Bedingung, die vor jedem Schleifendurchlauf geprüft wird, und einen Befehl, der nach jedem Durchlauf ausgeführt wird. Jedes dieser Elemente darf leer sein, die Semikolons sind aber Pflicht. `$i++` ist eine Abkürzung für `$i = $i + 1`.

Schließlich kennt die PowerShell noch eine `foreach`-Schleife, die in etwa dasselbe tut wie das im ersten Teil erwähnte Cmdlet `ForEach-Object`, aber ohne die Pipeline auskommt und daher in Skripten etwas einfacher zu lesen ist. Sie sieht so aus:

```
foreach($element in $liste)
{
  TuWas-Mit $element
}
```

Den Namen der Laufvariablen (`$element`) können Sie frei wählen; die Liste kann statt aus einer Variablen auch direkt aus einem Cmdlet kommen.

Lesbarkeit

Erfahrungsgemäß entstehen die wenigsten PowerShell-Skripte von vornherein mit der Absicht, sie als Dauerlösung für irgendein größeres Problem zu verwenden. Deutlich häufiger passiert es, dass eine Aufgabe zunächst mit ein, zwei Befehlen zu lösen scheint, sich dann aber doch komplizierter als gedacht darstellt und ein Skript einfach dazu dient, Befehle nicht immer wieder neu eintippen zu müssen. Schnell wachsen Umfang und Komplexität des Skripts über ein Maß hinaus, das es gestattet, seine Funktion auf einen Blick zu erfassen. Außerdem tendieren solche Skripte dazu, eben doch zur dauerhaften Lösung für das anstehende Problem zu werden und viel länger zu leben, als das am Anfang absehbar war.

Wenn Sie so ein Skript nach ein paar Wochen oder Monaten wieder zur Hand nehmen, um ein Detail zu ändern oder einen Fehler zu korrigieren, werden Sie feststellen, dass es gar nicht so einfach ist, sich an die Überlegungen zu erinnern, die Sie bei seiner Entstehung angestellt haben. Es lohnt sich also, Skripte von vornherein les- und verstehbar zu gestalten. Dabei helfen verschiedene Techniken.

Den kleinsten Aufwand bedeutet es, sich anzugewöhnen, „sprechende“ Variablennamen zu verwenden, also beispielsweise nicht

```
$c = (dir *.tmp).Count
```

zu schreiben, sondern etwa

```
$anzahlDateien = (dir *.tmp).Count
```

Solange Sie nur für den eigenen Bedarf skripten, ist es Geschmackssache, ob Sie dabei lieber deutsche oder englische Bezeichner verwenden, bei öffentlichen Skripten ist Englisch der Standard. Wenn Sie zum Erstellen Ihrer Skripte einen auf PowerShell-Skripte spezialisierten Editor verwenden, nimmt Ihnen dessen automatische Ergänzung einen Großteil der mehr zu tippenden Zeichen ab. Brauchbare Beispiele sind das Integrated Scripting Environment (ISE) der Windows PowerShell oder der Microsoft-Editor Visual Studio Code mit PowerShell-Erweiterung.

Höchst empfehlenswert ist es auch, selbst kleine Skripte von Anfang an mit sinnvollen Kommentaren zu versehen. Dabei handelt es sich um Text, der in das Skript eingebettet, aber so markiert ist, dass die PowerShell ihn ignoriert. Kommentare dienen somit ausschließlich dazu, menschlichen Lesern – vor allem später auch Ihnen – bestimmte Details des Skriptes zu erläutern.

Die PowerShell kennt zwei verschiedene Formen von Kommentaren: Zum einen ignoriert sie alles, was hinter einem #-Zeichen auf derselben Zeile steht; wenn die Zeile nicht mit dem Kommentar beginnt, muss man davor mindestens ein Leerzeichen einfügen. Außerdem gilt alles, was von den Markierungen `<#` und `#>` eingeschlossen ist, als Kommentar. Solche Erläuterungen dürfen auch mehrere Zeilen umfassen. Wenn man sich beim Format seiner beschreibenden Texte an den Aufbau hält, den `Get-Help about_Comment_Based_Help` beschreibt, kann man sogar dafür sorgen, dass das Cmdlet `Get-Help` Informationen zu eigenen Skripten oder Funktionen anzeigt.

Das „Auskommentieren“ einzelner Zeilen oder auch größerer Code-Blöcke ist eine beliebte Methode, Teile eines Skripts vorläufig außer Betrieb zu setzen, ohne sie endgültig zu löschen. Benutzt wird das häufig zur Fehlersuche: Man streut an strategischen Stellen `Write-Host`-Aufrufe in seinen Code ein, um zu überprüfen, ob er erwartungsgemäß abläuft. Sind schließlich alle Fehler beseitigt, stellt man jedem dieser Aufrufe ein #-Zeichen voran: So stört die Debug-Ausgabe im laufenden Betrieb nicht mehr, ist aber schnell wiederhergestellt, wenn man den Code später doch noch mal ändern muss.

Die in diesem Artikel verwendete Formatierung von Code-Blöcken mit geschweiften Klammern auf eigenen Zeilen und Einrückungen ist zwar nicht Pflicht, vereinfacht es aber auch, Skripte zu lesen und ihre Struktur schnell zu erfassen.

Funktionen

Eine weitere Möglichkeit, Skripte zu strukturieren, besteht darin, logisch zusammengehörende Code-Passagen in selbst definierte Funktionen zu verpacken. Eine Funktion ist nichts anderes als ein Block Code, der einen Namen trägt. In ihrer einfachsten Form sieht die Definition einer Funktion so aus:

```
function myFunc
{
  Tu-Was
  Tu-WasAnderes
  # Beliebig viele weitere Befehle
}
```

Damit wird die Funktion `myFunc` definiert, die sich nachfolgend einfach über ihren

```
function LogsLöschen($Ordner = ".", $Alter = 7)
{
  $datum = (Get-Date) - (New-TimeSpan -Days $Alter)
  dir "$Ordner\*.log" | ? LastWriteTime -lt $datum | Remove-Item
}
```

Eigene Funktionen können wie die eingebauten Befehle der PowerShell Parameter entgegennehmen; Standardvorgaben sind möglich.

Namen aufrufen lässt, wobei die PowerShell Groß-/Kleinschreibung ignoriert:

```
myFunc
```

Funktionen, die innerhalb ein und desselben Skripts definiert und auch gleich benutzt werden, dienen nicht nur der Übersicht, sondern sind noch aus einem weiteren Grund sinnvoll: Dadurch, dass man sie nur einmal zu definieren braucht und dann beliebig oft aufrufen kann, helfen sie dabei, dass man Befehlsfolgen, die an mehreren Stellen im Skript vorkommen, nur einmal hinzuschreiben braucht. Das macht das Skript nicht nur kürzer, sondern auch wartungsfreundlicher: Sollte sich herausstellen, dass die Funktion fehlerhaft ist, braucht man sie nur an einer Stelle zu korrigieren. Als Faustregel gilt: Sobald Sie sich dabei erwischen, mehr als eine Zeile Code zu kopieren, um sie an anderer Stelle im selben Skript zu duplizieren, sollten Sie besser darüber nachdenken, diesen Code in eine eigene Funktion auszulagern.

Marke Eigenbau

Aus einem anderen Blickwinkel betrachtet sind Funktionen nichts anderes als selbstgebastelte, eigene Befehle. Wie die eingebauten PowerShell-Befehle können auch Funktionen Argumente übergeben bekommen und ein Ergebnis zurückliefern. Ersteres erreicht man, indem man bei der Definition der Funktion die Namen der erwarteten Parameter in eine Klammer hinter den Befehlsnamen schreibt und durch Kommas trennt – siehe den Code auf Seite 169.

Die Funktion `LogsLöschen()` entfernt alle .log-Dateien in einem bestimmten Ordner, die älter als die gewünschte Anzahl von Tagen sind. Das Beispiel zeigt auch gleich noch, wie man den Parametern einer Funktion Standardwerte zuweist, die zum Zuge kommen, wenn der Anwender beim Aufruf keine eigenen Werte übergibt: Als Ordner ist das aktuelle Verzeichnis vorgegeben, als maximales Alter sieben Tage.

Welche der Argumente der Aufrufer befüllt, ist ihm selbst überlassen. All diese Aufrufe sind legal:

```
LogsLöschen
LogsLöschen Temp
LogsLöschen Temp 30
LogsLöschen -Alter 14
```

Bemerkenswert ist die letzte Zeile: Sie zeigt, dass die Parameter einer Funktion sich genauso verhalten wie die Optionen der eingebauten Befehle, das heißt, man kann ihnen auch durch Voranstellen des Namens einen Wert zuweisen. Sinnvoll ist das vor allem dann, wenn man wie hier für das erste Argument den Standardwert akzeptieren, aber andere Parameter selbst bestimmen will.

Soll eine Funktion einen Wert liefern, stellt man dem zurückzugebenden Ausdruck einfach das Schlüsselwort `return` voran. Das kann nicht nur wie im untenstehenden Beispiel in der letzten Zeile passieren, sondern auch vorher. Sinnvoll ist das dann aber nur innerhalb einer Bedingung oder einer Schleife, denn ein `return` beendet die Abarbeitung der Funktion für den aktuellen Aufruf sofort.

Die Funktion `BinFormat` dient dazu, große Zahlen übersichtlicher anzuzeigen, indem sie aus den Präfixen Kilo, Mega, Giga und so weiter denjenigen heraussucht, der den Wert am kompaktesten darstellt. Dazu teilt die Funktion den übergebenen Wert so lange durch 1024, bis er kleiner als 1024 wird oder ihr die Einheiten ausgehen.

Neu an der Funktion ist neben dem `return` der Umgang mit einer Liste: Die erste Zeile speichert in der Variablen `$units` die Liste (für Puristen: ein .NET-Array) der Präfixe, die dazu einfach durch Kommas getrennt im Code stehen. Die Variable `$unit` bildet den Index, der am Ende bestimmt, welches Element der Liste zum Zuge kommt. Wie man auf einen bestimmten Eintrag einer Liste zugreift, zeigt der letzte Ausdruck im Code: `$units[$unit]` bedeutet „das `$unit`-ste Element der Liste `$units`“, wobei zu beachten ist, dass die Zählung bei 0 beginnt. Deshalb wird `$unit` auch mit einer 0 initialisiert und jedes Mal um 1 erhöht (`$unit++`), wenn der `$Wert` durch 1024 geteilt wird.

Einen Parameter wie eine normale lokale Variable zu behandeln und seinen Wert innerhalb einer Funktion zu ändern ist übrigens durchaus gängig und bleibt außerhalb der Funktion ohne Nebenwirkungen. Nach der Abarbeitung der Zeilen

```
$groesse = 100000
BinFormat $groesse
```

enthält die Variable `$groesse` nach wie vor 100.000. Die Funktion `BinFormat` bekommt im Parameter `$Wert` nur eine Kopie übergeben.

Aufgefallen dürfte Ihnen im Beispiel-Code auch der Ausdruck sein, der hinter dem `return` steht. Er verwendet den Operator `-f`, mit dem sich in der PowerShell Zeichenketten formatieren lassen. Sein Aufbau folgt dem Muster

```
Formatstring -f Argumente
```

Dabei steckt in `Argumente` eine (durch Kommas getrennte) Liste der zu formatierenden Werte. Der `Formatstring` ist eine Zeichenkette, die im einfachsten Fall die Platzhalter `{0}`, `{1}` und so weiter enthält. Die werden dann entsprechend der Nummerierung (sie beginnt auch hier bei 0) durch die Argumente ersetzt. Durch Zusätze in den Platzhaltern kann man das Format für die Argumente noch genauer bestimmen – im Beispiel bedeutet `{0:f2}`, dass das erste Argument als Gleitkommazahl mit zwei Nachkommastellen angezeigt werden soll. Die Syntax entspricht dabei der „kombinierten Formatierung“, wie sie auch das .NET Framework verwendet (siehe ct.de/yr5z).

Man kann einer PowerShell-Funktion auch über die Objekt-Pipeline Parameter übergeben, Argumente als obligatorisch oder optional kennzeichnen und ihren Datentyp vorgeben. Das sprengt aber den Rahmen dieses Artikels und wird in einer späteren c’t-Ausgabe behandelt werden.

```
function BinFormat($Wert)
{
  $units = "", "Kilo", "Mega", "Giga", "Tera", "Peta", "Exa", "Zetta", "Yotta"
  $unit = 0
  while($Wert -ge 1024 -and $unit -lt $units.Length - 1)
  {
    $Wert = $Wert / 1024
    $unit++
  }
  return "{0:f2} {1}" -f $Wert, $units[$unit]
}
```

Das Schlüsselwort return dient dazu, aus einer Funktion einen Wert zurückzuliefern. Das Beispiel formatiert eine große Zahl übersichtlich.

Ungeduldigen seien der Befehl `help about_Functions_Advanced` sowie die dort unter „See also“ genannten Hilfe-Artikel ans Herz gelegt.

Profile

Die bislang als Beispiele gezeigten Funktionen sind durchaus dazu geeignet, nicht nur im Rahmen eines Skripts verwendet zu werden, sondern beim interaktiven Gebrauch der PowerShell als Befehle zur Verfügung zu stehen. Dazu wäre es aber blöd, wenn man sie jedes Mal vor der Benutzung erst eintippen müsste. Eine Lösung wäre ein Skript, das ihre Definitionen enthält und das man bei Bedarf etwa mit

```
. MeineFunktionen.ps1
```

aufruft. Aber es geht noch besser: Jedes Mal, wenn die PowerShell startet, versucht sie automatisch, mehrere Skripte zu laden und abzuarbeiten. Die Namen dieser Skripte bekommen Sie mit dem Aufruf

```
$profile | fl * -Force
```

zu sehen. Zwei davon liegen normalerweise im Installationsordner der PowerShell unter \Windows\System32\WindowsPowerShell\v1.0 und gelten für alle Benutzer auf diesem Rechner; sie sind nur mit Administratorrechten zu bearbeiten. Die beiden Dateien, deren Bezeichner mit „CurrentUser“ beginnen, gehören zu Ihrem Benutzerkonto; Sie können sie bei Bedarf einfach neu anlegen und editieren. Der Unterschied zwischen den Dateien, deren Bezeichner auf „AllHosts“ endet, und denen mit „CurrentHost“ besteht darin, dass erstere bei jedem PowerShell-Start zum Zuge kommen und letztere sich unterscheiden, je nachdem, ob Sie eine normale PowerShell-Konsole oder das ISE öffnen. So können Sie bestimmte Aktionen oder Definitionen auf eine der beiden Umgebungen beschränken.

Die Profile-Dateien können beliebigen Code enthalten. Dazu gehört neben Funktions- und Alias-Definitionen auch Code, der sich direkt auswirkt. In der Datei Microsoft.PowerShell_profile.ps1 für die Textkonsole des Verfassers dieser Zeilen findet sich beispielsweise der nebenstehend gezeigte Code. Er sorgt dafür, dass PowerShell-Fenster, die mit Administratorrechten geöffnet werden, eine auffällige Hintergrundfarbe bekommen.

Falls Ihnen diese Zeilen und der folgende Absatz wie böhmische Dörfer vorkommen: Die Aufrufe im Detail zu verstehen ist gar nicht nötig. Sie sollen vielmehr demonstrieren, wie einfach man die Klassen und Methoden des .NET Framework für Aufgaben in der PowerShell einspannen kann. Wer in dieser Umgebung bereits programmiert hat, dem steht so ein nahezu unerschöpflicher Vorrat an Funktionen zu Gebote.

```
function BinIchAdmin() {
  $identity = [System.Security.Principal.WindowsIdentity]::GetCurrent()
  $princ = New-Object System.Security.Principal.WindowsPrincipal($identity)
  return $princ.IsInRole(
                [System.Security.Principal.WindowsBuiltInRole]::Administrator)
}
& {
  $ui = (Get-Host).UI.RawUI
  if(BinIchAdmin) {
    $ui.BackgroundColor = "DarkRed"
    Clear-Host
  }
}
```

Wem der eingebaute Befehlsumfang der PowerShell nicht ausreicht, der kann auch sämtliche Klassen und Methoden des .NET Framework verwenden. Das Beispiel färbt das Konsolenfenster, wenn der Aufrufer Administratorrechte besitzt.

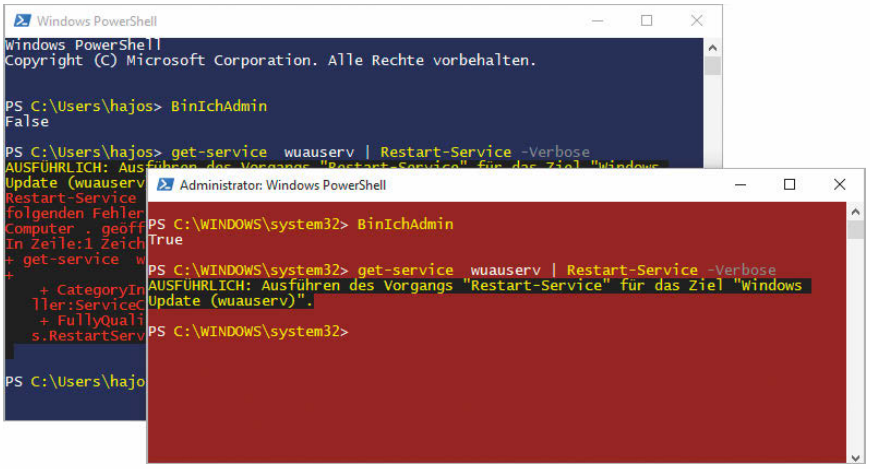
Ob die aktuelle PowerShell mit Admin-Rechten läuft, findet die Funktion `BinIchAdmin()`heraus, indem sie auf die Klasse `WindowsIdentity` aus dem Namensraum `System.Security.Principal` zurückgreift. Diese besitzt eine statische Methode namens `GetCurrent()`, die eine Repräsentation des gerade angemeldeten Benutzers liefert. Mit `New-Object` erzeugt `BinIchAdmin()` ein Objekt vom Typ `WindowsPrincipal` aus demselben Namensraum, das die Rechte des übergebenen Benutzers repräsentiert. Dessen Methode `IsInRole()` prüft schließlich, ob der Benutzer Administratorrechte besitzt.

Der im Skript folgende Code wertet das Ergebnis von `BinIchAdmin` per `if()` aus und setzt im Erfolgsfall die Hintergrundfarbe (`BackgroundColor`) der Konsole (`(Get-Host).UI.RawUI`) neu. Dass dieser Code in der Profile-Datei für ein normales PowerShell-Textfenster steht, hat den Grund, dass eine geänderte Hintergrundfarbe im Direktfenster des ISE extrem hässlich aussieht. *(hos@ct.de)* c't

Literatur

[1] Hajo Schulz, Aufbruch mit Power, PowerShell: Loslegen mit Microsofts mächtiger Kommandozeile, c't 2/2018, S. 166

Tools und Online-Doku: ***ct.de/yr5z***



Mit der auffälligen roten Farbe warnt dieses PowerShell-Fenster den Anwender, dass er Admin-Rechte besitzt und bei der Befehlseingabe besonders vorsichtig sein sollte.