Scripting

Das Praxisbuch für Administratoren
und DevOps-Teams

Kapitel 4

PowerShell

Die älteste »Shell« für Windows ist das Programm `cmd.exe`, also die sogenannte »Eingabeaufforderung«. `cmd.exe` repräsentiert die Oberfläche von MS DOS, also dem textorientierten Vorgänger von Microsoft Windows. Batch-Dateien (Endung `*.bat`) ermöglichen eine einfache Script-Programmierung. `cmd.exe` ist allerdings kein zeitgemäßes Werkzeug mehr, sondern ein Relikt aus der IT-Steinzeit.

2006 stellte Microsoft die erste Version der *Windows PowerShell* als leistungsfähigen Nachfolger von `cmd.exe` vor. Inzwischen sind wir bei Version 7.*n* angelangt. Das Programm läuft auch unter Linux und macOS; weil die Shell somit nicht mehr Windows-spezifisch ist, wurde ihr Name zu *PowerShell* vereinfacht. Die PowerShell ist heute *das* Scripting-Werkzeug in der Windows-Welt. Die Sprache ist unersetzlich, wenn Sie Windows-nahe Funktionen (z. B. Microsoft-Cloud-Setups oder Active Directories) administrieren möchten.

Die größte technische Errungenschaft der PowerShell im Vergleich zu anderen Shells ist ihr objektorientierter Ansatz. PowerShell-Kommandos liefern keinen Text, sondern echte Objekte zurück. Das eröffnet ganz neue Möglichkeiten für die Weiterverarbeitung. Im Vergleich zur Bash ist die Syntax der PowerShell deutlich logischer (aber dennoch nicht immer intuitiv und auf jeden Fall sehr ausschweifend und langatmig).

In diesem Kapitel vermittle ich Ihnen einen ersten Eindruck von den Grundfunktionen der PowerShell. Eine Menge Beispiele folgt dann in den weiteren Kapiteln.

Die ExecutionPolicy

Aus Sicherheitsgründen ist die Ausführung von (unsignierten) Scripts auf Firmenrechnern oft verboten. Hintergründe und Einstellmöglichkeiten sind in Abschnitt 4.5, »Das erste Script«, beschrieben.

KI-Tools und PowerShell

Es ist naheliegend, bei der Entwicklung von Scripts KI-Tools zu Hilfe zu nehmen, je nach Geschmack nur zur Recherche oder auch für das eigentliche Coding und die Fehlersuche. Mittlerweile funktionieren alle gängigen KI-Plattformen – von ChatGPT über Claude, Gemini bis hin zum europäischen Mistral gut.

Aber »gut« ist nicht immer gut genug! Während die KI-Hilfe bei einfachen Python- und Bash-Scripts in der Regel großartig ist, scheitern viele KI-Tools an PowerShell-Code. Und genau da gab es zuletzt spürbare Unterschiede zwischen den Anbietern. Die Zeitschrift c't stellte im Mai 2026 fest, dass Claude 4.6 Opus viel bessere Ergebnisse als ChatGPT 5.4 lieferte (und selbst Claude machte Fehler). Ich habe zum gleichen Zeitpunkt an diesem Buch gearbeitet und teile diese Ansicht: Claude ist, gerade beim PowerShell-Coding, spürbar zuverlässiger.

Überbewerten Sie diese Empfehlung nicht. Bis das Buch erschienen ist, gibt es schon wieder neue KI-Modelle, die hoffentlich besser funktionieren. Behalten Sie aber im Hinterkopf, dass sich bei KI-Ärger ein Test mit konkurrierenden Anbietern oder mit anderen Sprachmodellen durchaus lohnen kann, ganz speziell bei der PowerShell!

4.1 Installation

Die PowerShell ist unter aktuellen Windows-Versionen vorinstalliert – allerdings selten in der aktuellsten Version. Um die Version der installierten PowerShell zu ermitteln, suchen Sie im Startmenü nach *Terminal* und führen das gleichnamige Programm aus. (Bei älteren Windows-Versionen führt die Suche nach *PowerShell* zum Ziel.)

Im Terminal geben Sie `$PSVersionTable` ein und drücken ↵. Als Antwort erhalten Sie eine Tabelle mit den Eckdaten der PowerShell-Version:

```
> $PSVersionTable

  Name           Value
  ----           -----
  PSVersion      7.6.0
  PSEdition      Core
  ...
```

Sollte die Versionsnummer 5.1 lauten, dann haben Sie die veraltete Windows PowerShell gestartet (siehe auch den folgenden Hinweiskasten). Bei den meisten Windows-Installationen sind die Windows PowerShell und die neuere PowerShell parallel verfügbar. Öffnen Sie im Terminal einen neuen Tab und achten Sie darauf, dass Sie dabei den Eintrag **PowerShell** und nicht **Windows PowerShell** auswählen. Dann wiederholen Sie den obigen Test.

Sollte die PowerShell wirklich nicht installiert sein, schafft `winget` Abhilfe (siehe auch Abschnitt 8.1, »Paketmanager«). Die Installation von Version 7.*n* erfolgt parallel zur vorinstallierten Version 5.1.

```
> winget install -e --id Microsoft.PowerShell
```

Falls die PowerShell beim Start auf eine neuere Version hinweist, gelingt das Update wie folgt:

```
> winget upgrade Microsoft.PowerShell
```

Anstatt mit `winget` können Sie die PowerShell auch über den Microsoft Store installieren oder als MSI-Datei von der Microsoft-Website herunterladen. `winget` führt am schnellsten zum Ziel. `winget` ist ein integraler Bestandteil von Windows 11 und gibt Zugang zu Microsoft-Store-Funktionen über das Terminal.

Vermeiden Sie die Windows PowerShell!

Microsoft pflegt seit mehr als einem Jahrzehnt die alte Windows PowerShell (Versionsnummer 5.1, erhält seit 2016 nur noch Sicherheits-Updates) und die neue PowerShell (ohne vorangestelltes »Windows«). Weil es diverse Legacy-Scripts gibt, die mit der modernen PowerShell nicht funktionieren, sind auf den meisten Windows-Rechnern beide Varianten parallel installiert.

Stellen Sie mit `$PSVersionTable` sicher, dass Sie nicht mit der veralteten PowerShell arbeiten! Sie können bei den Terminal-Einstellungen festlegen, welche Shell per Default benutzt werden soll. Dieses Buch berücksichtigt ausschließlich die PowerShell ab Version 7.*n*!

Installation unter Linux

Seit die PowerShell einer Open-Source-Lizenz untersteht, steht die PowerShell auch unter Linux zur Verfügung. Besonders einfach funktioniert die Installation unter Ubuntu:

```
sudo snap install --classic powershell
```

Die PowerShell kann nun mit dem Kommando `pwsh` gestartet werden. Anleitungen für andere Distributionen finden Sie hier:

https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-on-linux

Installation unter macOS

Auf vielen macOS-Rechnern, die zur Software-Entwicklung eingesetzt werden, ist das Paketverwaltungssystem *Homebrew* installiert (siehe auch *https://brew.sh*). Wenn das der Fall ist, gelingt die PowerShell-Installation mit einem simplen Kommando im Terminal:

```
$ brew install powershell
```

Spätere Updates führen Sie so durch:

```
$ brew upgrade powershell
```

Wie unter Linux starten Sie die PowerShell auch unter macOS mit dem Kommando `pwsh`.

Einschränkungen der PowerShell unter Windows versus Linux/macOS

Auch wenn die PowerShell viele gute Ideen realisiert, hält sich die Sehnsucht von Linux- oder macOS-Anwenderinnen und -Anwendern nach einer weiteren Shell in Grenzen. Die Auswahl war schon bisher groß.

Das Hauptproblem bei der Anwendung der PowerShell außerhalb von Windows besteht darin, dass eine viel geringere Anzahl von Kommandos zur Auswahl steht. Mit `Get-Command` können Sie eine Liste aller Kommandos ermitteln. (Genau genommen handelt es sich bei den Kommandos um CmdLets, Funktionen etc. Auf diese Differenzierung gehe ich später ein.) Mit `Measure-Object` können Sie die Anzahl der Kommandos zählen. Das folgende Listing fasst die unter Windows, Linux und macOS ermittelten Ergebnisse im Frühjahr 2026 zusammen:

```
> Get-Command | Measure-Object
  Count: 2112   (Windows 11 Pro)
  Count:  298   (Linux/macOS)
  ...
```

Sie sehen schon, um wie viel kleiner die Anzahl unter Linux und macOS ist. Das hat damit zu tun, dass viele PowerShell-Kommandos Windows-spezifische Aufgaben erledigen, also z. B. einen neuen Windows-Benutzer einrichten. Derartige Kommandos würden unter Linux oder macOS keinen Sinn ergeben. Generell ist die Infrastruktur unter Windows eine ganz andere. Unter Linux und macOS gibt es keine *Scheduled Jobs*, keine *Common Information Model* (CIM), keine *Windows Management Instrumentation* (WMI) usw. (Linux und macOS haben natürlich vergleichbare Funktionen, aber die sind ganz anders realisiert.) Aus diesem Grund funktionieren manche der in diesem Buch präsentierten Beispiele *nicht* unter Linux oder macOS.

Das ist aber kein Grund, den Kopf in den Sand zu stecken! Die PowerShell bietet die Möglichkeit, Erweiterungsmodule zu installieren. Es gibt eine Menge Module, deren Funktionen gleichermaßen unter Windows, Linux und macOS geeignet sind, z. B. zur Administration externer Dienste wie der Amazon Cloud (AWS).

Fazit: PowerShell-Scripts zur Windows-Administration funktionieren unter Linux und macOS nicht, weil die Kommandos und das zugrunde liegende Software-Fundament fehlen. Scripts, die allgemeingültige, nicht auf Windows fokussierte Aufgaben erfüllen und dabei auf Module zugreifen, können aber sehr wohl plattformübergreifend entwickelt werden.

PowerShell-Konfiguration

Anfänglich besteht selten eine Notwendigkeit, die PowerShell zu konfigurieren. Sobald Sie aber etwas Erfahrung im Umgang mit der PowerShell gewonnen haben, wollen Sie vielleicht, dass beim Start der PowerShell Funktionen oder Kommandoabkürzungen (sogenannte Aliasse) definiert werden, Module automatisch geladen werden etc. Der

richtige Ort für solche Anpassungen ist die Profile-Datei, deren Pfad aus der Variablen `$PROFILE` hervorgeht (hier nur aus Platzgründen über zwei Zeilen verteilt):

```
> $PROFILE

  C:\Users\kofler\Documents\PowerShell\
    Microsoft.PowerShell_profile.ps1
```

Um die Datei zu verändern, starten Sie den Editor am besten direkt in der PowerShell:

```
> mkdir Documents\PowerShell # falls das Verz. nicht existiert
> edit $PROFILE       # edit (winget install microsoft.edit)
> notepad $PROFILE    # Notepad.exe
> code $PROFILE       # Visual Studio Code
```

4.2 Das Windows-Terminal

In der Einleitung dieses Abschnitts habe ich Sie gebeten, ein Terminal zu starten, um die PowerShell-Version herauszufinden. Das ist Ihnen vielleicht widersprüchlich erschienen. Was ist also der Unterschied zwischen der PowerShell und dem Terminal, und wie hängen die beiden Programme zusammen?

Das Terminal ist ein Programm, das verschiedene Shells ausführen kann. Windows-taugliche Shells sind z. B. die veraltete Eingabeaufforderung (`cmd.exe`), die Windows PowerShell 5.*n*, die PowerShell 7.*n*, die Azure Shell, die Git-Bash sowie die Bash von Linux (Letzteres über das *Windows Subsystem for Linux*, kurz WSL).

Die Aufgabe des Terminals besteht lediglich darin, Ihre Tastatureingaben entgegenzunehmen, diese an die aktive Shell weiterzuleiten und das Ergebnis der Shell dann wieder als Text anzuzeigen. Daneben erfüllt das Terminal noch ein paar mehr Funktionen, aber die spielen für uns eine untergeordnete Rolle.

Das Terminal ist seit 2019 ein integraler Bestandteil von Windows. In der Vergangenheit erledigten `cmd.exe` bzw. die PowerShell die Terminal-Funktionen selbst bzw. delegierten diese Aufgabe an den *Windows Console Host*. Unter Linux und macOS hat sich die Aufgabentrennung zwischen Shell und Terminal seit Jahrzehnten bewährt; und so hat sich schließlich auch Microsoft dazu aufgerafft, eine »richtige« Terminal-App zu entwickeln. Bei aktuellen Windows-Versionen ist das Terminal vorinstalliert. Sollte das nicht der Fall sein, führen Sie `winget install -e --id Microsoft.WindowsTerminal` aus.

Was kann das Terminal nun? Die vielleicht wichtigste Neuerung im Vergleich zur traditionellen Ausführung von `cmd.exe` oder der PowerShell besteht darin, dass Sie mehrere Shell-Instanzen starten und nebeneinander in »Tabs« (Dialogblättern) ausführen können. Das ist oft praktisch, um verschiedene Funktionen parallel zu testen, in verschiedenen Verzeichnissen zu arbeiten oder um in einem Tab die PowerShell, in einem zweiten aber die Bash (Linux) auszuführen.

Terminal mit Administratorrechten ausführen

Die meisten PowerShell-Kommandos können Sie als gewöhnlicher Benutzer ausführen. Nur wenn Sie administrative Arbeiten erledigen möchten, benötigen Sie mehr Rechte.

Dazu suchen Sie im Startmenü nach *Terminal*, klicken den Eintrag mit der rechten Maus- oder Touchpad-Taste an und wählen **Als Administrator ausführen**.

Konfiguration

Besonders stolz ist Microsoft, dass das Terminal auch optisch mit entsprechenden Linux-Vorbildern mithalten kann. Wenn Sie möchten, können Sie die Vorder- und Hintergrundfarben des Terminals verändern, das Terminal mit einem Bild unterlegen oder transparent gestalten, sodass Sie sehen, was in den Fenstern unter dem Terminal vor sich geht. In den Screenshots für dieses Buch habe ich von diesen Spielereien Abstand genommen. Allerdings habe ich die Terminal-Farben verändert (heller Hintergrund, dunkle Textfarben), damit die Abbildungen im Druck besser lesbar sind.

Diese und unzählige andere Optionen können Sie in den Einstellungen verändern (siehe Abbildung 4.1). Das Programm unterscheidet zwischen globalen Grundeinstellungen, die für das gesamte Programm gelten, sowie zwischen Profilen. Jedes Profil ist einem Shell-Typ zugeordnet.

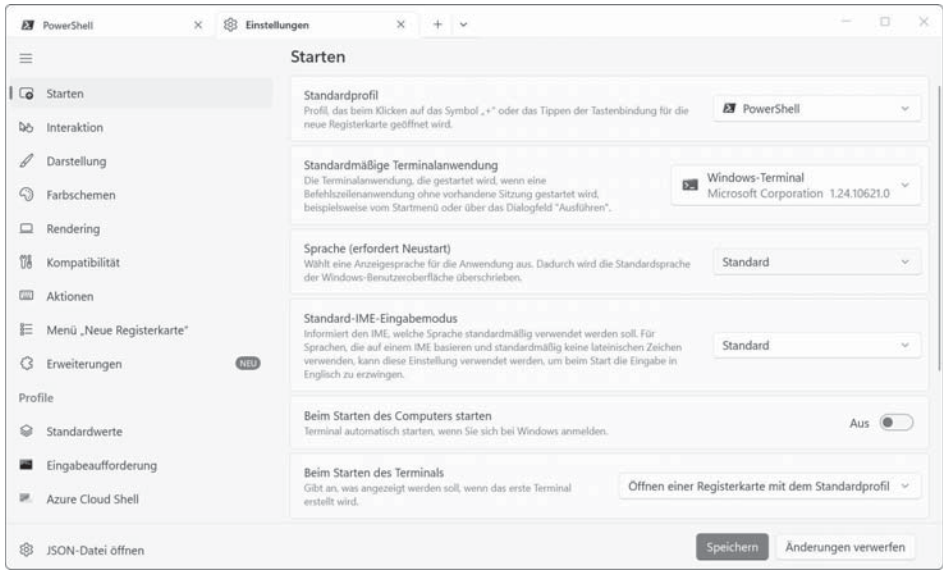

Abbildung 4.1: Das Aussehen und die Funktion des Terminals können durch unzählige Optionen beeinflusst werden.

Im Dialogblatt **Starten** der Einstellungen können (und sollten) Sie die PowerShell 7 zur Default-Shell machen (Eintrag **Standardprofil**). Außerdem sollten Sie das Terminal zum Defaultprogramm zur Ausführung von Shells machen (Eintrag **Standardmäßige Terminalanwendung**), wenn dies nicht ohnedies schon der Fall ist.

Bedienung

Wenn Sie in der Vergangenheit schon `cmd.exe`, alte PowerShell-Versionen oder ein Terminal in einem anderen Betriebssystem verwendet haben, wissen Sie bereits, dass Maus oder Touchpad im Terminal wenig Nutzen haben. Insbesondere ist es unmöglich, damit die Cursorposition zu verändern. Eingaben sind nur in der letzten Zeile des Terminals möglich. Innerhalb dieser Zeile können Sie den Cursor mit ◄ und ► bewegen. Mit ▲ und ▼ können Sie früher eingegebene Kommandos nochmals aufrufen und gegebenenfalls vorher verändern.

Immerhin können Sie mit der Maus einen Textbereich markieren und diesen dann mit Strg+C in die Zwischenablage kopieren. (Wenn gerade ein Kommando läuft, hat Strg+C eine andere Funktion: Es bricht die Ausführung ab.) Strg+V fügt den Inhalt der Zwischenablage an der Cursorposition ein.

Wenn Sie die ersten Buchstaben eines Kommandos angeben, wird in grauer Schrift die plausibelste Fortsetzung angezeigt. ► vervollständigt die Eingabe. Alternativ durchlaufen Sie mit der wiederholten Eingabe von →| weitere Ergänzungsmöglichkeiten. Analog funktioniert das auch für Kommandooptionen sowie für Dateinamen im aktuellen Verzeichnis. Strg+Leerzeichen zeigt alle möglichen Ergänzungen auf einmal an.

⇧+Strg+P führt in die Befehlspalette, eine lange Liste von Funktionen, die Sie innerhalb des Terminals ausführen können. Viele Funktionen beziehen sich auf die Verwaltung der Tabs (Tab öffnen, schließen, wechseln, umbenennen usw.) sowie auf optische Details (Schriftgröße, Farbe). Im Dialogblatt **Aktionen** sind nicht nur sämtliche Tastenkürzel aufgelistet, Sie können diese auch verändern.

4.3 Aufruf von CmdLets und Funktionen

Bevor Sie beginnen, erste Scripts zu programmieren, sollten Sie sich mit den Möglichkeiten der PowerShell interaktiv vertraut machen. Die folgenden Beispiele sollten ohne Grundkenntnisse verständlich sein. Auf die dabei eingesetzten Kommandos werden Sie später immer wieder stoßen. Details zu einzelnen Kommandos (eigentlich handelt es sich um »CmdLets«, aber mit diesem Begriff beschäftigen wir uns später) verrät bei Bedarf `Get-Help <kommandoname>`.

Im folgenden Listing ermittelt `Get-Location` den aktuellen Pfad. `Set-Location` wechselt in das Unterverzeichnis `Downloads`. Dort ermittelt `Get-ChildItem` alle Dateien mit der Endung `*.exe`. Schließlich löscht `Remove-Item` ein Setup-Programm:

```
> Get-Location

  C:\Users\kofler

> Set-Location Downloads

> Get-ChildItem *.exe

  Directory: C:\Users\kofler\Downloads

  Mode         LastWriteTime      Length Name
  ----         -------------      ------ ----
  -a---    29.01.2026 07:46    10752440 ChromeSetup.exe
  -a---    04.11.2025 10:13   569957808 Docker Desk Installer.exe
  -a---    27.03.2026 11:15     1127456 Okular Installer.exe

> Remove-Item ChromeSetup.exe
```

Ich habe einleitend erwähnt, dass die PowerShell mit Objekten arbeitet. Das ist aber nicht immer auf den ersten Blick erkenntlich. Beispielsweise liefert `Get-Date` das aktuelle Datum plus Uhrzeit. Standardmäßig zeigt die PowerShell nur eine Zeichenkette an, die die wichtigsten Daten zusammenfasst:

```
> Get-Date

  Freitag, 27. März 2026 15:45:00
```

Erst wenn Sie das Ergebnis `Get-Date` über den Pipe-Operator `|` an `Get-Member` weitergeben, wird klar, dass das Resultat von `Get-Date` den Datentyp `System.DateTime` hat und unzählige Eigenschaften und Methoden kennt. (Ich habe die Ausgabe hier aus Platzgründen stark gekürzt.)

```
> Get-Date | Get-Member

  TypeName: System.DateTime

  Name              MemberType     Definition
  ----              ----------     ----------
  Add               Method         datetime Add(timespan value)
  AddDays           Method         datetime AddDays(double value)
  AddHours          Method         datetime AddHours(double ...)
  AddMilliseconds   Method         datetime AddMilliseconds(...)
  ...
  Ticks             Property       long Ticks {get;}
  TimeOfDay         Property       timespan TimeOfDay {get;}
  Year              Property       int Year {get;}
```

Tipp

`Get-Member` liefert oft eine schier endlose Liste von Methoden und Eigenschaften. Unter Windows können Sie das Ergebnis mit `Get-Member | Out-GridView` in einem eigenen Fenster anzeigen. `Out-GridView` sowie diverse andere Möglichkeiten zur Formatierung bzw. zum Export von CmdLet-Ergebnissen stelle ich Ihnen in Abschnitt 7.6, »CmdLet-Ergebnisse verarbeiten«, vor.

Wenn Sie nicht das ganze Datum benötigen, sondern nur die Jahreszahl, stellen Sie `Get-Date` in runde Klammern, um so eine unmittelbare Auswertung zu erzwingen. Vom resultierenden Objekt werten Sie mit `.Year` nur eine Eigenschaft aus:

```
> (Get-Date).Year

  2026
```

Mit der Methode `AddHours(2)` berechnen Sie Datum und Uhrzeit in zwei Stunden:

```
> (Get-Date).AddHours(2)

  Freitag, 27. März 2026 17:45:00
```

CmdLets

Bis jetzt habe ich einfach gesagt, dass Sie in der PowerShell »Kommandos« ausführen können. Tatsächlich unterscheidet die PowerShell aber zwischen unterschiedlichen Typen von Kommandos: CmdLets, Funktionen, Aliasse und herkömmliche Kommandos. Bei Weitem am wichtigsten sind CmdLets: Das sind speziell für die PowerShell optimierte Kommandos.

Die Bezeichnung »CmdLet« rührt daher, dass ein CmdLet nicht ein eigenes Programm ist, sondern dass vielmehr ganze Gruppen von CmdLets in einer Bibliothek gesammelt sind. Noch technischer formuliert: Intern ist jedes CmdLet eine Instanz einer .NET-Klasse. Diese spezielle Organisation hat viele Vorteile. Insbesondere müssen gemeinsame Funktionen, z. B. die Verarbeitung von Parametern, nicht für jedes CmdLet neu implementiert werden. Auch die Verarbeitung von Objekten hat damit ein zentrales Fundament. Damit sollte auch klar sein, dass CmdLets keine PowerShell-Scripts sind. Bibliotheken mit CmdLets werden zumeist mit C# programmiert.

Die »Verb-Substantiv«-Nomenklatur

Die Namen von CmdLets folgen einem Schema und setzen sich aus einem Verb und einem Substantiv zusammen (*verb-noun pairs* in der Originaldokumentation). Das Verb beschreibt, was das Kommando tut, das Substantiv, welche Daten es verarbeitet. Microsoft hat sogar einen klaren Vorrat von Verben definiert, die Sie verwenden sollen, falls

Sie selbst Kommandos programmieren. Beispielsweise sollen Sie für Lösch-Operationen immer `Remove` verwenden, nicht aber `Delete`, `Eliminate` oder `Drop`:

https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands

Groß- und Kleinschreibung

Die PowerShell unterscheidet bei der Erkennung von Kommandos sowie bei CmdLet-Optionen, -Methoden und -Eigenschaften nicht zwischen der Groß- und Kleinschreibung. Daher liefern `(Get-Date).Month` und `(get-date).month` gleichermaßen die Nummer des aktuellen Monats.

Aliasse

`Get-ChildItem` ist eine sehr allgemeingültige Bezeichnung für ein Kommando, das den Inhalt eines Verzeichnisses ermittelt. Leider ist der Tippaufwand im Vergleich zu `dir` oder `ls` beachtlich. Deswegen gibt es für viele CmdLets Abkürzungen (»Aliasse«). Anstelle von `Get-ChildItem` sind gleichwertig auch `dir`, `ls` sowie `gci` erlaubt.

Eine Liste aller Aliasse liefert `Get-Alias` bzw. `gal`. Viele vordefinierte Abkürzungen setzen sich aus den Anfangsbuchstaben von Verb und Substantiv zusammen, also z. B. `fl` für `Format-List` oder `rnp` für `Rename-ItemProperty`. Teilweise kommen aber auch von Linux oder `cmd.exe` vertraute Bezeichnungen zum Einsatz, etwa `cp` für `Copy-Item` (siehe Tabelle 7.1).

Mit `Set-Alias` können Sie eigene Abkürzungen definieren. Die folgende Anweisung ermöglicht es, das Kommando `New-Item` zum Erzeugen einer leeren Datei auch mit dem Linux-typischen Namen `touch` auszuführen:

```
Set-Alias touch New-Item
```

Bei Bedarf speichern Sie Ihre `Set-Alias`-Anweisungen in `Documents/profile.ps1`, damit diese bei jedem PowerShell-Start automatisch ausgewertet werden.

Welches Kommando bezeichnet einen Alias?

Wenn Sie einen Alias kennen, aber nicht sicher sind, welches Kommando damit gemeint ist, führen Sie `Get-Command -Name <alias>` oder kürzer `Get-Alias <alias>` aus. `Get-Alias sls` zeigt, dass `sls` die Abkürzung für `Select-String` ist.

Parameter und Optionen

An die meisten Kommandos können Sie Parameter übergeben. Außerdem können Sie durch Optionen das Verhalten des Kommandos steuern. Das folgende Kommando durchsucht `C:\Users` sowie alle Unterverzeichnisse nach PNG-Dateien:

```
> Get-ChildItem -Path C:\Users\kofler -Recurse -Filter *.png

  Directory: C:\Users\kofler\OneDrive
  ... test.png

  Directory: C:\Users\kofler\Pictures
  ... img_1234.png
```

Welche Parameter in welcher Reihenfolge übergeben werden müssen und welche Optionen ein CmdLet vorsieht, fasst `Get-Help <cmd>` zusammen. An das Beispielkommando werden hier drei Optionen übergeben, `-Path`, `-Recurse` und `-Filter`. Optionen von CmdLets werden immer mit einem Bindestrich eingeleitet. Sie können abgekürzt werden, soweit eine eindeutige Ergänzung möglich ist. (Beispielsweise ist `-p` anstelle von `-pa` nicht erlaubt, weil es auch die Option `-PipelineVariable` gibt.)

```
gci -pa C:\Users -r -fi *.png
```

Bei der Script-Programmierung werden Abkürzungen nicht empfohlen, weil eine spätere Erweiterung eines Kommandos um eine Option dazu führen kann, dass Ihr Script dann nicht mehr funktioniert.

»Splatting«

Wenn CmdLets mit vielen Optionen aufgerufen werden sollen, ergeben sich oft sehr umfangreiche, schwer lesbare Anweisungen. Alternativ besteht die Möglichkeit, die Optionen vorweg in einer Hashtable zu speichern (siehe Abschnitt 4.7, »Arrays und Hashtables«) und diese dann an das CmdLet zu übergeben. Dabei müssen die Keys in der Hashtable mit den Optionsnamen des Kommandos übereinstimmen. Das folgende Beispiel greift die obige `Get-ChildItem`-Anweisung noch einmal auf und verdeutlicht die Syntax:

```
$opts = @{ Path = "C:\Users"; Recurse = $true; Filter = "*.png" }
Get-ChildItem @opts
```

Weitere Splatting-Varianten sind hier dokumentiert:

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_splatting

Funktionen

Bei der Script-Programmierung können Sie unkompliziert mehrere PowerShell-Kommandos zu einer Funktion zusammenfassen (siehe Abschnitt 4.11, »Funktionen und Parameter«). Um Funktionen weiterzugeben, speichern Sie den Code als Modul (Dateikennung `*.psm1`) in einem der dafür vorgesehenen Verzeichnisse (`$env:PSModulePath`).

Wenn Module korrekt installiert sind, lassen sich darin definierte Funktionen nicht von »echten« CmdLets unterscheiden. Auch Microsoft macht von dieser Möglichkeit intensiv Gebrauch: Von den rund 2000 Kommandos, die bei einer Standardinstallation der PowerShell unter Windows 11 zur Auswahl stehen, sind mehr als die Hälfte Funktionen:

```
> (Get-Command -CommandType Function | Measure-Object).Count
  1074

> (Get-Command -CommandType CmdLet | Measure-Object).Count
  906
```

Die obige Verkettung der Kommandos `Get-Command` und `Measure-Object` ist ein Beispiel für die Anwendung des Pipe-Operators, den ich Ihnen in Abschnitt 4.4, »Kommandos kombinieren«, näher vorstellen werde.

Herkömmliche Kommandos ausführen

Zu guter Letzt können Sie in der PowerShell auch herkömmliche Kommandos ausführen. Ein Beispiel für ein derartiges Kommando ist `ping` (üblicherweise `C:\Windows\System32\PING.EXE`) zum Testen von Netzwerkverbindungen. Traditionelle Kommandos haben allerdings den Nachteil, dass sie sich nicht wie CmdLets verhalten, also Text anstelle von Objekten zurückgeben.

Syntaktisch kompliziert wird es bei Kommandos, die sich in Verzeichnissen befinden, in denen die PowerShell nicht sucht (die also der Umgebungsvariablen `$Path` nicht bekannt sind). Der Versuch, einfach den gesamten Pfad anzugeben, scheitert dann oft an Leerzeichen:

```
> C:\Program Files\Git\usr\bin\nano.exe myfile.txt

  'C:\Program' is not recognized as a name of a cmdlet ...
```

Der Programmstart muss dann mit dem Call-Operator `&` erfolgen. Diesem wird im ersten Parameter eine Zeichenkette mit dem vollständigen Pfad übergeben. Davon getrennt können Parameter und Optionen folgen.

```
> & "C:\Program Files\Git\usr\bin\nano.exe"  "myfile.txt"
```

Online-Hilfe

Diesem Buch fehlt der Platz für eine umfassende Beschreibung aller CmdLets (werfen Sie aber einen Blick auf Kapitel 7, »CmdLets für die PowerShell«). Ich werde Ihnen daher in den Beispielen immer wieder neue CmdLets vorstellen.

Glücklicherweise ist die PowerShell mit großartigen Hilfefunktionen ausgestattet. `Get-Help <command>` (bei Bedarf ergänzt um die Optionen `-Examples` oder `-Detailed` oder `-Full`) liefert einen umfassenden Hilfetext zum betreffenden Kommando. Wenn Sie zusätzlich die Option `-Online` übergeben, erscheint der Hilfetext im Webbrowser.

Beim ersten Aufruf von `Get-Help` kann es passieren, dass die PowerShell darauf hinweist, dass die lokalen Hilfedateien unvollständig sind. Abhilfe schafft die einmalige Ausführung von `Update-Help -UICulture en-US`. Die Option `-UICulture` ist notwendig, weil die meisten Hilfetexte nur in englischer Sprache zur Verfügung stehen. Oft kommt es trotz dieser Option zu Download-Fehlern. Die haben damit zu tun, dass es für einige (oft kleinere) Module gar keine Hilfetexte gibt.

Wenn Sie auf der Suche nach einem Kommando sind, hilft `Get-Command` weiter. Ohne weitere Optionen liefert es einfach eine Liste aller Kommandos. Mit `Select-String` können Sie die schier endlose Ausgabe filtern. Das folgende Kommando erzeugt eine sortierte Liste aller Kommandos, die den Suchbegriff `VM` enthalten und offensichtlich virtuelle Maschinen steuern:

```
> Get-Command | Select-String VM | Sort-Object

  Add-NetEventVmNetworkAdapter
  Add-NetEventVmSwitch
  ...
```

`Get-Command` bietet noch viele weitere Suchmöglichkeiten – sehen Sie sich `Get-Help Get-Command -Examples` an!

> **Nach nicht installierten Kommandos suchen**
>
> `Get-Command` berücksichtigt nur CmdLets und Funktionen, die bereits installiert sind. Es gibt aber unzählige PowerShell-Erweiterungsmodule. Ein dort verstecktes Kommando entdecken Sie am schnellsten mit `Find-Command`. Mehr Tipps zur Verwendung von Zusatzmodulen folgen in Abschnitt 7.9, »Zusatzmodule installieren«.

CmdLets liefern Objekte. Welche Eigenschaften und Methoden ein derartiges Objekt zur Auswahl stellt, verrät die Methode `Get-Member`, die Sie auf ein Ergebnis anwenden. Das folgende Beispiel zeigt, dass `Get-ChildItem` normalerweise `System.IO.FileInfo`-Objekte zurückgibt, die mit unzähligen Eigenschaften und Methoden weiter ausgewertet bzw. verarbeitet werden können. (Beachten Sie, dass `Get-ChildItem` je nach Kontext

auch für andere Aufgaben verwendet wird, z. B. zum Auslesen von Registry-Einträgen. Dementsprechend ändert sich dann der Rückgabedatentyp.)

```
> Get-ChildItem somefile.txt | Get-Member

  TypeName: System.IO.FileInfo

  Name         MemberType     Definition
  ----         ----------     ----------
  Target       AliasProperty  Target = LinkTarget
  LinkType     CodeProperty   System.String LinkType{get=...;}
  AppendText   Method         System.IO.StreamWriter AppendText()
  ...
```

4.4 Kommandos kombinieren

Sie wissen nun, wie Sie einzelne Kommandos ausführen können. Die große Kunst der Script-Programmierung besteht darin, mehrere Kommandos sinnvoll miteinander zu kombinieren (siehe Tabelle 4.1). Die Operatoren `&&` und `||` stehen erst ab der PowerShell-Version 7 zur Verfügung.

Syntax	Funktion
`command1 | command2`	`command2` verarbeitet die Ergebnisse von `command1`
`command1 ; command2`	zuerst `command1` ausführen, dann `command2` (selbst dann, wenn `command1` einen Fehler ausgelöst hat)
`command1 && command2`	zuerst `command1` ausführen, dann `command2` (nur wenn `command1` fehlerfrei war)
`command1 || command2`	zuerst `command1` ausführen; `command2` wird nur ausgeführt, wenn `command1` einen Fehler ausgelöst hat

Tabelle 4.1: Kommandos kombinieren

Ich konzentriere mich im Weiteren auf den Pipe-Operator `|`, der in der Praxis am wichtigsten ist. Die Weiterverarbeitung von Zwischenergebnissen setzt voraus, dass das zweite Kommando mit dem Ergebnisdatentyp des ersten Kommandos zurechtkommt.

Beachten Sie, dass CmdLets intern immer Objekte liefern. Diese Objekte enthalten in der Regel viel mehr Eigenschaften, als am Bildschirm dargestellt werden. Die Bildschirmausgabe ist dahingehend optimiert, ein für Menschen (einigermaßen) übersichtliches Resultat zu liefern. Viele Eigenschaften der Objekte werden dabei weggelassen. Wenn Sie sich das Ergebnis mit allen Details ansehen wollen, leiten Sie die Ausgabe an `Format-List` oder an `Get-Member` weiter. (Probieren Sie z. B. `Get-ChildItem | Format-List` oder `Get-Process | Get-Member` aus!)

Im folgenden Beispiel ermittelt `Get-Process` alle Prozesse des Webbrowsers Edge. Die Prozessobjekte werden an `Stop-Process` weitergeleitet, dieses Kommando beendet den Webbrowser, sofern er denn läuft:

```
> Get-Process msedge | Stop-Process
```

Das nächste Kommando ermittelt die Anzahl der PNG-Dateien in einem Verzeichnis und allen Unterverzeichnissen:

```
> Get-ChildItem -Recurse *.png | Measure-Object

  Count             : 327
  Average           :
  Sum               :
  ...
```

`Measure-Object` liefert alle möglichen Eigenschaften. Hier ist aber nur `Count` von Interesse. Damit Sie die Eigenschaft auf das Objekt des Ergebnisses anwenden können, müssen Sie dieses klammern.

```
> (Get-ChildItem -Recurse *.png | Measure-Object).Count
  327
```

Wie viel Platz brauchen diese Dateien? `Measure-Object` kann von der `Get-ChildItem`-Ergebnisliste eine bestimmte Eigenschaft auswerten und summieren. In diesem Fall ist von den Ergebniseigenschaften nur `Sum` relevant:

```
> (Get-ChildItem -Recurse *.png |
   Measure-Object -Property Length -Sum).Sum

  97344806
```

Kommandoketten

Der Pipe-Operator kann natürlich auch mehrfach angewendet werden, also `command1 | command2 | command3` usw. Das eröffnet faszinierende Möglichkeiten. Das folgende Kommando ermittelt alle Dateien im Verzeichnis `Downloads`. Dann sortiert es die Ergebnisliste nach der Größe (die größte Datei zuerst), extrahiert die ersten zehn Dateien und ermittelt deren Platzbedarf.

```
> (Get-ChildItem -Path Downloads |
   Sort-Object -Property Length -Descending |
   Select-Object -First 10 |
   Measure-Object -Property Length -Sum).Sum

  22737476
```

Wenn Sie die zehn größten Dateien mit Rückfrage löschen möchten, gehen Sie folgendermaßen vor:

```
> Get-ChildItem -Path Downloads |
  Sort-Object -Property Length -Descending |
  Select-Object -First 10 |
  Remove-Item -Confirm
```

Mehrzeilige Anweisungen

Im Terminal können Sie beliebig lange Anweisungen eingeben, aber in diesem Buch ist die maximale Zeilenlänge vorgegeben. Deswegen muss ich komplexe Kommandos oft über mehrere Zeilen verteilen.

Wenn Sie mehrzeilige Kommandos eingeben, verhält sich die PowerShell recht intelligent: Sofern klar ist, dass das Kommando noch nicht fertig ist, können Sie einfach ↵ drücken und die Eingabe in der nächsten Zeile fortsetzen. Diese Voraussetzung ist immer dann gegeben, wenn es noch offene Klammern gibt oder wenn das letzte Zeichen in der Zeile ein Operator ist, der eine Fortsetzung erfordert. Das war bei den vorigen Beispielen der Fall (| am Zeilenende).

Wenn die Fortsetzung für die Shell hingegen nicht eindeutig erkennbar ist und Sie die Eingabe dennoch über mehrere Zeilen trennen wollen, muss die betroffene Zeile mit einem Leerzeichen und ` (einem *Backtick*) enden. Das Leerzeichen vor ` ist zwingend erforderlich. Nach ` muss unmittelbar ↵ folgen.

```
> Do-Something -LongOption 123 -OtherOption 456 `
    -YetAnotherOption "lorem ipsum"
```

Umgang mit Sonderzeichen

In der PowerShell haben viele Zeichen eine besondere Bedeutung: # leitet einen Kommentar ein, { und } bildet Code-Gruppen, > leitet die Ausgabe um usw. Manchmal ist es notwendig, solche Zeichen direkt zu verwenden, also ohne die PowerShell-spezifische Funktion. Das gilt z. B. beim Aufruf von externen Kommandos wie im folgenden Beispiel, bei dem die Zeichen > und # nicht interpretiert werden dürfen. (Das Kommando `magick` stelle ich Ihnen in Kapitel 17, »Bildverarbeitung«, näher vor.)

In solchen Fällen müssen Sie den vorhin schon erwähnten Backtick (also `) voranstellen. Der Backtick ist also das Quotierungszeichen der PowerShell und erfüllt dieselbe Aufgabe wie der Backslash \ in der Bash und vielen anderen Programmiersprachen. (Weil der Backslash in Windows Verzeichnisse trennt, kommt er für die PowerShell nicht als Quotierungszeichen in Frage.)

```
> magick in.png -resize 1024x1024`> -background `#733 out.jpg
```

In den meisten Fällen ist es einfacher, die betreffenden Zeichenketten in Anführungszeichen zu stellen.

```
> magick in.png -resize '1024x1024>' -background '#733' out.jpg
```

Wenn Sie in einer Zeichenkette einen Zeilenumbruch benötigen, verwenden Sie `` `n ``:

```
> Write-Output "line 1`nline 2"
  line 1
  line 2
```

4.5 Das erste Script

Die PowerShell bietet viele praktische Features für den interaktiven Betrieb. Mit eigenen Scripts gehen Sie noch einen Schritt weiter: Damit können Sie einmal interaktiv getestete Kommandos dauerhaft speichern. Das spart nicht nur in Zukunft Tipparbeit, sondern auch die Mühe, sich diverse Kommandos, Optionen usw. auswendig zu merken.

Um Scripts zu verfassen, benötigen Sie einen Editor. Für erste Experimente reicht *Notepad* aus. Schon deutlich mehr Funktionen bietet das kostenlose Programm *Notepad++*. Längerfristig empfehle ich Ihnen aber, Visual Studio Code zu installieren (siehe auch Kapitel 14, »Visual Studio Code«), wenn sich dieses Programm nicht ohnedies schon auf Ihrem Rechner befindet. Ganz egal, ob Sie PowerShell-, Bash- oder Python-Scripts entwickeln – VSCode unterstützt Sie dabei ausgezeichnet. (Nicht empfehlenswert ist dagegen die in vielen älteren Anleitungen beschriebene Entwicklungsumgebung *PowerShell ISE*. Das Programm wird von Microsoft nicht mehr weiterentwickelt und ist nur für PowerShell-Versionen bis 5.1 gedacht.)

Hello, World!

Das erste Script soll den Text »Hello, World!« ausgeben. (Keine Sorge, noch in diesem Abschnitt folgt ein originelleres Beispiel!) Öffnen Sie den Editor Ihrer Wahl, geben Sie die folgende Zeile ein und speichern Sie die Textdatei unter dem Namen `Hello.ps1` in einem leicht zu findenden Verzeichnis (z. B. in `Documents`). Das CmdLet `Write-Output` gibt die als Parameter übergebene Zeichenkette am Bildschirm aus.

```
Write-Output "Hello, World!"
```

Die Dateiendung `*.ps1` steht für PowerShell 1. Als Microsoft die zweite Version der PowerShell veröffentlichte, wollte man die dann schon etablierte Endung nicht mehr ändern – und so ist es absurderweise bis heute bei `*.ps1` geblieben.

Um das Script nun auszuführen, geben Sie einfach den Dateinamen samt vorangestelltem Verzeichnis an:

```
> Documents\Hello.ps1

  Hello, World!
```

Falls sich das Script im aktuellen Verzeichnis befindet, müssen Sie `.\` voranstellen, um das Verzeichnis explizit zum Ausdruck zu bringen.

```
> .\Hello.ps1

  Hello, World!
```

Dabei ist ».« eine Kurzschreibweise für das Verzeichnis, in dem Sie sich gerade befinden. Aus Sicherheitsgründen werden Scripts ohne Verzeichnisangabe nur dann ausgeführt, wenn sich die `*.ps1`-Dateien in einem Verzeichnis befinden, das in der Umgebungsvariablen `$PATH` genannt ist. Auf diese Details komme ich gleich ausführlicher zu sprechen.

Ärger mit der »Execution Policy«

Je nach Windows-Version wird der erste Versuch, ein eigenes Script auszuführen, mit der folgenden Fehlermeldung scheitern:

```
> .\Hello.ps1:

  File C:\Users\kofler\Documents\Hello.ps1 cannot be  loaded
  because running scripts is disabled on this system. For more
  information, see about_Execution_Policies at
  https://go.microsoft.com/fwlink/?LinkID=135170
```

Die Ursache für diese Fehlermeldung ist die sogenannte *Execution Policy*. Diese Richtlinie legt fest, unter welchen Umständen Scripts unter Windows ausgeführt werden dürfen. Es gibt vier mögliche Einstellungen:

- `Restricted`: Es dürfen gar keine Scripts ausgeführt werden.
- `AllSigned`: Nur signierte Scripts dürfen ausgeführt werden.
- `RemoteSigned`: Alle selbst erstellten Scripts dürfen ausgeführt werden; installierte bzw. heruntergeladene Scripts aber nur, wenn sie signiert wurden.
- `Unrestricted`: Alle Scripts dürfen ausgeführt werden.

Üblicherweise gilt unter Windows Server `RemoteSigned`, bei Desktop-Versionen aber `Restricted`:

```
> Get-ExecutionPolicy
  Restricted
```

Daher können auf vielen Windows-Installationen keine Scripts ausgeführt werden. Aus Sicherheitsgründen ist das eine sinnvolle Voreinstellung – aber sie ist natürlich ungeeignet, um Scripting zu lernen. Das folgende Kommando erlaubt für den aktuellen Benutzer die Ausführung eigener Scripts sowie signierter fremder Scripts:

```
> Set-ExecutionPolicy -Scope CurrentUser RemoteSigned
```

Sie können das obige Kommando auch ohne die Option `-Scope CurrentUser` ausführen – dann gilt die Einstellung für alle Benutzer des Rechners. Die Änderung der Execution Policy auf Systemebene ist aber nur erlaubt, wenn Sie die PowerShell bzw. das Terminal mit Administratorrechten ausführen.

Noch mehr Einstellungsmöglichkeiten

Die Execution Policy kann auf unterschiedlichen Ebenen (Rechner, Benutzer, Prozess) konfiguriert werden. Details zu den Konfigurationsmöglichkeiten finden Sie hier: *https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies*

Was bedeutet signiert?

Ein signiertes Script enthält einen Kommentarblock mit einer digitalen Signatur, also einem kryptografischen Code. Dieser gibt an, wer das Script verfasst hat. Der Code gilt nur für den Zustand des Scripts zum Zeitpunkt der Signatur. Jede nachträgliche Änderung macht die Signatur ungültig.

Eigene Scripts können Sie mit `Set-AuthenticodeSignature` signieren. Das setzt voraus, dass Sie über ein Zertifikat verfügen – entweder (für Testzwecke) ein selbst erstelltes oder ein richtiges Zertifikat von einer Zertifizierungsstelle. Mehr Informationen können Sie hier nachlesen:

https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_signing

Ein eigenes Script-Verzeichnis einrichten

Beim ersten Script spielt es keine Rolle, wo Sie es speichern. Aber bevor Sie in den nächsten Wochen immer mehr Scripts verfassen, ist es eine gute Idee, ein Verzeichnis für Ihre Scripts einzurichten und dieses Verzeichnis der Umgebungsvariablen `PATH` hinzufügen.

Ich gehe hier davon aus, dass Sie das Verzeichnis `myscripts` nennen und es direkt in Ihrem Arbeitsverzeichnis erstellen. `Set-Location` macht das Benutzerverzeichnis zum aktuellen Verzeichnis, sollte das nicht schon der Fall sein. `New-Item` erzeugt das Verzeichnis.(Wenn Sie bisher mit `cmd.exe` oder der Bash gearbeitet haben: `cd` und `mkdir myscripts` führen mit weniger Tippaufwand und Kopfzerbrechen auch ans Ziel. Persönlich verwende ich normalerweise diese Kommandos, auch weil sie mir von Linux und macOS vertraut sind. Aber ich bemühe mich in diesem Kapitel, Ihnen die »richtigen« PowerShell-CmdLets zu präsentieren.)

```
> Set-Location
> New-Item -ItemType Directory myscripts
```

Jetzt geht es noch darum, die `Path`-Variable einzurichten. Diese Variable enthält eine Liste aller Verzeichnisse, in denen die PowerShell nach ausführbaren Programmen und Scripts sucht. Wenn Sie das `myscripts`-Verzeichnis dieser Liste hinzufügen, können Sie dort gespeicherte Scripts ohne Pfadangabe ausführen. Ganz egal, welches Verzeichnis

gerade aktiv ist, reicht nun die Eingabe von `Hello`, um das Hello-World-Script zu starten. Dank der `Path`-Variable müssen Sie weder den Speicherort noch die Kennung `.ps1` angeben.

Zur Konfiguration dieser Variable suchen Sie im Startmenü nach **Systemumgebungsvariablen bearbeiten** und klicken dann im Dialog **Systemeigenschaften** auf den Button **Umgebungsvariablen**. Im gleichnamigen Dialog wählen Sie bei ihren Benutzervariablen `Path` aus und fügen mit **Bearbeiten • Neu • Durchsuchen** Ihr Script-Verzeichnis hinzu (siehe Abbildung 4.2).

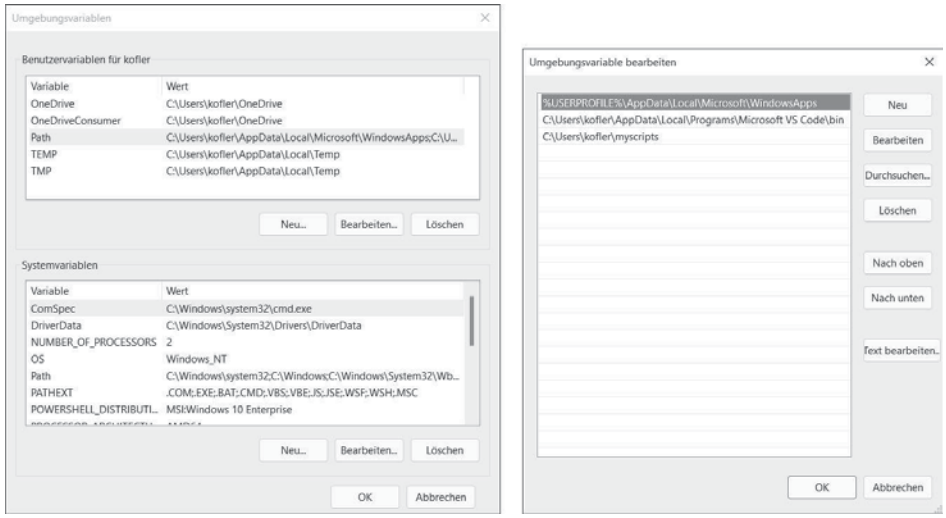

Abbildung 4.2: Eigenes Script-Verzeichnis zu »Path« hinzufügen

Beachten Sie, dass die Änderung von `Path` erst nach einem Neustart des Terminals bzw. der PowerShell wirksam wird. Anstatt `Path` umständlich in den Systemdialogen zu verändern, gelingt dies auch mit einem Kommando in der PowerShell. Passen Sie aber auf, dass Ihnen dabei keine Tippfehler unterlaufen!

```
> Set-Location myscripts

[Environment]::SetEnvironmentVariable("PATH",
  $env:PATH + ";$PWD", "User")
```

Script-Ausführung unter Linux und macOS

Ich gehe in diesem Kapitel davon aus, dass Sie unter Windows arbeiten. Selbstverständlich können Sie PowerShell-Scripts auch unter Linux oder macOS ausführen. Zwar gibt es dort keine *Execution Policy*, dafür müssen dort andere Regeln beachtet werden:

- Zum einen muss die Script-Datei mit der sogenannten Shebang-Zeile beginnen (siehe auch Abschnitt 3.4, »Das erste Bash-Script«):

```
#!/usr/bin/env pwsh
```

Diese Zeile gibt an, dass das Kommando `env` die Shell `pwsh` suchen und den folgenden Code damit ausführen soll. Für plattformunabhängige Scripts können Sie diese Zeile auch unter Windows verwenden. Sie gilt dort einfach als Kommentar, stört aber nicht.

- Zum anderen müssen Sie die Script-Datei mit `chmod` als »ausführbar« markieren (*executable*, daher `x`):

```
$ chmod  +x Hello.ps1      (Linux)
$ chmod a+x Hello.ps1      (macOS)
```

Wenn Sie sich um diese beiden Voraussetzungen gekümmert haben, können Sie das Script auch unter Linux und macOS ausführen:

```
$ ./Hello.ps1
  Hello, World!
```

Beispiel: Downloads-Verzeichnis aufräumen

Nach dem minimalistischen Hello-World-Script möchte ich im zweiten Beispiel nochmals eine Idee aus Abschnitt 4.4, »Kommandos kombinieren«, aufgreifen: Das Script `Tidy-Downloads.ps1` soll die *n* größten Dateien aus dem Downloads-Verzeichnis suchen und nach einer Rückfrage löschen. Auf diese Weise kann mit minimalem Zeitaufwand rasch Platz geschaffen werden. (Oft sind es ja nur wenige riesige Dateien, die den Großteil des Platzes beanspruchen.)

Im Vergleich zum Beispiel aus Abschnitt 4.4 gibt es zwei wesentliche Neuerungen:

- Das Script verwendet die Variable `DownPath`, die den Ort des Downloads-Verzeichnisses enthält. Der einfachste Weg, diesen Ort zu ermitteln, wäre `$DownPath = "$HOME\Downloads"`. Dieses Verfahren funktioniert allerdings nicht in jedem Fall (siehe *https://stackoverflow.com/questions/57947150*). Beispielsweise könnte es sein, dass manche Benutzer andere Orte für ihre Download-Verzeichnisse eingestellt haben. Die hier gewählte Vorgehensweise ist etwas umständlicher und verwendet die KnownFolders-API. Allerdings hat auch dieses Verfahren einen Nachteil: Es funktioniert nur unter Windows, nicht unter Linux oder macOS.
- An das Script kann über einen optionalen Parameter die Anzahl der zu löschenden Dateien übergeben werden. Fehlt der Parameter, sucht das Script nach den zehn größten Dateien.

Mehr Details zum Umgang mit Variablen und Parametern folgen im weiteren Verlauf des Kapitels. Die Grundkonzepte sollten aber klar sein: In der PowerShell werden Variablen mit einem vorangestellten `$`-Zeichen gekennzeichnet. Sämtliche Parameter eines Scripts oder einer Funktion werden mit `param()` deklariert, wobei Sie dabei einen Datentyp, einen Defaultwert sowie eine Menge anderer Zusatzinformationen festlegen können.

(Sofern Sie keine Funktionen verwenden, muss `param` die erste Anweisung im Script sein!)

```
# Beispieldatei Tidy-Downloads.ps1

# $NoOfFiles gibt an, wie viele Dateien gelöscht werden sollen
# (per Default: 10)
param([int] $NoOfFiles = 10)

# $DownPath enthält den Ort des Downloads-Verzeichnisses
$DownPath = (New-Object -ComObject Shell.Application).
               NameSpace('shell:Downloads').Self.Path

# die größten Dateien im Downloads-Verzeichnis mit
# Rückfrage löschen
Get-ChildItem -Path $DownPath |
     Sort-Object -Property Length -Descending |
     Select-Object -First $NoOfFiles |
     Remove-Item -Confirm
```

Kommentare

Wie bei den meisten Script-Sprachen werden Kommentare in PowerShell-Scripts mit dem Zeichen `#` eingeleitet und reichen dann bis zum Ende der Zeile. Mehrzeilige Kommentare werden mit `<#` eingeleitet und enden mit `#>`. Derartige Kommentare sind insbesondere dazu gedacht, Hilfetexte für `Get-Help` zu formulieren. Die Syntax ist hier dokumentiert:

https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comment_based_help

Differenzierung zwischen PowerShell-Versionen

Auf vielen Rechnern sind die PowerShell-Versionen 5.*n* und 7.*n* parallel installiert. Dennoch teilen sich alle Scripts die Dateiendung `*.ps1`. Wenn Ihr Code explizit eine bestimmte PowerShell-Version voraussetzt oder andere Anforderungen stellt, bauen Sie in das Script eine oder mehrere `#Requires`-Anweisungen ein:

- `#Requires -Version <n>`: Das Script kann nur mit der angegebenen oder einer neueren Version ausgeführt werden.
- `#Requires -PSEdition Core`: Das Script kann nur mit der Core-Variante ausgeführt werden (trifft für alle PowerShell-Versionen ab 6.0 zu).
- `#Requires -PSEdition Desktop`: Das Script kann nur mit der Desktop-Variante ausgeführt werden (alle Windows-Power-Shell-Versionen bis einschließlich 5.1).

- `#Requires -Modules <name>`: Das Script setzt das angegebene Modul voraus.
- `#Requires -RunAsAdministator`: Das Script kann nur mit Administratorrechten ausgeführt werden.

`#Requires`-Anweisungen können an einer beliebigen Stelle im Script platziert werden.

4.6 Variablen, Zeichenketten und Objekte

Erste Bekanntschaft mit PowerShell-Variablen haben Sie im vorigen Abschnitt gemacht. Prinzipiell ist der Umgang mit Variablen ganz einfach: Mit `$Name = ...` weisen Sie einer Variablen einen Inhalt zu. Später können Sie diesen Inhalt wieder verwenden. Um den Inhalt von Variablen auszugeben, können Sie `Write-Output` verwenden; noch einfacher ist es, die Variable oder eine Zeichenkette als eigenständige Anweisung zu nennen. Beachten Sie, dass Variablen in Zeichenketten, die zwischen doppelten Apostrophen stehen, durch ihren Inhalt ersetzt werden. Mehr Details zum Umgang mit Zeichenketten folgen später.

```
> $UserName = "Michael"

> Write-Output $UserName
  Michael

> $UserName
  Michael

> $UserId = 123

> Write-Output "$UserName hat die ID $UserId."
  Michael hat die ID 123.

> "$UserName hat die ID $UserId."
  Michael hat die ID 123.
```

Wenn Sie versehentlich eine nicht initialisierte Variable verwenden, verhält sich die PowerShell normalerweise gnädig: Die Variable ist leer, aber die Auswertung verursacht keinen Fehler:

```
> "$UserName hat die ID $UID."
  Michael hat die ID .
```

Allerdings können nicht initialisierte Variablen Logikfehler verursachen. Deswegen ist es vernünftig, am Beginn eines Scripts `Set-StrictMode -Version 1.0` einzubauen. Die Verwendung einer nicht initialisierten Variable führt dann zu einer Fehlermeldung. (Allerdings läuft das Script trotzdem weiter. Mehr Informationen zu `Set-StrictMode` und zum Umgang mit Fehlern in Scripts folgen dann in Abschnitt 4.13, »Fehlerabsicherung«.)

Variablennamen

Die PowerShell erlaubt nahezu alle Zeichen zur Benennung von Variablen, Funktionen usw. Wenn Sie Leer- oder Sonderzeichen verwenden möchten, ist die Schreibweise `${name}` zweckmäßig, also z. B. `${auch das ist ein Variablenname!}`. Die geschwungenen Klammern machen hier klar, wo der Variablenname beginnt und endet.

Trotz dieser syntaktischen Freiheiten ist es zweckmäßig, Variablennamen nur aus Buchstaben, Ziffern und wenigen Sonderzeichen (_) zusammenzusetzen.

Windows-typisch differenziert die PowerShell nicht zwischen der Groß- und Kleinschreibung. Sie könnten also im obigen Beispiel ebenso gut `$userid` oder `$USERID` auswerten – diese Namen sind für die PowerShell gleichwertig.

In Variablen können Sie auch das Ergebnis von anderen Kommandos speichern:

```
> $Images = Get-ChildItem *.jpg
```

Benutzereingaben führen Sie mit `Read-Host` durch:

```
> $Name = Read-Host "Geben Sie Ihren Namen an!"
  Geben Sie Ihren Namen an!: Michael

> "Hello, $Name!"
  Hello, Michael!
```

Datentypen

Der Versuch, mit einer per `Read-Host` eingegebenen Zahl zu rechnen, scheitert daran, dass die PowerShell zwischen verschiedenen Datentypen differenziert:

```
> $N = Read-Host "Eine Zahl, bitte"
  Eine Zahl, bitte: 123

> $N * 7
  123123123123123123123
```

Das Ergebnis von `Read-Host` wird als Zeichenkette betrachtet. Für Zeichenketten gilt `*` aber nicht als Multiplikation, sondern als Vervielfachung.

Abhilfe schafft die Deklaration der Variable mit einem passenden Datentyp (siehe Tabelle 4.2). Dazu stellen Sie den gewünschten Typ in eckigen Klammern voran. Für das obige Beispiel kommt `[int]` oder `[double]` in Frage. Beachten Sie, dass eine Eingabe wie »abc« nun einen Fehler auslöst.

```
> [int] $N = Read-Host "Eine Zahl, bitte"
  Eine Zahl, bitte: 123
```

```
> $N * 7
  861
```

Die Typangabe ist optional. Bei Variablenzuweisungen im Code wählt die PowerShell zumeist den richtigen Datentyp, was Sie später mit `Get-Member` verifizieren können:

```
> $Price = 100.0

> $Price | Get-Member
  TypeName: System.Double
  ...
```

Wenn Sie in einem Script je nach Datentyp unterschiedliche Code-Zweige durchlaufen möchten, helfen die Operatoren `-is` bzw. `-isnot` weiter:

```
> $Price -is [double]
  True
```

Typisierte Variablen haben aber den Vorteil, dass eine spätere Zuweisung von Daten im falschen Typ einen Fehler auslöst:

```
> [double] $Price = 100.0

> $Price = "abc"
  Error: Cannot convert value "abc" to type "System.Double".
```

Datentyp	Bedeutung
`bool`	boolescher Wert (`$true` oder `$false`)
`int/long`	32- bzw. 64-Bit-Integerzahl mit Vorzeichen
`float/double`	32- bzw. 64-Bit-Fließkommazahl
`decimal`	128-Bit-Festkommazahl
`String`	Zeichenkette

Tabelle 4.2: Die wichtigsten PowerShell-Datentypen

Rechnen und vergleichen

Sofern der Datentyp stimmt, funktionieren in der PowerShell alle gängigen Grundrechenarten. Die PowerShell unterstützt auch die Inkrement- und Dekrement-Operatoren `++` und `--` sowie mit einer Berechnung kombinierte Zuweisungen:

```
> $Price = 100.0
> $Price * 1.19 + 10
  129

> $Cnt = 27
> $Cnt++    # entspricht $Cnt = $Cnt + 1
> $Cnt+=3   # entspricht $Cnt = $Cnt + 3
```