Aufbruch mit Power

PowerShell: Loslegen mit Microsofts mächtiger Kommandozeile

Microsoft treibt die Verbreitung der PowerShell als universelle Textkonsole immer weiter voran – und das nicht nur unter Windows. Es lohnt sich also mehr denn je, sich mit ihr anzufreunden. Als Belohnung warten ein nahezu unerschöpflicher Befehlsvorrat und endlich eine Skriptsprache, die diese Bezeichnung verdient hat.

Von Hajo Schulz

Schon seit einigen Versionen hat Windows ein Programm an Bord, um die arg in die Jahre gekommene Eingabeaufforderung abzulösen: die PowerShell. Bei Administratoren in größeren Organisationen erfreut sie sich wachsender Beliebtheit, nicht zuletzt weil sich damit Rechner bequem aus der Ferne warten lassen. In letzter Zeit hat Microsoft einiges getan, um der PowerShell auf noch breiterer Front zum Durchbruch zu verhelfen: Beispielsweise hat sie seit dem vorletzten größeren Update für Windows 10 den Platz der Eingabeaufforderung im Windows+X-Menü eingenommen.

Die PowerShell ist auch nicht mehr exklusiv Windows-Anwendern vorbehalten: Für Linux- und macOS-Benutzer gibt es die „PowerShell Core“. Entstanden ist sie aus Quellcode, den Microsoft unter einer Open-Source-Lizenz veröffentlicht hat – siehe Textkasten.

Bei vielen Anwendern fristet die PowerShell trotz allem immer noch ein Schattendasein – zu Unrecht, wie wir finden: Bei der interaktiven Benutzung bietet sie Komfort-Funktionen wie Tab-Completion für Befehle und Optionen oder eine unerreicht ausführliche Hilfe. Ihr größter Schatz ist aber die eingebaute Skriptsprache, mit der sich Aufgaben automatisieren lassen. Im Vergleich dazu mutet die Batch-Sprache der Eingabeaufforderung geradezu vorsintflutlich an. Zudem hält die PowerShell sowohl in der Konsole als auch in Skripten einen beinahe unerschöpflichen Befehlsvorrat bereit.

Zugegeben: Wenn Sie bislang mit der Eingabeaufforderung arbeiten oder auch schon die eine oder andere Batch-Datei geschrieben haben, werden Sie nicht darum herumkommen, ein paar neue Befehle und Strukturen zu lernen. Aber spätestens wenn Ihr erstes PowerShell-

Skript läuft, ohne dass Sie zum x-ten Mal die kryptischen Optionen von Befehlen wie `for` oder `set` nachschlagen mussten, hat sich der Aufwand gelohnt. Auf gehts!

Erste Schritte

Die Bedienung der PowerShell fußt auf demselben Prinzip wie die der Eingabeaufforderung: Man tippt einen Befehl ein und schickt ihn mit einem Druck auf die Return-Taste ab. Wenn man sich nicht vertippt hat, wird das Kommando ausgeführt und unter der Eingabe erscheint Text, der je nach Befehl die angeforderten Informationen enthält oder über Erfolg oder Fehlschlag der Aktion informiert. Danach beginnt das Spiel von vorne.

Als Befehl kommen bei der PowerShell Vertreter verschiedener Kategorien in Frage. Zunächst einmal lassen sich unter Windows wie in der Eingabeaufforderung alle Kommandos benutzen, die aus einer externen .exe-Datei bestehen, darunter Befehle wie `robocopy` oder `bcdedit`. Kommandozeilenprogramme mit eigener Befehlseingabe wie `netsh` oder `diskpart` lassen sich genauso wie Windows-Anwendungen, etwa `notepad` oder `calc`, durch Eingabe ihres Namens starten. Von den in die Eingabeaufforderung eingebauten Befehlen versteht die PowerShell die wichtigsten. So kann man etwa `cd`, `md`, `dir` und `del` auch hier nutzen, muss sich aber bei dem einen oder anderen an eine neue Syntax für die Optionen gewöhnen. Wer die Kommandos aus der Unix-Shell Bash lieber mag als die der Eingabeaufforderung, findet auch Kommandos wie `ls` oder `rm`.

Den größten Vorrat an Kommandos bilden die PowerShell-eigenen Befehle, auch Cmdlets genannt. Sie sind grundsätzlich nach dem Muster `Verb-Nomen` aufgebaut, also beispielsweise `Get-FileHash`, `Remove-PrintJob` oder `Stop-Process`. Diese Konvention sieht erst mal nach viel Tipparbeit aus, die sich aber schnell relativiert: Für die am häufigsten benutzten Befehle gibt es sogenannte Aliasse, unter denen sie alternativ aufrufbar sind. Auch die oben erwähnten, aus der Eingabeaufforderung bekannten Kommandos sind Aliasse, etwa `cd` für `Set-Location`. Für Ihre persönlichen Lieblingsbefehle können Sie sich auch selbst Aliasse erstellen. Eine Liste aller gerade definierten Aliasse liefert `gal` – der Befehl ist ein Alias für `Get-Alias`.

Das Eintippen langer Befehle wird noch durch eine zweite Eigenschaft der PowerShell erleichtert: Sämtliche zur Verfügung stehenden Befehle und ihre Optionen sind nämlich auch der Tab-Completion bekannt. Das bedeutet, dass Sie nach der Eingabe der ersten paar Zeichen eines Befehls die Tab-Taste drücken können und die PowerShell ergänzt die restlichen. Bis zum Bindestrich sollten Sie aber Befehle schon selbst eintippen, denn sonst kann es Ihnen passieren, dass die PowerShell Ihre Eingabe mit Dateinamen aus dem aktuellen Ordner ergänzt – sie kann ja nicht wissen, ob Sie gerade einen Befehl eingeben oder eine Datei starten wollen. War Ihre Eingabe trotzdem noch nicht eindeutig, kann zunächst ein anderer als der von Ihnen gewünschte Befehl erscheinen – dann wiederholen Sie einfach den Druck auf die Tab-Taste; Umschalt+Tab hilft, falls Sie zu oft Tab gedrückt haben. Übrigens ist der PowerShell die Groß-/Kleinschreibung egal; Sie können alle Befehle auch komplett in Kleinbuchstaben eintippen.

Hilfe!

Eines der wichtigsten Cmdlets wird am Anfang Ihrer PowerShell-Gehversuche wahrscheinlich `Get-Help` (Alias: `help`) sein. Damit es aussagekräftige Informationen liefert, sollten Sie zunächst eine PowerShell mit Administratorrechten starten und dort einmal den Befehl `Update-Help` ausführen. Damit lädt die PowerShell die erweiterten Hilfedateien für die vorinstallierten Module von Microsofts Servern herunter und speichert sie auf der lokalen Festplatte. Für Anwender, die sich keine Admin-Rechte verschaffen können, bleibt noch der Ausweg, sich anzugewöhnen, `Get-Help` stets mit der Option `-Online` aufzurufen: Dann lädt die PowerShell die gewünschte Information als Webseite in Ihrem Standard-Browser.

Um Details über einen Befehl nachzulesen, können Sie dessen Namen einfach an das Kommando `help` anhängen, also beispielsweise `help Set-Location`. Existiert zu dem nachgeschlagenen Befehl ein Alias, funktioniert auch der: `help cd` liefert dieselben Informationen. Wer außer einer Beschreibung des Befehls auch genauere Informationen über seine Parameter und Beispiele zu seiner Verwendung lesen möchte, hängt am Ende noch die Option `-Detailed` an, die sich auch zu `-det` abkürzen lässt: `help cd -det`.

Außer Einträgen zu einzelnen Cmdlets enthält das Hilfesystem etliche Artikel, die Konzepte der PowerShell erklären. Ihre Titel beginnen alle mit „about_“; eine Liste liefert `help about_*`.

Objekte

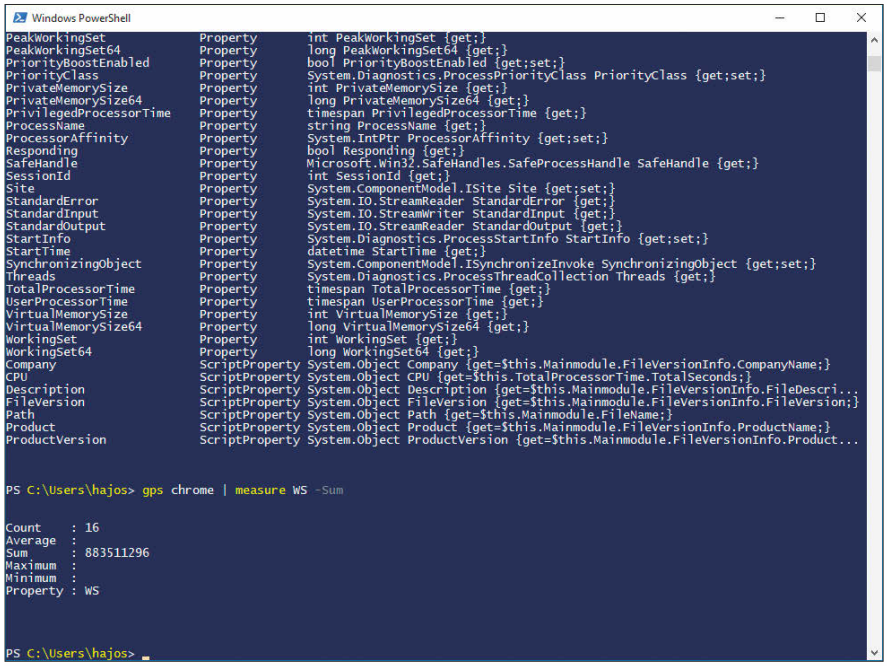
Eine der gewöhnungsbedürftigsten, aber auch mächtigsten Eigenschaften der PowerShell ist, dass ihre Cmdlets als Ausgabe eigentlich keinen Text liefern, sondern sogenannte Objekte. Was das bedeutet, lässt sich am besten anhand von Beispielen erklären. Um nicht vom Wesentlichen abzulenken, haben wir uns bemüht, sie möglichst einfach zu halten. Für sich betrachtet mögen Sie einige der gezeigten Befehle trotzdem als unnötige Verkomplizierung empfinden: Wenn Sie etwa wissen wollen, wie viel Hauptspeicher Ihr Browser gerade belegt, können Sie das natürlich bequem im Task-Manager ablesen. Sobald Sie aber den Wert beispielsweise regelmäßig in eine Log-Datei schreiben

PowerShell für alle

War die PowerShell anfangs ein Bestandteil von Windows, ist sie mittlerweile nicht mehr den Benutzern von Betriebssystemen aus Redmond vorbehalten: Microsoft hat große Teile der PowerShell unter der quelloffenen MIT-Lizenz veröffentlicht. Als „PowerShell Core“ wird dieser Entwicklungszweig von einer aktiven Community gepflegt und steht für Windows, Linux und macOS zur Verfügung – siehe ct.de/yzf2. Die aktuelle Versionsnummer von PowerShell Core lautet 6.0.

In Windows enthalten ist nach wie vor die „Windows PowerShell“, deren aktuellen Quellcode Microsoft wie gehabt unter Verschluss hält und deren derzeitige Versionsnummer die 5.1 ist. Wer lieber mit der quelloffenen Variante arbeiten möchte, kann PowerShell Core unter Windows parallel installieren und hat dann die Wahl, mit `powershell` die Windows PowerShell oder mit `pwsh` PowerShell Core zu starten.

Ausgewählte Verbesserungen aus der PowerShell-Core-Entwicklung will Microsoft in künftige Versionen der Windows PowerShell übernehmen und die Core-Entwickler legen großen Wert auf Kompatibilität zwischen beiden Zweigen. In diesem Artikel ist deshalb allgemein von der PowerShell die Rede; gemeint sind damit beide Ausgaben.



Rein optisch versprüht die PowerShell einen eher spröden Charme. Wer sich davon nicht abschrecken lässt, dem steht ein mächtiges Werkzeug zu Gebote.

wollen, um ihn im Zeitverlauf zu beobachten, oder wenn Sie ihn von mehreren Rechnern einsammeln müssen, werden Sie sich dazu wahrscheinlich ein Skript schreiben. Und dann können Sie das eine oder andere unserer Beispiele sicher gebrauchen – zumal sich die damit erklärten Konzepte ja auch auf andere Anwendungsfälle übertragen lassen.

Wenn Ihr Browser Edge, Chrome oder Firefox heißt, startet er für jeden Tab einen eigenen Prozess. Um den gesamten Speicherverbrauch zu ermitteln, müssen Sie also die Werte mehrerer Prozesse addieren. Das PowerShell-Cmdlet, das eine Liste der gerade laufenden Prozesse liefert, heißt `Get-Process` (Alias: `gps`). Als Parameter kann man ihm unter anderem einen Prozessnamen oder eine Suchmaske wie „*edge*“ mitgeben, dann enthält das Ergebnis nur noch Prozesse mit passendem Namen. Die sichtbare Ausgabe besteht aus einer Tabelle; den Hauptspeicherverbrauch enthält die Spalte „WS(K)“ – „WS“ steht für WorkingSet, „(K)“ für KByte. Hinter „PM(K)“ und „NPM(K)“ verbergen sich die Größen des ausgelagerten und des nicht auslagerbaren Hauptspeichers, auch jeweils in KByte.

Das eigentliche Ergebnis von `Get-Process` ist aber nicht diese Tabelle, sondern eine Liste von Prozess-Objekten. Objekte zeichnen sich dadurch aus, dass sie je nach Typ einen Satz von Eigenschaften besitzen – Prozesse eben unter anderem einen Namen, einen Speicherverbrauch und eine Prozess-ID. In der PowerShell kann man Objekte über eine sogenannte Pipeline von einem Cmdlet an das nächste weiterschicken, indem man die beiden Befehle in eine Zeile schreibt und durch einen senkrechten Strich (Pipe-Symbol; auf deutschen Tastaturen AltGr+<) voneinander trennt. Das sieht dann zum Beispiel so aus:

```
gps chrome | gm -MemberType Properties
```

`gm` ist ein Alias für `Get-Member`, ein Cmdlet, das ein Objekt oder eine Liste gleichartiger Objekte als Eingabe erwartet und als Antwort die Namen aller Bestandteile liefert, die diese Objekte ausmachen. Mit `-MemberType Properties` wird diese Liste auf die Eigenschaften beschränkt – Objekte haben in der Regel auch noch Methoden oder können Ereignisse melden, die hier aber zunächst nicht interessieren. Trotz des Filters ist die Liste der Eigenschaften von Prozess-Objekten beeindruckend: Man kann ihnen über 60 Merkmale entlocken.

Um sich statt der Tabelle, die `Get-Process` normalerweise ausgibt, andere Attribute von Prozessen anzeigen zu lassen, kann man eines der Cmdlets `Format-Table` (Alias: `ft`) oder `Format-List` (Alias: `fl`) benutzen: Ersteres liefert eine Tabelle, letzteres eine Listendarstellung. Beide verdauen als erstes Argument eine Liste der gewünschten Eigenschaften, also zum Beispiel:

```
gps win* | ft Name, Company, Path
```

Ein * anstelle der Eigenschaftenliste sorgt dafür, dass sämtliche Merkmale ausgegeben werden. So kann man die Floskel

```
| fl *
```

an beinahe jede PowerShell-Eingabe anhängen, um detailliertere als die standardmäßig ausgegebenen Ergebnisse des davorstehenden Befehls anzuzeigen.

Zurück zur Aufgabe, den Speicherverbrauch des Browsers herauszufinden: Für einfache Rechenaufgaben ist das Cmdlet `Measure-Object` (Alias: `measure`) zuständig. Ohne weitere Argumente zählt es einfach die ihm übergebenen Objekte. Als ersten Parameter kann man ihm einen oder eine Liste von Eigenschaftennamen übergeben und anschließend eine oder mehrere der Optionen `-Average`, `-Sum`, `-Maximum` und `-Minimum`, um sich den Mittelwert, die Summe, den größten oder den kleinsten Wert der gewünschten Eigenschaften anzeigen zu lassen. So liefert die Zeile

```
gps chrome | measure WS -Sum
```

die Anzahl der Bytes, die der Google-Browser gerade im Hauptspeicher belegt.

Filtern und sortieren

Dadurch, dass Cmdlets Objekte als Resultat liefern, kann man mit deren Eigenschaften nicht nur rechnen, sondern sie auch beinahe beliebig als Filter- oder Sortierkriterien verwenden. Sie brauchen beispielsweise die Angabe, welche Dienste gerade auf Ihrem System laufen? Kein Problem: Eine Liste aller installierten Dienste liefert `Get-Service` (Alias: `gsv`). Die Objekte, aus denen diese Liste besteht, haben unter anderem eine Eigenschaft `Status`, die entweder `Running` oder `Stopped` enthält. Als universelles Filter-Cmdlet dient `Where-Object` (Alias: `where`). Weil man es relativ oft verwendet, haben die PowerShell-Macher ihm als Abkürzung auch noch das einfache Fragezeichen spendiert. Alle laufenden Dienste bekommt man also mit der Eingabe

```
gsv | ? Status -eq Running
```

angezeigt. Dabei ist `-eq` der Vergleichsoperator, der auf Gleichheit prüft (englisch: equal). Alternativ stehen unter anderem noch `-ne` für „ungleich“ (not equal), `-gt` und `-lt` für „größer als“ (greater than)

und „kleiner als“ (less than) sowie `-ge` und `-le` für „mindestens“ (greater than or equal) und „höchstens“ (less than or equal) zur Verfügung. Mit `-match` kann man außerdem prüfen, ob eine Eingabe zu einem regulären Ausdruck passt. Näheres zu den Vergleichsoperatoren erzählt `help about_Comparison_Operators`.

Zum Sortieren dient das Cmdlet `Sort-Object` (Alias: `sort`). Als Parameter verdaut es einen oder mehrere Namen von Eigenschaften, nach denen es seine Eingabe dann aufsteigend sortiert; die Option `-Descending` dreht die Sortierreihenfolge um. Eine Liste aller Dateien im aktuellen Verzeichnis in absteigender Reihenfolge ihrer Größe liefert also beispielsweise der Befehl

```
dir | sort Length -Descending
```

Rechnen

Die PowerShell ist nicht nur dazu gedacht, einfache Befehle oder vergleichsweise simple Befehlsfolgen wie die bisher gezeigten auf der Konsole einzugeben und sich die Ergebnisse direkt anzeigen zu lassen. Für komplexere Aufgaben enthält sie auch eine Programmiersprache. Einige der Strukturen, die diese Sprache ausmachen, lassen sich aber durchaus auch von Nicht-Programmierern an der Konsole sinnvoll einsetzen.

Dazu gehört, dass man mit der PowerShell rechnen kann: Sie verdaut klaglos Eingaben wie `12 + 13` oder `3 * 5`. Bei der Eingabe längerer Ausdrücke beachtet sie die Regel „Punkt- vor Strichrechnung“; mit Klammern lässt sich die Reihenfolge der Berechnung beeinflussen. Details dazu liefert `help about_Arithmetic_Operators`.

Bestimmte Rechenoperatoren funktionieren nicht nur mit Zahlen, sondern zum Beispiel auch mit Zeichenketten: `"Power" + "Shell"` liefert etwa `"PowerShell"`, `"xyz" * 3` ergibt `"xyzxyzxyz"`.

Gewöhnungsbedürftig ist das Rechnen mit Kalenderdaten und Uhrzeiten: Das aktuelle Datum mit Uhrzeit liefert zunächst einmal `Get-Date`. Einen bestimmten Datumswert kann man mit einer Eingabe wie `Get-Date 24.12.2017` definieren, für eine bestimmte Uhrzeit muss man `Get-Date` einen oder mehrere der Parameter `-Hour`, `-Minute` und `-Second` mitgeben. Datumswerte lassen sich nicht direkt addieren, sondern dazu muss man `New-TimeSpan` verwenden; zieht man zwei Datumswerte voneinander ab, kommt ebenfalls ein `TimeSpan`-Objekt dabei heraus. „Heute in fünf Tagen“ würde man beispielsweise so ausdrücken:

```
(Get-Date) + (New-TimeSpan -Days 5)
```

Die Klammern sind notwendig: Sobald man mit den Ergebnissen von Cmdlets etwas anderes tun will, als sie über die Pipeline weiterzuleiten, benötigt die PowerShell sie, um zu erkennen, wo ein Befehl endet. Die PowerShell spart aber auch nicht mit Fehlermeldungen, wenn sie Klammern vermisst.

Variablen

Um die Ergebnisse von Berechnungen, aber auch von beliebigen anderen Befehlen oder Befehlsfolgen in späteren Kommandos wiederzuverwenden, kann man sie in sogenannten Variablen aufheben. Variablen sind nichts anderes als benannte Speicherplätze für irgendwelche Daten. Variablennamen beginnen stets mit einem $-Zeichen und dürfen Ziffern und Buchstaben enthalten; zwischen Klein- und Großbuchstaben wird nicht unterschieden.

Die PowerShell erzeugt Variablen automatisch bei ihrer ersten Verwendung. So existiert nach dem Ausführen des Ausdrucks

```
$antwort = 2 * 21
```

eine Variable namens `$antwort`, ihr Inhalt ist die Zahl 42. Sollten Sie mehrere PowerShell-Fenster geöffnet haben, besitzt jedes seinen eigenen Satz an Variablen. Sobald Sie eine PowerShell-Sitzung beenden, sind die darin definierten Variablen vergessen. Einen neuen Wert bekommt eine Variable, indem man einfach den alten überschreibt. Wichtig zu wissen ist, dass Variablen das Resultat einer Operation speichern, nicht die Operation selbst. Wenn Sie etwa die Zeile

```
$jetzt = Get-Date
```

eingeben, steht in `$jetzt` das heutige Datum und eine Uhrzeit von beispielsweise 12:15 Uhr. Fünf Minuten später ergibt `Get-Date` eine Uhrzeit von 12:20 Uhr, aber in `$jetzt` steht immer noch 12:15 Uhr.

Variablen speichern nicht nur einfache Werte wie Zahlen, Zeichenketten oder Datumswerte, sondern etwa auch Prozess-Objekte oder Dateilisten. So enthält nach der Befehlsfolge

```
$1woche = (Get-Date) - ↵
              ↳ (New-TimeSpan -Days 7)
$kannweg = dir *.log | ↵
          ↳ ? LastWriteTime -lt $1woche
```

die Variable `$kannweg` eine Liste aller Log-Dateien aus dem aktuellen Verzeichnis, die vor mehr als einer Woche geschrieben wurden. Wie das Ergebnis eines Cmdlet kann man auch eine Variable wieder in die Objekt-Pipeline einspeisen:

```
$platz = $kannweg|measure Length -Sum
```

Wie Ihnen vielleicht schon weiter oben aufgefallen ist, liefert `measure` die Gesamtgröße der Dateien nicht einfach als Zahl, sondern offenbar eingebettet in ein Objekt; die PowerShell gibt es in Listendarstellung aus und es enthält die gewünschte Zahl in einer Eigenschaft namens `Sum`. Um an diese Zahl heranzukommen, damit Sie sie etwa in einer anderen Variablen speichern oder mit ihr rechnen können, müssen Sie die Punkt-Schreibweise verwenden:

```
$MByte = $platz.Sum / (1024 * 1024)
```

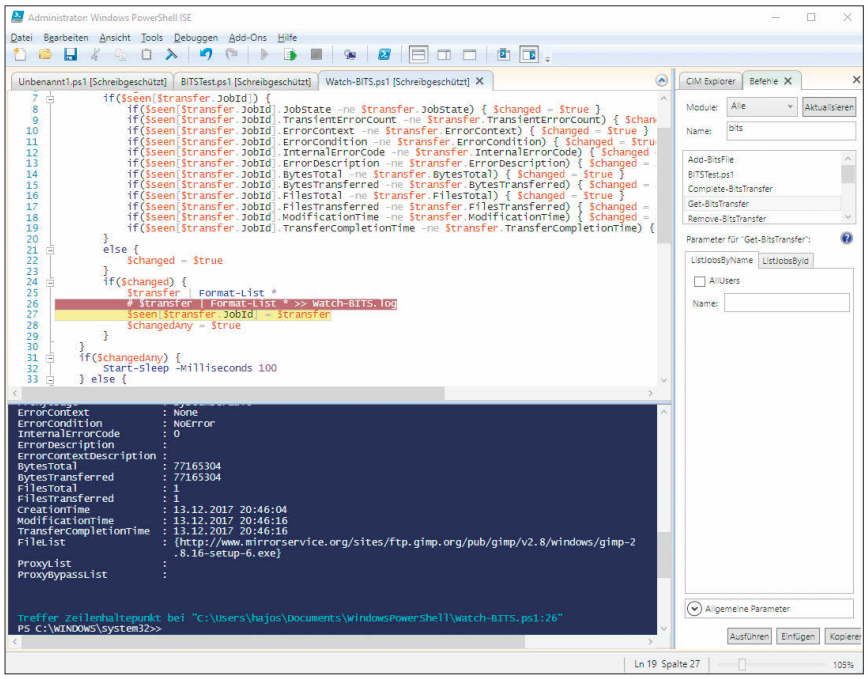
Ein Ausdruck nach dem Muster `Objekt.Eigenschaft` funktioniert mit allen Objekten, nicht nur mit Variablen. Wenn Sie aus dem Ergebnis eines Cmdlet direkt eine Eigenschaft herausoperieren wollen, müssen Sie den Aufruf samt aller eventuellen Parameter einklammern: Mit `(Get-Date).Year` erfahren Sie etwa die aktuelle Jahreszahl.

Einer für alle

Sie können die Objekt-Pipeline nicht nur zum Filtern, Sortieren und Anzeigen von Ergebnissen verwenden, sondern das, was hinten rauskommt, auch an Cmdlets weiterreichen, die Aktionen mit den jeweiligen Objekten ausführen: Dienst-Objekte können Sie an `Restart-Service` verfüttern, Prozess-Objekte an `Stop-Process` (Alias: `kill`), Datei-Objekte an `Remove-Item` (Alias: `del`) und so weiter. Ob Sie die Objekte zwischendurch in einer Variablen speichern, ist dabei egal: Nach Eingabe des obigen Beispiels räumt

```
$kannweg | del
```

die alten Log-Dateien anstandslos von der Platte. Was dabei hinter den Kulissen passiert, ist, dass die PowerShell das Cmdlet `Remove-Item` mehrfach aufruft und ihm bei jedem Durchgang ein anderes Objekt aus der in `$kannweg` gespeicherten Liste übergibt. So etwas funktioniert aber nur dann ohne weitere Verrenkungen, wenn der Wert aus der Liste der einzige Parameter ist, den das Ziel-Cmdlet verlangt. Für alle anderen Fälle gibt es das Cmdlet `ForEach-Object` mit dem Alias `foreach`. Weil es häufig gebraucht wird, können Sie es auch einfach mit einem %-Zeichen abkürzen.



Die Windows PowerShell hat mit dem Integrated Scripting Environment einen brauchbaren Code-Editor samt Debugger an Bord.

Seine Benutzung ist anhand eines Beispiels schnell erklärt: Angenommen, Sie wollen die alten Dateien in einen Archiv-Ordner verschieben, statt sie zu löschen. Dann sieht der Aufruf so aus:

```
$kannweg | % { move $_ D:\Archiv }
```

Der Ausdruck innerhalb der geschweiften Klammern ist ein sogenannter Skript-Block. Das Cmdlet ruft ihn nacheinander für jeden Eintrag von `$kannweg` einmal auf. Zuvor speichert es diesen Eintrag jeweils in der automatisch erzeugten Variablen `$_`. Die können Sie also als Platzhalter für das jeweilige Element der Liste verwenden.

Übrigens kann auch das Filter-Cmdlet `Where-Object` mit einem Skript-Block umgehen. Das ist immer dann nützlich, wenn Ihr Filter mehr als eine Eigenschaft der übergebenen Objekte prüfen soll. So liefert der Ausdruck

```
gps | ? { ↵
  ↳ $_.Company -match 'Microsoft' ↵
  ↳ -and $_.WS -ge 10 * 1024 * 1024 }
```

alle laufenden Programme, die von Microsoft stammen und (`-and`) mindestens 10 MByte im Hauptspeicher belegen.

Alles ist ein Laufwerk

Neben der Objekt-Pipeline dürfte das überraschendste Merkmal der PowerShell für Umsteiger von der Eingabeaufforderung wohl die Flexibilität sein, mit der sich die von Dateioperationen gewohnten Befehle einsetzen lassen. Argumente von `cd`, `dir`, `del` und Konsorten können nämlich nicht nur Datei- und Ordnerpfade, sondern auch Registry-Schlüssel oder Inhalte des Zertifikatspeichers sein: Die PowerShell bildet sie auf sogenannte PSDrives ab, so etwas wie virtuelle Laufwerke. Dasselbe gilt für die Aufbewahrungsorte von Aliassen und Funktionen sowie von PowerShell- und Umgebungsvariablen. Auf diesen „Laufwerken“ gibt es aber keine Ordner; sie sind flache Verzeichnisse.

Wie die zugehörigen Laufwerke heißen, lässt sich mit dem Befehl `Get-PSDrive` eruieren. In der Spalte „Name“ liefert er neben den gewöhnlichen Laufwerksbuchstaben unter anderem die Einträge HKCU und HKLM für die Registry (HKU fehlt leider aus unerfindlichen Gründen). Sie können also etwa mit

```
cd "HKCU:\Control Panel\Desktop\"
```

in den Registry-Schlüssel mit den wichtigsten Einstellungen für Ihren Desktop wechseln – Registry-Schlüssel funktionieren wie Dateiordner. Allerdings finden Registry-Werte ihre Entsprechung nicht in Dateien, sondern in sogenannten Item Properties. Nach dem obigen `cd` liefert etwa

```
gp . WallPaper
```

den Dateinamen Ihres Bildschirmhintergrundes, wobei `gp` ein Alias für `Get-ItemProperty` ist. Das zweite Argument (`WallPaper`) ist der Name des gesuchten Registry-Wertes; hier sind Wildcards erlaubt. Das Pendant zum Setzen von Registry-Werten heißt `Set-ItemProperty` (Alias: `sp`), das zum Löschen `Remove-ItemProperty` (`rp`). Eine detaillierte Liste aller Befehle, die im Zusammenhang mit der Registry sonst noch interessant sind, liefert `help Registry`.

Die virtuellen Laufwerke `Alias:`, `Function:`, `Env:` und `Variable:` werden Sie in der Praxis kaum benutzen. Neugierige können ihnen mit `dir` ein Verzeichnis der jeweils gespeicherten Objekte entlocken.

Interessant und hauptsächlich bei den Umgebungsvariablen praktisch ist noch eine besondere Schreibweise: Um auf die Umgebungsvariable zuzugreifen, die Sie aus der Eingabeaufforderung als `%PATH%` kennen, können Sie nämlich

```
$env:path
```

schreiben – und zwar zum Auslesen genauso wie zum Setzen.

Scripting

Apropos Eingabeaufforderung: Was der ihre Batch-Dateien sind, sind der PowerShell die Skripte. Auch dabei handelt es sich um Textdateien, deren Name aber die Endung .ps1 trägt. Bearbeiten lassen sie sich im Prinzip mit jedem Texteditor, es gibt aber deutlich bessere Werkzeuge dafür: Die Windows PowerShell bringt das „Integrated Scripting Environment“ mit, das sich aus einer normalen PowerShell-Sitzung heraus mit dem Kurzbefehl `ise` starten lässt. Wer die PowerShell an die Taskleiste anheftet, findet das ISE auch im Kontextmenü des Symbols.

Ein erstklassiger Editor für PowerShell-Skripte ist auch Visual Studio Code in Zusammenarbeit mit der dafür verfügbaren PowerShell-Erweiterung. Editor und Plug-in gibt es von Microsoft kostenlos (siehe ct.de/yzf2) und das nicht nur für Windows, sondern auch für macOS und Linux, also für alle Systeme, unter denen PowerShell Core läuft. Visual Studio Code – oder kurz Code – ist nicht zu verwechseln mit Microsofts großer Entwicklungsumgebung Visual Studio: Es ist um Größenordnungen schlanker, schneller installiert und gestartet und steht unter einer Open-Source-Lizenz.

Ob Sie als Windows-PowerShell-Anwender lieber zum ISE oder zu Code greifen, ist Geschmackssache: Beide bieten einen Code-Editor mit IntelliSense, also

automatisch aufpoppenden Listen, in denen man sinnvolle Fortsetzungen für den gerade getippten Ausdruck findet. Auch einen Debugger haben beide an Bord, wobei der in Code mit seinen bedingten Haltepunkten und einer Watch-Liste etwas besser ausgestattet ist. Auch bei der dynamischen Überprüfung des gerade bearbeiteten Skriptes auf falsche Syntax und typische Programmierfehler hat Code die Nase vorn. Dafür glänzt das ISE mit einer integrierten Befehlsliste, in der man nach unbekannten Cmdlets suchen und deren Parameter bequem in ein Formular eingeben kann.

Bevor Sie Skripte ausführen können, gibt es in der Windows PowerShell (nicht in PowerShell Core) noch eine Hürde zu überwinden: Aus Sicherheitsgründen erlaubt Microsoft das in der Grundeinstellung nämlich unverständlicherweise nicht. Aufheben lässt sich diese Einschränkung, indem Sie die PowerShell einmal mit Administratorrechten starten, dort den Befehl

```
Set-ExecutionPolicy RemoteSigned
```

eingeben und die Sicherheitsfrage bejahen. Das stellt die sogenannte Ausführungsrichtlinie so ein, dass die PowerShell alle Skripte ausführt, die nicht über ihre Eigenschaften als aus dem Internet stammend gekennzeichnet sind. Alternativ kommen auch die Richtlinien `Unrestricted` oder `Bypass` in Frage, aber die schützen nicht vor fremden Skripten. Genauere Informationen finden Sie unter `help about_Execution_Policies`.

Ein einfaches PowerShell-Skript enthält – wie eine Batch-Datei – schlicht Befehle, wie Sie sie auch in einer interaktiven Sitzung eingeben würden. Um ein Skript auszuführen, geben Sie in ein PowerShell-Fenster dessen kompletten Dateinamen ein. Zum Starten von MeinSkript.ps1 genügt aber auch die Eingabe `.\MeinSkript`, wenn Sie sich bereits in dem Verzeichnis befinden, wo das Skript gespeichert ist.

Beim Abarbeiten eines Skripts liest die PowerShell die Datei Befehl für Befehl und führt alle nacheinander aus, ganz so, als hätten Sie sie gerade in Ihrer PowerShell-Sitzung eingegeben. Einen kleinen Unterschied gibt es aber: Variablen, Aliasse und Funktionen, die Sie in einem Skript definieren, gelten nur innerhalb des Skripts – wenn es beendet ist, verschwinden sie wieder beziehungsweise nehmen ihren vorherigen Wert an. Vergleichbar ist das mit Batch-Dateien, die den Befehl `SETLOCAL` verwenden. Um ein Skript wirklich so auszuführen, als hätten Sie die Befehle gerade eingetippt, stellen Sie seinem Aufruf einen Punkt gefolgt von einem Leerzeichen voran:

```
. .\MeineAliasse.ps1
```

Programmieren

An Kontrollstrukturen bringt die Skriptsprache der PowerShell das mit, was man von einer modernen Programmiersprache erwarten kann. So lässt sich etwa vor dem Ausführen eines oder mehrerer Befehle prüfen, ob eine Bedingung erfüllt ist:

```
if(<Bedingung>)
{
  <Befehl(e)>
}
elseif(<Andere Bedingung>)
{
  <Andere Befehle>
}
else
{
  <Noch mehr Befehle>
}
```

So ein Statement darf dabei null bis beliebig viele `elseif`-Zweige und null bis einen `else`-Zweig besitzen. Als Bedingungen kommen unter anderem Vergleichsoperatoren in Frage, wie Sie sie schon von den Filtern kennen:

```
if($size -gt 1000)
{ ...
```

Die Existenz einer Datei oder eines Ordners prüft man mit `Test-Path`:

```
if(Test-Path "$env:TEMP\error.log")
{ ...
```

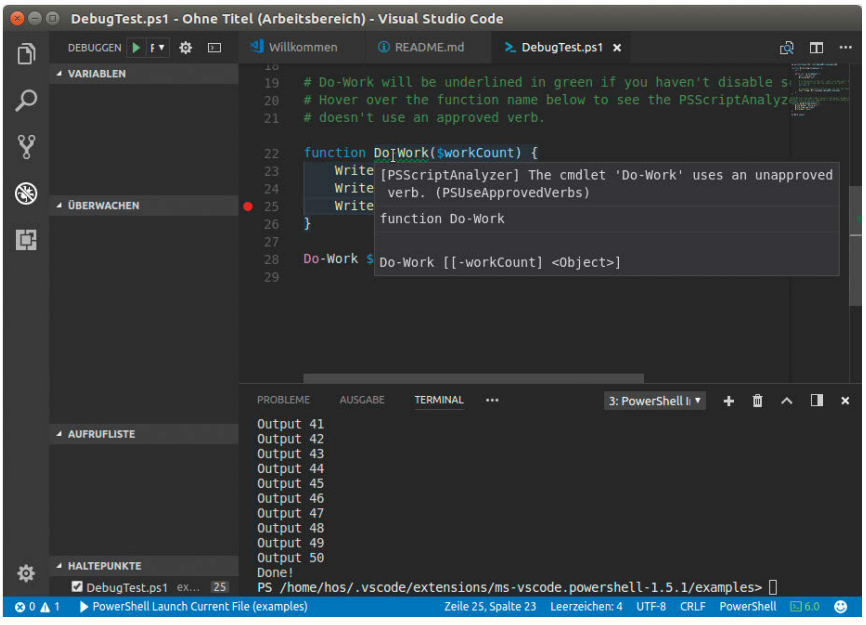
Außerdem kann man praktisch jeden Rückgabewert von Cmdlets und Berechnungen als Bedingung verwenden:

```
$np = Get-Process notepad
if($np)
{ ...
```

Als unwahr zählen dabei die Zahl 0, eine leere Zeichenkette, die Konstanten `$null` und `$false` sowie eine Liste ohne Elemente oder mit einem der vorgenannten Objekte als einzigem Element; alles andere erfüllt die Bedingung. Negieren kann man eine Bedingung durch ein vorangestelltes `-not` – Klammern nicht vergessen!

Zu den fortgeschrittenen Programmiertechniken, die Sie in PowerShell-Skripten einsetzen können, gehören unter anderem Schleifen, die Definition eigener Funktionen, die Übergabe von Parametern und der Zugriff auf die Objekt-Pipeline. Genug Stoff also für weitere Artikel zur PowerShell in kommenden c't-Ausgaben. *(hos@ct.de)* c't

Dokumentation, Tools, PowerShell Core: ***ct.de/yzf2***



PowerShell Core steht unter einer Open-Source-Lizenz und läuft unter Windows, Linux und macOS. Dasselbe gilt für die Skript-Entwicklungsumgebung Visual Studio Code.