Michael Kofler

# Scripting

## Das Praxisbuch für Administratoren & DevOps-Teams

+ Sicher mit Python, Bash und der PowerShell arbeiten

+ Automatisierungslösungen für Backups, Datenbanken, Web-Scraping und die Cloud

+ Tools & Workflows: SSH, Git, REST-APIs, RegExps, Filter und Pipes

# Liebe Leserin, lieber Leser,

wenn Sie die Komplexität der modernen IT mit all ihren Stolpersteinen kennen, werden Sie sicherlich zustimmen, dass der Schlüssel zur guten Programmierung darin besteht, schwierige Probleme in kleine, lösbare Schritte zu zerlegen. Diese Erkenntnis ist jedoch nicht neu. Bereits 1972 brachte Edsger Dijkstra in seiner Rede zur Verleihung des Turing-Preises den Gedanken präzise auf den Punkt:

> *I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs. If someone fears that this restriction is so severe that we cannot live with it, I can reassure him: the class of intellectually manageable programs is still sufficiently rich to contain many very realistic programs for any problem capable of algorithmic solution.*

Diesem Leitgedanken folgend sollten Programme so entwickelt werden, dass sie handhabbar und übersichtlich bleiben. In der UNIX-Welt spielte diese Philosophie eine besonders wichtige Rolle und trug dazu bei, dass sich der kleine, modulare Linux-Kernel gegenüber größeren und komplexeren Konkurrenten durchsetzen konnte.

Auch heute noch ist dieser Ansatz eine gute Richtlinie: Versuchen Sie, Probleme möglichst einfach zu lösen. Oftmals reichen wenige durchdachte Zeilen in einer Skriptsprache bereits für beeindruckende Ergebnisse aus, wie Ihnen Michael Kofler in diesem Buch beweisen wird. Und falls Sie den Vortrag *The Humble Programmer* von Edsger Dijkstra noch nicht kennen: Obwohl er inzwischen mehr als ein halbes Jahrhundert alt ist, kann er auch heute noch mit Gewinn gelesen werden.

Abschließend noch ein Wort in eigener Sache: Dieses Werk wurde mit großer Sorgfalt geschrieben, geprüft und produziert. Sollte dennoch einmal etwas nicht so funktionieren, wie Sie es erwarten, freue ich mich, wenn Sie sich mit mir in Verbindung setzen. Ihre Kritik und konstruktiven Anregungen sind jederzeit willkommen.

**Ihr Dr. Christoph Meister**
Lektorat Rheinwerk Computing

christoph.meister@rheinwerk-verlag.de
www.rheinwerk-verlag.de
Rheinwerk Verlag · Rheinwerkallee 4 · 53227 Bonn

# Hinweise zur Benutzung

Dieses E-Book ist **urheberrechtlich geschützt.** Mit dem Erwerb des E-Books haben Sie sich verpflichtet, die Urheberrechte anzuerkennen und einzuhalten. Sie sind berechtigt, dieses E-Book für persönliche Zwecke zu nutzen. Sie dürfen es auch ausdrucken und kopieren, aber auch dies nur für den persönlichen Gebrauch. Die Weitergabe einer elektronischen oder gedruckten Kopie an Dritte ist dagegen nicht erlaubt, weder ganz noch in Teilen. Und auch nicht eine Veröffentlichung im Internet oder in einem Firmennetzwerk.

Die ausführlichen und rechtlich verbindlichen Nutzungsbedingungen lesen Sie im Abschnitt *Rechtliche Hinweise*.

Dieses E-Book-Exemplar ist mit einem **digitalen Wasserzeichen** versehen, einem Vermerk, der kenntlich macht, welche Person dieses Exemplar nutzen darf:

Exemplar Nr. 7k3x-q9yh-tmdc-jivs
zum persönlichen Gebrauch für
Thomas Bartholomäus,
minduxde,
thomas.bartholomaeus@mindux.de

# Impressum

Dieses E-Book ist ein Verlagsprodukt, an dem viele mitgewirkt haben, insbesondere:

**Lektorat** Christoph Meister
**Korrektorat** Isolde Kommer, Großerlach
**Herstellung E-Book** Norbert Englert
**Covergestaltung** Lisa Kirsch
**Satz E-Book** Michael Kofler

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Bitte teilen Sie uns doch Ihre Meinung mit und lesen Sie weiter auf den *Serviceseiten*.

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

**ISBN 978-3-8362-9425-6 (E-Book)**
**ISBN 978-3-8362-9427-0 (Bundle)**

1. Auflage 2023

© Rheinwerk Verlag GmbH, Bonn 2023
www.rheinwerk-verlag.de

# Inhalt

Vorwort ... 11

## TEIL I Scripting-Sprachen

### 1 Scripting: Do one thing ... 17

1.1 Was heißt Scripting? ... 17
1.2 Script-Sprachen ... 20
1.3 Die Qual der Wahl ... 24

### 2 Zehn mal zehn Zeilen ... 27

2.1 Markdown-Rechtschreibkorrektur (Bash) ... 27
2.2 Bilder nach Aufnahmedatum sortieren (PowerShell) ... 28
2.3 JSON-Datei in das XML-Format umwandeln (Python) ... 29
2.4 Tägliches Server-Backup (Bash) ... 30
2.5 Web Scraping (Python) ... 31
2.6 Wetterdaten protokollieren (Python) ... 32
2.7 Hyper-V-Aufräumaktion (PowerShell) ... 33
2.8 Statistische Auswertung einer Logging-Datei (Bash) ... 34
2.9 Datei-Upload in die Cloud (PowerShell) ... 35
2.10 Virtuelle Maschinen klonen (Bash) ... 35

### 3 Bash und Zsh ... 37

3.1 Terminal, Shell und Bash ... 37
3.2 Installation ... 38
3.3 Kommandos interaktiv ausführen ... 42
3.4 Zsh als Bash-Alternative ... 45
3.5 Das erste Bash-Script ... 47
3.6 Kommandos ausführen ... 53
3.7 Standardeingabe und Standardausgabe ... 56

3.8 Globbing, Brace Extension, Umgang mit Datei- und Verzeichnisnamen ..... 59
3.9 Variablen ..... 63
3.10 Zeichenketten ..... 69
3.11 Verzweigungen ..... 77
3.12 Schleifen ..... 83
3.13 Funktionen ..... 87
3.14 Umgang mit Fehlern ..... 88

**4 PowerShell ..... 91**

4.1 Installation ..... 91
4.2 Das Windows-Terminal ..... 96
4.3 Aufruf von CmdLets und Funktionen ..... 98
4.4 Kommandos kombinieren ..... 105
4.5 Das erste Script ..... 108
4.6 Variablen, Zeichenketten und Objekte ..... 114
4.7 Arrays und Hashtables ..... 125
4.8 Ausgabeumleitung ..... 128
4.9 Schleifen ..... 132
4.10 Verzweigungen ..... 135
4.11 Funktionen und Parameter ..... 137
4.12 Module ..... 142
4.13 Fehlerabsicherung ..... 145

**5 Python ..... 149**

5.1 Python installieren ..... 150
5.2 Python in einem Terminalfenster kennenlernen ..... 152
5.3 Eigene Scripts programmieren ..... 153
5.4 Elementare Syntaxregeln ..... 155
5.5 Zahlen ..... 157
5.6 Zeichenketten ..... 159
5.7 Listen ..... 166
5.8 Tupel, Sets und Dictionaries ..... 169
5.9 Variablen ..... 172
5.10 Operatoren ..... 176
5.11 Verzweigungen (if) ..... 178
5.12 Schleifen (for und while) ..... 180

5.13 Funktionen ... 186
5.14 Textdateien verarbeiten ... 190
5.15 Fehlerabsicherung ... 193
5.16 Systemfunktionen ... 194
5.17 Module ... 197
5.18 Zusatzmodule installieren mit »pip« ... 199

## TEIL II Arbeitstechniken und Werkzeuge

### 6 Linux-Toolbox ... 207

6.1 Verzeichnisse und Dateien ... 208
6.2 Dateien suchen ... 209
6.3 Dateien komprimieren und archivieren ... 211
6.4 Arbeiten mit root-Rechten ... 213
6.5 Prozesse verwalten ... 215
6.6 Software-Installation ... 219
6.7 Sonstige Kommandos ... 220

### 7 CmdLets für die PowerShell ... 223

7.1 Verzeichnisse und Dateien ... 223
7.2 Dateien suchen ... 227
7.3 Dateien komprimieren und archivieren ... 231
7.4 Prozessverwaltung ... 232
7.5 Registrierdatenbank und Systeminformationen ... 236
7.6 CmdLet-Ergebnisse verarbeiten ... 239
7.7 Sonstige CmdLets ... 244
7.8 Zusatzmodule installieren ... 244
7.9 Standard-Aliasse ... 247

### 8 Textauswertung mit Filtern und Pipes ... 249

8.1 grep, sort, cut und uniq ... 250
8.2 Beispiel: Automatische Coronazahlenauswertung ... 257
8.3 Beispiel: ping-Auswertung ... 259
8.4 Beispiel: Apache-Log-Analyse ... 260
8.5 CSV-Dateien ... 264

**9 Reguläre Muster ... 269**

9.1 Syntaxregeln für reguläre Ausdrücke ... 270
9.2 Gruppen und Alternativen ... 272
9.3 Reguläre Muster in der Bash (grep, sed) ... 278
9.4 Reguläre Muster in der PowerShell ... 283
9.5 Reguläre Muster in Python (re-Modul) ... 287

**10 JSON, XML und INI ... 291**

10.1 JSON in der PowerShell ... 291
10.2 JSON und Python ... 295
10.3 JSON in der Bash ... 299
10.4 XML in der PowerShell ... 302
10.5 XML und Python ... 306
10.6 XML in der Bash ... 310
10.7 INI-Dateien ... 312

**11 Scripts automatisch ausführen ... 315**

11.1 Cron ... 315
11.2 Beispiel: Webserver-Monitoring ... 319
11.3 Windows Task Scheduler ... 321
11.4 Beispiel: Währungskurse speichern ... 326
11.5 Änderungen im Dateisystem verfolgen ... 327

**12 SSH ... 331**

12.1 SSH-Client und -Server installieren ... 332
12.2 Mit SSH arbeiten ... 335
12.3 scp und rsync ... 340
12.4 SSH-Authentifizierung mit Schlüsseln ... 342
12.5 Beispiel: Bilder-Upload auf einen Linux-Webserver ... 345
12.6 Beispiel: Auswertung virtueller Maschinen ... 348

**13 Visual Studio Code ... 351**

13.1 Einführung ... 351
13.2 Sprachspezifische VSCode-Erweiterungen ... 353
13.3 Remote-SSH-Erweiterung ... 355

**14 Git** ..... 359

14.1 Git-Crashkurs ..... 360
14.2 Der richtige Umgang mit Einstellungen und Passwörtern ..... 368
14.3 Git-Automatisierung ..... 371
14.4 Git Hooks ..... 374

**TEIL III Anwendungen und Beispiele**

**15 Backups** ..... 379

15.1 Verzeichnisse auf externen Datenträger synchronisieren ..... 379
15.2 WordPress-Backup ..... 384
15.3 SQL-Server-Backup ..... 387

**16 Bildverarbeitung** ..... 389

16.1 Bilddateien manipulieren ..... 389
16.2 Fotos nach Aufnahmedatum sortieren ..... 394
16.3 EXIF-Metadaten in SQL-Kommandos umwandeln ..... 397

**17 Web Scraping** ..... 403

17.1 Websites mit wget herunterladen ..... 405
17.2 Web Scraping mit regulären Mustern ..... 408
17.3 Web Scraping mit Beautiful Soup ..... 409
17.4 Web Scraping mit Requests-HTML ..... 414
17.5 Web Scraping mit der PowerShell ..... 417

**18 REST-APIs nutzen** ..... 419

18.1 curl und wget ..... 420
18.2 REST-APIs in der PowerShell nutzen ..... 425
18.3 Beispiel: Aktuelles Wetter ermitteln ..... 428
18.4 REST-APIs in Python nutzen ..... 429
18.5 Beispiel: Strompreise ermitteln und grafisch darstellen ..... 430

**19 Datenbanken** ... 435

19.1 Datenbanken aktualisieren und warten ... 436
19.2 Neuen Kunden-Account einrichten ... 437
19.3 EXIF-Metadaten in einer Datenbank speichern ... 440
19.4 JSON-Daten in eine Tabelle importieren ... 444

**20 Scripting in der Cloud** ... 447

20.1 AWS CLI ... 447
20.2 Beispiel: Verschlüsselte Backup-Dateien in die Cloud hochladen ... 452
20.3 AWS-PowerShell-Modul ... 454
20.4 Beispiel: Große Dateien einer Webseite in die Cloud auslagern ... 457

**21 Virtuelle Maschinen** ... 461

21.1 Virtuelle Maschinen einrichten und ausführen (KVM) ... 461
21.2 Netzwerkkonfiguration automatisieren (KVM) ... 464
21.3 Hyper-V steuern ... 467

**22 Docker und Scripting** ... 473

22.1 Beispiel: EXIF-Sorter als Docker-Image ... 474
22.2 Beispiel: Markdown/Pandoc-Umgebung ... 477

Index ... 485

# Vorwort

Ein Script ist ein kleines Stück Code zur Lösung eines konkreten Problems oder zur Automatisierung einer lästigen Aufgabe. Sie brauchen zur Entwicklung eines Scripts weder eine Entwicklungsumgebung noch einen Compiler – ein Editor genügt. Beim Scripting ist Minimalismus die Devise. Es geht darum, mit minimalem Mittel maximale Wirkung zu erzielen. Salopp formuliert:

**Scripting ist die Kunst,**
**IT-Probleme in zehn Zeilen Code zu lösen.**

In meinem Arbeitsumfeld sind Scripts allgegenwärtig. Ich verwende Scripts, um den Energiesparmodus meines Notebooks einzustellen, um aus Markdown-Dateien die PDF-Datei für den Druck dieses Buchs zu erzeugen, um auf Servern das Backup zu automatisieren, um E-Books mit Wasserzeichen auszustatten, um in einer Webapplikation neue Kunden einzurichten, um Dutzende von Datenbanken nach logischen Fehlern zu durchsuchen, um virtuelle Maschinen für den Unterricht zu erstellen, um Fotos zu sortieren usw.

## Bash, PowerShell oder Python?

Die »klassische« Script-Sprache ist die Bash. Ihr Name *Bourne Again Shell* ist ein Wortspiel. Der Vorgänger war die unter Unix verbreitete Bourne-Shell. Für Linux war die Lizenz der Bourne-Shell ungeeignet, weswegen ein neues Projekt entstand. Heute ist die Bash die dominierende Linux-Shell, sowohl im interaktiven Betrieb zur Ausführung von Kommandos als auch zur Script-Programmierung. macOS setzt auf die weitgehend kompatible Zsh, die auch unter Linux immer mehr Anhänger findet.

Die Bash ist populär, Ihre antiquierte Syntax gewinnt aber keine Schönheitspreise. Im Gegenteil, Bash-Scripts sehen mitunter grauenhaft aus. Insofern kann man verstehen, dass Microsoft gar nicht erst versucht hat, die Bash für Windows zu adaptieren. Stattdessen hat Microsoft mit der PowerShell Grundideen klassischer Unix-Shells mit den Konzepten objektorientierter Programmiersprachen zu einer vollkommen neuen Sprache kombiniert. Das ist überraschend gut gelungen! Nicht ohne Grund schwören Windows-Administratorinnen und -Administratoren auf die PowerShell.

Python ist eigentlich keine typische Script-Sprache. Je nach Einsatzzweck ist Python die Basis für KI-Entwicklungen, ein Tool für (Natur-)Wissenschaftler oder die erste Sprache für den Einstieg in die Programmierung. Es gibt wohl keine andere Sprache, die so universell eingesetzt wird! Die Eleganz der Python-Syntax und das schier uner-

schöpfliche Angebot von Erweiterungsmodulen haben dazu geführt, dass Python auch zur Systemadministration, zur Umwandlung von Dateien zwischen verschiedenen Formaten, als Datenbank-Tool oder zur Hardware-Steuerung (Raspberry Pi) verwendet wird. Python spielt seine Stärken umso mehr aus, je komplexer die Aufgabenstellung ist.

## Über dieses Buch

Weil es nicht die eine perfekte Script-Sprache gibt, startet dieses Buch nach der Einleitung mit Crash-Kursen für die Sprachen Bash, PowerShell und Python. Wenn Sie möchten, können Sie sich für den Start auf eine dieser drei Sprachen konzentrieren und Ihren Sprachwortschatz nach und nach vergrößern.

Teil II des Buchs stellt Werkzeuge und Arbeitstechniken vor, die Sie in Scripts typischerweise verwenden: Dazu zählen Kommandos zur Verarbeitung von Textdateien, CmdLets zur Anwendung regulärer Ausdrücke und Funktionen zum Umgang mit JSON- und XML-Dateien. Ich zeige Ihnen, wie Sie Scripts regelmäßig und automatisiert ausführen, wie Sie dank SSH Code auch auf anderen Rechnern ausführen bzw. Dateien dorthin kopieren und wie Sie dank Git die Versionsverwaltung Ihres Codes und Scripting kombinieren. Viele praktische Beispiele runden das Informationsangebot an.

In Teil III geht es schließlich um konkrete Anwendungen: Zu den wichtigsten Themen zählen Backups, Bildverarbeitung, Web Scraping, die Nutzung von REST-APIs, Datenbanken, die Nutzung der Cloud und die Administration virtueller Maschinen.

Ich habe mich bemüht, das Buch so gut wie möglich nach einem Baukastensystem zusammenzusetzen. Sie müssen also nicht alle Kapitel linear lesen, sondern können gezielt nach den Informationen suchen, die Ihnen gerade wichtig sind. Alle Kapitel von Teil II und III beginnen mit einem kurzen Informationskasten, der die Voraussetzungen für das Kapitel zusammenfasst.

## Scripting und KI-Tools

In den letzten Monaten hat ChatGPT einen Umbruch in vielen Berufen eingeleitet. Wie so oft hängt es von der Sichtweise ab, ob Sie das Glas halb voll oder halb leer sehen: KI-Tools sind heute alles andere als perfekt, aber sie sind definitiv (schon jetzt und noch mehr in den kommenden Jahren) ein Hilfsmittel, das die Software-Entwicklung revolutionieren wird.

Wenn Sie ChatGPT oder den GitHub Copilot in VSCode um ein PowerShell-Script bitten, das eine JSON-Datei lesen und die darin enthaltenen Daten in einer PostgreSQL-Datenbank speichern soll, wird es Ihnen innerhalb von Sekunden ein plausibel aussehendes Script ausspucken.


© Rheinwerk Verlag, Bonn 2024

Im Idealfall funktioniert das Script auf Anhieb. Nach meinen Erfahrungen ist der Code aber selten optimal – umso weniger, je komplexer die Fragestellung ist. Oft enthält der Code eindeutige Fehler, z. B. nicht existierende Optionen oder Kommandos. Mitunter gibt es auch logische Fehler, die schwerer zu erkennen sind. Mit etwas Scripting-Erfahrung lassen sich die Probleme zumeist rasch beheben. Sie können das KI-Tool sogar ganz gezielt bitten, einen bestimmten Aspekt des Codes zu verbessern.

Mein persönlicher Eindruck ist, dass sich KI-Tools umso besser einsetzen lassen, je solider das vorhandene Vorwissen ist. Ein KI-Tool und eine erfahrene Entwicklerin, ein routinierter Programmierer – das ist aus meiner Sicht das Dream-Team der Zukunft. Dieses Buch vermittelt genau das dafür erforderliche Fundament, ohne sich mit allzu vielen enzyklopädischen Details aufzuhalten. Für das Programmieren der Zukunft müssen Sie nicht hundert Syntaxvarianten von regulären Ausdrücken auswendig kennen. Aber Sie müssen wissen, dass es reguläre Ausdrücke gibt, wie sie prinzipiell funktionieren und wo die Fallstricke sind. Genau da liegt der Fokus dieses Buchs.

### Scripting als Kernkompetenz für effiziente IT-Arbeit

Ganz egal, in welcher Sparte der IT-Industrie Sie tätig sind, an welchen Projekten Sie gerade arbeiten: Ein wenig Scripting-Know-how macht Sie in Ihrer Arbeit noch effizienter. Konzentrieren Sie sich auf das Wesentliche, überlassen Sie die lästigen Nebenaufgaben einem Script! Dabei wünsche ich Ihnen viel Erfolg!

Michael Kofler (*https://kofler.info*)

PS: Alle Beispieldateien zu diesem Buch finden Sie hier zum Download:

*https://rheinwerk-verlag.de/5672*

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

# TEIL I

# Scripting-Sprachen

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

# Kapitel 1
# Scripting: Do one thing ...

Dieses kurze Einführungskapitel beschäftigt sich mit der Frage: »Was ist Scripting?« Es erklärt die Unterschiede zwischen »vollwertigen« Programmiersprachen und klassischen Scripting-Sprachen und zeigt gleich, dass diese Grenzen (speziell bei Python) fließend sind.

In diesem Kapitel ist es mir ein Anliegen, ein bisschen auf die Hintergründe und die Philosophie des Scriptings einzugehen. Besonders wichtig erscheint mir das, wenn Sie Ihren Code bisher mit einer Sprache wie Java oder C# entwickelt haben. Sie haben gelernt, die richtigen Datentypen zu verwenden und Ihren Code objektorientiert zu strukturieren. Sie haben sich, so gut es geht, an alle Regeln der Kunst gehalten.

Was spricht dann dagegen, das Backup-Script oder ein Testprogramm für eine REST-API ebenfalls in einer streng typisierten Programmiersprache zu entwickeln? Welche Vorteile bietet eine syntaktisch wesentlich großzügigere Script-Sprache wie die Bash, die als einzigen Datentyp Zeichenketten kennt? Warum sollten Sie sich mit der PowerShell anfreunden, wenn diese doch das gleiche .NET-Fundament nutzt wie C#?

## 1.1 Was heißt Scripting?

Scripting ist,

- wenn 20 Zeilen Code ausreichen, um einmal täglich ein Backup Ihrer Datenbank zu erstellen, dieses zu verschlüsseln und in einen Cloud-Speicher hochzuladen,
- wenn Sie die unzähligen Fotos einer Hochzeit mit einem winzigen Programm auf eine Auflösung von max. 1024 × 768 Pixel reduzieren, mit einem Wasserzeichen versehen und in ein verstecktes Verzeichnis Ihres Webservers hochladen können, damit Ihre Kundinnen und Kunden die Bilder dort ansehen und auswählen,
- wenn Sie unkompliziert alle Rechner im lokalen Netzwerk nach einer bestimmten Sicherheitslücke oder einer veralteten Software-Version durchsuchen,

Persönliches Exemplar für Thomas Bartholomäus

- wenn Sie den Speicherstand von Deutschlands Gasspeicher, die Preisentwicklung Ihres auserkorenen nächsten Notebooks oder die neuesten Corona-Fallzahlen von öffentlichen Websites abgreifen und in einem Diagramm visualisieren,
- wenn Sie aus einer Logging-Datei mit 50.000 Zeilen die 20 für Sie relevanten Fehlermeldungen herausfiltern.

### Scripting versus Programmierung

Scripts sind also auch »nur« Programme. Sie unterscheiden sich aber durch die Art der Programmierung, durch die eingesetzten Werkzeuge/Komponenten und durch ihre Zielsetzung von großen Software-Projekten:

- Typischerweise erfüllen Scripts überschaubare, relativ einfache Aufgaben. Oft helfen sie dabei, administrative Aufgaben zu automatisieren oder zumindest effizienter durchzuführen.
- Scripts werden im Textmodus (im Terminal) verwendet oder im Hintergrund automatisiert ausgeführt. Es gibt keine grafische Benutzeroberfläche.
- Zur Programmierung kommen spezielle Script-Sprachen zum Einsatz, die ein unkompliziertes, effizientes Entwickeln möglich machen. Script-Sprachen erfordern weder aufwendige Entwicklungswerkzeuge noch einen Compiler. Die Syntax ist minimalistisch (mitunter leider auch veraltet und gewöhnungsbedürftig). Zentrale, aus großen Projekten vertraute Grundsätze wie Objektorientierung oder eine strenge Typenkontrolle der Variablen spielen eine untergeordnete Rolle oder lassen sich gar nicht realisieren.
- Populäre Script-Sprachen zeichnen sich dafür durch ein riesiges Angebot von Kommandos oder Erweiterungsmodulen aus. Das hilft bei der Durchführung grundlegender Arbeitsschritte (Dateien lesen und auswerten, Benutzer einrichten, Netzwerkoperationen durchführen etc.). Beim Scripting haben Sie keine Zeit, das Rad neu zu erfinden! Sie bedienen sich so gut wie möglich aus einer riesigen Toolbox vorhandener Komponenten und Bausteine.
- Der Codeumfang von Scripts ist klein, typischerweise unter 100 Zeilen (Kommentare und Zeichenketten nicht mitgerechnet).

Bei großen Software-Projekten ist es äußerst wichtig, dass Code »sauber« entwickelt wird, dass er von allen Mitgliedern im Team verstanden und später erweitert oder modifiziert werden kann, dass also sämtliche Richtlinien modernen Software-Designs beachtet werden.

Beim Scripting geht es hingegen darum, ein kleines Problem rasch und pragmatisch zu lösen (*Getting Things Done*). Das heißt natürlich nicht, dass Sie sich bemühen sollen, Ihre Scripts schlampig zu entwickeln! Auch Scripts sollen aussagekräftige Varia-


© Rheinwerk Verlag, Bonn 2024

blennamen verwenden, mit Kommentaren dokumentiert werden, eine grundlegende Fehlerabsicherung enthalten usw. Aber die Prioritäten und Entwicklungsziele eines Scripts, das möglichst innerhalb eines Tags fertig sein soll, sind ganz andere als für ein Software-Projekt, dessen Code womöglich ein Jahrzehnt lang gewartet werden muss.

### Glue Languages und Glue Code

Im Zusammenhang mit Scripting ist manchmal von »Glue Code« die Rede. Entsprechend werden Scripting-Sprachen gelegentlich als »Glue Languages« bezeichnet. Was bedeuten diese Begriffe?

In größeren Projekten ist oft sprichwörtlich *Glue* (englisch Kleber) notwendig, um voneinander unabhängige oder miteinander inkompatible Software-Bausteine zu verbinden. Im einfachsten Fall ruft ein Script einige externe Kommandos auf, die an sich nichts miteinander zu tun haben: Eines erstellt ein Backup einer Datenbank, das zweite verschlüsselt die resultierende Datei, das dritte überträgt die Datei per HTTP auf einen anderen Server usw. Jedes der eingesetzten Kommandos wurde unabhängig von allen anderen entwickelt. Aber indem Sie die Kommandos durch Ihr Script verbinden, entsteht eine neue, sinnvolle Komponente.

Glue Code ist manchmal auch notwendig, um moderne Werkzeuge zur Software-Entwicklung effizient anzuwenden. Beispielsweise entwickelt Ihr Team ein großes Projekt in JavaScript und verwendet dabei Werkzeuge wie Git (Versionskontrolle) und Docker (Container für lokale Testumgebungen). Jedes Mal, wenn ein Team-Mitglied eine neue Testversion fertigstellt (»eincheckt«), soll der aktuelle Code auf einen externen Test-Server übertragen (»deployed«) werden. Diesen Arbeitsschritt könnte ein kleines Script erledigen.

Theoretisch können Sie derartige Aufgaben mit jeder Programmiersprache erledigen. Besonders gut geeignet sind aber Script-Sprachen – wegen ihrer einfachen Syntax, des minimalen Overheads bei der Entwicklung und des Umstands, dass die Scripts keine oder nur wenige neue Projektabhängigkeiten verursachen.

### Do One Thing and Do It Well

Unix-Programme wurden unter dem von Doug McIlroy formulierten Motto *Do One Thing and Do It Well* entwickelt und werden auch heute noch nach diesem Maßstab bewertet. Aus Unix wurde Linux und macOS, was damals Programm genannt wurde, ist heute ein Kommando. Aber am Prinzip hat sich nichts geändert: Von einem Kommando wie `ls`, `grep` oder `find` wird erwartet, dass es eine ganz spezifische Aufgabe erfüllt – und diese richtig gut.

Persönliches Exemplar für Thomas Bartholomäus

Was hat das Unix-Motto mit Scripting zu tun? Wenn Sie unter Linux und macOS Scripts in den Sprachen Bash oder Zsh entwickeln, tun Sie dies auf einem Fundament von mehreren Hundert Kommandos, die der Unix-Empfehlung entsprechen. Sie sind gut beraten, das Motto auch für Ihre eigenen Projekte zu übernehmen. Schreiben Sie Scripts, die *eine* Aufgabe ordentlich erfüllen.

## 1.2 Script-Sprachen

Rein formal unterscheiden sich Script-Sprachen von anderen, »höheren« Programmiersprachen dadurch, dass der Code interpretiert wird. Der Code wird also in einer Textdatei formuliert und dann direkt durch die Bash, die PowerShell oder den Python-Interpreter ausgeführt. Der Code braucht nicht vorher kompiliert (also in eine binäre Darstellung umgewandelt) werden.

Dieses Konzept hat den Vorteil, dass Scripts ohne lange Vorbereitungsarbeiten sofort ausgeführt werden können. Das beschleunigt den Entwicklungsprozess.

Die Verwendung eines Interpreters hat allerdings den Nachteil, dass Scripts zumeist etwas langsamer laufen als kompilierte Programme. Deswegen ist eine Script-Sprache selten die ideale Wahl, um rechenintensive Algorithmen zu entwickeln. Da viele Scripts überwiegend aus Aufrufen anderer Kommandos bestehen, spielt der Effizienzverlust durch den fehlenden Compiler gar keine Rolle.

Als »klassische« Script-Sprachen gelten sämtliche Linux-Shells, also z. B. Bourne Shell, die Korn Shell, Bash und Zsh. Eine Shell ist eigentlich ein Kommandointerpreter, also ein Programm, das Kommandos entgegennimmt und diese ausführt. Werden mehrere solche Kommandos in einer Textdatei gespeichert, entsteht die ursprüngliche Form eines Scripts.

Im Laufe der Zeit wurden unzählige Script-Sprachen entwickelt, die mehr syntaktische Möglichkeiten als traditionelle Shells boten und oft für spezifische Aufgaben optimiert wurden. Dazu zählen beispielsweise JavaScript, Python, PHP und Tcl.

In der Windows-Welt übernahm ursprünglich das unsägliche Programm `cmd.exe` die Rolle der Shell. Darauf aufbauende `*.bat`-Dateien sind trotz der äußerst bescheidenen Scripting-Möglichkeiten bis heute im Einsatz. Es folgten VBScript, die für den Office-Einsatz optimierte Sprache VBA und schließlich die PowerShell. Erst damit hatte Microsoft Erfolg: Die PowerShell gilt heute als *die* Sprache, wenn es darum geht, große Windows-Netzwerkinstallationen zu warten und zu administrieren.


© Rheinwerk Verlag, Bonn 2024

### Compiler für Script-Sprachen

Das in der Vergangenheit etablierte Kriterium Interpreter/Compiler zur Unterscheidung zwischen Script- und anderen Sprachen ist heute obsolet. Für viele Sprachen, deren Code anfänglich durch einen Interpreter ausgeführt wurde, gibt es mittlerweile Compiler. Oft handelt es sich um *Just-in-Time-Compiler*, die den Code unmittelbar vor der Ausführung und unbemerkt durch die Anwenderinnen und Anwender kompilieren. Das trifft unter anderem für JavaScript, PHP und Python zu.

### Bash und Zsh

Es liegt auf der Hand, dass ich in diesem Buch nicht auf alle populären Script-Sprachen eingehen kann. Ich habe mich vielmehr auf drei (mit der Zsh vier) Sprachen fokussiert, die für administrative Aufgaben sowie im DevOps-Umfeld am wichtigsten sind. Diese Sprachen möchte ich Ihnen im Folgenden kurz vorstellen.

Der Name *Bash* ist eine Abkürzung für *Bourne Again Shell*. Die Bourne Shell war vor mehr als 30 Jahren unter Unix sehr beliebt. Allerdings stand dieses Programm nicht unter einer Open-Source-Lizenz zur Verfügung. Das führte zur Entwicklung der weitgehend kompatiblen Bash, die sich später als Standard-Shell bei den meisten Linux-Distributionen durchsetzte.

Wenn im Linux-Umfeld ohne weitere Erläuterungen von *Scripting* die Rede ist, dann ist als Scripting-Sprache nahezu immer die Bash gemeint. Egal, ob Server-Prozesse gestartet, Netzwerkverbindungen eingerichtet oder Firewall-Regeln verändert werden sollen – ganz häufig kommen dazu bereits auf Betriebssystemebene Bash-Scripts zum Einsatz. Daher ist es naheliegend, auch eigene Aufgaben mit der Bash zu erledigen.

Die weite Verbreitung der Bash lässt manchmal darüber hinwegsehen, dass die Wurzeln und die Syntax der Bash sehr alt sind. Dementsprechend ist die Syntax der Sprache mitunter inkonsistent, ab und zu auch einfach grauenhaft. Statt einfacher Funktionen müssen unzählige Sonderzeichen herhalten, um ganz triviale Aufgaben zu erledigen (Zeichenketten bearbeiten, Berechnungen durchführen). Neben Zeichenketten und Arrays gibt es keine weiteren Datentypen. Objektorientierung ist in der Bash sowieso ein Fremdwort.

Auf der Plusseite steht die schier grenzenlose Auswahl von Unix-Tools, die in Scripts genutzt und kombiniert werden können. Die Stärke der Bash liegt also nicht ihren sprachlichen Möglichkeiten, sondern bei den Kommandos, die Sie in Scripts unkompliziert aufrufen können. (Und Sie wissen ja schon: Diese Kommandos wurden unter dem Motto *Do One Thing* … entwickelt.)

Persönliches Exemplar für Thomas Bartholomäus

Einschränkend muss ich noch erwähnen, dass es Einsteigerinnen und Einsteigern schwerfällt, sich in der Bash- und Linux-Kommandowelt zu orientieren. Zwar ist (fast) jedes Kommando für sich gut dokumentiert, aber es gibt keine zentrale Übersicht. (Das begründet den Erfolg meines Buchs »Die Linux-Kommandoreferenz«, das im Rheinwerk Verlag bereits mehrere Auflagen erlebt hat.)

### Bash versus Zsh

Die Zsh ist weitgehend kompatibel zur Bash. Für die Script-Programmierung sind die Unterschiede nur minimal, und natürlich können Sie in beiden Shells die gleichen Kommandos aufrufen. Bei der interaktiven Verwendung zeichnet sich die Zsh aber durch viele Vorteile und bessere Erweiterungsmöglichkeiten aus. Deswegen gewinnt die Zsh unter Linux immer mehr Fans und wird von manchen Distributionen sogar schon als Default-Shell verwendet. (Bei anderen Distributionen kann die Zsh mit wenigen Handgriffen installiert werden.)

macOS hat 2019 einen Wechsel von der Bash auf die Zsh vollzogen. Die Motivation von Apple lag weniger bei den technischen Vorzügen, sondern hatte vielmehr mit Lizenzfragen zu tun: Aktuelle Versionen der Bash verwenden die Lizenz GPL 3, die Apple vermeidet. Die Zsh hat dagegen eine liberalere, BSD-artige Lizenz.

Soweit es dieses Buch betrifft, spielt es keine große Rolle, ob Sie die Bash oder die Zsh bevorzugen. Einen kurzen Überblick der wichtigsten Vor- und Nachteile gebe ich in Abschnitt 3.4, »Zsh als Bash-Alternative«.

### PowerShell

Microsoft hat lange Zeit auf grafische Benutzeroberflächen gesetzt – nicht nur im Office-Bereich, sondern auch zur Server-Administration. Auf den ersten Blick schien das ein Vorteil im Vergleich zu Linux zu sein: Ein paar Mausklicks erschließen sich leichter als dubiose Konfigurationsdateien.

Für Administratorinnen und Administratoren hat sich das zum Albtraum entwickelt: Das Hauptproblem besteht darin, dass die Konfigurationsarbeit nicht skaliert. Zehn Server zu administrieren, dauert per Mausklick eben zehn Mal länger als bei einem Server. Anders als unter Linux gab es kaum Möglichkeiten, derartige Arbeiten zu automatisieren.

Das änderte sich mit der Präsentation der PowerShell 2006. Microsoft hat die Gelegenheit des Neuanfangs gut genutzt: Im Vergleich zur Bash punktet die PowerShell mit einer deutlich logischeren Syntax. Das technisch interessanteste Feature der PowerShell besteht darin, dass der Datentransport von einem Kommando zum nächsten nicht in Textform erfolgt, sondern dass dabei vollwertige Objekte übertragen werden. Das ermöglicht weitreichende Verarbeitungsmöglichkeiten durch das Ausle-


© Rheinwerk Verlag, Bonn 2024

sen von Eigenschaften und den Aufruf von Methoden. Der objektorientierte Ansatz funktioniert allerdings nur mit speziell für die PowerShell optimierten Kommandos, die Microsoft *CmdLets* nennt. (Der Aufruf traditioneller Kommandos ist auch möglich, unterliegt aber Einschränkungen.)

Ein weiterer Erfolgsfaktor der PowerShell ist im Umfeld zu suchen: Microsoft hat begonnen, viele Windows-Komponenten und Server-Dienste vollständig durch CmdLets konfigurierbar zu machen. Während früher maximal Grundeinstellungen per Script verändert werden konnten, andere Optionen aber doch nur per Mausklick erreichbar waren, gilt jetzt: *PowerShell first*.

Neben den standardmäßig ausgelieferten CmdLets gibt es im Internet unzählige Erweiterungsmodule mit CmdLets für bestimmte Aufgaben. Rund um die PowerShell ist eine aktive Community entstanden. Seit 2018 ist die PowerShell zudem ein Open-Source-Projekt und kann auch unter Linux und macOS installiert werden. Allerdings ist das CmdLet-Angebot außerhalb der Windows-Welt deutlich kleiner. Typische administrative Aufgaben (Benutzer einrichten, Netzwerkkonfiguration ändern usw.) funktionieren nur unter Windows. Plattformübergreifend lässt sich die PowerShell nur für Aufgaben nutzen, die nicht Windows-spezifisch sind.

### Python

Die erste Python-Version wurde 1991 veröffentlicht. Damit ist Python beinahe so alt wie die Bash, deren erste Version 1989 erschien. Aber anders als der Bash merkt man Python das Alter kaum an: Python zeichnet sich durch eine elegante, gut durchdachte Syntax aus, die bis heute Maßstäbe setzt.

Python ist insofern eine Script-Sprache, als der Code ursprünglich von einem Interpreter ausgeführt wurde. Bei aktuellen Versionen wird der Code aus Performance-Gründen allerdings zuerst in ein binäres Zwischenformat (den sogenannten Byte-Code) kompiliert. Bei der Anwendung von Python bemerken Sie davon nichts. Anders formuliert: Python verhält sich wie eine interpretierte Sprache, benutzt hinter den Kulissen aber sehr wohl einen Compiler.

Python wurde nicht in erster Linie zur Automatisierung von administrativen Vorgängen konzipiert. Python ist vielmehr äußerst universell verwendbar. Sie können mit Python gleichermaßen programmieren lernen oder KI-Probleme lösen!

Ein Grundkonzept von Python besteht darin, dass der Sprachkern sehr kompakt ist. Dafür kann die Sprache einfach durch Module erweitert werden. Diese Module sind der Grund, warum Python heute (auch) als Script-Sprache im Sinne dieses Buchs so populär ist: Im Laufe der Zeit entstanden immer mehr Erweiterungsmodule, um Cloud-Dienste zu nutzen, Netzwerkfunktionen anzuwenden, auf Datenbanken zuzu-

Persönliches Exemplar für Thomas Bartholomäus

greifen usw. Für nahezu jede denkbare administrative Aufgabe kann im Handumdrehen ein passendes Python-Modul installiert werden!

Python ist allerdings nur mäßig geeignet, um vorhandene Kommandos aufzurufen. Insofern sind die Module zugleich Fluch und Segen. Während Ihnen die für ein Bash-Script erforderlichen Kommandos vielleicht schon bekannt sind, müssen Sie sich in ein adäquates Python-Modul mit ähnlichen Funktionen erst einarbeiten. Oft kommen mehrere Module infrage. Nicht immer ist klar, welches Modul besser geeignet ist, welches auch in Zukunft noch gewartet wird. Insofern lohnt sich der Einsatz von Python vor allem dann, wenn die Aufgabenstellung einigermaßen komplex ist, wenn die Vorteile von Python den Nachteil einer längeren Einarbeitung in ein bestimmtes Zusatzmodul kompensieren.

### Viele Ähnlichkeiten, noch mehr Unterschiede

Die Behandlung von gleich *drei* Script-Sprachen in einem Buch ist zugegebenermaßen eine intellektuelle Herausforderung – für den Autor ebenso wie für Sie, die Leserin oder den Leser! Natürlich zeichnen sich alle drei Sprachen durch viele Gemeinsamkeiten aus. Im besonderen Maß gilt dies für Bash und PowerShell. Gleichzeitig gibt es aber unzählige syntaktische Abweichungen, die den raschen Wechsel zwischen den Sprachen mühsam machen. Mein Tipp: Verwenden Sie einen Editor, der die jeweilige Script-Sprache gut unterstützt – dann erkennt der Editor die meisten Fehler bzw. Syntaxverwechslungen bereits vor dem ersten Testlauf.

## 1.3 Die Qual der Wahl

Wenn Sie gerade in die Welt des Scriptings einsteigen, wäre es Ihnen vermutlich am liebsten, ich könnte Ihnen hier sagen: »Lernen Sie die Sprache Xxx, die ist für alle Zwecke geeignet.« Leider ist die IT-Welt nicht so einfach. Welche Script-Sprache überhaupt oder sogar ideal geeignet ist, hängt sehr stark von der Aufgabenstellung und dem Betriebssystem ab, auf dem Ihr Script laufen soll.

- Für Scripts zur Administration von Windows-Rechnern und -Netzwerken ist die PowerShell klar die beste Wahl. Viele Windows-spezifische Funktionen können durch PowerShell-eigene Kommandos und Module am besten gesteuert werden.
- Analog ist die Bash (oder, nahezu gleichwertig, die Zsh) die ideale Sprache, wenn Sie administrative Scripts auf Linux-Rechnern oder -Servern sowie unter macOS ausführen möchten.
- Für Aufgabenstellungen, die plattformunabhängig sind und nicht von betriebssystemspezifischen Bibliotheken abhängen, eignen sich die PowerShell, die Bash sowie Python gleichermaßen. In diesem Fall empfehle ich Ihnen pragmatisch die


© Rheinwerk Verlag, Bonn 2024

Sprache, mit der Sie am besten umgehen können bzw. deren Umfeld (Kommandos, Erweiterungsmodule) Sie am besten kennen.

- Je größer die Komplexität der Aufgabenstellung ist, je aufwendiger die Steuerung des Codes ist (Schleifen, Verzweigungen, Funktionen etc.), je umfangreicher die erwartete Code-Menge, je größer die Anzahl der benötigten Variablen und Datenstrukturen, desto stärker geht meine Tendenz zu Python.

  Wenn die Gefahr besteht, dass aus Ihrem Script ein »richtiges« Programm mit mehreren 100 Zeilen Code wird, überwiegen die Vorteile von Python (klarere Syntax, bessere Entwicklungswerkzeuge). Allerdings entfernen wir uns dann vom eigentlichen Thema dieses Buchs ...

Irgendwo müssen Sie starten. Wenn Sie unter Windows zu Hause sind und Ihre Scripts dort ausführen möchten, würde ich an Ihrer Stelle mit der PowerShell beginnen. Analog rate ich Linux- und macOS-Fans, zuerst die Grundzüge der Bash bzw. der Zsh zu erlernen. (Diese Empfehlung gilt auch dann, wenn Sie zwar Windows als Arbeitsumgebung verwenden, Ihre Scripts aber auf Linux-Servern laufen sollen. Das klingt widersprüchlich, ist in der Praxis aber absolut üblich. Nicht ohne Grund laufen selbst in der Microsoft-eigenen Cloud *Azure* mehr Linux- als Windows-Instanzen. Es spricht nichts dagegen, die Vorzüge von Windows auf dem Desktop mit jenen von Linux auf dem Server zu kombinieren.)

Sobald Sie mit der Bash oder mit der PowerShell ein wenig vertraut sind, sollten Sie sich mit Python bekannt machen. Python punktet mit seiner bestechend eleganten Syntax und ist diesbezüglich der Bash ebenso wie der PowerShell meilenweit überlegen. Allerdings kommen diese Vorzüge eher bei komplexen, plattformunabhängigen Aufgabenstellungen zur Geltung. Insofern ist Python keine ganz »klassische« Script-Sprache, sondern hat eine viel universellere Zielsetzung. Wenn Ihr Script in erster Linie elementare Linux-Tools wie `find`, `grep`, `adduser` und `gzip` aufrufen oder grundlegende Windows-Administrationsaufgaben erledigen soll, sind die Bash oder die PowerShell dazu trotz all ihrer Syntaxeinheiten besser geeignet als Python.

Letztlich ist entscheidend, dass Sie einfach zu programmieren beginnen. Lassen Sie sich dabei nicht von der Reihenfolge der Kapitel dieses Buchs beeinflussen, sondern steigen Sie ein, wo es für Sie am logischsten erscheint. Ich habe mich beim Schreiben sehr bemüht, alle Kapitel möglichst unabhängig voneinander zu konzipieren. Das PowerShell-Kapitel setzt kein Bash-Wissen voraus – und umgekehrt!

Zum Abschluss dieser Betrachtungen habe ich in Tabelle 1.1 eine ganz persönliche, durchaus subjektive Bewertung der in diesem Buch vorgestellten Sprachen vorgenommen. Es wird Ihnen auffallen, dass Python dabei sehr gut abschneidet. Das ändert nichts daran, dass bei einfachen Aufgabenstellungen je nach Betriebssystem die Bash oder die PowerShell diejenigen Werkzeuge sind, mit denen Sie am schnells-

Persönliches Exemplar für Thomas Bartholomäus

ten zum Ziel kommen und die Sie daher vermutlich am häufigsten einsetzen werden. (Zumindest geht es mir so ...)

| Kriterium | Bash/Zsh | PowerShell | Python |
|---|---|---|---|
| geeignet für Windows | 4 | 10 | 10 |
| geeignet für macOS | 10 | 3 | 10 |
| geeignet für Linux | 10 | 3 | 10 |
| OS-spezifische, administrative Aufgaben | 10 | 10 | 5 |
| plattformunabhängige Aufgaben | 7 | 5 | 10 |
| konsistente Syntax | 3 | 6 | 10 |
| einfacher Aufruf von externen Kommandos | 10 | 9 | 5 |
| Toolbox-Größe (Kommandos, Module etc.) | 9 | 5 | 10 |
| Entwicklungswerkzeuge/Debugging | 2 | 5 | 8 |
| Hilfesystem/Dokumentation | 2 | 6 | 8 |
| einfache, »klassische« Scripts | 8 | 8 | 3 |
| komplexer Code | 2 | 4 | 10 |

**Tabelle 1.1** Persönliche Bewertung ausgewählter Script-Sprachen (0 Punkte: miserabel; 10 Punkte: großartig)


© Rheinwerk Verlag, Bonn 2024

# Kapitel 2
# Zehn mal zehn Zeilen

Die nachfolgenden Kapitel schaffen das Fundament für die Script-Programmierung. Je nachdem, in welcher Sprache Sie arbeiten wollen und welche Vorkenntnisse Sie besitzen, können Sie einige dieser Kapitel durchaus überspringen. Wenn Sie aber gerne linear lesen, dann stehen jetzt recht viele Grundlagenseiten vor den wesentlich spannenderen Scripts. Deswegen präsentiere ich Ihnen in diesem Kapitel – quasi als Appetitanreger – zehn kurze Scripts mit je maximal zehn Zeilen Code.

Wenn Sie am Start Ihrer Scripting-Laufbahn stehen, werden Sie die Funktionsweise dieser Scripts nicht oder nur ansatzweise verstehen. Das macht aber nichts! Jedes Script endet mit dem Querverweis auf ein Kapitel, in dem ein ähnliches Script oder eine umfangreichere Variante erläutert wird. An dieser Stelle geht es ausschließlich darum, Ihnen zu beweisen, was für großartige Möglichkeiten selbst winzige Scripts bieten.

Aus langjähriger Erfahrung weiß ich: Nichts ist öder als Wissensvermittlung, bei der das Ziel unklar ist. Genau das möchte ich vermeiden. Das Wissen aus diesem Buch soll Sie in die Lage versetzen, alltägliche IT-Aufgaben mit minimalem Code-Aufwand zu lösen – so wie in diesem Kapitel!

**Längere Scripts ...**

Verstehen Sie mich nicht falsch! Natürlich sind »echte« Scripts oft deutlich länger und umfassen gerne einmal 100 oder 200 Zeilen. Solche Scripts erfüllen dann komplexere Aufgaben, validieren die Eingabeparameter, zeigen Hilfetexte an usw. Ich will hier nur zeigen, wie weit Sie mit nur zehn Zeilen Code kommen – und das ohne schwer verständliche Spezialfunktionen zur Code-Minimierung.

## 2.1 Markdown-Rechtschreibkorrektur (Bash)

Ich verfasse meine Bücher nicht in Word, sondern verwende die Markdown-Syntax und arbeite in einem Editor ohne Rechtschreibkorrektur. Das mag Ihnen altmodisch erscheinen, aber ich versichere Ihnen, dass mein Schreibprozess ziemlich effizient ist.

Allerdings passieren mir immer wieder bestimmte Tipp- und Rechtschreibfehler. Um meine Korrekturleserinnen nicht in die Verzweiflung zu treiben, wende ich vor Beginn der Korrektur ein Script an, das in allen Kapiteldateien bestimmte Wörter ersetzt.

Das folgende Bash-Script liest in der ersten Schleife zeilenweise die Einträge aus `corrections.txt` und bildet daraus ein `sed`-Kommando (siehe Abschnitt 9.3, »Reguläre Muster in der Bash (grep, sed)«). In der zweiten Schleife wendet es `sed` auf alle an das Script übergebenen Dateien an. Dabei wird ein Backup der ursprünglichen Datei erzeugt (Option `-i.bak`).

```
# Beispieldatei correct.sh
sedcmd=""
while read -r findtxt replacetxt; do
    sedcmd+="s/$findtxt/$replacetxt/g;"
done < corrections.txt
for filename in $*; do
    echo "Korrigiere Datei $filename"
    sed -i.bak "$sedcmd" $filename
done
```

Der Aufruf des Scripts sieht so aus:

```
$ ./correct.sh *.md
```

Wenn Sie möchten, können Sie sich nach dem Aufruf mit `diff test.md test.md.bak` von der korrekten Funktion überzeugen. `corrections.txt` enthält einfach durch Leerzeichen getrennte Suchen- und Ersetzen-Wörter, in der ersten Spalte die falsche Schreibweise, in der zweiten die richtige:

```
Addin           Add-in
Code-Beispiel   Codebeispiel
Github          GitHub
...
```

## 2.2 Bilder nach Aufnahmedatum sortieren (PowerShell)

Kürzlich bekam ich zur Gestaltung eines Fotoalbums mehrere Tausend Fotos von meinen Geschwistern. Glücklicherweise stammte ein Großteil der Fotos aus dem digitalen Zeitalter: Die meisten Bilddateien enthielten daher EXIF-Informationen mit dem Aufnahmedatum. Um rasch einen zeitlichen Überblick zu gewinnen, wollte ich die Bilder monatsweise in Verzeichnisse verschieben (also z. B. `2015-03` für die Fotos, die im März 20215 entstanden sind).


© Rheinwerk Verlag, Bonn 2024

Diese Aufgabe erledigt das folgende PowerShell-Script. Es setzt das Programm exiftool voraus, das vorweg installiert werden muss (siehe Abschnitt 16.2, »Fotos nach Aufnahmedatum sortieren«).

Die erste Schleife durchläuft alle an das Script übergebenen Parameter (z. B. *.jpg und *.jpeg, wenn das Script in der Form .\sort-images.ps1 *.jpg *.jpeg gestartet wurde). Die zweite Schleife verarbeitet die dem Muster entsprechenden Dateien, ermittelt mit exiftool das Aufnahmedatum und formatiert dieses in der Form yyyy-mm. New-Item erzeugt – wenn erforderlich – das entsprechende Verzeichnis. Move-Item verschiebt die Bilddatei dorthin.

```
# Beispieldatei sort-images.ps1
foreach ($arg in $args) {
    foreach ($file in Get-Item $arg) {
        $yearmonth = exiftool -s3 -d '%Y-%m' `
                              -DateTimeOriginal $file
        if ($yearmonth) {
            $targetdir = New-Item -ItemType Directory `
                                  -Path $yearmonth -Force
            Move-Item $file $targetdir
        }
    }
}
```

## 2.3 JSON-Datei in das XML-Format umwandeln (Python)

Gegeben ist die JSON-Datei employees.json mit dem folgenden Aufbau:

```
[
  {
    "emp_no" : 10001,
    "birth_date" : "1953-09-02",
    "first_name" : "Georgi",
    "last_name" : "Facello",
    ...
  }, ...
```

Sie müssen daraus eine XML-Datei bilden, die wie folgt aussehen soll:

```
<?xml version="1.0"?>
<employees>
  <employee no='10001' birth_date='1953-09-02'>Georgi
    Facello</employee>
  ...
</employees>
```

Persönliches Exemplar für Thomas Bartholomäus

Das ist genau die Art von Aufgabenstellung, für die Python prädestiniert ist! `with` öffnet die Quell- und die Zieldatei. `json.load` liest die JSON-Datei ein und macht daraus eine Python-Liste mit Dictionaries. Die `for`-Schleife durchläuft alle Listenelemente und bringt die Mitarbeiternummer, das Geburtsdatum und den Namen in das gewünschte XML-Format.

```
# Beispieldatei json2xml.py
import json
fmt = "  <employee no='%s' birth_date='%s'>%s %s</employee>\n"
with open('employee.json', 'r') as jsonfile, \
     open('employee.xml', 'w') as xmlfile:
    data = json.load(jsonfile)
    xmlfile.write('<?xml version="1.0"?>\n<employees>\n')
    for item in data:
        xmlfile.write(fmt % (item['emp_no'],
          item['birth_date'], item['first_name'],
          item['last_name']))
    xmlfile.write('</employees>\n')
```

Grundlagen zum Umgang mit JSON- und XML-Dateien finden Sie in Kapitel 10, »JSON XML und INI«.

## 2.4 Tägliches Server-Backup (Bash)

Auf einem LAMP-Server (Linux, Apache, MySQL/MariaDB, PHP) läuft eine Webapplikation. Der Inhalt der Datenbank sowie ein Verzeichnis mit Webdateien sollen einmal täglich gesichert werden. Das gelingt mit einem kurzen Bash-Script:

```
# Beispieldatei lamp-backup.sh
dbfile="/localbackup/sql.gz"
mysqldump -u backupuser --single-transaction dbname | \
  gzip -c > $dbfile
htmlfile="/localbackup/html.tar.gz"
tar czf $htmlfile -C /var/www/html/applicationdir .
```

Das Kommando `mysqldump` setzt voraus, dass es eine Konfigurationsdatei `/root/.my.cnf` gibt, in der das Passwort für den Backup-User hinterlegt ist. Damit das Script jede Nacht um 4:30 mit `root`-Rechten ausgeführt wird, bauen Sie in `/etc/crontab` die folgende Zeile ein (siehe auch Kapitel 11, »Scripts automatisch ausführen«:

```
# in /etc/crontab
30 4 * * * root /path/to/lamp-backup.sh
```


© Rheinwerk Verlag, Bonn 2024

Grundlagen und Tipps zur Gestaltung von Backup-Scripts folgen in Kapitel 15, »Backups«. Wie Sie Ihre Backups verschlüsseln und in die Cloud hochladen, zeige ich Ihnen in Abschnitt 20.2.

## 2.5 Web Scraping (Python)

Web Scraping ist die Kunst, aus einer Webseite Informationen zu entnehmen, z. B. aus einer Produktseite den aktuellen Preis. Das folgende Script soll aus der Website *https://git-scm.com* (siehe Abbildung 2.1) die aktuelle Versionsnummer sowie den Link auf die Release Notes extrahieren.

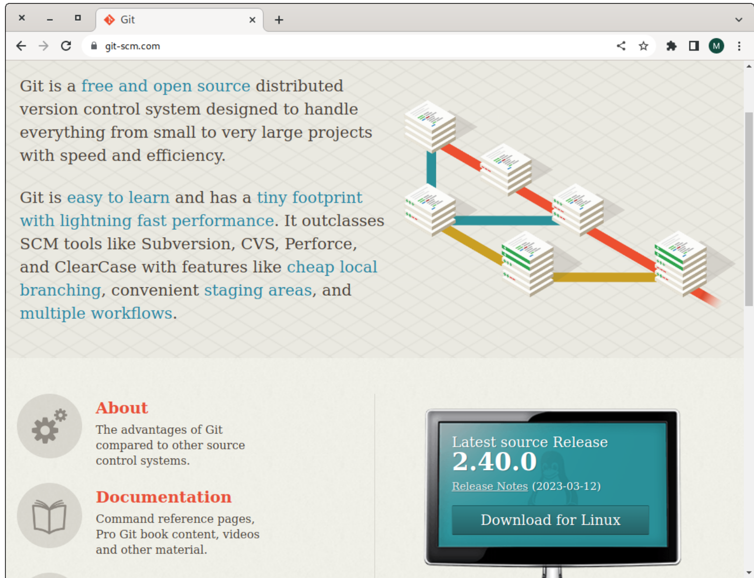



**Abbildung 2.1** Die Git-Homepage zeigt rechts unten die aktuelle Versionsnummer an.

Ein Blick in den HTML-Code der Seite offenbart das folgende Code-Fragment:

```
...
<div class="monitor">
  <h4> Latest source Release </h4>
  <span class="version">
      2.40.0
  </span>
  <a href="https://raw.github.com/git/git/master/Documentation/\
          RelNotes/2.40.0.txt">Release Notes</a>
  ...
```

Persönliches Exemplar für Thomas Bartholomäus

Das folgende Python-Script setzt voraus, dass Sie vorweg zwei Module installieren. Je nach Betriebssystem verwenden Sie dazu `pip` oder `pip3`:

```
$ pip install requests
$ pip install beautifulsoup4
```

Das Miniprogramm lädt den HTML-Code herunter und verwendet die Bibliothek *Beautiful Soup*, um das Span-Tag zu suchen. Dessen `text`-Eigenschaft liefert die gewünschte Versionsnummer. Um den Link auf die Release Notes zu ermitteln, greift `parent` auf die darüberliegende Ebene zu (hier das `div`-Tag mit der Klasse `monitor`). `find` sucht darin das erste `a`-Tag.

```
import requests
from bs4 import BeautifulSoup
response = requests.get("https://git-scm.com/downloads")
dom = BeautifulSoup(response.content, 'html.parser')
version = dom.find('span', class_='version')
print("Git version:", version.text.strip())
url = version.parent.find('a')
print("What's new:", url.attrs['href'])
```

Als ich die Arbeit an diesem Kapitel abgeschlossen habe, lieferte das Script die folgende Ausgabe:

```
Git version: 2.40.0
What's new: https://raw.github.com/git/git/master/Documentation/\
            RelNotes/2.40.0.txt
```

Beachten Sie, dass dieses Script nur funktioniert, solange sich der Aufbau der Git-Projektseite nicht ändert. Auf dieses Grundproblem des Web Scrapings sowie auf diverse Programmiertechniken in den Sprachen Bash, PowerShell und Python gehe ich in Kapitel 17 ein.

## 2.6 Wetterdaten protokollieren (Python)

Es gibt im Internet unzählige Sites, auf denen Sie über sogenannte REST-APIs interessante Daten ermitteln können. Bei vielen Diensten müssen Sie sich vorher registrieren. Oft ist – wie bei *https://www.weatherapi.com* – eine eingeschränkte kostenlose Nutzung erlaubt.

Das folgende Script ermittelt das Wetter in Graz und protokolliert dieses in die Datei `weather.csv`. Diese Datei hat den folgenden Aufbau:

```
2023-04-14 11:30;Light rain;5.0
2023-04-14 12:13;Light rain;6.0
...
```


© Rheinwerk Verlag, Bonn 2024

Die drei Spalten enthalten also den Zeitpunkt, eine Kurzbeschreibung des Wetters und die Temperatur. Das dazugehörende Script führt einen GET-Request durch und macht aus den resultierenden JSON-Daten eine Python-Datenstruktur. Aus dieser können in der Folge ganz unkompliziert die gewünschten Daten extrahiert werden. Wie das vorige Beispiel setzt auch dieses Script das `requests`-Modul voraus.

```
# Beispieldatei log-weather.py
import requests
key = "7901xxxx"
location = "Graz"
base = "https://api.weatherapi.com/v1/current.json"
url = base + "?key=" + key + "&q=" + location
data = requests.get(url).json()
temp = data['current']['temp_c']
condition = data['current']['condition']['text']
time = data['location']['localtime']
with open("weather.csv", 'a') as f:
    f.write("%s;%s;%s\n" % (time, condition, temp))
```

Optional könnten Sie nun den periodischen Aufruf des Scripts durch Cron oder den Windows Task Scheduler automatisieren (siehe Kapitel 11, »Scripts automatisch ausführen«). Dabei müssen Sie nur beachten, dass Sie dem Dateinamen `weather.csv` einen absoluten Pfad voranstellen. Anleitungen, wie Sie REST-APIs in Python- und PowerShell-Scripts nutzen können, finden Sie in Kapitel 18, »REST-APIs nutzen«.

## 2.7 Hyper-V-Aufräumaktion (PowerShell)

Wenn Sie Hyper-V als Virtualisierungssystem verwenden, besteht die Gefahr, dass immer mehr alte Snapshots Speicherplatz fressen. Das folgende PowerShell-Script löscht alle Snapshots, die älter als 30 Tage sind:

```
# Beispieldatei delete-old-snapshots.ps1
$days = 30
$aMonthAgo = (Get-Date).AddDays(-$days)
foreach ($snapshot in Get-VMSnapshot -VMName *) {
    if ($snapshot.CreationTime -lt $aMonthAgo) {
       $vmname = $snapshot.VMName
       $snapname = $snapshot.Name
       Write-Output "Lösche '$snapname' der VM '$vmname'"
       Remove-VMSnapshot -VMName $vmname -Name $snapname -Confirm
    }
}
```

Persönliches Exemplar für Thomas Bartholomäus

Das Script setzt voraus, dass auf Ihrem Windows-Rechner Hyper-V mit allen Administrations-Tools installiert ist. Damit steht automatisch auch das PowerShell-Modul `Hyper-V` zur Verfügung. Das Script erfordert außerdem Administrationsrechte: Sie müssen also zuerst ein PowerShell-Terminal als Administrator öffnen, bevor Sie das Script darin ausführen können.

Sicherheitshalber fragt das Aufräum-Script vor dem Löschen jedes Snapshots nach, ob Sie das wirklich wollen. Diese Rückfragen entfallen, wenn Sie die Option `-Confirm` entfernen. Weitere Beispiele zur Administration virtueller Maschinen mit Hyper-V folgen in Abschnitt 21.3, »Hyper-V steuern«.

## 2.8 Statistische Auswertung einer Logging-Datei (Bash)

Der Ausgangspunkt für dieses Beispiel ist eine aus Datenschutzgründen anonymisierte Logging-Datei eines Webservers. Jede Zeile dieser Datei ist wie im folgenden Listing aufgebaut (hier aus Platzgründen umbrochen):

```
160.85.252.207 - - [05/Feb/2023:00:00:26 +0100]
  "GET /research/business/only HTTP/1.1" 200 25158
  "https://example.com/less/response/edge/policy"
  "Mozilla/5.0 ..."
```

Die erste Spalte gibt jeweils die IP-Adresse an, von der die Website kontaktiert wurde. Das Script, das in seiner Kürze nicht zu unterbieten ist, erstellt eine Liste der 20 am häufigsten vorkommenden IP-Adressen. Die Zeichenkombination `$1` ist der erste an das Script übergebene Parameter.

```
# Beispieldatei ip-address-statistic.sh
cut -d ' ' -f 1 $1 | sort | uniq -c | sort -n -r | head -n 20
```

Wenn Sie mit dem Script die Datei `sample.log` auswerten, erhalten Sie das folgende Ergebnis:

```
$ ./ip-address-statistics.sh

   6166  65d3:f5b9:e9e5:4b1c:331b:29f3:97c1:c18f
   6048  3547:0b26:4c84:4411:0f66:945e:7741:d887
   5136  186.107.89.128
   4620  d741:a4ea:f6e1:6a17:78b1:1694:f518:c480
    ...
```

Es gab also 6166 Requests von der IPv6-Adresse 65d3:f5b9:...:c18f usw. Eine ausführliche Erklärung, wie die im Script vorkommenden Kommandos `cut`, `grep`, `uniq`, `sort` und `head` funktionieren, finden Sie in Kapitel 8, »Textauswertung mit Filtern und Pipes«.


© Rheinwerk Verlag, Bonn 2024

## 2.9 Datei-Upload in die Cloud (PowerShell)

Die Aufgabenstellung für das vorletzte Beispiel lautet, alle Bilddateien im aktuellen Verzeichnis, die sich seit dem letzten Ablauf des Scripts geändert haben oder die seither neu dazugekommen sind, in ein Bucket des Cloud-Services AWS-S3 hochzuladen (*Amazon Web Services, Simple Storage Service*).

Das PowerShell-Script setzt voraus, dass die Module `AWS.Tools.Common` und `AWS.Tools.S3` installiert sind und dass die Zugriffsdaten für das Bucket mit `Set-AWSCredential` für den Profil-Namen `MyProfile` hinterlegt wurden (siehe auch Abschnitt 20.3, »AWS-PowerShell-Modul«).

Die `if`-Konstruktion testet, ob die Datei `last-run` existiert. Wenn das der Fall ist, wird der Zeitpunkt der letzten Änderung ausgelesen. Anschließend wird die Eigenschaft `LastWriteTime` auf den aktuellen Zeitpunkt aktualisiert. Existiert die Datei dagegen nicht, wird sie erzeugt. Das Script berücksichtigt alle Bilddateien seit Anfang 2000.

Nun ermittelt `Get-ChildItem` alle Bilddateien im aktuellen Verzeichnis. `Where-Object` filtert die heraus, die neu oder verändert sind. `Write-S3Object` lädt diese Dateien in ein S3-Bucket hoch.

```
# Beispieldatei upload-images-to-aws.ps1
if (Test-Path -Path "last-run") {
    $lastRunTime = (Get-Item -Path "last-run").LastWriteTime
    (Get-ChildItem "last-run").LastWriteTime = Get-Date
} else {
    $lastRunTime = New-Object DateTime(2000, 1, 1)
    New-Item "last-run" | Out-Null
}
Get-ChildItem -Path "*.jpg", "*.jpeg", "*.png" |
Where-Object { $_.LastWriteTime -gt $lastRunTime } |
ForEach-Object {
    Write-S3Object -BucketName "my.aws.bucket" `
      -ProfileName "MyProfile" -File $_.Name -Key $_.Name
}
```

Ein wenig widerwillig muss ich zugeben, dass die Länge dieses Scripts das Zehn-Zeilen-Motto dieses Kapitels knapp überschreitet ...

## 2.10 Virtuelle Maschinen klonen (Bash)

Das letzte Beispiel stammt aus der Server-Administration. Auf einem meiner Linux-Server läuft das Virtualisierungssystem KVM. Zu Semesterbeginn erzeuge ich für alle Studentinnen und Studenten meiner Linux-Übung virtuelle Maschinen mit durch-

Persönliches Exemplar für Thomas Bartholomäus

laufenden Nummern. Beispielsweise führt das Kommando `./make-vms.sh 20 44` dazu, dass 25 Klone meines Basissystems erzeugt werden, `vm20` bis `vm44`.

```
# Beispieldatei make-vms.sh
orig='vm-base'    # Name der zu klonenden Basis-VM
for (( nr=$1; nr<=$2; nr++ )); do
    echo "create vm-$nr"
    disk=/var/lib/libvirt/images/vm-$nr-disk.qcow2
    virt-clone --name "vm-$nr" --original $orig \
       --mac 52:54:00:01:00:$nr --file $disk1
done
```

Das obige Script hält die Zehn-Zeilen-Regel wieder ein – aber ich gebe zu, dass ich ein wenig geschwindelt habe. Die »richtige« Version dieses Scripts testet, ob wirklich zwei Parameter an das Script übergeben wurden, fährt die eventuell noch laufende virtuelle Ausgangsmaschine herunter, richtet für jede Übungs-VM mehrere Datenträger und mehrere Netzwerkschnittstellen ein und einiges mehr.

Grundlagen und weitere Details können Sie in Abschnitt 21.1, »Virtuelle Maschinen einrichten und ausführen (KVM)«, nachlesen.


© Rheinwerk Verlag, Bonn 2024

# Kapitel 3
# Bash und Zsh

Die Bash (*Bourne Again Shell*) ist die Standard-Shell der meisten Linux-Distributionen. Sie erfüllt zwei Aufgaben: Einerseits nimmt sie im Terminal Kommandos entgegen, führt diese aus und zeigt das Ergebnis an. Andererseits gilt die Bash als *die* klassische Scripting-Programmiersprache. In diesem Kapitel gehe ich auf die interaktive Anwendung nur kurz ein und konzentriere mich vielmehr darauf, Ihnen die wichtigsten Bash-Konzepte vorzustellen.

Als Alternative zur Bash nimmt die Zsh immer mehr Fahrt auf. Während die Unterschiede im Vergleich zur Bash bei der Script-Programmierung vernachlässigbar sind, punktet die Zsh durch viele kleine Verbesserungen bei der interaktiven Bedienung. Populäre Erweiterungen (eine hat den Namen *Oh my Zsh*) machen die Zsh noch komfortabler und verbessern das Erscheinungsbild. Unter macOS ist die Zsh seit 2019 die Standard-Shell. Unter Linux gelingt die Umstellung von der Bash auf die Zsh bei Bedarf in einer Minute.

## 3.1 Terminal, Shell und Bash

In der kurzen Einleitung sind einige Begriffe vorgekommen, die ich hier kurz nochmals erklären möchte. Beginnen wir mit dem »Terminal«: Das ist ein Programm, in dem Sie textorientierte Kommandos ausführen können. Unter Linux und macOS war die Existenz des Terminals immer eine Selbstverständlichkeit. Während es unter macOS ein *Terminal*-Programm gibt, stehen unter Linux unzählige Programme zur Auswahl. Gebräuchlich sind z. B. `gnome-terminal`, die neue Konsole des Gnome-Projekts (Programmname `kgx`) sowie `konsole`. Sogar unter Windows gibt es neuerdings das *Windows Terminal*.

Innerhalb des Terminals läuft die Shell: Das ist ein Programm, das Ihre Texteingaben entgegennimmt, das resultierende Kommando ausführt und das Ergebnis wieder in Textform anzeigt. Die Shell bietet darüber hinaus eine Menge Zusatzfunktionen, die nicht auf den ersten Blick erkennbar sind: Sie kann die Ergebnisse eines Kommandos in eine Datei umleiten, mehrere Kommandos miteinander verbinden usw.

Persönliches Exemplar für Thomas Bartholomäus

Im Laufe der Unix- und Linux-Geschichte wurden schon diverse Male Shells entwickelt: Bourne Shell, C Shell, Bash, Dash, Zsh usw. In diesem Kapitel geht es aber nur um die beiden aktuell populärsten Shells, eben Bash und Zsh.

**Warum die Trennung zwischen Terminal und Shell?**

Die Verschachtelung von Terminal und Shell wirkt auf den ersten Blick verwirrend. Sie hat damit zu tun, dass das Terminal nur dann notwendig ist, wenn Sie auf einem grafischen Desktop-System arbeiten. Die meisten Betriebssysteme können aber auch im Textmodus oder über eine Netzwerkverbindung (SSH) gesteuert werden – dann läuft die Shell *ohne* Terminal!

Die Trennung zwischen Terminal und Shell hat den weiteren Vorteil, dass Sie, abweichend von den Voreinstellungen von Linux oder macOS, Ihr Lieblings-Terminal mit Ihrer Lieblings-Shell kombinieren können. Diese Freiheit ist vor allem Linux-Profis wichtig.

### Die aktuelle Shell ermitteln

Um festzustellen, welche Shell auf Ihrem Rechner läuft, führen Sie das Kommando `echo $SHELL` aus. Es zeigt den Inhalt der Variablen `$SHELL` an, die den Pfad zur Default-Shell Ihres Accounts angibt (siehe Abbildung 3.1). Wenn es sich um die Bash handelt, können Sie deren Versionsnummer mit dem zweiten Kommando `bash --version` feststellen.

```
kofler@ubuntu:~$ echo $SHELL
/bin/bash
kofler@ubuntu:~$ bash --version
GNU bash, version 5.1.16(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
kofler@ubuntu:~$
```

**Abbildung 3.1** Auf diesem Rechner läuft die Bash in der Version 5.1

## 3.2 Installation

Bei fast allen Linux-Distributionen ist die Bash vorinstalliert. Wenn Sie also unter Linux arbeiten, erübrigt sich dieser Abschnitt. Sollte Ihre Distribution die Zsh verwenden, empfehle ich Ihnen, für die interaktive Anwendung dabei zu bleiben. Um


© Rheinwerk Verlag, Bonn 2024

möglichst portable Scripts zu entwickeln, ist die parallele Installation der Bash aber empfehlenswert. Das gelingt mit dem Paketverwaltungskommando Ihrer Distribution, unter Debian und Ubuntu beispielsweise mit `sudo apt install bash`.

Unter macOS steht die Bash nur in der veralteten Version 3.2 zur Verfügung. In zukünftigen macOS-Versionen wird die Bash voraussichtlich ganz entfernt. Dafür bietet die Zsh einen kompatiblen und in vielerlei Hinsicht sogar besseren Ersatz. Sollten Sie doch die Bash bevorzugen bzw. zur Script-Programmierung einsetzen, richten Sie am besten zuerst den Paketmanager *Homebrew* ein (siehe *https://brew.sh*). Anschließend installieren Sie die Bash im Terminal mit dem Kommando `brew install bash`. Zum Start der Bash führen Sie `/opt/homebrew/bin/bash` aus.

Mühe bereitet die Installation der Bash nur unter Windows. Ich präsentiere Ihnen hier verschiedene Wege, wie Sie die Bash unter Windows nutzen können. Sie sind mit unterschiedlichen Vor- und Nachteilen verbunden. So viel vorweg: Wirklich ideal ist keine Variante.

#### Bash in einer virtuellen Maschine verwenden

Der vielleicht naheliegendste Weg zur Bash besteht darin, in einem Virtualisierungsprogramm (also in VirtualBox, VMWare oder Hyper-V) eine gängige Linux-Distribution zu installieren, z. B. Ubuntu. Innerhalb der virtuellen Maschine steht Ihnen dann die Bash im Terminalfenster zur Verfügung.

Zum Lernen der Bash ist diese Lösung ideal. Allerdings hat dieser Weg den Nachteil, dass Sie mit Ihren Scripts keinen Zugriff auf das Windows-Dateisystem haben. Wenn Sie also z. B. ein Script entwickeln, das Bilddateien verarbeitet, können Sie das Script nur auf Dateien innerhalb des Linux-Dateisystems anwenden. (Manche Virtualisierungssysteme bieten die Möglichkeit, ein Verzeichnis zwischen Windows und der virtuellen Linux-Installation zu teilen. Gerade bei VirtualBox ist die Vorgehensweise aber ziemlich umständlich.)

Ein weiterer Nachteil der Virtualisierungslösung besteht darin, dass Sie eigentlich nicht mehr unter Windows arbeiten, sondern unter Linux. Sie müssen sich an einen unbekannten Desktop gewöhnen und zumindest ein wenig in die Linux-Administration einsteigen. Sie wollen aber die Bash lernen, oder? (Aus meiner ganz persönlichen Sicht ist die Verwendung von Linux ein Vorteil, kein Nachteil. Aber ich will hier niemanden zu Linux bekehren.)

Noch ein Argument spricht gegen die Virtualisierungslösung: Das kostenlose und daher entsprechend beliebte Programm VirtualBox zickt nach meinen Erfahrungen häufig auf Rechnern, die zur Software-Entwicklung verwendet werden. Das hat damit zu tun, dass auf solchen Rechnern diverse Hyper-V-Funktionen aktiv sind. Theoretisch sollte VirtualBox damit kompatibel sein, in der Praxis hakt es aber oft.

Persönliches Exemplar für Thomas Bartholomäus

### Windows-Subsystem für Linux (WSL)

Die aus meiner Sicht beste Lösung zur Anwendung der Bash unter Windows bietet das *Windows-Subsystem für Linux*, kurz WSL. Diese Funktion ermöglicht es, Linux in einem Terminalfenster auszuführen. Großartig ist dabei, dass Sie in Windows auf die Linux-Dateien und in WSL ebenso unkompliziert auf Ihre Windows-Dateien zugreifen können. (Mit aktuellen WSL-Versionen ist es sogar möglich, Linux im Grafikmodus auszuführen. Für unsere Zwecke ist das aber gar nicht notwendig.)

Die gerade aktuellste WSL-Version ist im Microsoft Store kostenlos erhältlich. Nach der Installation, die mit einem obligatorischen Windows-Neustart verbunden ist, suchen Sie abermals im *Microsoft Store* nach *Ubuntu 22.04 LTS* (bzw. ab April 2024 nach der dann aktuellen Version 24.04). Die Installation und der erste Start dauern ca. zwei Minuten. Sie müssen dabei zweimal ein Passwort angeben. Merken Sie sich dieses Passwort! Sie werden es später brauchen, wenn Sie mit `sudo` zusätzliche Software-Komponenten installieren wollen. (Neben Ubuntu stehen im Microsoft Store auch andere Distributionen zur Wahl, unter anderem AlmaLinux, Debian, openSUSE sowie Oracle Linux. Verwenden Sie, womit Sie vertraut sind.)

In der Folge können Sie jederzeit im Windows-Terminal oder im Startmenü (Programmname *Ubuntu n.n*) ein Fenster mit der Bash öffnen (siehe Abbildung 3.2).

```
kofler@KVMWIN:~$ echo $SHELL
/bin/bash
kofler@KVMWIN:~$ grep PRETTY /etc/os-release
PRETTY_NAME="Ubuntu 22.04 LTS"
kofler@KVMWIN:~$ df -h | egrep -v 'none|tmpfs'
Filesystem      Size  Used Avail Use% Mounted on
rootfs          200G   36G  165G  18% /
C:\             200G   36G  165G  18% /mnt/c
F:\            1006M   17M  990M   2% /mnt/f
kofler@KVMWIN:~$
```

**Abbildung 3.2** Ubuntu mit der Bash in einem Terminal-Fenster unter Windows

Innerhalb der Bash können Sie unter dem folgenden Pfad auf Ihr Windows-Benutzerverzeichnis zugreifen (einmal vorausgesetzt, Ihre Windows-Installation befindet sich auf dem Datenträger `C:`):

```
/mnt/C/Users/<name>
```

Umgekehrt finden Sie Ihr Linux/Bash-Benutzerverzeichnis im Windows-Explorer über den Eintrag LINUX ganz unten in der Seitenleiste (siehe Abbildung 3.3) bzw. über die folgende Adresse:

```
\\wsl.localhost\Ubuntu-22.04\home\<name>
```


© Rheinwerk Verlag, Bonn 2024

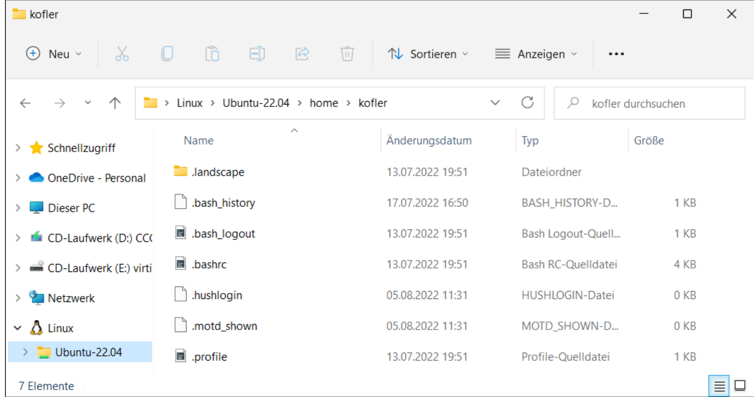



**Abbildung 3.3** Einfacher Zugriff auf das Linux-Home-Verzeichnis im Windows-Explorer

## Git Bash

Einen eleganten Bash-Einstieg bietet das Versionsverwaltungsprogramm Git. Wenn Sie Git unter Windows installieren (siehe *https://git-scm.com/download/win*), wird zugleich die Git Bash installiert. Wenn Sie in der Software-Entwicklung tätig sind, ist das höchstwahrscheinlich ohnedies schon der Fall. (Sie arbeiten doch nicht ohne Git, oder?) Dann können Sie im Startmenü *Git Bash* ausführen. Damit erscheint ein Terminal-Fenster, in dem die Bash in der Version 4.4 läuft. Das ist zwar nicht die aktuellste Version, aber sie ist für unsere Zwecke ausreichend.

Die Git Bash nutzt direkt das Windows-Dateisystem. Im Git-Bash-Fenster stehen die wichtigsten Kommandos zur Verfügung, die auch unter Linux oder macOS gebräuchlich sind – also `ls`, `cat`, `less`, `more`, `find`, `grep` usw.

Das Problem der Git Bash besteht darin, dass Sie keine weiteren Kommandos installieren können. Früher oder später (eher früher) werden Sie zu dem Punkt kommen, an dem Sie in Ihrem Script ein Kommando aufrufen möchten, das nicht Teil der Grundausstattung der Git Bash ist. Was die Bash-Programmierung betrifft, sind Sie dann in einer Sackgasse gelandet.

## Docker

Sofern Sie mit dem Containersystem Docker vertraut sind, können Sie Bash-Scripts auch in einem Docker-Container ausführen. Diese Vorgehensweise eignet sich aber eher für Profis zur Weitergabe von Scripts. Wenn Sie die Bash dagegen interaktiv kennenlernen möchten, ist Docker nicht der ideale Weg.

Persönliches Exemplar für Thomas Bartholomäus

### Bash-Konfiguration (/etc/profile und .bashrc)

Es gibt mehrere Dateien, die das Verhalten der Bash beeinflussen. Am wichtigsten sind /etc/profile mit globalen Einstellungen sowie .bashrc in Ihrem Heimatverzeichnis mit individuellen Optionen. Beachten Sie, dass bei der Ausführung von Scripts außerhalb Ihres Accounts nur /etc/profile berücksichtigt wird.

Für den normalen Einsatz müssen Sie an beiden Dateien keine Änderungen vornehmen. Der häufigste Grund, eine der Dateien doch zu verändern, ist der Aufruf von Kommandos oder Programmen, die abseits der üblichen Verzeichnisse installiert wurden. Dazu müssen Sie die Umgebungsvariable PATH erweitern. Das folgende Beispiel zeigt, wie Sie außer den voreingestellten Verzeichnissen zusätzlich /my/own/scripts berücksichtigen:

```
# am Ende von .bashrc oder /etc/profile
export PATH=/my/own/scripts:$PATH
```

## 3.3 Kommandos interaktiv ausführen

Sofern Sie schon Routine im Umgang mit dem Terminal haben, egal unter welchem Betriebssystem, können Sie diesen Abschnitt getrost überspringen. Andernfalls ist es eine gute Idee, sich zuerst ein wenig mit der interaktiven Ausführung von Kommandos und mit der Bedienung des Terminals vertraut zu machen, bevor Sie mit dem ersten Script beginnen.

Auch später wird es so sein, dass Sie parallel zu Ihrem Script-Editor in einem Terminal-Fenster einzelne Kommandos ausprobieren. Erst wenn die Kommandos so funktionieren, wie Sie sich das vorstellen, bauen Sie diese in Ihr Script ein.

Im folgenden Beispiel wechseln Sie zuerst mit cd (*change directory*) in das Verzeichnis Downloads und sehen sich dann mit ls (*list*) an, welche Dateien dort gespeichert sind:

```
$ cd Downloads
$ ls

  00a_intro_pn.pdf
  00b_about_pn.pdf
  00c_next_pn.pdf
  ...
```

ls mit der Option -l (*long*) zeigt mehr Details zu jeder Datei – unter anderem die Zugriffsrechte, den Besitzer und die Gruppenzuordnung, die Größe, den Zeitpunkt der letzten Änderung:

```
$ ls -l
```


© Rheinwerk Verlag, Bonn 2024

```
  -rw-r--r-- 1 kofler kofler  674076 ... 11:49  00a_intro_pn.pdf
  -rw-r--r-- 1 kofler kofler  943948 ... 11:53  00b_about_pn.pdf
  -rw-r--r-- 1 kofler kofler  714852 ... 11:58  00c_next_pn.pdf
  ...
```

Wenn Sie wissen möchten, wie viele Dateien sich im Verzeichnis befinden, leiten Sie die Ausgabe von `ls` an `wc` (*word count*) weiter. Das Kommando liefert drei Zahlen. Die erste gibt die Anzahl der Zeilen, die zweite die Anzahl der Worte und die dritte die Anzahl der Zeichen an. Da `ls` jede Datei in einer eigenen Zeile ausgibt, ist nur die Anzahl der Zeilen relevant. Die `wc`-Option `-l` (*lines*) reduziert die Ausgabe entsprechend.

```
$ ls | wc

  116     159    3194

$ ls | wc -l

  116
```

Mit diesen trivialen Kommandos haben Sie schon eine Menge gelernt:

- Die in der Bash üblichen Kommandos haben überwiegend sehr kurze Namen. Für den Einstieg sind diese Namen wenig intuitiv. Wenn Sie sich aber einmal daran gewöhnt haben, ist der Tippaufwand minimal.
- Das Verhalten von Kommandos lässt sich durch Optionen steuern. Viele Optionen bestehen aus nur einem Buchstaben. Ihnen wird ein Bindestrich vorangestellt. Oft gibt es alternativ auch eine Langform, der *zwei* Bindestriche vorangestellt werden. Beispielsweise sind `wc --lines` und `wc -l` gleichwertig.
- Kommandos können mit dem Pipe-Operator | miteinander verknüpft werden. Auf diesen Operator werde ich später noch genauer eingehen. Kurz zusammengefasst bedeutet `command1 | command2`, dass `command1` seine Textausgaben nicht anzeigt, sondern zur Verarbeitung an `command2` weitergibt.

### Der Prompt

Als »Prompt« wird die Zeichenfolge bezeichnet, die bei der Eingabe neuer Kommandos links vom Cursor angezeigt wird. Unter Linux ist es üblich, dass der Prompt bei gewöhnlichen Benutzern mit dem Zeichen $ endet. Wenn hingegen der Systemadministrator `root` arbeitet, endet der Prompt mit #.

Der Prompt ist einstellbar (Umgebungsvariable `PS1`) und kann deswegen je nach Distribution und Konfiguration ganz unterschiedlich aussehen. Oft werden vor dem Zeichen $ bzw. # das aktuelle Verzeichnis und/oder der Hostname angezeigt.

Persönliches Exemplar für Thomas Bartholomäus

In diesem Buch ist Kommandos, die in der Bash oder Zsh auszuführen sind, das Zeichen $ vorangestellt. (Es sollte Ihnen mittlerweile klar sein, dass dieses Zeichen *nicht* einzugeben ist.) Vor PowerShell-Kommandos steht dagegen das Zeichen >, weil der Prompt unter Windows dieses Zeichen verwendet.

Ein Sonderfall ist dieses Kapitel: Da ohnedies klar ist, dass sich die Beispiele auf die Bash beziehen, verzichte ich auf die Angabe des Prompts oft ganz. Das gilt insbesondere für die Präsentation von Arbeitstechniken, die gleichermaßen interaktiv im Terminal als auch als Code in einem Script funktionieren.

## Bedienung des Terminals

Es ist Ihnen vermutlich bereits aufgefallen, dass die Maus im Terminal weitgehend nutzlos ist. Sie können damit nicht wie in anderen Programmen die Cursorposition verändern. Eingaben können nur in der letzten Zeile des Terminals erfolgen. (Unter Linux können Sie immerhin mit der Maus Text markieren und mit der mittleren Maustaste blitzschnell an der aktuellen Cursor-Position einfügen – also ohne den Text umständlich in die Zwischenablage zu kopieren.)

Die magere Mausunterstützung wird durch viele Tastenkürzel kompensiert, mit denen Sie die aktuelle Eingabezeile bearbeiten und auf vergangene Eingaben zurückgreifen können (siehe Tabelle 3.1).

| Tastenkürzel | Bedeutung |
|---|---|
| ⇆ | vervollständigt ein Kommando oder einen Namen |
| Strg+A | setzt den Cursor an den Beginn der Zeile |
| Strg+C | bricht die Eingabe ab bzw. stoppt ein laufendes Kommando |
| Strg+D | löscht das Zeichen an der Cursorposition |
| Strg+E | setzt den Cursor an das Ende der Zeile |
| Strg+K | löscht alles bis zum Ende der Zeile |
| Strg+R abc | sucht nach einem früher ausgeführten Kommando abc |
| Strg+Y | fügt den zuletzt gelöschten Text wieder ein |
| Alt+◄ | löscht ein Wort rückwärts |

**Tabelle 3.1** Tastenkürzel im Terminal

Besonders praktisch ist das Tastenkürzel ⇆, das viel Tipparbeit erspart: Sie müssen nur die ersten Buchstaben eines Kommandos, eines Verzeichnisses oder einer Datei


© Rheinwerk Verlag, Bonn 2024

angeben. Anschließend drücken Sie [↹]. Wenn der Kommando-, Verzeichnis- oder Dateiname bereits eindeutig erkennbar ist, wird er vollständig ergänzt, sonst nur so weit, bis sich mehrere Möglichkeiten ergeben. Ein zweimaliges Drücken von [↹] zeigt die Liste aller möglichen Vervollständigungen an.

### Onlinehilfe

Kommandos wie `ls` oder `wc` verfügen über kein Hilfe-Menü. Es gibt aber auch für diese Kommandos Hilfetexte, die durch verschiedene Kommandos gelesen werden können. Als ersten Versuch sollten Sie <command> --help aufrufen (also z. B. `ls --help`). Damit erhalten Sie in der Regel eine Liste aller Optionen samt einer kurzen Erklärung zu ihrer Bedeutung.

man <command> zeigt bei vielen Kommandos einen längeren Infotext an. `man ls` erklärt also, wie das Kommando `ls` zu nutzen ist und welche Optionen es dabei gibt. Durch den mehrseitigen Text können Sie mit den Cursortasten navigieren. Die Leertaste blättert eine ganze Seite nach unten. Mit [/] können Sie im Hilfetext einen Ausdruck suchen. [N] springt bei Bedarf zum nächsten Suchergebnis. [Q] beendet die Hilfe.

### Kommandoreferenz

Im Laufe dieses Buchs werde ich Ihnen eine Menge Linux-Kommandos vorstellen. Einen guten Startpunkt bietet diesbezüglich Kapitel 6, »Linux-Toolbox«.

## 3.4 Zsh als Bash-Alternative

Ich habe es in der Kapiteleinleitung schon erwähnt: Für das Kernthema dieses Buchs, also für die Programmierung von Scripts, spielt die Zsh nur eine untergeordnete Rolle. Aus Kompatibilitätsgründen ist es besser, hierfür die Bash zu verwenden und sogar auf den Einsatz einiger Bash-spezifischer Erweiterungen gegenüber anderen Shells zu verzichten. Damit erreichen Sie, dass Ihre Scripts wirklich in fast jeder Umgebung funktionieren.

Der Reiz der Zsh liegt vielmehr in der interaktiven Anwendung: Die Zsh geht bei der Expansion von Kommando-, Datei- und Verzeichnisnamen durch [↹] noch intelligenter vor als die Bash. Sie müssen nicht die Anfangsbuchstaben angeben, stattdessen reicht eine Buchstabenkombination, die irgendwo im Namen vorkommt. Außerdem bietet die Zsh mehr Konfigurationsmöglichkeiten und Erweiterungs-Module. Seit ich den Komfort der Zsh in Kombination mit der Erweiterung »Oh My Zsh« kennengelernt habe, ist mir die »gewöhnliche« Bash nicht mehr gut genug!

Persönliches Exemplar für Thomas Bartholomäus

### Installation

macOS sowie ausgewählte Linux-Distributionen setzen schon jetzt standardmäßig auf die Zsh. Bei allen anderen Linux-Distributionen können Sie die Zsh mühelos mit dem jeweiligen Paketverwaltungs-Tool installieren. Unter Debian und Ubuntu funktioniert das so:

```
$ sudo apt update
$ sudo apt install zsh
```

Im nächsten Schritt aktivieren Sie die Zsh für Ihren Account:

```
$ chsh -s $(which zsh)
```

Damit die Änderung wirksam wird, müssen Sie sich ab- und neu anmelden. Wenn Sie nun das erste Mal ein Terminal öffnen, läuft einmalig ein Konfigurations-Script:

```
This is the Z Shell configuration function for new users,
zsh-newuser-install. (...) You can:
(q) Quit and do nothing.  The function will be run again next
    time.
(0) Exit, creating the file ~/.zshrc containing just a comment.
    That will prevent this function being run again.
(1) Continue to the main menu.
(2) Populate your ~/.zshrc with the configuration recommended
    by the system administrator and exit.
```

Sie machen nichts verkehrt, wenn Sie bei der manuellen Konfiguration (Punkt 1) einfach die vom Script vorgeschlagenen Default-Einstellungen oder überhaupt die von Ihrer Distribution vorgegebene Default-Konfigurationsdatei `.zshrc` übernehmen (Punkt 2). Bei Bedarf können Sie die Konfiguration später mit `zsh-newuser-install` wiederholen und anpassen.

Zur Nutzung der Zsh unter Windows gelten dieselben Empfehlungen wie bei der Bash: Installieren Sie Linux im WSL oder als virtuelle Maschine, und führen Sie die Zsh dort aus! Die Umstellung von der Bash auf die Zsh erfolgt wie oben beschrieben.

### Zsh-Konfiguration (/etc/zshrc und .zshrc)

Ähnlich wie die Bash wertet auch die Zsh beim Start einige Konfigurationsdateien aus. Am wichtigsten sind `/etc/zshrc` für globale Einstellungen sowie `.zshrc` in Ihrem Heimatverzeichnis.

### Oh My Zsh

Für die Zsh gibt es im Internet eine Menge Plugins, die zusätzliche Funktionalität liefern, sowie Themes, die das Aussehen der Kommandoprompts verändern. *Oh My Zsh*


© Rheinwerk Verlag, Bonn 2024

ist ein Script, das bei der Verwaltung solcher Erweiterungen hilft. Diverse Plugins und Themes sind gleich enthalten, weitere können später hinzugefügt werden.

Zur Installation laden Sie von der Projektseite *https://github.com/ohmyzsh/ohmyzsh* ein kleines Script herunter und führen es aus. Das Script setzt voraus, dass das Kommando `git` aus dem gleichnamigen Paket zur Verfügung steht. Anstatt das folgende Kommando abzuschreiben, kopieren Sie den Code besser von der GitHub-Seite.

```
$ sh -c "$(curl -fsSL https://raw.githubusercontent.com/ohmyzsh/\
                        ohmyzsh/master/tools/install.sh)"
```

Während der Installation wird `.zshrc` überschrieben. Der bisherige Inhalt landet in der Backup-Datei `.zshrc.pre-oh-my-zsh`. *Oh My Zsh* testet in Zukunft beim Start, ob es Updates gibt. Dementsprechend erscheint beim Öffnen eines neuen Terminalfensters oft die Rückfrage, ob Sie *Oh My Zsh* aktualisieren möchten.

In der Basiskonfiguration wird der Prompt durch das Theme `robbyrussell` vorgegeben. Als einziges aktives Plugin steuert `git` unzählige Aliasse bei, die bei der Bedienung von `git` helfen. Eine Anleitung zu den vielen Funktionen sowie Tipps zur weiteren Konfiguration von *Oh My Zsh* finden Sie z. B. unter *https://ohmyz.sh* oder im folgenden Blog:

*https://stackabuse.com/pimp-my-terminal-an-introduction-to-oh-my-zsh*

## 3.5 Das erste Bash-Script

Nach diesem kurzen Ausflug zur Zsh kehren wir zurück zur Bash. Bevor ich Ihnen auf den weiteren Seiten dieses Kapitels die Syntax der Bash inklusive diverser Programmier- und Arbeitstechniken näher vorstelle, möchte ich Ihnen in diesem Abschnitt erklären, was konkret ein Script ist. Eigentlich ist die Sache ganz einfach: Ein Script ist eine Textdatei, in der Sie die gleichen Kommandos formulieren, die Sie auch interaktiv im Terminal eingeben. Es gibt nur zwei Besonderheiten: Die erste Zeile des Scripts muss den sogenannten Shebang enthalten, und die Datei muss mit `chmod` als ausführbar markiert werden.

### Hash-Bang (Shebang)

Die erste Zeile eines Scripts, das unter Linux oder macOS ausgeführt werden soll, muss mit den Zeichen `#` (»Hash« oder »Sharp«) und `!` (»Bang«) sowie der Pfadangabe des Interpreters beginnen. »Shebang« ist eine sprachliche Verkürzung von »Sharp Bang«.

Persönliches Exemplar für Thomas Bartholomäus

Der Interpreter ist das Programm, das die Script-Datei liest und verarbeitet – in diesem Kapitel also die Bash (oder in Ausnahmefällen die Zsh). Als Interpreter kommen aber auch andere Programmiersprachen in Frage, z. B. Python oder Perl.

Das Programm `bash` wird unter Linux zumeist in das Verzeichnis `/bin` installiert. Deswegen beginnt ein Bash-Script mit dieser Zeile:

```
#!/bin/bash
```

### Shebang-Varianten

Je nachdem, in welcher Distribution oder in welchem Unix-ähnlichen System Sie arbeiten, kann es sein, dass die Bash in einem anderen Verzeichnis installiert ist, z. B. in `/usr/bin`. Mit `which bash` können Sie den Ort auf Ihrem Rechner feststellen:

```
$ which bash

  /usr/bin/bash
```

Oft stellen Links sicher, dass sowohl `/usr/bin/bash` als auch `/bin/bash` funktionieren. Eine allgemeingültige Hash-Bang-Formulierung, die unabhängig vom Speicherort funktioniert, greift auf das Kommando `/usr/bin/env` zurück. Dieses Kommando durchsucht alle gängigen Orte:

```
#!/usr/bin/env bash
```

Wenn die Bash auf Ihrem Rechner gar nicht installiert ist und Sie mit der Zsh arbeiten, ersetzen Sie `/bin/bash` durch `/bin/zsh` (oder den Ort, an dem die Zsh auf Ihrem Rechner installiert ist):

```
#!/bin/zsh
```

Falls Ihr Script unabhängig davon funktionieren soll, ob nun die Bash oder die Zsh zur Verfügung steht, führt der folgende Hash-Bang zum Ziel:

```
#!/bin/sh
```

`sh` ist bei den meisten Linux-Systemen ein Link, der auf die Standard-Shell verweist. Es muss Ihnen aber bewusst sein, dass bei der Ausführung Ihres Scripts jetzt je nach Rechner/Distribution unterschiedliche Shells zur Anwendung kommen, unter Debian und Ubuntu z. B. die auf Geschwindigkeit optimierte Dash. Bei der Formulierung Ihres Scripts müssen Sie darauf achten, auf keinen Fall irgendwelche Bash-spezifischen Erweiterungen zu verwenden, die in anderen Shells nicht zur Verfügung stehen.


© Rheinwerk Verlag, Bonn 2024

**Was denn nun?!**

Ich weiß, es ist ärgerlich, dass es in der IT-Welt zu jeder Regel drei Ausnahmen und dann noch fünf Sonderfälle gibt. Solange keine zwingenden Gründe dagegen sprechen, verwenden Sie als Hash-Bang `#!/bin/bash`. Auch die Beispieldateien zu diesem Buch verwenden diesen Code.

### Das Script ausführbar machen (chmod +x)

Alle Unix-ähnlichen Systeme inklusive Linux und macOS speichern zusammen mit jeder Datei einige Zugriffsbits, die Auskunft darüber geben, wer die Datei lesen, schreiben und ausführen darf. Für Scripts ist das Execute-Bit (kurz `x`) entscheidend. Nur mit diesem Bit kann ein Script später wie ein Kommando ausgeführt werden.

Um das Execute-Bit zu setzen, führen Sie im Terminal einmalig das folgende Kommando aus. Beachten Sie, dass die `chmod`-Variante von macOS die Kurzschreibweise `+x` nicht kennt. Sie müssen explizit angeben, für wen das Execute-Bit gesetzt wird. `a` bedeutet: für alle.

```
$ chmod +x my-script-file.sh    (Linux, Git Bash)
$ chmod a+x my-script-file.sh   (macOS)
```

`chmod` dient dazu, die Zugriffsbits zu verändern. Das Kommando unterstützt recht vielfältige Syntaxvarianten. (Führen Sie `man chmod` aus, wenn Sie sich für die Details interessieren.) Für uns ist hier nur die Variante `chmod +x` relevant, um das Execute-Bit zu setzen.

**Die Kennung ».sh«**

Sämtliche Bash-Beispieldateien in diesem Buch enden mit `.sh`. Diese Endung ist zwar üblich, aber keineswegs vorgeschrieben. Bash-Scripts funktionieren auch ohne Kennung (und sogar mit einer falschen Kennung) – vorausgesetzt, der Hash-Bang ist korrekt.

Da dieses Buch neben der Bash auch die PowerShell (`.ps1`) und Python (`.py`) behandelt, hilft die Kennung bei der Zuordnung der Beispieldateien.

### Hello, World!

Um Ihr erstes Script zu verfassen, starten Sie Ihren Lieblingseditor: Visual Studio Code ist eine gute Wahl, aber jeder Editor ist in Ordnung, auch das minimalistische Programm `nano`, das Sie direkt in einem Terminal ausführen können. Nun geben Sie die folgenden zwei Zeilen ein:

Persönliches Exemplar für Thomas Bartholomäus

```
#!/bin/bash
echo "Hello, World!"
```

Nun speichern Sie diese Datei unter dem Namen `hello-world.sh`. Mit `chmod` machen Sie das Script ausführbar.

```
$ chmod  +x hello-world.sh     (Linux, Git Bash)
$ chmod a+x hello-world.sh     (macOS)
```

Um das Script auszuprobieren, geben Sie dessen Namen ein und drücken ↵. Beachten Sie, dass Sie dem Namen ./ voranstellen müssen. Damit weisen Sie die Bash darauf hin, dass sie Ihr Script im aktuellen Verzeichnis suchen soll. (`.` ist eine Kurzschreibweise für das gerade aktive Verzeichnis.)

```
$ ./hello-world.sh
  Hello, World!
```

**Wenn die Script-Ausführung nicht funktioniert**

Wenn die Script-Ausführung einen Fehler auslöst, kann dies verschiedene Gründe haben:

- Haben Sie den Dateinamen korrekt angegeben?
- Haben Sie an `chmod +x` gedacht?
- Haben Sie die Hash-Bang-Zeile korrekt angegeben?
- Gibt es auf Ihrem Rechner gar keine Bash? Wenn Sie das Script mit der Zsh ausführen möchten, ändern Sie den Hash-Bang zu `#!/bin/zsh` oder `#!/bin/sh`.

### Elementare Syntaxregeln

Außer dem Hash-Bang gibt es einige weitere Regeln, die Sie in Ihren Scripts beachten müssen:

- Zeilen, die mit # beginnen, sind Kommentare und werden nicht ausgewertet.
- Mit dem Zeichen \ können Sie lange Kommandos über mehrere Zeilen verteilen. Das Zeichen \ muss exakt am Zeilenende stehen, danach dürfen keine Leerzeichen mehr folgen!
- Einrückungen im Code sind erlaubt, aber optional.
- Die Bash ist äußerst pingelig, was die Verwendung von Leerzeichen im Code betrifft. `s="abc"` ist korrekt, die Varianten `s ="abc"` oder `s= "abc"` sowie `s = "abc"` sind alle drei falsch und lösen Fehler aus!

  Umgekehrt gibt es aber auch Fälle, in denen Leerzeichen vor oder nach Klammern erforderlich sind, damit die Bash die Konstruktion korrekt erkennt. Wenn Ihr Script


© Rheinwerk Verlag, Bonn 2024

nicht wie erwartet funktioniert und die Fehlermeldungen vage oder irreführend sind, werfen Sie zuerst einen Blick auf die Verwendung der Leerzeichen!

### Ein »richtiges« Beispiel: Backup-Script

Natürlich lesen Sie dieses Buch nicht, um Hello-World-Scripts auszuführen. Das zweite Beispiel ist etwas länger, erfüllt dafür aber eine reale Aufgabe: Es erzeugt ein komprimiertes Backup aller Dateien in Ihrem `Dokumente`-Verzeichnis und speichert das Backup im Verzeichnis `mybackups`:

```
#!/bin/bash
# Beispieldatei backup-documents.sh
# den Inhalt dieses Verzeichnisses speichern
documentdir=~/Dokumente
# Backup-Ort
backupdir=~/mybackups
# Backup-Verzeichnis erzeugen
mkdir -p $backupdir
# liefert z. B. date=27 for 2023-03-27
day=$(date '+%d')
# Backup erzeugen, als documents-<day>.tar.gz speichern
echo "Backup file: $backupdir/documents-$day.tar.gz"
tar czf $backupdir/documents-$day.tar.gz -C $documentdir .
```

Dieses Script erfordert einige Erklärungen.

- `documentdir=~/Dokumente` speichert den Namen Ihres Dokumente-Verzeichnisses in der Variablen `documentdir`. `~` ist dabei eine Kurzschreibweise für Ihr Heimatverzeichnis (unter Linux z. B. `/home/kofler`, unter macOS dagegen `/Users/kofler`).

  Je nachdem, unter welchem Betriebssystem bzw. unter welcher Distribution Sie arbeiten, müssen Sie `Dokumente` durch `Documents` ersetzen. Wenn macOS im Finder bei einer deutschen Spracheinstellung `Dokumente` anzeigt, lautet der wirkliche Verzeichnisname tatsächlich `Documents`!

- `backupdir=~/mybackups` speichert in einer weiteren Variablen den gewünschten Backup-Ort.

- `mkdir -p $backupdir` erzeugt das Backup-Verzeichnis. Die Option `-p` verhindert dabei Fehlermeldungen, wenn das Verzeichnis schon existiert (was spätestens beim zweiten Ausführen des Scripts der Fall ist).

- `day=$(date '+%d')` speichert den Tag des Monats in der Variablen `day`. `date` ist ein Kommando, das normalerweise das gesamte Datum samt Uhrzeit liefert. Mit dem zusätzlichen Parameter `+%d` wird stattdessen nur der Tag des Monats (`01` bis `31`) geliefert.

Persönliches Exemplar für Thomas Bartholomäus

Die Schreibweise `$(...)` bedeutet: Führe das in den Klammern enthaltene Kommando aus und gib das Ergebnis zurück.

- Das Kommando `echo` gibt den genauen Pfad der neuen Backup-Datei auf dem Bildschirm aus. Die Ausgabe dient nur als Feedback für Sie, damit Sie nachvollziehen können, dass das Script wie vorgesehen funktioniert.

  Beachten Sie, dass Sie Variablennamen bei der Zuweisung ohne `$` angeben, beim Auslesen aber `$` voranstellen müssen. Das ist eine der vielen merkwürdigen Syntaxregeln der Bash.

- Das Kommando `tar` erzeugt schließlich ein Backup aller Dateien im Verzeichnis `$documentdir`. Auch die Syntax dieses Kommandos ist merkwürdig:
  - `czf` gibt an, was das Kommando tun soll (*create*, *zip*, *file*). Der nächste Parameter gibt den Ort der Backup-Datei an, z. B. `/home/kofler/documents-27.tar.gz`.
  - `-C` bestimmt, welches Verzeichnis während der Ausführung von `tar` aktiv sein soll. Bei diesem Beispiel soll es sich nicht um das gerade aktive Verzeichnis handeln, sondern um das Verzeichnis, in dem sich die zu sichernden Dateien befinden.
  - Und `.` am Ende des Kommandos bedeutet, dass der gesamte Inhalt des Verzeichnisses gesichert werden soll (und nicht nur ausgewählte Dateien, was auch denkbar wäre).

  Eine ausführlichere Beschreibung der wichtigen `tar`-Optionen folgt in Abschnitt 6.3, »Dateien komprimieren und archivieren«.

Wenn Sie dieses Script einmal täglich ausführen, befinden sich nach einiger Zeit 31 Backup-Versionen im Verzeichnis `mybackups`, von `documents-01.tar.gz` bis hin zu `documents-31.tar.gz`. Das ist zwar ein wenig verschwenderisch, gibt Ihnen aber die Möglichkeit, alle innerhalb des letzten Monats irrtümlich überschriebenen oder gelöschten Dokumente wiederherzustellen. (Wie Sie die tägliche Backup-Erstellung automatisieren, verrate ich Ihnen in Kapitel 11, »Scripts automatisch ausführen«. Andere Varianten zur Durchführung von Backups beschreibe ich in Kapitel 15, »Backups«.)

Für den Einstieg war das ein bisschen viel auf einmal. Aber keine Angst, im Verlauf dieses Kapitels folgen noch eine Menge Grundlageninformationen, die den Code besser verständlich machen. Es ist mir in diesem Beispiel primär darum gegangen, Ihnen zu zeigen, wie »reale« Scripts aussehen.

**Beispieldateien**

Dieses Script befindet sich zusammen mit allen anderen längeren Listings in den Beispieldateien zu diesem Buch. Diese können Sie unter *https://rheinwerk-verlag.de/5672* herunterladen.


© Rheinwerk Verlag, Bonn 2024

### Scripts ohne ».‌/« ausführen

Finden Sie es lästig, dass Sie bei der Ausführung von Scripts immer ./ voranstellen müssen (oder sogar den vollständigen Pfad, wenn Sie sich gerade in einem anderen Verzeichnis befinden)? Dafür gibt es eine Lösung!

- **Eigenes Script-Verzeichnis:** Zum Ersten richten Sie ein eigenes Verzeichnis für Ihre Scripts ein, beispielsweise `myscripts` in Ihrem Heimatverzeichnis.
- **PATH-Variable erweitern:** Und zum Zweiten ergänzen Sie die Umgebungsvariable `PATH` um den vollständigen Pfad zu diesem Verzeichnis. `PATH` enthält standardmäßig diverse Verzeichnisse, in denen die Bash nach Kommandos sucht. Wenn Sie also `ls` ausführen, durchsucht die Bash der Reihe nach alle in `PATH` enthaltenen Verzeichnisse. Wenn `PATH` auch `myscripts` enthält, dann wird auch dieses Verzeichnis durchsucht.

  Zur Veränderung von `PATH` öffnen Sie mit einem Editor die Datei `.bashrc` in Ihrem Heimatverzeichnis (bzw. `.zshrc`, falls Sie die Zsh verwenden). Am Ende dieser Datei fügen Sie die folgende Zeile hinzu:

  ```
  # am Ende von .bashrc bzw. .zshrc
  ...
  export PATH=/home/kofler/myscripts:$PATH
  ```

  Anstelle von `/home/kofler/myscripts` müssen Sie natürlich den vollständigen Pfad zu Ihrem Script-Verzeichnis angeben. Achten Sie auch auf die korrekte Syntax! Vor und nach `=` und `:` dürfen keine Leerzeichen stehen. `PATH` wird das erste Mal ohne, das zweite Mal aber mit `$`-Zeichen geschrieben.

  Damit diese Änderung wirksam wird, müssen Sie sich aus- und neu einloggen. Danach können Sie Scripts, die in diesem Verzeichnis gespeichert sind, einfach durch Namensnennung ausführen.

## 3.6 Kommandos ausführen

Nachdem Sie sich nun ein wenig mit der Bash oder Zsh anfreunden konnten, folgt in den weiteren Abschnitten Grundlagenwissen. Ich beziehe mich dabei zwar grundsätzlich auf die Bash, aber nahezu alle Informationen gelten ebenso für die Zsh.

Noch ein Hinweis, bevor es richtig losgeht: Alle in diesem und den folgenden Abschnitten erläuterten Techniken gelten gleichermaßen für die interaktive Anwendung der Bash als auch für das Scripting!

Persönliches Exemplar für Thomas Bartholomäus

### Serielle Ausführung

Im einfachsten Fall – so wie beim Backup-Beispiel im vorigen Abschnitt – besteht ein Script aus einer Abfolge von Kommandos, die der Reihe nach ausgeführt werden:

```
# Kommandos nacheinander ausführen
command1
command2
command3
```

Es ist auch erlaubt, mehrere Kommandos, getrennt durch Strichpunkte, in einer Zeile anzugeben. Das spart Platz und ist vor allem bei der interaktiven Verwendung der Bash praktisch, wenn Sie mehrere zeitaufwendige Aufgaben nacheinander erledigen möchten, ohne jedes Mal ein neues Kommando einzugeben:

```
# gleichwertig
command1; command2; command3
```

### Konditionelle Kommandoausführung

Jedes Kommando liefert einen normalerweise unsichtbaren Fehlercode zurück. 0 bedeutet, dass kein Fehler aufgetreten ist. Jeder andere Wert weist auf einen Fehler bzw. auf ein negatives Ergebnis hin. Die Bezeichnung »Fehler« ist je nach Kontext eigentlich zu stark: Auch wenn eine mit dem Kommando `test` formulierte Bedingung nicht zugrifft oder wenn das Filterkommando `grep` den Suchbegriff nicht findet, gilt das Kommando als nicht erfolgreich ausgeführt und liefert einen Rückgabe-Code ungleich 0.

Die Bash bietet die Möglichkeit, die Ausführung des zweiten Kommandos vom Ergebnis des ersten abhängig zu machen. Die erste Variante verwendet die Zeichenkombination `&&`. Sie drückt aus, dass das zweite Kommando nur dann ausgeführt werden soll, wenn das erste erfolgreich war. (Analog können Sie natürlich auch drei und mehr Kommandos auf diese Weise verknüpfen.)

```
# Kommando 2 nur ausführen, wenn Kommando 1 erfolgreich war
command1 && command2
```

Die Anwendung von `&&` ist beispielsweise zweckmäßig, wenn das erste Kommando einen Test ausführt (existiert die Datei?) und das zweite Kommando nur bei einem positiven Testergebnis laufen soll.

Ein anderes Beispiel: Sie wollen ein Update durchführen. Zuerst aktualisieren Sie die Paketquellen (in Debian oder Ubuntu mit `apt update`). Nur wenn dieser Vorgang fehlerfrei gelingt, wird im zweiten Schritt das eigentliche Update durchgeführt (`apt full-upgrade`). Tritt hingegen schon bei `apt update` ein Fehler auf, weil z. B. gerade keine Internetverbindung besteht, ist die Ausführung des zweiten Kommandos sinnlos.


© Rheinwerk Verlag, Bonn 2024

```
# testet, ob data.csv existiert, und gibt
# dann eine entsprechende Meldung aus
test -f data.csv && echo "data.csv exists"

# Paketquellen aktualisieren;
# sofern erfolgreich, danach das eigentliche Update
# durchführen
apt update && apt full-upgrade
```

Alternativ zu && können Sie zwei Kommandos auch mit || verknüpfen. Dann gilt eine inverse Logik: Kommando 2 wird nur ausgeführt, wenn Kommando 1 einen Fehler verursacht:

```
# Kommando 2 nur ausführen, wenn Kommando 1 scheitert
command1 || command2
```

Anstelle von && und || können Sie natürlich auch `if`-Konstruktionen verwenden, die ich Ihnen in Abschnitt 3.11, »Verzweigungen«, vorstelle. && und || ermöglichen aber die Formulierung kompakteren Codes, der dem Stil traditioneller Shell-Scripts eher entspricht.

### Kommandos im Hintergrund ausführen

Normalerweise blockiert die Ausführung eines Kommandos das Terminal bzw. das Script. Das nächste Kommando kann erst gestartet werden, wenn das vorige Kommando fertig ist. Abhilfe schafft `command &`, um ein Kommando im Hintergrund auszuführen (also asynchron).

Die asynchrone Ausführung von Kommandos ist in Scripts nur selten zweckmäßig. Viel häufiger werden Sie % beim interaktiven Betrieb im Terminal brauchen. Beispielsweise können Sie mit `command &` ein Programm mit grafischer Benutzeroberfläche starten (z. B. den Webbrowser Firefox), ohne weitere Eingaben im Terminal zu blockieren.

Ein im Hintergrund laufender Prozess kann mit `fg` wieder zu einem Vordergrundprozess gemacht werden. Umgekehrt kann ein zeitintensives Kommando mit Strg+Z unterbrochen und dann mit `bg` im Hintergrund fortgesetzt werden.

### Kommandos in einer Subshell ausführen

Indem Sie mehrere Kommandos in Klammern stellen, starten Sie für deren Ausführung einen neuen Shell-Prozess und führen die Kommandos dort aus. Einzeilig gelingt das mit `(c1; c2; c3)`. In Scripts können Sie die Klammern frei platzieren:

```
command1    # in der Haupt-Shell ausführen
(           # die folgenden Kommandos in einer Subshell ausführen
command2
command3
command4
)
command5    # wieder in der Haupt-Shell
```

Innerhalb der Subshell stehen alle vorher definierten Variablen weiter zur Verfügung. Subshells haben aber den Vorteil, dass dort definierte und veränderte Variablen nur darin gelten und danach ihre Gültigkeit verlieren bzw. ihren vorherigen Wert wieder annehmen. Auch ein mit `cd` gewechseltes Verzeichnis gilt nur innerhalb der Subshell. Mit dem Ende der Subshell wird das zuvor gültige Verzeichnis wieder aktiv.

Subshells können zur besseren Strukturierung des Codes verwendet werden. Wenn Sie innerhalb einer Subshell `exit` ausführen, beenden Sie nur die Subshell, nicht aber das in der Haupt-Shell laufende Script.

## 3.7 Standardeingabe und Standardausgabe

In der Bash gibt es drei »Kanäle«, die Kommandos zur Ein- und Ausgabe verwenden:

- **Standardausgabe (Standard Output = stdout):** Ergebnisse eines Kommandos werden an die sogenannte »Standardausgabe« geleitet. Wenn Sie `ls` ausführen, dann ist das Ergebnis dieses Kommandos die Liste aller Dateien im lokalen Verzeichnis. Dieses Ergebnis wird in Textform ausgegeben und normalerweise im Terminal angezeigt.
- **Fehlermeldungen (Standard Error = stderr):** Auch Fehlermeldungen werden normalerweise im Terminal angezeigt. Intern werden Fehlermeldungen aber an einen anderen Kanal geleitet, woraus sich die Möglichkeit ergibt, normale Ausgaben und Fehlermeldungen unterschiedlich zu behandeln.
- **Standardeingabe (Standard Input = stdin):** Viele häufig benutzte Variablen verarbeiten Textdateien, deren Name als Parameter übergeben wird. Fehlt diese Angabe, erwartet das Kommando den zu verarbeitenden Text von der Standardeingabe. Sie müssen also im Terminal Text eingeben, den das Kommando dann empfängt. Um auszudrücken, dass die Eingabe zu Ende ist, drücken Sie in diesem Fall `Strg`+`D`.

Auf den ersten Blick scheint es sich bei den drei Standardkanälen um Selbstverständlichkeiten zu handeln: Wo sonst als im Terminal soll ein Kommando seine Ergebnisse anzeigen, wo sonst als durch Ihre Eingaben soll ein Kommando Input erhalten?


© Rheinwerk Verlag, Bonn 2024

### Ein- und Ausgabe umleiten

Der Clou an der Sache ist die Möglichkeit, Ein- und Ausgabe umzuleiten. Mit dem Aufruf `command > file` können Sie das Ergebnis eines Kommandos in einer Textdatei speichern. Vorsicht: Falls die Datei bereits existiert, wird sie überschrieben.

Analog liest `command < file` die zu verarbeitenden Daten aus einer Textdatei, anstatt auf Terminal-Eingaben zu warten. Tabelle 3.2 fasst die Syntax für einige weitere Spielarten zusammen. Auf die Heredoc- und Herestring-Syntax gehe ich in Abschnitt 3.10, »Zeichenketten«, näher ein.

| Kommando | Funktion |
|---|---|
| `command > out.txt` | speichert die Standardausgabe in der Datei |
| `command >> out.txt` | fügt die Ausgabe am Ende der Datei hinzu |
| `command 2> errors.txt` | speichert Fehlermeldungen in der Datei |
| `command &> all.txt` | speichert Ausgaben und Fehler in der Datei |
| `command >& all.txt` | wie oben |
| `command >&2` | leitet die Standardausgabe in den Fehlerkanal um |
| `command < in.txt` | verarbeitet die Datei |
| `command << EOF` | betrachtet alle Zeilen bis EOF als Standardeingabe (Heredoc) |
| `command <<< $var` | betrachtet den Inhalt der Variablen als Standardeingabe (Herestring) |

**Tabelle 3.2** Operatoren zur Ein- und Ausgabeumleitung

Im einfachsten Fall dient die Ausgabeumleitung dazu, Ergebnisse dauerhaft zu speichern:

```
ls  > filelist.txt
```

Die Eingabeumleitung ist bei interaktiven Kommandos praktisch. So erwartet der MySQL-Client `mysql` normalerweise die Eingabe von SQL-Kommandos. Bei der folgenden Anwendungen werden diese Kommandos aus einer Textdatei gelesen, beispielsweise um eine Datenbank aus einem Backup wiederherzustellen:

```
mysql < backup.sql
```

Das Kommando `cat`, das normalerweise einfach eine Textdatei am Bildschirm ausgibt, kann dazu verwendet werden, um eine neue Datei zu erzeugen:

Persönliches Exemplar für Thomas Bartholomäus

```
cat > newfile.txt
  first input line
  second input line
  <Strg>+<D>
```

Weil cat der Parameter mit der auszugebenden Datei fehlt, erwartet das Kommando interaktive Eingaben am Terminal. Diese werden in die Datei newfile.txt umgeleitet. Mit diesem »Trick« können Sie kleine Textdateien erstellen, ohne einen Editor aufzurufen.

**Vorsicht**

Sie können allerdings nicht eine Datei lesen und gleichzeitig schreiben. Das Kommando `sort < file > file` sortiert nicht den Inhalt der Textdatei und speichert dann das Ergebnis, sondern löscht stattdessen `file` (weil `> file` zuerst ausgeführt wird).

**Heredoc-Syntax**

Ein Sonderfall der Eingabeumleitung ist die Heredoc-Syntax. Darauf gehe ich in Abschnitt 3.10, »Zeichenketten«, getrennt ein.

### Der Pipe-Operator »|«

Ganz sind wir mit der Ein- und Ausgabeumleitung noch nicht fertig. Das vielleicht coolste Feature fehlt noch: Mit `command1 | command2` produziert `command1` ein Ergebnis. Dieses wird aber nicht angezeigt, sondern an `command2` weitergeleitet. Der Pipe-Operator | macht also die Standardausgabe von `command1` zur Standardeingabe von `command2`. Der Pipe-Operator kann mehrfach angewendet werden, wobei jedes Mal die Ausgabe des einen Kommandos an das nächste Kommando weitergegeben wird.

Daraus ergeben sich derart vielfältige Möglichkeiten zur Analyse und Verarbeitung von Textdateien, dass ich diesem Thema ein eigenes Kapitel gewidmet habe. An dieser Stelle müssen die folgenden einfachen Beispiele reichen, die sich auf die Linux-spezifische Datei /etc/passwd beziehen. Diese Datei enthält Informationen zu allen Accounts des Systems. Die Kommandos grep, cut, sort und head erläutere ich dann in Kapitel 8, »Textverarbeitung«, näher.

```
# zeigt alle Zeilen aus /etc/passwd an, die 'nologin' enthalten
grep nologin /etc/passwd

# extrahiert aus den Ergebniszeilen die erste Spalte
# mit dem Account-Namen
grep nologin /etc/passwd | cut -d ':' -f 1
```


© Rheinwerk Verlag, Bonn 2024

```
# sortiert die Namen
grep nologin /etc/passwd | cut -d ':' -f 1 | sort

# und zeigt die ersten fünf Ergebnisse an
grep nologin /etc/passwd | cut -d ':' -f 1 | sort | head -n 5
```

### Ausgaben umleiten und gleichzeitig anzeigen

Das Kommando `tee` dupliziert die Standardeingabe und zeigt diese auf der Standardausgabe an *und* leitet sie an eine Datei weiter. Beim folgenden Kommando wird das Ergebnis von `command` also im Terminal angezeigt *und* in der Datei `out.txt` gespeichert.

```
command | tee out.txt
```

## 3.8 Globbing, Brace Extension, Umgang mit Datei- und Verzeichnisnamen

Was passiert, wenn Sie an ein Kommando `*.txt` oder `directory??/*.pdf` übergeben, weil Sie alle Textdateien oder alle PDF-Dateien in einigen Unterverzeichnissen verarbeiten möchten? Der schwer zu übersetzende Begriff »Globbing« beschreibt die Mechanismen, die die Bash für derartige Aufgaben zur Verfügung stellt. Tabelle 3.3 fasst die Bedeutung der »Joker-Zeichen« der Bash zusammen. (Die Zeichen sind eine sehr stark vereinfachte Variante der »regulären Ausdrücke«, die Sie in Kapitel 9 kennenlernen werden.)

| Zeichen | Bedeutung |
|---|---|
| `?` | ein beliebiges Zeichen |
| `*` | beliebig viele (auch null) beliebige Zeichen |
| `**` | rekursives Globbing (globstar) |
| `[aeiou]` | eines der angegebenen Zeichen |
| `[a-z]` | Zeichenbereich |
| `[a-zäöüß]` | alle Kleinbuchstaben |
| `[A-Fa-f0-9]` | eine hexadezimale Stelle |
| `[!a-f]` | keines der Zeichen von `a` bis `f` |
| `[^a-f]` | keines der Zeichen von `a` bis `f` |

**Tabelle 3.3** Globbing-Zeichen

Persönliches Exemplar für Thomas Bartholomäus

Entscheidend ist, dass für das Globbing die Bash und nicht das jeweilige Kommando zuständig ist. Wenn Sie also z. B. `cp *.jpg some/directory/` ausführen, um einige Bilder in ein Unterverzeichnis zu kopieren, dann kümmert sich die Bash und *nicht* das `cp`-Kommando darum, die Liste der zu kopierenden Dateien zu ermitteln. Das ist sinnvoll, weil andernfalls beinahe jedes Kommando entsprechenden Code benötigen würde. Weil aber die Bash zuständig ist, erfolgt die Auswertung von Zeichen wie `*` oder `?` zentral und immer nach den gleichen Regeln: Die Bash wertet zuerst alle Globbing-Zeichen aus, bringt diese mit den Dateinamen im gerade aktuellen Verzeichnis in Übereinstimmung und übergibt dann die fertige Liste von Parametern an das betreffende Kommando. `cp` sieht also nie `*.jpg`, sondern nur die fertige Liste der Dateinamen, z. B. `img_1.jpg`, `img_2.jpg` und `img_3.jpg`.

Falls Sie unsicher sind, welche Dateinamenliste ein Globbing-Ausdruck liefert, führen Sie `echo <ausdruck>` aus, z. B. so:

```
echo img_*.raw
  img_23433.raw img_23434.raw img_23435.raw
```

Noch mehr Komfort bietet die Zsh: Dort können Sie einfach [Tab] drücken, um den Globbing-Ausdruck durch die im aktuellen Verzeichnis passenden Dateinamen zu ersetzen.

### Keine Globbing-Ergebnisse

Globbing funktioniert nur, wenn es zumindest *eine* passende Datei gibt. Sollte sich im aktuellen Verzeichnis gar keine JPEG-Datei befinden, dann bleibt die Auswertung von `*.jpg` ergebnislos. Die Bash übergibt dann die originale Zeichenkette an `cp`. Das Kommando sucht nach einer Datei, die Zeichen für Zeichen `*.jpg` heißt. Diese Datei ist nicht zu finden, weswegen `cp` die Fehlermeldung *file not found* liefert.

Derartige Fehler können Sie mit `shopt -s nullglob` verhindern. Die Nullglob-Option bewirkt, dass das Muster entfernt wird, wenn es keine passenden Dateien gibt. `cp *.jgp some/path/` löst allerdings selbst dann einen Fehler aus, diesmal, weil an das Kommando nur ein Parameter übergeben wurde (es müssen mindestens zwei sein).

Wenn Sie das Globbing verhindern möchten, stellen Sie die Zeichenkombination in einfache Apostrophe. Vielleicht haben Sie eine Datei erzeugt, deren Dateiname `???` lautet. (Das ist zwar nicht zielführend, aber grundsätzlich erlaubt.) Um diese Datei zu löschen, führen Sie `rm '???'` aus.

## Rekursives Globbing

Die Zeichenkombination `**` erfasst rekursiv alle (Unter-)Verzeichnisse. Damit hat `ls **/*.pdf` eine ähnliche Bedeutung wie das Suchkommando `find . -name '*.pdf'`. Weil das rekursive Durchsuchen aller Verzeichnisse viel Zeit kosten kann, ist dieses


© Rheinwerk Verlag, Bonn 2024

Verhalten standardmäßig deaktiviert. Bevor Sie die Funktion nutzen können, müssen Sie diese mit `shopt -s globstar` aktivieren. Das folgende Beispiel kopiert sämtliche PDF-Dateien in das zentrale Verzeichnis `all-my-pdfs`.

```
shopt -s globstar
mkdir all-my-pdfs
cp **/*.pdf all-my-pdfs
```

## Zugriff auf wichtige Verzeichnisse

Losgelöst von den Globbing-Regeln gibt es für den Zugriff auf oft benötigte Verzeichnisse einige Kurzschreibweisen (siehe Tabelle 3.4).

| Zeichen | Bedeutung |
|---|---|
| `.` | das aktuelle Verzeichnis |
| `..` | das übergeordnete Verzeichnis (Parent-Verzeichnis) |
| `~` | das eigene Heimatverzeichnis |
| `~name` | das Heimatverzeichnis von `name` |

**Tabelle 3.4** Kurzschreibweisen für wichtige Verzeichnisse

## Brace Extension

In der Bash können Sie in geschwungenen Klammern durch Kommas getrennte Aufzählungen oder mit `..` gebildete Bereiche formulieren. Vor der Ausführung des Kommandos werden alle möglichen Kombinationen gebildet, der in Klammern angegebene Ausdruck wird also »expandiert«. Im Unterschied zum Globbing nimmt die Brace Extension aber keine Rücksicht darauf, ob entsprechende Dateien schon existieren oder nicht. Am einfachsten ist der Mechanismus anhand von Beispielen zu verstehen:

```
echo {a..f}.txt
  a.txt b.txt c.txt d.txt e.txt f.txt

echo /{etc,usr,var}
  /etc /usr /var

echo {1..3}{a,e,i,o,u}
  1a 1e 1i 1o 1u 2a 2e 2i 2o 2u 3a 3e 3i 3o 3u
```

Auf den ersten Blick sieht das wie eine Spielerei aus. Tatsächlich gibt es aber durchaus Anwendungen. Stellen Sie sich vor, sie wollen für das aktuelle Jahr (ermittelt mit `date '+%Y'`) für jeden Monat ein Unterverzeichnis erstellen. In der Bash ist das ein Einzeiler:

```
mkdir -p $(date '+%Y')/{01..12}
```

Dabei liefert `$(date ...)` das Ergebnis dieses Kommandos zurück. Die Option `-p` bei `mkdir` bewirkt, dass nicht nur einzelne Verzeichnisse erzeugt werden, sondern bei Bedarf ganze Verzeichnisketten (für das Jahr 2023 also zuerst `2023` und darin `01`, `02` usw.).

Die Brace-Extension kann auch zur Bildung von Schleifen verwendet werden. `i` ist dabei eine Variable. Grundlagen zu Variablen und Schleifen folgen in Abschnitt 3.9 und in Abschnitt 3.12:

```
for i in {1..10}; do echo $i; done
  1
  2
  ...
```

### Datei- und Verzeichnisnamen mit Leerzeichen

In der Bash werden an ein Kommando übergebene Parameter durch Leerzeichen getrennt. `cp a.txt b.txt /my/directory` kopiert also die Dateien `a.txt` und `b.txt` in ein Verzeichnis.

Problematisch wird dieses Verhalten, wenn Dateien selbst Leerzeichen enthalten. (Eingefleischte Linux-Anwenderinnen und -Anwender vermeiden diesen Fall nach Kräften!) `cp filename with blanks.txt /my/directory` glaubt, dass Sie die drei Dateien `filename`, `with` und `blanks.txt` kopieren wollen, und erkennt nicht, dass `filename with blanks.txt` *eine* Datei ist! Um diese Doppeldeutigkeit zu klären, müssen Sie den betreffenden Namen in Anführungszeichen stellen, also:

```
cp "filename with blanks.txt" /my/directory
```

Obwohl es an dieser Stelle abermals ein Vorgriff auf die noch gar nicht behandelten Schleifen ist (siehe Abschnitt 3.12, »Schleifen«): Die Anführungszeichen sind auch dann notwendig, wenn Sie automatisiert Dateien verarbeiten. So kann der Globbing-Ausdruck `*.txt` durchaus auch Dateinamen mit Leerzeichen liefern. Die folgende Schleife, die einfach Detailinformationen zu jeder entsprechenden Datei anzeigen soll, löst dann Fehler aus:

```
# löst Fehler aus, wenn eine *.txt-Datei Leerzeichen enthält
for fn in *.txt; do ls -l $fn; done
```

Der Code funktioniert erst dann korrekt, wenn Sie beim `ls`-Kommando die Variable `fn` in Anführungszeichen stellen:

```
# korrekte Variante des vorigen Beispiels
for fn in *.txt; do ls -l "$fn"; done
```


© Rheinwerk Verlag, Bonn 2024

Noch komplizierter wird es, wenn die an ein Bash-Script übergebenen Parameter auswerten soll und dabei Dateinamen mit Leerzeichen vorkommen. Eine korrekte Verarbeitung gelingt nur, wenn Sie vorher die Bash-Variable `IFS` verändern (siehe Abschnitt 3.9, »Variablen«, und Abschnitt 3.12, »Schleifen«).

## 3.9 Variablen

Der Umgang mit Variablen in der Bash ist gewöhnungsbedürftig, insbesondere dann, wenn Sie schon in anderen, »höheren« Programmiersprachen gearbeitet haben. Ich beginne einmal mit drei Grundregeln:

- Variablennamen muss bei der Auswertung (beim Lesen) meistens ein Dollarzeichen vorangestellt werden, bei der Zuweisung (beim Schreiben) hingegen nicht. Sie werden noch sehen, es gibt Ausnahmen von dieser Regel.
- Bei der Zuweisung `myvar=value` sind vor und nach dem Zeichen `=` keine Leerzeichen erlaubt!
- Bash-Variablen speichern normalerweise Zeichenketten. (Ja, die Bash kann auch mit Zahlen umgehen und unterstützt sogar Arrays. Aber das sind Sonderfälle, die zum Teil syntaktische Klimmzüge erfordern.)

Ein erstes Beispiel illustriert den Umgang mit Variablen. Erläuterungen zur `if`-Konstruktion mit dem Vergleichsoperator `-gt` (*greater than*) folgen in Abschnitt 3.11, »Verzweigungen«.

```
myvar="123"
myvar='123'             # hier gleichwertig
myvar=123               # hier gleichwertig
echo $myvar             # Ausgabe: 123
echo "Inhalt: $myvar"   # Ausgabe: Inhalt: 123
if [ $myvar -gt 100 ]; then
  echo "myvar ist größer als 100"
fi
```

### Variablen initialisieren und löschen

Bei der Zuweisung von Variablen können Sie mit `var="abc"` oder auch mit `var='abc'` den Zeichenkettencharakter verdeutlichen. Syntaktisch erforderlich sind Apostrophe nur, wenn der zu speichernde Ausdruck Leer- oder andere Sonderzeichen enthält, also etwa für `var="abc efg"` oder `var='$x'`.

Den Unterschied zwischen den beiden Apostrophen erläutere ich in Abschnitt 3.10, »Zeichenketten«. Kurz gefasst wird bei `"abc $myvar abc"` der Inhalt der Variablen

myvar eingesetzt (wie beim vorigen echo-Kommando), während 'abc $myvar abc' die Zeichenkette unverändert übernimmt.

Wenn Sie in einer Variablen das Ergebnis eines anderen Kommandos speichern möchten, verwenden Sie die Schreibweise myvar=$(command). Die etwas sperrige Bezeichnung für den Ausdruck $(...) lautet »Kommandosubstitution«, weil das angegebene Kommando ausgeführt und durch seinen Inhalt ersetzt wird. Alternativ ist auch die gleichwertige Formulierung mit nach rechts gerichteten Apostrophen erlaubt, also myvar=`command`.

Um eine Variable zu löschen, führen Sie unset myvar aus oder weisen einfach keinen Inhalt zu (also myvar=).

### Variablen deklarieren

In der Bash ist es nicht notwendig, Variablen vor ihrer Verwendung zu deklarieren – und dennoch gibt es ein declare-Kommando! Damit können Sie das Verhalten von Variablen beeinflussen, wie dies im folgenden Listing zusammengefasst ist:

```
# Konstante, kann später nicht verändert werden (r = readonly)
declare -r const="abc"

# myvar kann nur ganze Zahlen speichern (i = integer)
declare -i myvar
myvar="abc"    # kein Fehler, aber myvar=0

# normales und assoziatives Array (Details folgen gleich)
declare -a myarray
declare -A mymap
```

### Rechnen in der Bash

Die Bash »denkt« in Zeichenketten. Berechnungen bzw. die Auswertung mathematischer Ausdrücke sind nur in einem speziellen Kontext mit let oder innerhalb von doppelten runden Klammern möglich (siehe Tabelle 3.5). Sämtliche mathematischen Berechnungen verwenden prinzipiell nur ganze Zahlen.

| Ausdruck | Bedeutung |
|---|---|
| ((expression)) | führt den enthaltenen mathematischen Ausdruck aus |
| $((expression)) | wie oben, gibt aber ein Ergebnis zurück |
| let myvar=expression | speichert das Ergebnis des Ausdrucks in einer Variablen |

**Tabelle 3.5** Mathematik in der Bash


© Rheinwerk Verlag, Bonn 2024

### Mathematische Operatoren

Beachten Sie, dass mathematische Operatoren wie + und *, aber auch Vergleichsoperatoren wie < oder >=, ausschließlich im Kontext von Tabelle 3.5 zulässig sind. Insbesondere < und > dienen in der Bash ja üblicherweise zur Ein- und Ausgabeumleitung.

Die folgenden Zeilen geben einige Beispiele für mathematische Ausdrücke in der Bash. Dabei sind einige sonst übliche Regeln aufgeweicht. Beispielsweise funktionieren innerhalb von `((...))` Zuweisungen auch mit Leerzeichen vor und nach =. (Das gilt aber nicht für `let`!) Außerdem ist es zulässig, beim Auslesen von Variablen auf das vorangestellte Zeichen $ zu verzichten.

```
x=2
y=$x+3              # Vorsicht, falsch: y enthält "2+3"!
let y=x+3           # y=5, x kann ohne $ ausgewertet werden
((y=x+3))           # gleichwertig
(( y = x + 3 ))     # gleichwertig, innerhalb von (( ... )) sind
                    # Leerzeichen erlaubt
y=$((x+3))          # gleichwertig
y=$(( x + 3 ))      # gleichwertig
(( y++ ))           # y=6
echo $(( y > 10 )) # trifft nicht zu
  0
echo $(( y < 10 )) # trifft zu
  1
```

## Arrays

Neben einfachen Variablen kennt die Bash auch Arrays. Gewöhnliche Arrays verwenden ganze Zahlen als Index. Beachten Sie die von vielen anderen Programmiersprachen abweichende Syntax `${feld[n]}` für den Zugriff auf das n-te Element.

```
x=()               # Definition eines leeren Arrays
x[0]='a'           # Array-Elemente zuweisen
x[1]='b'
x[2]='c'

x=('a' 'b' 'c')    # Kurzschreibweise für die obigen 4 Zeilen

echo ${x[1]}       # ein Array-Element lesen
echo ${x[@]}       # alle Array-Elemente lesen
```

Persönliches Exemplar für Thomas Bartholomäus

### Bash versus Zsh

Ich habe in diesem Kapitel mehrfach darauf hingewiesen, dass die Bash und die Zsh in den Grundfunktionen kompatibel zueinander sind.

Eine der wenigen Ausnahmen betrifft die Array-Indizes. Während diese bei der Bash von 0 bis zur Elementanzahl minus 1 reichen, hat das erste Arrayelement in der Zsh den Index 1! Der Index 0 ist nicht erlaubt (es sei denn, Sie aktivieren die Option `KSH_ARRAYS`). Weitere Hintergründe können Sie hier nachlesen:

*https://stackoverflow.com/questions/50427449*

Mit `mapfile` können Sie eine ganze Textdatei zeilenweise in die Elemente eines Arrays einlesen:

```
mapfile z < textdatei
```

Um die Elemente eines Arrays mit den Spalten einer Zeile zu initialisieren, verwenden Sie am besten das Kommando `read` mit der Array-Option `-a`. Details zu `read` sowie zu mit <<< formulierten Herestrings folgen in Abschnitt 3.10, »Zeichenketten«.

```
data="abc efg ijk opq uvw"
read -a myvar <<< $data
echo ${myvar[2]}  # Ausgabe ijk
```

Wenn die Spalten bzw. »Worte« einer Zeichenkette nicht durch Whitespace getrennt sind, verändern Sie vorübergehend den in der vordefinierten Variablen `IFS` enthaltenen *Internal Field Separator*. Standardmäßig enthält diese Variable ein Leerzeichen, ein Tabulatorzeichen und ein Zeilenumbruchzeichen (also die drei wichtigsten Whitespace-Zeichen). Im folgenden Beispiel sind die Worte dagegen durch das Zeichen : getrennt. Weil die `IFS`-Deklaration in derselben Zeile wie das `read`-Kommando erfolgt, gilt die Änderung nur für dieses eine Kommando. In der Folge hat `IFS` wieder die Defaulteinstellung. (Das ist wichtig, weil `IFS` Auswirkungen auf viele Kommandos hat.)

```
data="abc:efg:ijk lmn:opq:uvw"
IFS=':' read -a myvar <<< $data
echo ${myvar[2]}  # Ausgabe ijk lmn
```

Neben gewöhnlichen Arrays kennt die Bash auch assoziative Arrays. Anstelle von Zahlen sind beliebige Zeichenketten als Indizes erlaubt. Je nach Programmiersprache kennen Sie diese Datenstruktur auch als Dictionary oder Map. Zur Verwendung assoziativer Arrays müssen Sie die Variable explizit mit `declare -A` deklarieren.


© Rheinwerk Verlag, Bonn 2024

```
declare -A y       # Definition eines leeren assoziativen Arrays
y['abc']='123'     # Element eines assoziativen Arrays zuweisen
y['efg']='xxx'
y=( [abc]=123 [efg]=xxx )     # Kurzschreibweise
echo ${y[abc]}                # ein Array-Element lesen
```

### Vordefinierte Variablen

Bash-Scripts können auf vordefinierte Variablen zugreifen (siehe Tabelle 3.6). Diese Variablen können nur gelesen, aber nicht verändert werden. Die Abkürzung PID steht dabei für *Process ID*, also für die interne Prozessnummer.

| Variable | Bedeutung |
|---|---|
| $# | Anzahl der an das Script übergebenen Parameter |
| $0 | Dateiname des ausgeführten Scripts |
| $1 bis $9 | Parameter 1 bis 9 |
| ${10}, ${11} | Zugriff auf weitere Parameter |
| $* oder $@ | Gesamtheit aller übergebenen Parameter |
| $? | Rückgabewert des letzten Kommandos |
| $! | PID des zuletzt gestarteten Hintergrundprozesses |
| $$ | PID der aktuellen Shell |

**Tabelle 3.6** Vordefinierte bash-Variablen

In der Praxis besonders wichtig ist die Auswertung der an ein Script übergebenen Parameter. Deren Anzahl verrät `$#`. Auf die einzelnen Parameter können Sie mit `$1`, `$2` usw. zugreifen. Beispiele für die Auswertung vieler Parameter in Schleifen oder durch `case`-Konstruktionen folgen in Abschnitt 3.11, »Verzweigungen«, sowie in Abschnitt 3.12, »Schleifen«.

### Umgebungsvariablen

Gewöhnliche Variablen verlieren nach der Ausführung eines Scripts ihre Gültigkeit. Ausgenommen von dieser Regel sind Umgebungsvariablen, die mit `export myvar=...` deklariert oder verändert werden. `printenv` listet alle Umgebungsvariablen mit ihrem aktuellen Wert auf. Wichtige Umgebungsvariablen (siehe Tabelle 3.7) werden in `/etc/profile`, in `.profile` sowie in `.bashrc` oder `.zshrc` voreingestellt.

| Variable | Bedeutung |
|---|---|
| HOME | das Heimatverzeichnis |
| LANG | Lokalisierungseinstellungen, also Sprache und Zeichensatz |
| PATH | Verzeichnisse, in denen nach Programmen gesucht wird |
| PS1 | Inhalt/Aussehen des Kommando-Prompts |
| PWD | das aktuelle Verzeichnis |
| USER | der Login-Name des aktiven Nutzers |
| SHELL | Name der aktiven Shell |
| EDITOR | Name des Default-Editors (oft vi oder nano) |
| IFS | Internal Field Separator (Zerlegung von Zeichenketten in Worte) |

**Tabelle 3.7** Wichtige Umgebungsvariablen

Abweichend von »gewöhnlichen« Variablen können Umgebungsvariablen mit `ENVVAR=... command` nur für die Ausführung eines Kommandos verändert werden. Ein Beispiel dafür habe ich vorhin schon beim Einlesen von Teilen einer Zeichenkette in ein Array gegeben:

```
# der veränderte IFS-Inhalt gilt nur für read
IFS=':' read -a myvar
```

Eine andere häufige Anwendung sieht so aus:

```
# das Kommando ohne die aktuellen Spracheinstellungen ausführen
LANG= command
```

Damit wird die Umgebungsvariable `LANG` mit den Spracheinstellungen nur für die Ausführung von `command` gelöscht. `command` wird daher ohne die sonst gültigen Spracheinstellungen ausgeführt und liefert gegebenenfalls Fehlermeldungen in englischer Sprache. (Englisch gilt als Defaultsprache, wenn `LANG` leer ist. Englische Fehlermeldungen sind hilfreich, wenn Sie im Internet nach der Ursache eines Problems suchen.) `LANG` wird aber nicht dauerhaft gelöscht, sondern steht bei den weiteren Kommandos wieder zur Verfügung.

### Keine vorübergehende Änderung für Bash-Ablaufstrukturen

Die Syntax `ENVVAR=... command` kann nur für gewöhnliche Kommandos verwendet werden, nicht aber für Schleifen oder Verzweigungen, die Sie mit `if`, `for`, `while` etc. einleiten.


© Rheinwerk Verlag, Bonn 2024

## 3.10 Zeichenketten

Zeichenketten sind der zentrale Datentyp der Bash – so viel habe ich ja schon erwähnt. Dennoch mutet der Umgang mit Zeichenketten bisweilen befremdlich an. Dieser Abschnitt fasst die wichtigsten Funktionen rund um Zeichenketten zusammen.

### Einfache versus doppelte Apostrophe

In der Bash können Zeichenketten sowohl in einfache als auch in doppelte Apostrophe gestellt werden:

- 'abc$efg': In diesem Fall wird die Zeichenkette exakt übernommen.
- "abc": Bei der Variante ersetzt die Bash im Text enthaltene Variablen (hier $efg) durch ihren Inhalt und führt eventuell auch andere Substitutionsverfahren aus, die ich im weiteren Verlauf dieses Abschnitts erläutern werde.

In der Bash können mehrere Zeichenketten ohne Operator aneinandergefügt werden. Die einzige Voraussetzung ist, dass die Bash die Zeichenketten als solche erkennen kann. In der ersten Zeile des folgenden Listings ist das nicht der Fall.

```
x=abc efg            # Fehler!
x=abc                # OK
x="abc"              # besser
x='abc'              # hier gleichwertig
y="123"$x"456"       # -> 123abc456
y='123'$x'456'       # -> 123abc456
y="123 $x 456"       # -> 123 abc 456
y='123 $x 456'       # -> 123 $x 456
y=123 $x 456         # Fehler!
```

Es ist erlaubt, Zeichenketten über mehrere Zeilen zu verteilen. Damit bei der Ausgabe durch echo die Zeilenumbrüche erhalten bleiben, müssen Sie die Variable in Anführungszeichen setzen (siehe auch den folgenden Abschnitt):

```
myvar='1. Zeile
2. Zeile'
echo "$myvar"
```

### Zeichenketten ausgeben (echo)

Das Kommando echo zur Ausgabe von Zeichenketten kennen Sie natürlich schon:

```
myname="Michael"
echo "Hello, $myname!"
  Hello, Michael!
```

Persönliches Exemplar für Thomas Bartholomäus

An dieser Stelle möchte ich auf einige echo-Besonderheiten hinweisen. Mit der Option -n verhindern Sie, dass echo nach der Ausgabe eine neue Zeile beginnt. So können Sie die Ausgabe einer Zeile über mehrere Kommandos verteilen:

```
echo -n "Hello, "
echo $myname
```

Mit der Option -e erreichen Sie, dass in der Zeichenkette enthaltene Backslash-Sequenzen korrekt interpretiert werden, dass also die beiden Zeichen \ und n als Zeilenumbruch erkannt werden:

```
myvar="abc\nefg"
echo $myvar
  abc\nefg
echo -e $myvar
  abc
  efg
```

Verwirrend ist auch der Umgang mit echten Zeilenumbrüchen. Bei echo $myvar werden diese durch Leerzeichen ersetzt. Erst echo "$myvar" mit Anführungszeichen liefert eine mehrzeilige Ausgabe:

```
myvar="abc
efg"
echo $myvar
  abc efg
echo "$myvar"
  abc
  efg
```

Fehlermeldung leiten Sie mit >&2 in den Stderr-Kanal um:

```
echo "Fehlermeldung" >&2
echo >&2 "Fehlermeldung"   # gleichwertig
```

**printf**

Die aus vielen Programmiersprachen bekannte Funktion printf steht unter Linux und macOS als Kommando zur Verfügung. Es eignet sich besser als echo, um Ausgaben zu formatieren. Syntaxdetails verrät man 1 printf.

## Farben

Sie können Text auch farbig ausgeben. Zur Umstellung der Farbe bauen Sie in die Ausgabe sogenannte ANSI-Escape-Sequenzen ein (siehe Tabelle 3.8). Zur Veränderung der Hintergrundfarbe ersetzen Sie 3n durch 4n.


© Rheinwerk Verlag, Bonn 2024

| Farbe | Code | Farbe | Code |
|---|---|---|---|
| Schwarz | 0;30 | Grau | 1;30 |
| Blau | 0;34 | Hellblau | 1;34 |
| Grün | 0;32 | Hellgrün | 1;32 |
| Cyan | 0;36 | Hell-Cyan | 1;36 |
| Rot | 0;31 | Hellrot | 1;31 |
| Violett | 0;35 | Rosa | 1;35 |
| Braun | 0;33 | Gelb | 1;33 |
| Hellgrau | 0;37 | Weiß | 1;37 |

**Tabelle 3.8** ANSI-Farbcodes für Vordergrundfarben

Die Farbcodes müssen zwischen den Steuersequenzen `\033[` und `m` platziert werden. Damit Ihr Code nicht vollständig unleserlich wird, speichern Sie die Farbcodes am besten in Variablen. `echo` verarbeitet die Farbcodes nur dann korrekt, wenn Sie die Option `-e` verwenden. Die Schreibweise `${var}` anstelle von `$var` ist hier zweckmäßig, um die Variablennamen ohne Leerzeichen vom restlichen Text abzugrenzen.

```
BLUE='\033[0;34m'
RED='\033[0;31m'
NOCOLOR='\033[0m'
echo -n -e "Textausgabe in ${RED}Rot${NOCOLOR} "
echo    -e "und ${BLUE}Blau${NOCOLOR}."
```

Das tatsächliche Aussehen der Farben kann je nach Terminal(konfiguration) variieren. Anstatt Farben heller darzustellen (z. B. `1;34` für Hellblau) erfolgt die Ausgabe je nach Terminal in der Grundfarbe, aber dafür mit fetter Schrift. Eine Referenz weiterer Codes, mit denen Sie auch die Cursorposition verändern können, finden Sie hier:

*https://en.wikipedia.org/wiki/ANSI_escape_code*
*https://tldp.org/HOWTO/Bash-Prompt-HOWTO/c327.html*

### Schlechte Lesbarkeit, schwierige Weiterverarbeitung

Der Einsatz von Farben ist mit Nachteilen verbunden. Einerseits kann darunter die Lesbarkeit leiden, zumal Sie sich nicht auf eine bestimmte Hintergrundfarbe im Terminal verlassen können. Viele Benutzerinnen und Benutzer verwenden Terminals mit dunklem Hintergrund, andere ziehen einen hellen oder gar weißen Hintergrund vor.

Persönliches Exemplar für Thomas Bartholomäus

Außerdem erschweren Farb-Codes die Weiterverarbeitung von Textausgaben. Falls Sie sich für den Einsatz von Farben entscheiden, sollte Sie eine Option vorsehen, um diese Funktion bei Bedarf zu deaktivieren.

### Zeichenketten eingeben bzw. einlesen (read)

Interaktive Eingabeumleitungn führen Sie am einfachsten mit `read` durch. Damit die Benutzerinnen und Benutzer wissen, was sie eingeben sollen, führen Sie vorher mit `echo` eine Ausgabe durch. `-n` bewirkt, dass der Cursor hinter der Ausgabe stehen bleibt.

```
echo -n "Geben Sie bitte Ihren Namen an: "
read myvar
echo "Der Name lautet: $myvar"
```

`read` kann in Kombination mit einer Eingabeumleitung auch verwendet werden, um eine Datei zeilenweise zu lesen. Ein entsprechendes Beispiel folgt in Abschnitt 3.12, »Schleifen«.

`read -a myarray` liest eine Zeile und speichert deren Worte in Elementen der angegebenen Array-Variablen.

#### Hilfe zu »read«

`read` ist ein `bash`-internes Kommando. Details zu den diversen `read`-Optionen liefert deswegen `help read` und nicht wie sonst üblich `man read`.

### Substitutions- und Expansionsmechanismen

Die Bash kennt diverse »Substitutions- bzw. Expansionsmechanismen«, wobei diese Begriffe weitgehend synonym sind und in der Dokumentation uneinheitlich verwendet werden. Auf jeden Fall wird ein mit `$` eingeleiteter Ausdruck durch seinen Inhalt bzw. seine Auswertung ersetzt:

- `$myvar` bzw. `${myvar}`: Bei weitem am wichtigsten ist die **Variablensubstitution**, die Sie ja schon in etlichen Beispielen kennengelernt haben. Die alternative Schreibweise `${myvar}` ist zweckmäßig, wenn der Variablenname von weiterem Text abgegrenzt werden soll, z. B. `${myvar}txt`.
- `$(command)` bzw. `` `command` ``: Bei der ebenfalls schon behandelten **Kommandosubstitution** wird das Kommando ausgeführt und durch sein Ergebnis (also die Standardausgabe) ersetzt.
- `$((mathexpression))`: Bei der **arithmetischen Substitution** wird der in doppelten Klammern enthaltene mathematische Ausdruck ausgewertet und durch sein Ergebnis ersetzt.


© Rheinwerk Verlag, Bonn 2024

- `${___var___}`: Durch die sogenannte **Parametersubstitution** stellt die Bash einige Möglichkeiten zur Verfügung, die in der Variablen enthaltene Zeichenkette zu verarbeiten. Der Ausdruck liefert das Ergebnis zurück. Anstelle von ___ geben Sie diverse, leider nur schwer zu merkende Sonderzeichen an, auf die ich im nächsten Abschnitt anhand von Beispielen eingehe.

### Parametersubstitution

Der Bash fehlen die in anderen Programmiersprachen üblichen Funktionen zur Bearbeitung von Zeichenketten. Dafür kennt die Bash unter dem Begriff *Parametersubstitution* einige durchaus leistungsfähige Mechanismen, die Informationen aus Zeichenketten extrahieren, die in Variablen gespeichert sind. Beachten Sie, dass sämtliche Konstruktionen zwar ein Ergebnis zurückgeben, aber die angegebene Variable nie direkt verändern.

- `${var:n}` gibt die in `var` gespeicherte Zeichenkette ab dem *n*-ten Zeichen aus, wobei die Zählung bei 0 beginnt.

```
var="abcdefghij"
echo ${var:3}        # Ausgabe defghij
```

- `${var:offset:len}` überspringt `offset` Zeichen und gibt dann `len` Zeichen aus:

```
var="abcdefghij"
echo ${var:3}        # Ausgabe fgh
```

- `${var:-default}`: Wenn die Variable leer ist, liefert die Konstruktion die Defaulteinstellung als Ergebnis, andernfalls den Inhalt der Variablen. Die Variable wird nicht geändert.

```
var=
echo ${var:-abc}     # Ausgabe abc
var=123
echo ${var:-abc}     # Ausgabe 123
```

- `${var:=default}`: Wie oben, es wird aber gleichzeitig der Inhalt der Variablen geändert, wenn sie bisher leer war.
- `${var:+neu}`: Wenn die Variable leer ist, bleibt sie leer. Wenn die Variable dagegen bereits belegt ist, wird der bisherige Inhalt durch eine neue Einstellung ersetzt. Die Konstruktion liefert den neuen Inhalt der Variablen.
- `${var:?fehlermeldung}`: Wenn die Variable leer ist, werden der Variablenname und die Fehlermeldung ausgegeben, und das Shell-Programm wird anschließend beendet. Andernfalls liefert die Konstruktion den Inhalt der Variablen.
- `${#var}`: Liefert die Anzahl der in der Variablen gespeicherten Zeichen als Ergebnis oder `0`, falls die Variable leer ist. Die Variable wird nicht geändert.

Persönliches Exemplar für Thomas Bartholomäus

```
x='abcde'
echo ${#x}          # Ausgabe 5
```

- `${var#muster}`: Vergleicht den Anfang der Variablen mit dem angegebenen Muster. Wenn das Muster erkannt wird, liefert die Konstruktion den Inhalt der Variablen abzüglich des kürzestmöglichen Textes, der dem Suchmuster entspricht. Wird das Muster dagegen nicht gefunden, wird der ganze Inhalt der Variablen zurückgegeben. Im Suchmuster können die vom Globbing bekannten Zeichen verwendet werden, also `*`, `?` und `[abc]`.

```
dat=/home/pi/Bilder/img123.png
echo ${dat#*/}      # Ausgabe home/pi/Bilder/img123.png
echo ${dat#*.}      # Ausgabe png
```

- `${var##muster}`: Wie oben, allerdings wird jetzt die größtmögliche Zeichenkette, die dem Muster entspricht, eliminiert:

```
echo ${dat##*/}     # Ausgabe img123.png
echo ${dat##*.}     # Ausgabe png
```

- `${var%muster}`: Wie `${var#muster}`, allerdings erfolgt der Mustervergleich jetzt am Ende des Variableninhalts. Es wird die kürzestmögliche Zeichenkette vom Ende der Variablen eliminiert. Die Variable selbst bleibt unverändert.

```
echo ${dat%/*}      # Ausgabe /home/pi/Bilder
echo ${dat%.*}      # Ausgabe /home/pi/Bilder/img123
```

- `${var%%muster}`: Wie oben, allerdings wird nun die größtmögliche Zeichenkette eliminiert:

```
echo ${dat%%/*}     # keine Ausgabe (leere Zeichenkette)
echo ${dat%%.*}     # Ausgabe /home/pi/Bilder/img123
```

- `${var/find/replace}`: Ersetzt das erste Auftreten des Musters `find` durch `replace`:

```
x='abcdeab12ab'
echo ${x/ab/xy}     # Ausgabe xycdeaab12ab
```

- `${!var}` liefert den Inhalt derjenigen Variablen, deren Name in der Zeichenkette enthalten ist:

```
abc=123
efg='abc'
echo ${!efg}        # Ausgabe 123
```

Zum Abschluss möchte ich Ihnen noch eine typische Anwendung der Parametersubstitution zeigen: Sie wollen mit `convert` (siehe Kapitel 16, »Bildverarbeitung«) einige PNG-Dateien im gerade aktuellen Verzeichnis in das JPEG-Format umwandeln. Dazu müssen Sie beim Dateinamen in der Variablen `pngname` die Kennung `.png` durch `.jpg` ersetzen. Das gelingt am einfachsten so:


© Rheinwerk Verlag, Bonn 2024

```
#!/bin/bash
# Beispieldatei png2jpg.sh
shopt -s nullglob
for pngname in *.png; do
    # .png durch .jpg ersetzen
    jpgname=${pngname%.png}.jpg
    convert "$pngname" "$jpgname"
done
```

`shopt -s nullglob` vermeidet eine Fehlermeldung, wenn es im Verzeichnis gar keine `*.png`-Dateien gibt. Details zur Funktionsweise der `for`-Schleife folgen in Abschnitt 3.12, »Schleifen«. Beachten Sie, dass die Variablen `pngname` und `jpgname` beim Aufruf des `convert`-Kommandos in Anführungszeichen stehen. Nur so funktioniert das Script auch für Dateinamen mit Leerzeichen.

**Viel einfacher gelingt die Verarbeitung von Zeichenketten in Python!**

Wenn Sie sich einmal an die merkwürdigen Bash-Mechanismen zur Bearbeitung von Zeichenketten gewöhnt haben, können Sie damit erstaunlich viele Probleme lösen. Dennoch gilt: Für Scripts, in denen Sie viele Zeichenketten manipulieren müssen, verwenden Sie besser Python! Ich kenne keine Sprache, die für den Umgang mit Zeichenketten derart umfassende und doch einfach anzuwendende Funktionen bietet.

### Heredocs und Herestrings

*Here documents* (kurz Heredocs) sind in das Script eingebettete Textblöcke, die mit der durch << angegebenen Zeichensequenz enden (oft `EOF` für *End of file*). Das folgende Beispiel illustriert die Syntax sehr anschaulich. (Das `mail`-Kommando setzt voraus, dass auf dem Rechner ein Mail-Server läuft.)

```
name="Michael"
amount=1200
to="spamvictim@spamforever.com"

mail -s "Sicher investieren" $to << EOF
Hallo $name, anbei ein tolles Investitionsangebot
ohne jedes Risiko. Wenn du $amount US$ auf dieses Konto
überweist, bekommst du ...
EOF
```

Mehrzeilige Textausgaben können Sie ganz einfach mit `cat` ausführen:

```
cat << EOF
Das ist
```

Persönliches Exemplar für Thomas Bartholomäus

```
ein langer
Hilfetext.
EOF
```

Intern sind Heredocs eine besondere Form der Eingabeumleitung. `mail` bzw. `cat` erwartet den Mail-Text aus der Standardeingabe. Stattdessen werden die folgenden Textzeilen verwendet, wobei in dem Text die übliche Variablensubstitution (`$name`, `$amount`) stattfindet.

Für Heredocs gibt es einige Syntaxvarianten:

- In der Form `<<- EOF` werden führende Tabulatorzeichen aus dem Heredoc-Text eliminiert. Das ermöglicht es, den Text einzurücken. Leider funktioniert diese Textumwandlung nicht bei Leerzeichen.
- `<< "EOF"` verhindert die Variablensubstitution.
- `<<< $myvar` leitet den Inhalt der Variablen als Standardeingabe weiter. Dieses Verfahren ist unter dem Namen *Herestring* bekannt.

### Der Backslash

Gefühlt hat fast jedes Sonderzeichen in der Bash eine besondere Bedeutung. Wenn Sie das Zeichen als solches in einer Zeichenkette verwenden möchten, stellen Sie die Zeichenkette in einfache Apostrophe oder stellen einen Backslash als Quoting-Zeichen voran:

```
echo \$myvar
  $myvar
```

Auch Dateien mit Leerzeichen im Namen können Sie mit einem Backslash ansprechen. Noch einfacher ist aber auch hier die Verwendung von Apostrophen:

```
touch filename\ with\ blanks.txt
ls "filename with blanks.txt"
rm 'filename with blanks.txt'
```

Im Gegensatz zu vielen anderen Programmiersprachen macht die Bash aus `\n` keinen Zeilenumbruch. Vielmehr werden die Zeichen `\` und `n` separat ausgegeben bzw. gespeichert. Erst `echo -e` wertet diese Zeichensequenz wie erwartet aus.

```
echo -e "Line 1\nLine 2"
```

Schließlich können Sie mit dem Backslash lange Kommandos über mehrere Zeilen verteilen:

```
command --with --many --options and even \
  more parameters
```


© Rheinwerk Verlag, Bonn 2024

## 3.11 Verzweigungen

Wie in nahezu jeder Programmiersprache formulieren Sie auch in der Bash Verzweigungen mit if. Die Syntax sieht wie folgt aus:

```
if bedingung1; then
    kommando1a
    kommando1b
[ elif bedingung2; then
    kommandos2 ]
[ else
    kommandos3 ]
fi
```

Ein konkretes Beispiel sieht so aus:

```
if [ $# -ne 2 ]; then
    echo "Dem Kommando müssen zwei Parameter übergeben werden!"
    exit 2  # Fehlercode für falsche Parameter
else
    echo "Parameter 1: $1, Parameter 2: $2"
fi
```

Zur merkwürdigen Syntax für die eigentliche Bedingung komme ich gleich. Die Einrückungen im Listing sind optional, verbessern aber die Lesbarkeit.

Syntaktisch irritierend (und oft vergessen) sind die Strichpunkte nach den Bedingungen. Sie können die Strichpunkte einsparen, wenn Sie für then eine eigene Zeile opfern. Das macht den Code aber auch nicht schöner:

```
if bedingung
then
    kommando1
    kommando2
fi
```

### if-Kurzschreibweise mit && oder ||

Durch die korrekte Platzierung von Strichpunkten können Sie if-Konstruktionen in einer einzigen Zeile formulieren. Wenn im Rahmen einer if-Konstruktion nur ein einziges Kommando ausgeführt werden soll, ist die konditionelle Ausführung von Kommandos mit && oder || eleganter und platzsparender (siehe auch Abschnitt 3.6, »Kommandos ausführen«). Allerdings macht das den Code schwerer verständlich, insbesondere für Personen, die mit der Bash nicht vertraut sind.

```
if bedingung; then
    kommando
fi
```

Persönliches Exemplar für Thomas Bartholomäus

```
# gleichwertig
if bedingung; then kommando; fi

# auch gleichwertig: das Kommando wird ausgeführt,
# wenn die Bedingung erfüllt ist
bedingung && kommando

# inverse Logik: das Kommando wird ausgeführt,
# wenn die Bedingung nicht erfüllt ist
inversebedingung || kommando
```

### Bedingungen

Die `if`-Syntax ist also einfach. Schon erheblich schwieriger ist das Formulieren von Bedingungen. Die Bash sieht dafür diverse Varianten vor, die ich im folgenden Listing vorweg zusammenfasse und im Weiteren erläutere. Ein vorangestelltes Ausrufezeichen negiert eine Bedingung.

```
x=5
test "$x" -gt 3  && echo "true"   # -> true
test "$x" -eq 2  && echo "true"   # (keine Ausgabe)
[ "$x" -gt 3 ]   && echo "true"   # -> true
[[ "$x" -gt 3 ]] && echo "true"   # -> true
(( x > 3 ))      && echo "true"   # -> true
! (( x > 3 ))    && echo "true"   # (keine Ausgabe)
```

**Stellen Sie Variablen in Anführungszeichen!**

`test $x -gt 3` ist syntaktisch korrekt – aber nur, solange die Variable nicht leer ist. Wenn `$x` keinen Inhalt hat, ergibt sich `test -gt 3`. Diese Anweisung ist unsinnig und führt zu einem Syntaxfehler. Deswegen sollten Sie es sich angewöhnen, Variablen für Vergleiche immer in Anführungszeichen zu stellen!

In der ursprünglichsten Form wird das Kommando `test` verwendet. Es verarbeitet einen oder mehrere Parameter, wobei die Bedingung mit Optionen ausgedrückt wird. `-gt` steht für *greater than*, `-eq` für *equals*. Vielleicht fragen Sie sich, warum `test $x > 5` nicht funktioniert. Das hat damit zu tun, dass die Zeichen `>` und `<` für die Ein- und Ausgabeumleitung reserviert sind.

Anstelle von `test condition` existiert die Kurzschreibweise `[ condition ]` (siehe Tabelle 3.9). Wichtig ist, dass vor und hinter den eckigen Klammern Leerzeichen platziert werden müssen! Beachten Sie auch, dass die Vergleichsoperatoren vom Datentyp abhängen: `-eq` vergleicht Zahlen, `=` vergleicht Zeichenketten. Soweit im Testausdruck Variablen vorkommen, muss deren Namen `$` vorangestellt werden. `dat` bezieht sich auf Zeichenketten mit Dateinamen.


© Rheinwerk Verlag, Bonn 2024

| Testausdruck | Bedeutung |
|---|---|
| `[ var ]` | wahr, wenn die Variable nicht leer ist |
| `[ -n var ]` | wie oben |
| `[ zk1 = zk2 ]` | wahr, wenn die Zeichenketten übereinstimmen |
| `[ z1 -eq z2 ]` | wenn die Zahlen gleich groß sind (*equal*) |
| `[ z1 -ne z2 ]` | wenn die Zahlen ungleich sind (*not equal*) |
| `[ z1 -gt z2 ]` | wenn `z1` größer als `z2` ist (*greater than*) |
| `[ z1 -ge z2 ]` | wenn `z1` größer gleich `z2` ist (*greater equal*) |
| `[ z1 -lt z2 ]` | wenn `z1` kleiner als `z2` ist (*less than*) |
| `[ z1 -le z2 ]` | wenn `z1` kleiner gleich `z2` ist (*less equal*) |
| `[ -d dat ]` | wenn `dat` ein Verzeichnis ist (*directory*) |
| `[ -f dat ]` | wenn `dat` eine Datei ist (*file*) |
| `[ -r dat ]` | wenn die Datei gelesen werden darf (*read*) |
| `[ -w dat ]` | wenn die Datei verändert werden darf (*write*) |
| `[ dat1 -nt dat2 ]` | wenn Datei 1 neuer als Datei 2 ist (*newer than*) |
| `! [ ... ]` | negiert die Bedingung |

**Tabelle 3.9** Die wichtigsten Syntaxvarianten zur Formulierung von Bedingungen mit »[ ]«

`[ condition ]` steht in den meisten Shell-Varianten zur Verfügung. Bash und Zsh erlauben alternativ auch Bedingungen, die zwischen doppelten eckigen Klammern gestellt werden, also `[[ condition ]]` (siehe Tabelle 3.10). In `muster` können Sie die vom Globbing bekannten Zeichen verwenden, also `*`, `?` sowie `[a-z]`.

| Testausdruck | Bedeutung |
|---|---|
| `[[ zk = muster ]]` | wahr, wenn die Zeichenkette dem Muster entspricht |
| `[[ zk == muster ]]` | wie oben |
| `[[ zk =~ regex ]]` | wahr, wenn der reguläre Ausdruck zutrifft (siehe Kapitel 9) |
| `[[ bed1 && bed2 ]]` | wenn beide Bedingungen erfüllt sind (*and*) |
| `[[ bed1 \|\| bed2 ]]` | wenn mindestens eine Bedingung erfüllt ist (*or*) |

**Tabelle 3.10** Bash-spezifische Formulierung von Bedingungen mit »[[ ]]«

Im folgenden Beispiel muss `myvar` einen Code wie A17 oder C99 enthalten:

```
if [[ "$myvar" == [ABC][0-9][0-9] ]]; then
    echo "ok"
fi
```

**Kompatibilitätsüberlegungen**

Die Syntax `[[ ]]` wird nicht von allen Shells unterstützt (wohl aber von der Zsh). Sie erhöhen die Kompatibilität Ihrer Scripts, wenn Sie auf `[[ ]]` verzichten. Manche Bash-spezifischen Zusatzfunktionen lassen sich leicht ersetzen:

`[[ bed1 && bed2 ]]` entspricht `[ bed1 ] && [ bed2 ]`
`[[ bed1 || bed2 ]]` entspricht `[ bed1 ] || [ bed2 ]`

Die bereits in Abschnitt 3.9, »Variablen«, vorgestellte Schreibweise `(( ))` für mathematische Ausdrücke eignet sich ebenfalls zur Formulierung von Bedingungen. Dabei sind gut lesbaren Vergleichsoperatoren wie `<`, `>` oder `!=` erlaubt. Innerhalb der doppelten Klammern können Sie auf die Kennzeichnung von Variablen mit `$` verzichten.

```
x=5
(( x == 5 )) && echo "true"  # -> true
```

Passen Sie auf, dass Sie für Vergleiche nicht versehentlich `=` verwenden: `(( var = value ))` führt eine Zuweisung aus und liefert immer »wahr«!

```
x=5
(( x = 6 )) && echo "true"  # -> true
echo $x                     # -> 6 !!
```

Vorsicht ist auch geboten, wenn Variablen gar nicht deklariert sind. Die Bash geht dann vom Wert 0 aus und zeigt weder eine Warnung noch einen Fehler an:

```
# immer true, die Bash nimmt emptyvar=0 an
if (( emptyvar < 100 )); then ...
```

### Verzweigungen mit »case«

Verzweigungen können auch mit `case` formuliert werden. Dabei wird ein Ausdruck (oft einfach der Inhalt einer Variablen) mit verschiedenen Mustern verglichen. Die Muster können die Globbing-Zeichen `*` und `?` sowie Zeichenbereiche wie `[a-z]` enthalten. Wie so oft ist die Syntax der Bash gewöhnungsbedürftig:

```
case ausdruck in
    muster1)
        kommando1a
        kommando1b
```


© Rheinwerk Verlag, Bonn 2024

```
        ...
        ;;
    muster2)
        kommando2a
        kommando2b
        ...
        ;;
    *)          # Default-Block (optional)
        ...
esac
```

Die Einrückungen und der Start einer neuen Zeile nach `muster)` sind optional, verbessern aber die Lesbarkeit. Die doppelten Strichpunkte beenden die `case`-Konstruktion. Wenn Sie die Strichpunkte bewusst weglassen oder ganz einfach vergessen, setzt die Bash die `case`-Konstruktion vor. Wenn sie dabei weitere zutreffende Muster erkennt, führt sie die zugeordneten Kommandos ebenfalls aus. Die folgenden Zeilen zeigen die Auswertung einer `yes`/`no`-Eingabe, wobei die Eingabe in beliebiger Groß- und Kleinschreibung sowie in den Kurzformen `y` oder `j` (ja) akzeptiert wird:

```
echo -n "yes/no? "
read answer
case $answer in
    [yYjJ] | [yY][eE][sS] ) echo "yes";;
    [nN] | [nN][oO] )       echo "no";;
    *)                      echo "invalid input";;
esac
```

### Parameterauswertung mit »case«

In meinen eigenen Scripts gehe ich `case` zumeist aus dem Weg und ziehe besser lesbare `if`-Konstruktionen vor. Manchmal ist `case` aber tatsächlich eine Hilfe, wie das folgende Beispiel zeigt: Dabei geht es um eine einfache Auswertung von Parametern (*command line arguments*), die an ein Script übergeben werden.

Das Script `myoptions.sh` akzeptiert drei Optionen `-a`, `-b` und `-c`, die in beliebiger Reihenfolge angegeben werden können. Dabei erwarten `-b` und `-c` jeweils einen Parameter. Im Anschluss an die Optionen können beliebig viele weitere Parameter übergeben werden. Das folgende Listing zeigt zwei Aufrufe des Scripts:

```
$ ./myoptions.sh -a -b lorem -c ipsum dolores est

  Option a
  Option b with parameter lorem
  Option c with parameter ipsum
  More parameters: dolores est
```

Persönliches Exemplar für Thomas Bartholomäus

```
$ ./myoptions.sh -c lorem -b ipsum dolores

  Option c with parameter lorem
  Option b with parameter ipsum
  More parameters: dolores
```

Zur Auswertung der übergebenen Optionen und Parameter greift das Script auf das Bash-interne Kommando `getopts` zurück.

- `getopts "abc"` erwartet die Optionen `-a`, `-b` und `-c` in beliebiger Reihenfolge, auch kombiniert (also `-bc` anstelle von `-b -c`).
- `getopts ":abc"` funktioniert wie oben, liefert aber keine Fehlermeldungen bei ungültigen Parametern.
- `getopts "ab:c:"` mit einem Doppelpunkt hinter `b` und `c` erwartet für jeden dieser Optionen einen Parameter. Bei der Auswertung kann der übergebene Parameter aus der Variable `$OPTARGS` gelesen werden.
- Jeder Aufruf wertet *eine* Option aus. Die Auswertung muss daher in einer Schleife erfolgen. Für dieses Beispiel habe ich `while` verwendet. (Details zu Schleifen folgen im nächsten Abschnitt.)
- Während der Verarbeitung der Optionen verweist `$OPTIND` auf das jeweils nächste Element der Parameterliste.

  Unter der Voraussetzung, dass zuerst die Optionen und dann die Parameter übergeben werden, können alle verarbeiteten Optionen mit `shift` aus `$*` entfernt werden. (`shift <n>` entfernt die ersten *n* Elemente aus `$*`. Weil `$OPTIND` aber schon auf das nächste Element zeigt, dürfen nur `$OPTIND -1` Elemente aus der Parameterliste herausgeschoben werden.)

```
# Beispieldatei myoptions.sh
while getopts ":ab:c:" opt; do
    case $opt in
       a) echo "Option a";;
       b) echo "Option b with parameter $OPTARG";;
       c) echo "Option c with parameter $OPTARG";;
       ?) echo "Invalid option"
          echo "Usage: myoptions [-a] [-b data] [-c data] [...]"
          exit 2
          ;;
    esac
done
# verarbeitete Optionen aus $* entfernen
shift $(( $OPTIND - 1 ))
echo "More parameters: $*"
```


© Rheinwerk Verlag, Bonn 2024

### getopts versus getopt

`getopts` ist ein relativ simples Kommando, das nur mit Ein-Buchstaben-Optionen zurechtkommt, nicht aber mit langen Optionen wie `--search`. Eine leistungsfähigere Alternative ist das externe Kommando `getopt`, das unter Linux zumeist im Paket `util-linux` enthalten ist. Ein Beispiel für die Anwendung dieses Kommandos sowie weitere Tipps zur Auswertung von Script-Parametern finden Sie auf StackOverflow:

*https://stackoverflow.com/questions/192249*

## 3.12 Schleifen

Die Bash stellt mehrere Kommandos zur Bildung von Schleifen zur Auswahl. In diesem Abschnitt beginne ich mit `for`, bevor ich Ihnen zum Abschluss kurz `while` und `until` vorstelle. Die prinzipielle Syntax der `for`-Schleife geht aus dem folgenden Listing hervor. Beachten Sie die Platzierung des Strichpunkts vor und nicht nach `do`! Sie können diesen Strichpunkt vermeiden, wenn Sie mit `do` eine neue Zeile beginnen.

```
for myvar in mylist; do
  kommando1
  kommando2
done

# gleichwertig
for myvar in mylist; do kommando1; kommando2; done

# Beispiel
for item in a b c; do echo $item; done
  a
  b
  c
```

Die Schleifenvariable `myvar` durchläuft also alle Werte der angegebenen Liste. (Sie werden gleich sehen, dass »Liste« ein recht generischer Begriff ist: Sie können mit `for` Schleifen über Parameter, Dateinamen, Zeilen einer Textdatei usw. bilden.) Achten Sie darauf, den Namen der Schleifenvariable nach `for` ohne `$` anzugeben! Das ist insofern konsequent, als die Variable an dieser Stelle verändert und nicht ausgelesen wird.

Die folgenden Zeilen geben noch einige Anwendungsbeispiele, die aus Platzgründen jeweils einzeilig formuliert sind. Beachten Sie, dass Dateinamen immer Leerzeichen enthalten können! Sie sollten daher die Schleifenvariable immer in Anführungszeichen setzen, damit die betreffende Datei korrekt verarbeitet wird. Als Ausgangspunkt für eine Schleife kann auch ein Brace-Extension-Ausdruck dienen (siehe das dritte Bei-

Persönliches Exemplar für Thomas Bartholomäus

spiel). Beachten Sie den Mustercharakter der Beispiele. Solange Sie nur eine einzige Operation ausführen, können Sie in der Bash oft ganz auf Schleifen verzichten.

```
# gibt alle an das Script übergebenen Parameter
# (command line arguments) aus
for para in $*; do echo $para; done

# alle *.jpg-Dateien in das images-Verzeichnis kopieren
for filename in *.jpg; do cp "$filename" images; done

# erzeugt file-00.txt bis file-99.txt
for fn in file-{00..99}.txt; do touch $fn; done

# gleichwertige Kommandos zu den obigen drei Schleifen
echo "$*"
cp *.jpg images
touch file-{00..99}.txt

# Schleife über Array-Elemente
myarray=("item" "other item" "third item")
for item in "${myarray[@]}"; do echo $item; done
```

In der Bash werden Sie nur selten den Bedarf nach einer Schleife haben, die numerisch einen Wertbereich durchläuft. Möglich sind solche Schleifen aber durchaus, wie das folgende Listing zeigt. Beachten Sie insbesondere das letzte Beispiel, das der klassischen `for`-Schleife der Programmiersprache C entspricht.

```
for i in {1..10}; do echo $i; done
# Ausgabe 1, 2, ..., 10

for i in {01..12}; do echo $i; done
# Ausgabe 01, 02, ..., 12

for ((i=1; i<=10; i++)); do echo $i; done
# Ausgabe 1, 2, ..., 10
```

### Dateinamen mit Leerzeichen verarbeiten

Das oben angegebene Beispiel `for para in $*; do ...` verarbeitet alle an ein Script übergebene Parameter. Vorsicht ist gegeben, wenn es sich dabei um Dateinamen handelt und diese Leerzeichen enthalten können! Nehmen wir an, es gibt die Datei `account names.txt` und Sie führen `./myscript.sh *.txt` mit dem folgenden Code aus:

```
# verarbeitet Dateinamen mit Leerzeichen fehlerhaft
for filename in $*; do
  ls -l "$filename"
done
```


© Rheinwerk Verlag, Bonn 2024

Dabei tritt zweimal ein Fehler auf. Das Kommando `ls` findet weder die Datei `account` noch `names.txt`. Für das Script ist nicht erkennbar, dass `account names.txt` *ein* Dateiname ist. Der Grund besteht darin, dass die übergebenen Parameter in der Schleife standardmäßig bei jedem *Whitespace*-Zeichen getrennt werden, sowohl bei Leerzeichen als auch bei Zeilenumbruchzeichen, das die Bash zur Trennung von Dateinamen verwendet.

Damit die Schleife korrekt verarbeitet wird, müssen Sie die Anweisung `IFS=$'\n'` voranstellen. Das bedeutet, dass die Zerlegung in Wörter nur bei einem Zeilenumbruchzeichen durchgeführt wird.

```
# verarbeitet Dateinamen mit Leerzeichen korrekt
IFS=$'\n'
for filename in $*; do
  ls -l "$filename"
done
```

Die Spezialvariable `IFS` (Internal Field Separator) habe ich Ihnen in Abschnitt 3.9, »Variablen«, schon vorgestellt. Das vorangestellte Dollar-Zeichen stellt sicher, dass `\n` korrekt interpretiert wird.

### »while« und »until«

Die `while`-Schleife wird ausgeführt, solange die Bedingung erfüllt ist:

```
while bedingung; do
  kommandos
done
# Beispiel (Ausgabe: 1, 2, 3, 4, 5)
i=1
while [ $i -le 5 ]; do
    echo $i
    (( $i++ ))
done
```

`while`-Schleifen können ausgezeichnet mit einer Eingabeumleitung oder mit dem Pipe-Kommando kombiniert werden:

```
# Datei zeilenweise lesen
while read filename; do
    echo "$filename"
done < files.txt

# Dateien verarbeiten
ls *.jpg | while read filename; do
  echo "$filename"
done
```

Persönliches Exemplar für Thomas Bartholomäus

until-Schleifen funktionieren ähnlich wie while-Schleifen. Der Unterschied besteht darin, dass die Schleife fortgesetzt wird, solange die Bedingung *nicht* erfüllt ist:

```
# Ausgabe: 1, 2
i=1
until [ $i -eq 3 ]; do
  echo $i
  (( i++ ))
done
```

**»break« und »continue«**

Die Schlüsselwörter break und continue funktionieren in der Bash wie bei den meisten Programmiersprachen: Mit break brechen Sie die Ausführung einer Schleife vorzeitig ab. Bei verschachtelten Schleifen gibt break n an, wie viele Schleifenebenen beendet werden sollen.

continue überspringt die restlichen Kommandos im Schleifenblock, setzt die Schleife dann aber fort.

### Schleifen über Textdateien

Recht oft besteht der Wunsch, eine Textdatei zu verarbeiten. In der Bash sind hierfür zwei Verfahren üblich, die allerdings nicht ganz gleichwertig sind.

Die folgende while-Schleife liest mit jedem Schleifendurchgang eine Zeile aus access.log. Beachten Sie, wie diese Textdatei bei done durch das Umleitungszeichen < als Standardeingabe für die gesamte Schleifenkonstruktion verwendet wird!

```
# Textdatei Zeile für Zeile verarbeiten
while read line; do
    echo $line
done < access.log
```

Alternativ können Sie mit $(cat file) eine Liste erzeugen und diese dann mit for verarbeiten. Diese Vorgehensweise ist nur für kleine Textdateien empfehlenswert. In der Schleife wird die Liste bei jedem Leer-, Tabulator- oder Zeilentrennzeichen zerlegt. Der Text wird somit Wort für Wort verarbeitet.

```
# Textdatei Wort für Wort verarbeiten,
# Worttrennung bei ' ', \t und \n
for word in $(cat words.txt); do
    echo $word
done
```


© Rheinwerk Verlag, Bonn 2024

An welchen Stellen `for` die `cat`-Ausgabe in Worte zerlegt, wird wiederum durch die Umgebungsvariable `IFS` gesteuert (Internal Field Separator). Im folgenden Beispiel habe ich `:` als Trennzeichen eingestellt. Außerdem gilt der Zeilenumbruch *immer* als Trennzeichen. Nach der Schleife wird der ursprüngliche Zustand von `IFS` wiederhergestellt.

```
# Textdatei Wort für Wort verarbeiten, Worttrennung bei : und \n
OLDIFS=$IFS
IFS=':'
for word in $(cat words.txt); do
    echo $word
done
IFS=$OLDIFS  # IFS wiederherstellen
```

## 3.13 Funktionen

Das Schlüsselwort `function` definiert eine Funktion, die im Script wie ein Kommando aufgerufen werden kann. Der Code der Funktion muss in geschwungene Klammern gesetzt werden. Funktionen müssen *vor* ihrem ersten Aufruf deklariert werden und stehen daher oft am Beginn des Scripts.

An Funktionen können Parameter übergeben werden. Anders als bei vielen Programmiersprachen werden die Parameter nicht in Klammern gestellt. Innerhalb der Funktion können die Parameter den Variablen `$1`, `$2` entnommen werden. Eine Funktion verarbeitet Parameter auf die gleiche Art und Weise wie das Script die *Command Line Arguments*. Das folgende Mini-Script gibt *Hello World, Bash!* aus:

```
#!/bin/bash
function myfunc {
    echo "Hello World, $1!"
}
myfunc "Bash"
```

Das Schlüsselwort `function` ist optional. Wenn Sie auf `function` verzichten, müssen dem Funktionsnamen allerdings zwei runde Klammern folgen. Somit ist die folgende Funktion gleichwertig zum vorigen Beispiel:

```
myfunc() {
    echo "Hello World, $1!"
}
```

Bash-Funktionen helfen dabei, den Code übersichtlich zu strukturieren. Außerdem können Sie mehrfach benötigte Code-Blöcke auslagern und so Redundanz vermeiden.

Persönliches Exemplar für Thomas Bartholomäus

Mit `return` verlassen Sie eine Funktion vorzeitig. Es gibt keine Möglichkeit, Daten zurückzugeben. Sie können aber natürlich Ausgaben mit `echo` durchführen oder Variablen verändern.

### Lokale Variablen

Normalerweise werden alle Variablen im gesamten Script »geteilt«, sind also auch in Funktionen zugänglich und können dort verändert werden. Das Schlüsselwort `local` gibt die Möglichkeit, lokale Variablen zu definieren.

```
function myfunc {
    a=4
    local b=4
}

a=3; b=3
myfunc
echo "$a $b"  # Ausgabe 4 3
```

## 3.14 Umgang mit Fehlern

Die Bash hat einen ausgesprochen saloppen Umgang mit Fehlern: Wenn ein Kommando in einem Script einen Fehler auslöst, setzt die Bash das Script einfach mit der nächsten Anweisung fort! Diese merkwürdige Strategie hat damit zu tun, dass der Rückgabe-Code vieler Kommandos oft gar nicht auf echte Fehler hinweist, sondern lediglich darauf, dass eine Bedingung nicht erfüllt war oder bei der Suche nach einer Datei diese nicht gefunden wurde. Das muss nicht unbedingt ein »richtiger« Fehler sein.

Es gibt allerdings eine Ausnahme: Bei offensichtlichen Syntaxfehlern, wenn also Anführungszeichen oder Klammern fehlen oder Kontrollstrukturen unvollständig sind (kein `fi` zu `if`), wird das Script gar nicht gestartet.

### Fehler feststellen

Der Rückgabecode jedes Kommandos wird in `$?` gespeichert. 0 bedeutet: alles in Ordnung. Jede andere Zahl weist auf einen Fehler hin.

```
command_might_fail
errcode=$?
if [ $errcode -ne 0 ]; then
  echo "Fehlercode $errcode"
fi
```



© Rheinwerk Verlag, Bonn 2024

### Bei Fehlern abbrechen

Wenn Ihr Script beim ersten Fehler beendet werden soll, fügen Sie in den Hash-Bang die Option `-e` ein:

```
#!/bin/bash -e
```

Bei miteinander verknüpften Kommandos gilt das Gesamtergebnis. Wenn `command1` im folgenden Script scheitert – und sei es, dass es das Kommando gar nicht gibt –, wird `command2` ausgeführt. Nur wenn auch dieses Kommando zu einem Fehler führt, wird das Script abgebrochen.

```
#!/bin/bash -e
command1 || command2
```

Anstatt das Fehlerverhalten im Hash-Bang für das ganze Script einzustellen, können Sie den strikten Fehlertest mit `set -e` für einige Code-Zeilen aktivieren und später mit `set +e` wieder deaktivieren.

#### Nicht empfohlen

Die meisten Bash-Handbücher und -FAQs raten davon ab, die Bash mit der Option `-e` auszuführen oder die Funktion mit `set -e` zu aktivieren. Der Schaden ist größer als der Nutzen, das Verhalten des Scripts kann unvorhersehbar werden:

*https://mywiki.wooledge.org/BashFAQ/105*

### exit

`exit [n]` beendet das Script. Dabei wird der Rückgabewert des letzten Kommandos oder der Wert `n` zurückgegeben.

| exit-Code | Bedeutung |
|---|---|
| 0 | okay, kein Fehler |
| 1 | allgemeiner Fehler |
| 2 | Fehler in den übergebenen Parametern |
| 3–255 | anderer, kommando- oder scriptspezifischer Fehler |

**Tabelle 3.11** Fehlercodes von Bash- und Linux-Kommandos

### Reaktion auf Signale (trap)

Das Konzept der Prozessverwaltung unter Unix/Linux sieht vor, dass Prozessen Signale gesendet werden können (siehe Tabelle 3.12). Noch mehr Signale listet

Persönliches Exemplar für Thomas Bartholomäus

kill -l auf. Wenn Sie beispielsweise während der Ausführung eines Bash-Scripts Strg+C drücken, wird das Signal SIGINT versendet, und die Script-Ausführung wird abgebrochen.

| Signalname | Code | Bedeutung |
| --- | --- | --- |
| SIGHUP | 1 | Aufforderung, die Konfiguration neu zu lesen |
| SIGINT | 2 | Unterbrechung durch Strg+C |
| SIGKILL | 9 | kill-Signal, kann nicht abgefangen werden. |
| SIGTERM | 15 | Aufforderung, das Programm zu beenden |

**Tabelle 3.12** Die wichtigsten Signale

Mit trap können Sie gezielt auf solche Signale reagieren. Die Syntax von trap ist einfach:

```
trap 'commands' signals
```

commands gibt ein oder mehrere Kommandos an, die ausgeführt werden sollen, wenn eines der aufgezählten Signale empfangen wird. Die Signale können wahlweise durch ihre Namen oder durch ihre Codes angegeben werden. Wenn Sie komplexen Code in Reaktion auf ein Signal ausführen möchten, definieren Sie den Code in einer Funktion und geben beim trap-Kommando den Funktionsnamen an.

trap wird oft dazu verwendet, vor einem Programmabbruch, der durch ein Signal ausgelöst wird, Aufräumarbeiten durchzuführen. So können Sie beispielsweise eine temporäre Datei löschen. Mit dem simplen Kommando trap '' 2 erreichen Sie, dass Ihr Script Strg+C einfach ignoriert.

### Timeout

Bei Kommandos, die sich auf Netzwerk- oder Datenbankverbindungen verlassen, ist nicht immer eindeutig, ob wirklich ein Fehler vorliegt oder ob der Verbindungsaufbau zu langsam ist. Mit timeout <time> <command> können Sie ein Kommando mit einem vorgegebenen Timeout ausführen. Wenn das Kommando länger als die erwartete Zeit braucht, wird die Ausführung abgebrochen. Der Rückgabewert von timeout ist 124, wenn ein Timeout auftritt. Das Kommando kennt einige weitere spezielle Fehlercodes (125, 126, 127 und 137 – siehe man timeout). Wenn command dagegen innerhalb des Timeouts beendet wird, gibt timeout den Exit-Code von command zurück. Das folgende Kommando testet, ob es für den aktuellen Zweig ein Git-Remote-Repository gibt (siehe auch Kapitel 14, »Git«):

```
timeout 30s git ls-remote
```


© Rheinwerk Verlag, Bonn 2024

# 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-365-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.

## 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     5.1.22000.1335
  PSEdition     Desktop
  ...
```

Das obige Ergebnis ist im Winter 2023 unter Windows 11 entstanden. Ein wenig verblüffend ist der Umstand, dass die PowerShell-Version 5.1 zum Einsatz kommt, die bereits 2016 ausgeliefert wurde. Gibt es wirklich keine neuere Version?

Tatsächlich war, als ich dieses Kapitel verfasst habe, Version 7.3 aktuell. Allerdings hat sich mit dem Sprung von Version 5.*n* auf 7.*n* mehr als nur die Versionsnummer geändert:

- Für die PowerShell gilt seit Version 6 eine Open-Source-Lizenz. Das Programm steht auch für andere Betriebssysteme kostenlos zur Verfügung.
- Dementsprechend lautet der Programmname nicht mehr »Windows PowerShell«, sondern einfach nur »PowerShell«.

Allerdings sind die Versionen 5.*n* und 7.*n* nicht vollständig kompatibel zueinander. Auch wenn die Weiterentwicklung von Version 5.*n* längst eingestellt wurde, bleibt diese Version aus Kompatibilitätsgründen vorinstalliert.

Im Weiteren gehe ich aber davon aus, dass Sie mit der aktuellen PowerShell-Version arbeiten. Zwar funktionieren die meisten in diesem Buch vorgestellten Kommandos und Scripts auch mit Version 5.1, getestet habe ich meine Beispiele aber ausschließlich mit der Version 7.3.

Zur Installation gibt es mehrere Wege. Ich empfehle Ihnen, in einem Terminal oder in `cmd.exe` einfach das Kommando `winget install Microsoft.PowerShell` auszuführen. `winget` ist eine relativ neue Komponente von Windows 10 und 11 und vereinfacht die Paketverwaltung.

Alternativ können Sie ein MSI-Paket von *https://aka.ms/PSWindows* herunterladen und ausführen (siehe Abbildung 4.1). Achten Sie darauf, dass Sie die 64-Bit-Version installieren (Datei `PowerShell-7.n-win-x64.msi`), nicht die 32-Bit-Version (`PowerShell-7.n-win-x86.msi`)!



© Rheinwerk Verlag, Bonn 2024

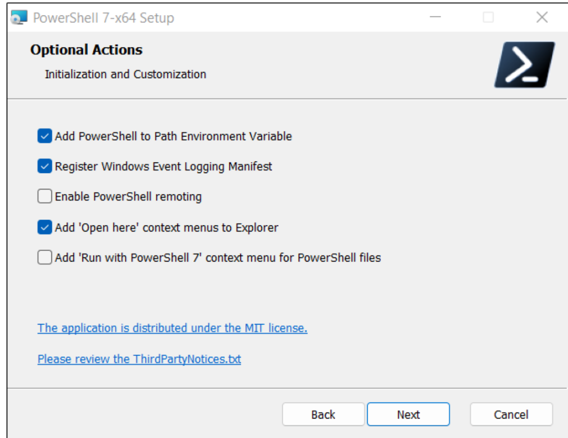



**Abbildung 4.1** Installieren Sie eine aktuelle PowerShell-Version!

Die Installation von Version 7.*n* erfolgt parallel zur vorhandenen Version 5.1. In der Folge können Sie wählen, mit welcher Version Sie arbeiten möchten. Nach einem Neustart des Terminals enthält dieses einen neuen Eintrag POWERSHELL (nicht WINDOWS POWERSHELL, dieser Eintrag bezeichnet die Version 5.1). Das Kommando `$PSVersionTable`, das einfach den Inhalt der Variable anzeigt, sollte nun dieses Ergebnis liefern:

```
> $PSVersionTable

  Name          Value
  ----          -----
  PSVersion     7.3.1
  PSEdition     Core
  ...
```

## Installation unter Linux

Seit die PowerShell einer Open-Source-Lizenz untersteht, stellt Microsoft auch Versionen für andere Betriebssysteme zur Verfügung. Die Installation ist unkompliziert. Ich konzentriere mich hier auf die Vorgehensweise unter Ubuntu. Die Zeichenkette bei `wget` muss ohne das Zeichen \ und ohne Leerzeichen in einer langen Zeile angegeben werden. `wget` lädt ein kleines Paket herunter, das eine Paketquelle einrichtet. Aus dieser Paketquelle bezieht Ubuntu dann das Paket `powershell`. Der Vorteil dieser auf den ersten Blick etwas umständlichen Vorgehensweise besteht darin, dass Sie in Zukunft im Zuge der üblichen Updates automatisch die jeweils neueste Version der PowerShell erhalten.

```
$ sudo apt update
```

Persönliches Exemplar für Thomas Bartholomäus

```
$ sudo apt install -y wget apt-transport-https \
                      software-properties-common

$ wget -q "https://packages.microsoft.com/config/ubuntu/\
         $(lsb_release -rs)/packages-microsoft-prod.deb"

$ sudo dpkg -i packages-microsoft-prod.deb
$ sudo apt update
$ sudo apt install -y powershell
```

Nach der Installation starten Sie die PowerShell in einem Terminalfenster mit dem Kommando `pwsh`. Anleitungen zur Installation der PowerShell 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 --cask powershell
```

Alternativ finden Sie auf der folgenden Webseite eine `*.pkg`-Datei mit der neuesten PowerShell-Version zum Download:

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

Ein Doppelklick startet den Installationsprozess, 30 Sekunden später ist alles erledigt. In der Folge starten Sie die PowerShell wie unter Linux in einem Terminalfenster 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


© Rheinwerk Verlag, Bonn 2024

Kommandos zählen. Das folgende Listing fasst die unter Windows, Linux und macOS ermittelten Ergebnisse im Frühjahr 2023 zusammen:

```
> Get-Command | Measure-Object
  Count: 1953   (Windows 11 Pro)
  Count:  273   (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:

```
> notepad $PROFILE      (Notepad.exe)
> code $PROFILE         (Visual Studio Code, falls installiert)
```

Persönliches Exemplar für Thomas Bartholomäus

## 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 eine relativ neue Komponente von Windows, die es erst seit 2019 gibt. In der Vergangenheit erledigten `cmd.exe` bzw. die PowerShell die Terminal-Funktionen einfach 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 bereits vorinstalliert. Sollte das nicht der Fall sein, führen Sie `winget install Microsoft.WindowsTerminal` aus oder installieren das Programm im Microsoft Store.

Denken Sie daran, hin und wieder ein Update auf die gerade aktuellste Version durchzuführen. Das Terminal wird aktuell intensiv weiterentwickelt und erhält fast monatlich neue Features!

```
> winget upgrade Microsoft.WindowsTerminal
```

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. Wenn Sie aber administrative Arbeiten erledigen möchten, benötigen Sie mehr Rechte.



© Rheinwerk Verlag, Bonn 2024

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 nun auch optisch mit entsprechenden Linux-Programmen 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 4.2). Das Programm unterscheidet zwischen globalen Grundeinstellungen, die für das gesamte Programm gelten, sowie zwischen Profilen. Jedes Profil ist einer Shell zugeordnet.

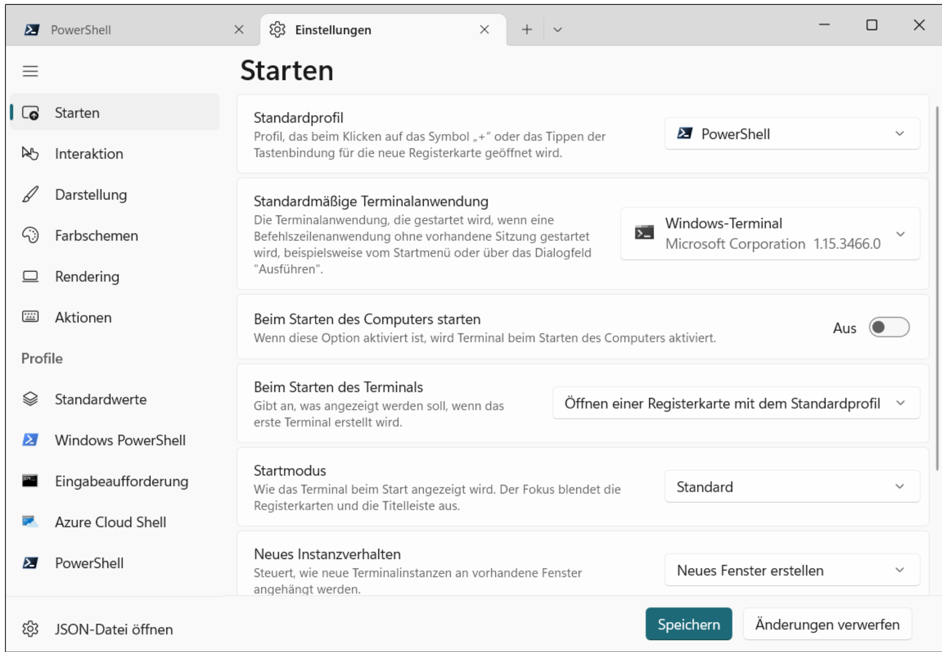



**Abbildung 4.2** Das Aussehen und die Funktion des Terminals können durch unzählige Optionen beeinflusst werden.

Im Dialogblatt Starten der Einstellungen können (und sollten) Sie die PowerShell 7 zur Default-Shell machen (Eintrag Standardprofil). Außerdem sollten Sie das Ter-

Persönliches Exemplar für Thomas Bartholomäus

minal zum Defaultprogramm zur Ausführung von Shells machen (Eintrag STANDARDMÄSSIGE TERMINALANWENDUNG), wenn dies nicht ohnedies schon der Fall ist.

**Tipp**

Wenn beim Öffnen der Einstellungen eine merkwürdige Datei in einem Editor erscheint, verwenden Sie eine veraltete Version der Terminals ohne einen richtigen Einstellungsdialog. Abhilfe: Führen Sie im Microsoft Store ein Update des Terminals durch.

### 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


© Rheinwerk Verlag, Bonn 2024

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

  Path
  ----
  C:\Users\kofler

> Set-Location Downloads

> Get-ChildItem *.exe

  Directory: C:\Users\kofler\Downloads

  Mode        LastWriteTime      Length Name
  ----        -------------      ------ ----
  -a---    18.07.2022 17:24     1414600 ChromeSetup.exe
  -a---    19.07.2022 11:49    49381480 Git-2.37.1-64-bit.exe
  -a---    11.01.2023 18:49    25218984 python-3.11.1-amd64.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

  Mittwoch, 3. Mai 2023 17:11:04
```

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
```

Persönliches Exemplar für Thomas Bartholomäus

```
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

  2023
```

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

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

  Mittwoch, 3. Mai 2023 19:13:03
```

## 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 müs-


© Rheinwerk Verlag, Bonn 2024

sen 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` mag eine generische Bezeichnung für ein Kommando sein, das den Inhalt eines Verzeichnisses ermittelt, aber der Tippaufwand im Vergleich zu `dir` oder `ls` ist 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.

Persönliches Exemplar für Thomas Bartholomäus

**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>` aus. Beispielsweise verrät `Get-Alias sls`, 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
```


© Rheinwerk Verlag, Bonn 2024

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 1900 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
  1028
> (Get-Command -CommandType CmdLet | Measure-Object).Count
  843
```

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 an 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 ...
```

Persönliches Exemplar für Thomas Bartholomäus

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 7.8, »Zusatzmodule installieren«.


© Rheinwerk Verlag, Bonn 2024

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.

Persönliches Exemplar für Thomas Bartholomäus

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 folgende 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:

```
> (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.


© Rheinwerk Verlag, Bonn 2024

```
> (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 16, »Bildverarbeitung«, näher vor.)

Persönliches Exemplar für Thomas Bartholomäus

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.

```
> 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 bereits im interaktiven Betrieb großartige Möglichkeiten. 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 13, »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!"
```


© Rheinwerk Verlag, Bonn 2024

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.

Persönliches Exemplar für Thomas Bartholomäus

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.

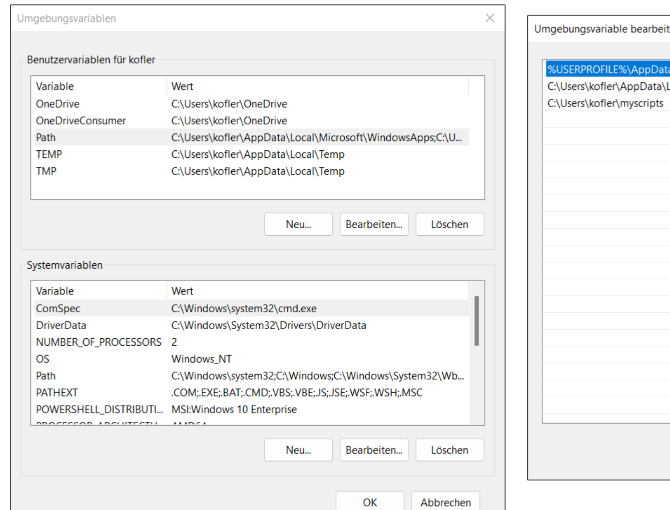

© Rheinwerk Verlag, Bonn 2024

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. `NewItem` 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. 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 4.3).



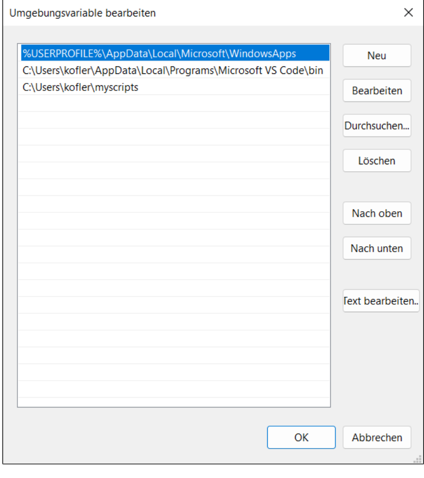



**Abbildung 4.3** Eigenes Script-Verzeichnis zu »Path« hinzufügen

Beachten Sie, dass die Änderung von `Path` erst nach einem Neustart des Terminals bzw. der PowerShell wirksam wird.

Persönliches Exemplar für Thomas Bartholomäus

### Script-Ausführung unter Linux und macOS

Ich gehe in diesem Kapitel normalerweise 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 3.5, »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 zu 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 Known-Folders-API. Allerdings hat auch dieses Verfahren einen Nachteil: Es funktioniert nur unter Windows, nicht unter Linux oder macOS.


© Rheinwerk Verlag, Bonn 2024

- 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

Persönliches Exemplar für Thomas Bartholomäus

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.
```


© Rheinwerk Verlag, Bonn 2024

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 `SetStrictMode` 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:

Persönliches Exemplar für Thomas Bartholomäus

```
> $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:

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

> $N * 7
  861
```

Beachten Sie, dass eine Eingabe wie »abc« nun einen Fehler auslöst.

| 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

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
```


© Rheinwerk Verlag, Bonn 2024

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".
```

### 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
> $Cnt
  31
```

Das Zeichen ^ kennt die PowerShell allerdings nicht. Zum Potenzieren müssen Sie die Funktion `Pow` verwenden:

```
> [Math]::Pow(2, 4)
  16
```

Beim obigen Ausdruck bezieht sich `[Math]` auf die gleichnamige Klasse, `::Pow` auf eine darin definierte Funktion. (Falls Sie mit den Konzepten der objektorientierten Programmierung vertraut sind: `Pow` ist eine statische Methode.) Neben `Pow` stellt `Math` unzählige weitere Funktionen zur Auswahl, z. B. `Sin`, `Cos` oder `Sqrt`.

Wesentlich umständlicher als das Rechnen ist das Vergleichen. Eine Bedingung wie `$Price > 140` kann die PowerShell nicht auswerten, weil die Zeichen < und > zur Ausgabeumleitung dienen (siehe Abschnitt 4.8). Stattdessen müssen Sie auf diverse Operatoren zurückgreifen, die wie Optionen aussehen (siehe Tabelle 4.3).

```
> $Price = 100.0
> $Price -gt 80
  True
> $Price -eq 80
  False
```

Persönliches Exemplar für Thomas Bartholomäus

| Operator | Bedeutung |
|---|---|
| -eq | gleich (*equal*) |
| -ne | ungleich (*not equal*) |
| -gt | größer (*greater than*) |
| -ge | größer oder gleich (*greater or equal*) |
| -lt | kleiner (*less than*) |
| -le | kleiner oder gleich (*less than or equal*) |

**Tabelle 4.3** Vergleichsoperatoren für Zahlen und Zeichenketten

Mit `-not (bedingung)` bzw. der Kurzschreibweise `!(bedingung)` können Sie eine Bedingung invertierten. Wenn Sie mehrere Bedingungen verknüpfen wollen, verwenden Sie `-and`, `-or` oder `-xor`:

```
> (100 -lt 110) -or (200 -lt 220)
  True
```

### Zeichenketten

Für viele Scripts sind Zeichenketten wichtiger als Zahlen. Wie die meisten anderen Shells unterscheidet auch die PowerShell zwischen zwei Formen, um Zeichenketten auszudrücken. Wenn Sie Zeichenketten in einfachen Apostrophen formulieren, wird der Inhalt unverändert übernommen. Verwenden Sie dagegen doppelte Apostrophe, dann werden innerhalb der Zeichenkette genannte Variablen durch ihren Inhalt ersetzt, also eine *Variablensubstitution* durchgeführt:

```
> $A = "abc"
> $B = "efg $A"
> $C = 'efg $A'
> $A, $B, $C

  abc
  efg abc
  efg $A
```

Wenn sich der Variablenname nicht klar abgrenzen lässt, stellen Sie diesen in geschwungene Klammern:

```
>  Write-Output "${A}${B}"
   abcefg abc
```


© Rheinwerk Verlag, Bonn 2024

Mehrzeilige Zeichenketten bzw. Zeichenketten, die selbst die Zeichen " und ' enthalten, setzen Sie am besten in Form von *Here-Strings* zusammen. Derartige Zeichenketten beginnen mit @" bzw. @' und enden mit "@ bzw. '@, je nachdem, ob Sie die Variablensubstitution verwenden möchten oder nicht. Die Start- und End-Codes müssen jeweils in einer eigenen Zeile stehen:

```
$json = @"
{
  "first": "item",
  "second": "item"
}
"@
```

Zum Vergleichen von Zeichenketten können Sie die bereits bekannten Operatoren verwenden (siehe Tabelle 4.3). Dabei müssen Sie beachten, dass die PowerShell nicht zwischen Groß- und Kleinschreibung unterscheidet:

```
> "abc" -eq "ABC"
  True
```

Zu allen Vergleichsoperatoren gibt es aber eine *case sensitive* Variante mit einem vorangestellten c:

```
> "abc" -ceq "ABC"
  False
```

Mit -like bzw. -notlike können Sie einen einfachen Mustervergleich durchführen. Dabei steht ? als Platzhalter für ein beliebiges Zeichen, * für mehrere Zeichen:

```
> "PowerShell" -like '*sh*'
  True
```

Wesentlich komplexere Vergleichsmuster auf der Basis regulärer Ausdrücke können mit -match und -notmatch gebildet werden (siehe Kapitel 9, »Reguläre Muster«).

Intern sind Zeichenketten Objekte (siehe auch die folgende Überschrift). Das bedeutet, dass Zeichenketten mit Methoden und Eigenschaften bearbeitet werden können. Die folgenden Zeilen zeigen ohne jeden Anspruch auf Vollständigkeit einige Beispiele. Eine vollständige Liste aller Methoden erhalten Sie mit "" | Get-Member.

```
> $UserName = "Michael Kofler"
> $UserName.Length
  14

> $UserName.ToUpper()
  MICHAEL KOFLER

> $UserName.StartWith("Michael")
  True
```

Persönliches Exemplar für Thomas Bartholomäus

```
> $UserName.Substring(8, 6)    # 6 Zeichen ab der Position 8
  Kofler

> $UserName.Split(" ")         # liefert Array
  Michael
  Kofler
```

Manche Bearbeitungsschritte können Sie statt mit Methoden auch mit speziellen Operatoren durchführen (siehe Tabelle 4.4):

```
> $UserName -split " "         # liefert Array
  Michael
  Kofler

> $UserName -replace "Michael", "Mike"
  Mike Kofler

> "Name: {0}  ID: {1}" -f "Kofler", 123
  Name: Kofler  ID: 123
```

| Operator | Bedeutung |
|---|---|
| `-eq`, `-ne` etc. | Vergleichsoperatoren (siehe Tabelle 4.3) |
| `-ceq`, `-cne` etc. | Vergleichsoperatoren mit Berücksichtigung der Groß- und Kleinschreibung |
| `-like`, `-notlike` | einfacher Mustervergleich |
| `-match`, `-notmatch` | Mustervergleich mit regulären Ausdrücken |
| `-f` | formatieren |
| `-replace` | suchen und ersetzen |
| `-split` | Zeichenkette zerlegen |

**Tabelle 4.4** Wichtige Operatoren für Zeichenketten

Beim Formatierungsoperator `-f` werden die Platzhalter `{0}`, `{1}` usw. durch die nachfolgenden Parameter ersetzt. Die PowerShell sieht dabei unzählige Formatierungsmöglichkeiten vor:

*https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings*

Ein Beispiel für die Anwendung von `-f` folgt in Abschnitt 4.10, »Verzweigungen«.

 © Rheinwerk Verlag, Bonn 2024

### Kommandosubstitution

Sie wissen schon, dass Sie dank der Variablensubstitution den Inhalt von Variablen in Zeichenketten einbauen bzw. ausgeben können:

```
> $account = "Maria"
> Write-Output "Account-Name: $account"

  Account-Name: Maria
```

Die Substitution funktioniert aber nicht, wenn Sie auf die Variable eine Methode anwenden möchten:

```
> Write-Output "Account-Name: $account.ToLower()"  # Fehler
  Account-Name: Maria.ToLower()
```

Oft enthalten PowerShell-Variablen Objekte (siehe dazu auch den folgenden Abschnitt). Auch der intuitive Zugriff auf deren Eigenschaften scheitert. Stattdessen gibt die PowerShell den Inhalt der Variable und danach den Eigenschaftsnamen aus:

```
> $process = Get-Process |  Select-Object -First 1

> Write-Output "Prozessname: $process.ProcessName"  # Fehler
  Prozessname:
  System.Diagnostics.Process (AggregatorHost).ProcessName
```

Abhilfe schafft die sogenannte *Kommandosubstitution* mit der Syntax `$( expr )`. Dabei wird der Ausdruck `expr` ausgewertet und sein Ergebnis in die Zeichenkette eingesetzt. Der Ausdruck muss sich nicht auf vorhandene Variablen beziehen, es kann vielmehr jedes beliebige CmdLet ausgeführt werden: Die Leerzeichen vor und nach den Klammern sind optional, können aber die Lesbarkeit verbessern. Das folgende Listing zeigt einige Beispiele für die Kommandosubstitution:

```
> Write-Output "Account-Name: $( $account.ToLower() )"
  Account-Name: maria

> Write-Output "Prozessname: $( $process.ProcessName )"
  Prozessname: AggregatorHost

> Write-Output "Datum und Uhrzeit: $(Get-Date)"
  Datum und Uhrzeit: 02/12/2023 10:08:34

> $now = Get-Date
> "Zeit in zwei Stunden: $($now.AddHours(2).ToShortTimeString())"
  Zeit in zwei Stunden: 12:08
```

### Objekte

Die meisten CmdLets liefern nicht wie Kommandos in herkömmlichen Shells Zeichenketten zurück, sondern vollwertige Objekte. Beispielsweise liefert `Get-Process` ein Array von `Process`-Objekten. Mit `Select-Object` können Sie sich ein solches Objekt herauspicken und dann mit `Get-Member` dessen Eigenschaften und Methoden ansehen:

```
> Get-Process | Select-Object -First 1 | Get-Member

  TypeName: System.Diagnostics.Process

  Name                  MemberType     Definition
  ----                  ----------     ----------
  Handles               AliasProperty  Handles = Handlecount
  Name                  AliasProperty  Name = ProcessName
  NPM                   AliasProperty  NPM = NonpagedSystemMem...
  ...
  Disposed              Event          System.EventHandler Dis...
  Exited                Event          System.EventHandler Exi...
  ...
  BeginOutputReadLine   Method         void BeginOutputReadLine()
  CancelErrorRead       Method         void CancelErrorRead()
```

Das folgende Kommando sortiert die Prozesse nach dem Startzeitpunkt und liefert dann die fünf zuletzt gestarteten Prozesse:

```
> Get-Process | Sort-Object -Property StartTime -Descending |
  Select-Object -First 5
```

Es ist nicht erlaubt, in der Form `cmdlet.property` direkt auf die Eigenschaften zuzugreifen, die ein CmdLet als Ergebnis zurückgibt. Stattdessen müssen Sie das CmdLet in runde Klammern stellen.

```
> Get-Date.Year          # error, term not recognized
> (Get-Date).Year        # OK
  2023
```

`(cmdlet).prop` ist eine Kurzschreibweise für `cmdlet | ForEach-Object { $_.prop }`. Die PowerShell verarbeitet also alle Objekte, die das CmdLet zurückgibt, und wertet dann die gewünschte Eigenschaft aus. `(Get-ChildItem *.txt).FullName` liefert dementsprechend die vollständigen Dateinamen der Textdateien im aktuellen Verzeichnis.


© Rheinwerk Verlag, Bonn 2024

### Eigene Objekte zusammenstellen

Sie können in Ihren Scripts nicht nur Objekte von vorgegebenen Klassen verarbeiten, Sie können auch selbst Ergebnisse in Form eigener, ad hoc zusammengestellter Objekte zurückgeben. Dazu gibt es mehrere Wege. Am einfachsten verwenden Sie ein `PSCustomObject` als Grundlage:

```
$MyObject = [PSCustomObject]@{
                UserName = "Michael"
                UserId   = 123
}
```

Oft erfüllt eine Hashtable (also ein einfacher Key-Value-Speicher) denselben Zweck. Der Hauptvorteil besteht darin, dass der Tippaufwand etwas kleiner ist:

```
$Data = @{
  UserName = "Michael"
  UserId = 123
}
```

Wenn Sie Zeilen sparen möchten, können Sie die Key-Value-Paare durch Strichpunkte voneinander trennen:

```
$Data = @{ UserName = "Michael"; UserId = 123 }
```

### Vordefinierte Variablen

Normalerweise müssen Sie sich um die Initialisierung Ihrer Variablen selbst kümmern. Es gibt aber auch eine Reihe vordefinierter Variablen, die in der Nomenklatur der PowerShell *automatic variables* heißen (siehe Tabelle 4.5). Diese Variablen enthalten Informationen über die PowerShell, den Zustand des aktuellen Scripts etc.

| Variable | Inhalt |
|---|---|
| `$?` | Status des zuletzt ausgeführten Kommandos (`$false`, wenn ein Fehler aufgetreten ist) |
| `$_` | das aktuelle Objekt, das gerade von einer Pipe verarbeitet wird (`$PSITEM`) |
| `$args` | Array mit den Parametern, die an ein Script oder eine Funktion übergeben wurden |
| `$Error` | Objekt, beschreibt den zuletzt aufgetretenen Fehler |
| `$Event` | Objekt, beschreibt das aktuell verarbeitete Ereignis |
| `$false` | boolescher Zustand »falsch« |

**Tabelle 4.5** Die wichtigsten vordefinierten Variablen der PowerShell

Persönliches Exemplar für Thomas Bartholomäus

| Variable | Inhalt |
|---|---|
| `$HOME` | Zeichenkette mit dem Heimatverzeichnis des aktuellen Benutzers |
| `$input` | über die Standardeingabe (Pipe) an eine Funktion übergebene Daten |
| `$IsLinux` | `$true`, wenn die PowerShell unter Linux läuft |
| `$IsMacOS` | `$true`, wenn die PowerShell unter macOS läuft |
| `$IsWindows` | `$true`, wenn die PowerShell unter Windows läuft |
| `$null` | null (nicht initialisierter Zustand) |
| `$PID` | aktuelle Prozessnummer |
| `$PROFILE` | Zeichenkette mit dem Pfad zur Profile-Datei zur Initialisierung/Konfiguration der PowerShell |
| `$PWD` | Objekt mit dem Pfad des aktuellen Verzeichnisses |
| `$true` | boolescher Zustand »wahr« |

**Tabelle 4.5** Die wichtigsten vordefinierten Variablen der PowerShell (Forts.)

Ich habe hier nur eine Referenz der wichtigsten Variablen zusammengestellt. Eine vollständige Beschreibung aller Variablen (es sind rund 50!) befindet sich wie immer in der Online-Dokumentation:

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

### Kein Globbing!

Wenn Sie ein Bash-Script in der Form `script.sh *.txt` ausführen, werden an das Script die Namen aller Textdateien im aktuellen Verzeichnis übergeben. Wenn Sie dagegen `*.txt` als Parameter an ein PowerShell-Script übergeben, dann wird exakt diese Zeichenkette übergeben.

Sie müssen sich im Script selbst darum kümmern, die entsprechenden Dateinamen herauszufinden. Wenn es sich nur um einen Parameter handelt, funktioniert `Get-Item $args`. Wenn an Ihr Script möglicherweise auch mehrere Parameter übergeben werden (z. B. `script.ps1 file1 file2` oder aber `script.ps1 *.jpg *.png`), dann müssen Sie *zwei* Schleifen bilden: Die erste verarbeitet die Parameter und führt `Get-Item` aus, die zweite die von `Get-Item` gelieferten Ergebnisse. Ein Code-Beispiel finden Sie in Abschnitt 16.2, »Fotos nach Aufnahmedatum sortieren«.



© Rheinwerk Verlag, Bonn 2024

### Umgebungsvariablen

Auch Umgebungsvariablen sind vordefinierte Variablen. Sie sind allerdings auf Betriebssystemebene definiert und stehen nicht nur der PowerShell, sondern allen Prozessen zur Verfügung. Viele Umgebungsvariablen enthalten die Orte wichtiger Verzeichnisse (siehe Tabelle 4.6).

| Variable | Inhalt |
|---|---|
| $Env:APPDATA | Verzeichnis für Programmeinstellungen |
| $Env:HOME | Heimatverzeichnis |
| $Env:HOMEDRIVE | Laufwerk des Heimatverzeichnisses |
| $Env:LOCALAPPDATA | Verzeichnis für Programmeinstellungen |
| $Env:Path | Verzeichnisliste für ausführbare Programme |
| $Env:PROCESSOR_ARCHITECTURE | CPU-Architektur (z. B. AMD64) |
| $Env:PSModulePath | Verzeichnisliste für PowerShell-Module |
| $Env:SHELL | Pfad zur aktiven Shell |
| $Env:TEMP, $Env:TMP | temporäres Verzeichnis |
| $Env:USER | aktueller Account-Name |
| $Env:windir | Pfad zum Windows-Verzeichnis (z. B. C:\Windows) |

**Tabelle 4.6** Die wichtigsten Umgebungsvariablen

Der Zugriff auf Umgebungsvariablen durch die PowerShell erfolgt in der etwas ungewöhnlichen Form `$Env:[<varname>]`, also z. B. `$Env:Path`. Sämtliche Umgebungsvariablen liefern Zeichenketten, keine Objekte. Ich habe hier einige besonders wichtige Umgebungsvariablen in der vordefinierten, recht inkonsequenten Groß- und Kleinschreibung angegeben.

## 4.7 Arrays und Hashtables

Ein Array ist eine Datenstruktur, um mehrere Objekte zu speichern. PowerShell macht intensiven Gebrauch von Arrays. Viele Kommandos geben Arrays zurück. Beispielsweise liefert `Get-ChildItem` normalerweise ein Array von `FileInfo`- und `DirectoryInfo`-Objekten (außer `Get-ChildItem` wird zur Verarbeitung von Registry-Einträgen verwendet).

Persönliches Exemplar für Thomas Bartholomäus

```
> $Files = Get-ChildItem
> $Files.Length
  4

> $Files.Count    # gleichwertig, Count ist ein Alias für Length
  4

> $Files.GetType()

  IsPublic   IsSerial   Name       BaseType
  --------   --------   ----       --------
  True       True       Object[]   System.Array
```

Der Pipe-Operator kommt ausgezeichnet mit Arrays zurecht und verarbeitet die Elemente nacheinander – dafür gab es in diesem Kapitel schon viele Beispiele. Mit `ForEach-Object` können Sie explizit eine Schleife bilden und Details zu jedem Array-Element ausgeben. Innerhalb der Schleife gibt `$_` (eine Kurzschreibweise für `$PSItem`) Zugriff auf das aktuelle Element. Mehr Details zu Schleifen folgen in Abschnitt 4.9, »Schleifen«.

```
> $Files | ForEach-Object {  "$($_.GetType()) $($_.FullName)" }

  System.IO.DirectoryInfo C:\Users\kofler\myscripts\mydir
  System.IO.FileInfo      C:\Users\kofler\myscripts\Hello.ps1
  System.IO.FileInfo      C:\Users\kofler\myscripts\MyTest.ps1
  System.IO.FileInfo      C:\Users\kofler\myscripts\Loop.ps
```

### Selbst Arrays erzeugen und verarbeiten

In eigenen Scripts können Sie unkompliziert Arrays erzeugen:

```
> $Data = 1, 17, 39, 45
> $Data = @(1, 17, 39, 45)   # gleichwertig
```

Die Größe von einmal erzeugten Arrays ist eigentlich unveränderlich. Dennoch können Sie mit dem Operator += unkompliziert weitere Elemente hinzufügen. Hinter den Kulissen wird dabei aber ein neues, etwas größeres Array erzeugt. Bei Arrays mit vielen Elementen ist das sehr ineffizient!

```
> $Data += 23, 27
> $Data
  1
  17
  39
  45
  23
  27
```


© Rheinwerk Verlag, Bonn 2024

Der Zugriff auf Array-Elemente erfolgt über einen Index in eckigen Klammern. Wie in den meisten Programmiersprachen reichen die erlaubten Index-Werte von 0 bis zur Elementanzahl minus eins:

```
> $Data[0]
  1
> $Data[5]
  27
```

Das Verhalten der PowerShell bei der Angabe eines ungültigen Index hängt vom Modus ab. Standardmäßig liefert die PowerShell einfach null zurück, löst aber keinen Fehler aus. Wenn Sie dagegen den Strict-Modus aktiviert haben (was in eigenen Scripts sehr zu empfehlen ist!), kommt es zu einem Fehler:

```
> $Data[6]
> Set-StrictMode -Version Latest
> $Data[6]
  OperationStopped: Index was outside the bounds of the array.
```

Es ist auch erlaubt, mehrere Index-Einträge anzugeben. Das Ergebnis ist dann ein neues Array mit den ausgewählten Einträgen:

```
> $Data[1, 2, 5]
  17
  39
  27
```

Die Schreibweise `start..end` liefert alle Elemente im angegebenen Bereich. Sollte `start` größer sein als `end`, werden die Elemente in umgekehrter Reihenfolge zurückgegeben. (Passen Sie auf, dass Sie nicht versehentlich drei Punkte angeben! Dann würde der dritte Punkt als Dezimalpunkt interpretiert, der nachfolgende Indexwert als Fließkommazahl.)

```
> $Data[3..5]
  45
  23
  27
> $Data[5..3]
  27
  23
  45
```

### Hashtables

Hashtables erlauben wie Arrays die Speicherung von Daten. Es gibt aber einen wesentlichen Unterschied: Der Zugriff auf die Elemente erfolgt nicht durch einen durchnummerierten Index, sondern über selbst gewählte Schlüssel. Aus diesem Grund werden

Persönliches Exemplar für Thomas Bartholomäus

Hashtables oft auch als Key-Value-Speicher bezeichnet. Sie können damit übersichtliche Datenstrukturen bilden. Der Umgang mit Hashtables ist ebenso einfach wie mit Arrays:

```
> $Data = @{ OS = "Windows"; Version = 11; ReleaseYear = 2021 }

> $Data.MinimumRamGB = 4     # Element hinzufügen

> $Data                      # alle Elementen lesen

  Name            Value
  ----            -----
  MinimumRamGB    4
  ReleaseYear     2021
  OS              Windows
  Version         11

> $Data.ReleaseYear          # ein Element lesen
  2021
```

Normalerweise kümmert sich eine Hashtable nicht um die Reihenfolge der Elemente. Sie dürfen sich also nicht darauf verlassen, dass eine Schleife die Elemente in der Reihenfolge liefert, in der Sie die Elemente eingefügt haben. Wenn die Reihenfolge für Sie wichtig ist, müssen Sie die Hashtable mit dem Schlüsselwort `[ordered]` erzeugen:

```
> $Data = [ordered]@{ Key1 = Value1; Key2 = Value2 ... }
```

## 4.8 Ausgabeumleitung

Standardmäßig wird das Ergebnis jedes Kommandos im Terminal angezeigt. Wenn Sie `Get-ChildItem` ausführen, enthält das Terminal danach eine Liste der gefundenen Dateien. Variablen oder Zeichenketten können Sie wahlweise mit dem schon bekannten CmdLet `Write-Output` oder einfach durch ihre Nennung ausgeben. Die beiden folgenden Anweisungen sind daher gleichwertig und führen jeweils zur Ausgabe von »Hello, World!«:

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

### Streams

Was auf den ersten Blick trivial einfach aussieht, ist hinter den Kulissen komplexer: Die PowerShell differenziert zwischen mehreren »Streams« (Kanälen), die für unterschiedliche Zwecke gedacht sind (siehe Tabelle 4.7).


© Rheinwerk Verlag, Bonn 2024

| Nummer | Beschreibung | Cmdlet |
|---|---|---|
| 1 | Success Stream | `Write-Output` sowie direkte Ausgaben |
| 2 | Error Stream | `Write-Error` |
| 3 | Warning Stream | `Write-Warning` |
| 4 | Verbose Stream | `Write-Verbose` |
| 5 | Debug Stream | `Write-Debug` |
| 6 | Information Stream | `Write-Information` |
| * | alle Streams | – |

**Tabelle 4.7** Streams

Während Sie für »normale« Ausgaben also `Write-Output` verwenden, sollten Sie Fehlermeldungen mit `Write-Error`, Warnungen mit `Write-Warning` usw. verarbeiten. Beachten Sie, dass Sie diese Kommandos auch mit dem Pipe-Operator kombinieren können. Das folgende Kommando schreibt die aktuelle Prozessliste in den Debug Stream:

```
> Get-Process | Write-Debug
```

Standardmäßig werden nur die Streams 1 bis 3 im Terminal angezeigt. Die Streams 4 bis 6 bleiben unsichtbar, um die Benutzer nicht mit Debugging-Informationen zu belästigen. Welcher Stream angezeigt bzw. nicht angezeigt wird, können Sie für die Streams 3 bis 6 durch `Preference`-Variablen steuern:

```
> $WarningPreference, $VerbosePreference, $DebugPreference,
    $InformationPreference

  Continue
  SilentlyContinue
  SilentlyContinue
  SilentlyContinue
```

Das obige Listing zeigt die Defaulteinstellungen. Andere zulässige Einstellungen sind die Zeichenketten `Inquire` und `Stop` (siehe auch Tabelle 4.9 am Ende dieses Kapitels). Damit führt jede Ausgabe im entsprechenden Stream zu einer Script-Unterbrechung. Bei `Inquire` kann das Script nach einer Rückfrage fortgesetzt werden, bei `Stop` nicht.

### Ausgabeumleitung

Der Clou der Streams besteht darin, dass sie in Dateien umgeleitet werden können. Der klassische Umleiteoperator für den Success Stream lautet `> file`. Damit werden alle Ausgaben des Success Stream nicht mehr angezeigt, sondern in einer Textdatei

Persönliches Exemplar für Thomas Bartholomäus

gespeichert. Falls die Datei bereits existiert, wird sie überschrieben. Innerhalb der PowerShell können Sie den Inhalt der Datei mit `Get-Content` ansehen:

```
> Get-ChildItem > files.txt
> Get-Content files.txt
  ...
```

Statt eine vorhandene Datei zu überschreiben, können Sie die Textausgaben auch am Ende hinzufügen. Dazu verwenden Sie statt `>` den Operator `>>`.

Die Operatoren `>` und `>>` betreffen nur den Success Stream. Alle weiteren Streams können Sie bei Bedarf mit `2>` bis `6>` extra umleiten. (Bei der Umleitung der Streams 3 bis 6 bleiben die `Preference`-Variablen gültig. Bei der Einstellung `SilentlyContinue` wird in der angegebenen Datei nichts gespeichert, weil die Ausgabe unterdrückt ist.)

Anstatt die Streams in getrennte Dateien umzuleiten, können Sie die Streams 2 bis 6 mit dem Success Stream zusammenführen und diesen dann in eine Datei umleiten. Beim folgenden Kommando wird zuerst der Error Stream dem Success Stream hinzugefügt. Die Ausgaben beider Streams werden dann in der Datei `out.txt` gespeichert.

```
> Do-Something 2>&1 > out.txt
```

Mit `*>` können Sie schließlich sämtliche Streams in eine gemeinsame Datei umleiten (siehe Tabelle 4.8).

| Operator | Funktion |
|---|---|
| `>` | Success Stream in eine Datei umleiten |
| `>>` | Success Stream einer Datei hinzufügen |
| `2>` bis `6>` | Stream 2 bis 6 in eine Datei umleiten |
| `2>>` bis `6>>` | Stream 2 bis 6 einer Datei hinzufügen |
| `2>&1` bis `6>&1` | Stream 2 bis 6 mit dem Success Stream zusammenführen |
| `*>` | alle Streams in eine Datei umleiten |
| `2>&1 >>` | Stream 1 und 2 zusammenführen und dann einer Datei hinzufügen |

**Tabelle 4.8** Operatoren zur Ein- und Ausgabeumleitung

**»Write-Host« versus »Write-Output«**

Das Kommandos `Write-Host` führt Ausgaben direkt im Terminal (auf dem »Host«) aus. Auf den ersten Blick sieht das Ergebnis wie bei `Write-Output` aus.



© Rheinwerk Verlag, Bonn 2024

Der entscheidende Unterschied besteht darin, dass für diese Ausgaben keine Umleitung möglich ist. Mit `Write-Host` verlieren Ihre Scripts an Flexibilität. Die Anwenderinnen und Anwender Ihrer Scripts können deren Ausgabe nicht einfach in eine Datei umleiten. Verwenden Sie `Write-Output`!

### Ausgabe duplizieren (»Tee-Object«)

Manchmal wollen Sie die Ausgaben eines CmdLets oder Scripts am Bildschirm sehen *und* in einer Datei speichern. Dabei hilft `Tee-Object`: Dieses Kommando wird mit dem Pipe-Operator verwendet. Es übernimmt den Success Stream, zeigt ihn an und leitet ihn gleichzeitig in die angegebene Datei um. Wenn Sie das folgende Kommando ausführen, sehen Sie die Dateien des aktuellen Verzeichnisses im Terminal; zugleich wird die Dateiliste in `files.txt` gespeichert.

```
> Get-ChildItem | Tee-Object files.txt
```

`Tee-Object` berücksichtigt nur den Success Stream. Sie können aber mit `n>&1` zuerst einen anderen Stream zum Success Stream hinzufügen:

```
> Get-ChildItem 2&>1 | Tee-Object files-and-errors.txt
```

### Die Ausgabe (nicht) formatieren

Standardmäßig formatiert die PowerShell alle Ausgaben. Beispielsweise liefert `Get-ChildItem` ein Array aus `FileInfo`- und `DirectoryInfo`-Objekte als Ergebnis. Die PowerShell zeigt aber nicht diese Objekte mit all ihren Eigenschaften an, sondern bildet daraus eine für Menschen einigermaßen gut lesbare Textdarstellung. Alternative Darstellungen mit mehr oder weniger Details erhalten Sie, wenn Sie die Ausgaben an diverse `Format`-Kommandos weiterleiten:

```
> Get-ChildItem | Format-List
> Get-ChildItem | Format-Table
> Get-ChildItem | Format-Custom Name,Length
```

Wenn Sie das Ergebnis eines PowerShell-Kommandos in eine Textdatei umleiten, wäre es oft zweckmäßig, jegliche Formatierung zu eliminieren. Beim Beispielkommando `Get-ChildItems` sind Sie vermutlich nur an den Dateinamen interessiert, nicht an einer Tabelle mit mehreren Spalten, Farbcodes zur optisch schöneren Darstellung der Spaltenbeschriftung usw. Dieser nahe liegende Wunsch lässt sich in der PowerShell leider nur umständlich realisieren.

Das folgende Beispiel zeigt eine mögliche Vorgehensweise: Das Ergebnis von `Get-ChildItem` wird an `ForEach-Object` weitergeleitet. `$_.Name` extrahiert aus jedem Objekt den Dateinamen. (Details zu `ForEach-Object` und zur Schreibweise `$_` folgen in

Persönliches Exemplar für Thomas Bartholomäus

Abschnitt 4.9, »Schleifen«.) `Out-String` gibt die resultierenden Namen ohne weitere Formatierung aus. > leitet schließlich sämtliche Ausgaben in die Datei `files.txt` um. Diese Datei enthält dann in jeder Zeile einen Datei- oder Verzeichnisnamen.

```
> Get-ChildItem |
  ForEach-Object {$_.Name} |
  Out-String > files.txt
```

### Keine Eingabeumleitung

Bei vielen Shells besteht mit < auch die Möglichkeit der Eingabeumleitung. Unter Linux ist daher `command1 | command2` gleichwertig mit der folgenden Kommandosequenz (wenn man einmal davon absieht, dass der Umweg über eine Datei Speicherplatz und Zeit kostet):

```
$ command1 > file.txt
$ command2 < file.txt
```

Die PowerShell unterstützt < leider nicht, weil PowerShell-Kommandos zumeist Objekte erwarten. Wenn Sie nun aber zuerst das Ergebnis eines PowerShell-Kommandos mit `command1 > file.txt` in eine Textdatei schreiben, gehen die Objektinformationen verloren. Die Textdatei enthält lediglich formatierten Text und kann in der Folge nicht als Basis für die Verarbeitung der Daten in einem weiteren Kommando dienen.

Problematisch wird die fehlende Möglichkeit zur Eingabeumleitung, wenn Sie mit Kommandos aus der Linux-Welt arbeiten, die genau das erfordern. Beispielsweise können Sie mit `mysqldump` ein Backup einer Datenbank machen und dieses später mit `mysql` wieder einspielen. Unter Linux und macOS funktioniert das so:

```
$ mysqldump dbname > backup.sql    # Backup erstellen
$ mysql dbname  < backup.sql       # Datenbank wiederherstellen
```

In der PowerShell funktioniert nur das erste, nicht aber das zweite Kommando. Sie können das Problem aber umgehen. Dazu führen Sie `Get-Content` aus, um den Inhalt der Datei `backup.sql` auszugeben. Diese Ausgabe leiten Sie mit | an `mysql` weiter, also:

```
> Get-Content backup.sql | mysql dbname
```

## 4.9 Schleifen

Häufig wollen Sie in Scripts mehrere Elemente nacheinander verarbeiten – jede Zeile einer Textdatei, jede Datei eines Verzeichnisses usw. Dabei helfen Schleifen, die in der PowerShell in mehreren Syntaxformen zur Auswahl stehen. In der Praxis am wichtigsten ist `foreach($item in $data)`. Damit werden alle Elemente aus `$data` (also einer Variablen mit einem Array oder einer anderen Aufzählung) der Reihe nach in die

 © Rheinwerk Verlag, Bonn 2024

Schleifenvariable `$item` eingesetzt und verarbeitet. Im folgenden Beispiel wird die Zeichenkette in `$Text` mit `Split` in Wörter zerlegt. Diese werden dann in einer Schleife am Bildschirm ausgegeben.

```
> $Text = "I love PowerShell!"

> foreach ($Word in $Text.Split(" ")) {
      $Word
  }

  I
  love
  PowerShell!
```

### ForEach-Object

Mit `ForEach-Object` stellt die PowerShell eine alternative Vorgehensweise zur Wahl. Der wesentliche Unterschied besteht darin, dass `ForEach-Object` die zu verarbeitenden Objekte als Eingabe erwartet. Damit eignet sich das Kommando besonders gut für Pipes. Innerhalb der Schleifenkonstruktion können Sie mit `$_` auf das gerade aktuelle Objekt zugreifen. Die folgende Schleife ist gleichwertig zum vorigen Beispiel:

```
$Text.Split(" ") | ForEach-Object { $_ }
```

Wenn Sie die Länge aller Wörter ausgeben möchten, gehen Sie so vor:

```
> $Text.Split(" ") | % { Write-Host $_.Length }
  1
  4
  11
```

Für diesen relativ häufigen Fall, also die Anwendung einer Eigenschaft oder einer Methode auf die Schleifenvariable, gibt es eine Kurzschreibweise:

```
> $Text.Split(" ") | ForEach-Object Length
  1
  4
  11
```

Und weil das Schlüsselwort `ForEach-Object` so häufig benötigt wird, kann es mit `%` noch weiter verkürzt werden:

```
> $Text.Split(" ") | % ToUpper()
  I
  LOVE
  POWERSHELL!
```

Persönliches Exemplar für Thomas Bartholomäus

Wenn `ForEach-Object` mit der Option `-Parallel` kombiniert wird, führt es die Anweisungen in der Schleife (also den Code zwischen den geschwungenen Klammern) in parallelen Threads aus. Auf leistungsfähigen Rechnern mit vielen CPU-Cores kann das viel Zeit sparen. Das ist beim folgenden Beispiel der Fall: `Get-ChildItem` ermittelt alle `*.bmp`-Dateien im aktuellen Verzeichnis. `Compress-Archive` komprimiert jedes Bild in einer Datei mit dem Namen `<filename.bmp>.zip`.

```
> Get-ChildItem *.bmp | ForEach-Object -Parallel {
      Compress-Archive -DestinationPath "$_.zip" $_
  }
```

### for, while, do-while und do-until

Die PowerShell kennt auch die aus anderen Programmiersprachen bekannten Schleifenkonstruktionen mit `for`, `while`, `do-while` und `do-until`. Diese Art der Schleifenbildung ist in Shell-Scripts eher unüblich, weswegen ich hier nur je ein kurzes Beispiel ohne weitere Erläuterungen angebe. Werfen Sie gegebenenfalls einen Blick in die Dokumentation.

```
> for ($i=0; $i -lt 5; $i++) { $i }

  # Ausgabe: 0, 1, ..., 4

> $i=0
> while ($i -lt 5) {
      Write-Host $i
      $i++
  }
  # Ausgabe wieder: 0, 1, ..., 4

> $i=0
> do {
      Write-Host $i
      $i++
  } while ($i -lt 5)
  # Ausgabe nochmals: 0, 1, ..., 4

> $i=0
> do {
      Write-Host $i
      $i++
  } until ($i -eq 5)
  # Ausgabe neuerlich: 0, 1, ..., 4
```


© Rheinwerk Verlag, Bonn 2024

## 4.10 Verzweigungen

Oft wollen Sie Code nur dann ausführen, wenn eine bestimmte Bedingung erfüllt ist. Je nachdem, welchen Zustand eine Variable hat, sollen unterschiedliche Zweige Ihres Codes ausgeführt werden. Am einfachsten gelingt dies mit einer `if-elseif-else`-Anweisung. Dabei sind beliebig viele `elseif`-Blöcke erlaubt. Sowohl `elseif` als auch `else` sind optional.

```
if (condition1) {
    statements
} elseif (condition2) {
    statements
} else {
    statements
}
```

### Leerzeichen

Die Schreibweisen `if(condition)` und `if (condition)` sind beide syntaktisch zulässig. Üblicher und besser lesbar ist die Verwendung eines Leerzeichens vor den runden Klammern. Das gilt auch für `for`, `foreach`, `while`, `until` usw.

Beachten Sie aber, dass es bei Funktionen und Methoden nicht erlaubt ist, ein Leerzeichen zwischen den Namen und der öffnenden Klammer anzugeben!

Das folgende Beispiel-Script erzeugt zuerst mittels `Get-CimInstance` ein Array von `Win32_LogicalDisk`-Objekten, die die Dateisysteme des aktuellen Computers beschreiben. CIM steht dabei für *Common Information Module*. Dieser Standard hilft bei der Verwaltung von IT-Ressourcen und wird von Microsoft durch das WMI-API (*Windows Management Infrastructure*) unterstützt. In unserem Beispiel interessieren wir uns nur für Datenträger des Typs 3 (*Local Disk*). Für jedes Dateisystem auf einer Festplatte oder SSD berechnet das Script den freien Speicher in Prozent und in GByte.

```
# Beispieldatei Get-DiskFree.ps1
$Drives = Get-CimInstance -ClassName Win32_LogicalDisk
foreach ($Drive in $Drives) {
    if ($Drive.DriveType -eq 3) {
        $PercentageFree = 1.0 * $Drive.FreeSpace / $Drive.Size
        $GBFree = $Drive.FreeSpace / 1000000000
        "{0:S} {1:P1}  / {2,5:F0} GB free" -f `
            $Drive.DeviceId, $PercentageFree, $GBFree
    }
}
```

Persönliches Exemplar für Thomas Bartholomäus

Zur Formatierung der Ausgabe verwendet das Script den Format-Operator `-f`. Die Grundidee sieht so aus:

```
"format code" -f value1, value2, value3, ...
```

Der Zeichenkette mit den Formatierungsanweisungen folgen also der Operator `-f` und dann ein Array mit Werten, die verarbeitet werden sollen. Innerhalb der Formatierungszeichenkette wird `{0}` durch den ersten Wert ersetzt, `{1}` durch den zweiten Wert usw. Der Nummer können weitere Formatierungscodes folgen. In diesem Beispiel habe ich die folgenden Codes verwendet:

- `{0:S}`: ersten Wert des nachfolgenden Arrays unverändert als Zeichenkette ausgeben
- `{1:P1}`: zweiten Wert als Prozentzahl mit einer Nachkommastelle formatieren
- `{2,5:F0}`: dritten Wert als Fließkommazahl ohne Nachkommastellen (`F0`) rechtsbündig mit fünf Stellen ausgeben

Eine gute Dokumentation zu den zulässigen Formatcodes, die die PowerShell wie so vieles andere vom .NET-Framework übernimmt, finden Sie hier:

*https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings*

Das Script liefert dann z. B. die folgende Ausgabe:

```
> .\Get-DiskFree.ps1

  C: 84,2 %  /   180 GB free
  F: 98,3 %  /    15 GB free
```

Das Script sollte die Anwendung von `if` illustrieren. Ich will Ihnen aber nicht verschweigen, dass der Code auch ganz anders formuliert werden kann. In den folgenden Zeilen filtert `Where-Object` die gewünschten Dateisysteme aus dem Array aus. `ForEach` verarbeitet die verbleibenden Resultate, wobei der Zugriff auf das aktuelle Element nun eben durch `$_` statt durch die Schleifenvariable `$Drive` erfolgt. Falls Sie – egal in welcher Sprache – funktional programmieren gelernt haben, wird Ihnen diese Lösung wahrscheinlich eleganter erscheinen:

```
$Drives = Get-CimInstance -ClassName Win32_LogicalDisk
$Drives | Where-Object -Property DriveType -eq 3 | ForEach {
    $PercentageFree = 1.0 * $_.FreeSpace / $_.Size
    $GBFree = $_.FreeSpace / 1000000000
    "{0:S} {1:P1}  /  {2,5:F0} GB free" -f `
         $_.DeviceId, $PercentageFree, $GBFree
}
```


© Rheinwerk Verlag, Bonn 2024

### Fallunterscheidungen mit »switch«

`Switch` kann mehrere `if`-Anweisungen ersetzen, wenn für einen Ausdruck unterschiedliche Fälle verarbeitet werden sollen. Das ist die Syntax im einfachsten Fall:

```
$option = "b"
switch ($option) {
    "A"     { Write-Output "Option A" }
    "B"     { Write-Output "Option B" }
    "C"     { Write-Output "Option C" }
    default { Write-Output "Ungültig" }
}
```

Denken Sie daran, dass Zeichenketten wie üblich ohne Berücksichtigung der Groß- und Kleinschreibung verglichen werden.

## 4.11 Funktionen und Parameter

Funktionen bieten die Möglichkeit, mehrere PowerShell-Kommandos zu bündeln. Das erleichtert die Organisation von umfangreichen Scripts. In der einfachsten Form sieht eine Funktion so aus:

```
function Do-Something {
    command1
    command2
    ...
}
```

Innerhalb eines Scripts können Funktionen erst *nach* ihrer Definition verwendet werden. Das bedeutet, dass eine Script-Datei zuerst alle Funktionen enthält und erst zum Schluss den Code, der letztlich ausgeführt werden soll.

**Namensregeln**

Die PowerShell-Dokumentation empfiehlt, eigene Funktionen wie CmdLets dem *Verb-Noun*-Muster entsprechend zu benennen. Der dafür erforderliche Bindestrich kann wie nahezu jedes andere Zeichen ohne besondere Kennzeichnung im Namen verwendet werden.

### Funktionsergebnisse

Sehr ungewöhnlich im Vergleich zu »richtigen« Programmiersprachen ist bei der PowerShell die Zusammensetzung des Rückgabewerts einer Funktion: Das Ergebnis ergibt sich aus allen in der Funktion entstandenen Ausgaben! Sämtliche Teilergeb-

Persönliches Exemplar für Thomas Bartholomäus

nisse, die die Kommandos innerhalb der Funktion verursachen, werden gesammelt zurückgegeben. Nicht berücksichtigt werden Anweisungen, deren Ergebnis in Variablen landet.

Wenn innerhalb der Funktion die folgenden drei Anweisungen ausgeführt werden, ist das Funktionsergebnis ein Array, das sich aus dem Ergebnis von `command1`, der Zeichenkette »lorem ipsum« und dem Ergebnis von `command2` ergibt.

```
command1
command2
Write-Output "lorem ipsum"
# Ergebnis: Array mit den Teilergebnissen aus command1 und
# command2 sowie einer Zeichenkette
```

Im zweiten Beispiel verursacht nur `$a + $b` eine Ausgabe in den Success Stream. Dabei handelt es sich um eine Zahl:

```
$a = 123
$b = command    # command gibt 456 als Ergebnis
$a + $b
# Ergebnis: Integer-Zahl 579
```

Auch im dritten Beispiel werden drei Anweisungen ausgeführt. Aber weil die Ergebnisse von `command1` und `command2` in Variablen gespeichert werden und weil `Write-Debug` nicht in den Success Stream schreibt, kommt kein Ergebnis zustande!

```
$a = command1
Write-Debug "lorem ipsum"
$b = command2
# kein Ergebnis (also $null)
```

Die hier skizzierte Zusammensetzung des Ergebnisses gilt selbstverständlich auch für Kommandos, die in Schleifen ausgeführt werden. Jede Ausgabe innerhalb der Schleife erweitert das Funktionsergebnis, das zum Schluss zurückgegeben wird.

**Vorsicht mit »Write-Output«!**

Zur Fehlersuche ist es oft naheliegend, in eine Funktion `Write-Output` einzubauen, um Zwischenergebnisse auszugeben. Tun Sie das nicht! `Write-Output` produziert Ausgaben, die Teil des Funktionsergebnisses werden. Verwenden Sie besser `Write-Debug`, und aktivieren Sie Debugging-Ausgaben mit `$DebugPreference = "Continue"`!

Alternativ können Sie auch `Write-Host` verwenden. Das hat aber den Nachteil, dass Sie diese Anweisungen später wieder entfernen müssen. Ausgaben von `Write-Host` können weder umgeleitet noch sonst deaktiviert werden. Im Vergleich zu `Write-Output` hat `Write-Host` immerhin den Vorteil, dass es das Funktionsergebnis nicht verändert.


© Rheinwerk Verlag, Bonn 2024

### return

Eine Funktion endet normalerweise mit der Ausführung des letzten Kommandos. Sie können die Funktion aber auch vorzeitig mit `return` beenden (z. B. innerhalb einer Schleife, sobald eine Bedingung erfüllt ist). Wenn Sie `return` ohne Parameter verwenden, setzt sich das Funktionsergebnis wie bisher aus allen vorherigen Ausgaben zusammen:

```
command1
command2
return
# Ergebnis: Array mit den Teilergebnissen von command1 und
    command2
```

Alternativ können Sie an `return` auch eine Variable, einen Ausdruck oder ein Kommando übergeben. In diesem Fall wird das Ergebnis dieses Ausdrucks den bisherigen Teilergebnissen hinzugefügt. (Eigentlich würde man ja annehmen, dass nur der `return`-Ausdruck das Ergebnis bestimmt. Das ist bei der PowerShell aber nicht der Fall!)

```
command1
command2
return command3
# Ergebnis: Array mit den Teilergebnissen von command1, command2
# und command3
```

### Parameter

Normalerweise sollen an Funktionen Parameter übergeben werden. Es ist zweckmäßig, Position, Name, Datentyp, Defaultwert und andere Eigenschaften vorweg mit `param` zu deklarieren. Die PowerShell bietet dabei eine Fülle syntaktischer Möglichkeiten, von denen ich hier allerdings nur die wichtigsten berücksichtige. Im einfachsten Fall legen Sie einfach eine Liste von Parametern fest, die zwingend übergeben werden müssen. Die Angabe des Datentyps in eckigen Klammern ist optional; wenn Sie sich dafür entscheiden, wird der Datentyp bei der Übergabe überprüft.

```
function Do-Something {
    param(
        [type1] $Parameter1
        [type2] $Parameter2
        [type3] $Parameter3
    )
}
```

Alternativ ist die folgende Kurzschreibweise erlaubt:

```
function Do-Something ([type1]$Para1, [t2]$P2, [t3]$P3) { ... }
```

Persönliches Exemplar für Thomas Bartholomäus

Wenn Sie für Parameter wie im folgenden Beispiel einen Defaultwert angeben, wird der Parameter optional, braucht also beim Aufruf nicht angegeben zu werden.

```
function Do-It ([int]$A, [int]$B = 2, [String]C = "abc") { ... }
```

**Funktionen ohne explizite Parameterliste**

Die Verwendung von `param` ist optional. Sämtliche Daten, die nicht der vordefinierten Parameterliste entsprechen, sind innerhalb der Funktion in der vordefinierten Variablen `$args` zugänglich. Wenn Sie auf die Deklaration einer Parameterliste verzichten, dann betrifft `$args` eben sämtliche Parameter. Die PowerShell-Dokumentation spricht in diesem Zusammenhang von *positional parameters*, weil die Parameter keinen Namen haben und nur anhand ihrer Position ausgewertet werden können.

Allerdings ist die Auswertung von `$args` fehleranfällig. Intern ist `$args` ein Array. `$args[0]` ist somit der erste Parameter, `$args[1]` der zweite usw. `$args.Length` gibt an, wie viele nicht mit `param` deklarierte Parameter übergeben wurden.

### Funktionen aufrufen

Denken Sie daran, dass Sie Parameter beim Aufruf *ohne* die Nennung runder Klammern übergeben! Dieses Verhalten entspricht dem Aufruf von CmdLets, weicht aber natürlich vom Verhalten fast aller anderen Programmiersprachen ab.

```
Do-It 1 2 "abc"
```

Parameter können wie Optionen verwendet werden. Die dazugehörenden Werte folgen entweder getrennt durch ein Leerzeichen oder in der Schreibweise `-Parametername:Wert`. Die Reihenfolge spielt dann keine Rolle mehr. Die beiden folgenden Aufrufe von `Do-It` sind gleichwertig zur vorigen Zeile:

```
Do-It -C "abc" -A 1 -B 2
Do-It -B:2 -C:"abc" -A:1
```

### Standardeingabe verarbeiten

Manchmal soll eine Funktion so programmiert werden, dass sie Objekte aus der Standardeingabe verarbeiten kann, also mit dem Pipe-Operator in der Form `... | My-Function` aufgerufen werden kann. Am einfachsten ist es in solchen Fällen, die vordefinierte Variable `$input` auszuwählen. Die folgende Funktion berechnet die Summe über die `Length`-Eigenschaften aller Objekte, dividiert das Ergebnis durch 1.000.000 und zeigt es dann an:


© Rheinwerk Verlag, Bonn 2024

```
function Get-TotalSize {
    $TotalSize = 0
    foreach($item in $input) {
        $TotalSize += $Item.Length
    }
    $TotalSize /= 1000000
    "Gesamtgröße: {0:F1} MB" -f $TotalSize
}
```

Um den Platzbedarf aller Dateien im `Downloads`-Verzeichnis auszurechnen, kann die Funktion wie folgt aufgerufen werden:

```
> Get-ChildItem -Recurse C:\Users\kofler\Downloads\ |
  Get-TotalSize

  Gesamtgröße: 247,4 MB
```

Das Script hat einen Schönheitsfehler: Es verlässt sich darauf, dass jedes über die Standardeingabe übergebene Objekt tatsächlich eine `Length`-Eigenschaft aufweist. Daran stört sich auch `Set-StrictMode`, sofern Sie eine Versionsnummer größer als eins übergeben. Die einfachste Lösung besteht darin, innerhalb der Schleife den Datentyp zu überprüfen und dann eine Zuweisung in eine typisierte Variable vorzunehmen:

```
foreach($Item in $input) {
    if ($Item.GetType().Name -eq "System.IO.FileInfo") {
        [System.IO.FileInfo] $FInfo = $Item
        $TotalSize += $FInfo.Length
    }
}
```

Sollte der Datentyp nicht passen, löst das Script einen Fehler aus.

### Syntaxvarianten

Für Funktionen, die die Standardeingabe verarbeiten sollen, kennt die PowerShell zwei Syntaxvarianten. Eine besteht darin, die Funktion in drei Blöcken `begin`, `process` und `end` zu organisieren:

```
function Process-Data {
    begin   { initialization ... }
    process { process data ...   }
    end     { output, cleanup ... }
}
```

Der `process`-Block wird für jedes Element der Standardeingabe ausgeführt, wobei `$_` Zugriff auf das gerade aktuelle Element gibt. Alle drei Blöcke sind optional. Wenn Sie in einer Funktion *einen* dieser Blöcke definieren, muss der gesamte Code der

Persönliches Exemplar für Thomas Bartholomäus

Funktion in begin, process und end angegeben werden. Get-TotalSize kann dann so programmiert werden:

```
function Get-TotalSize {
    begin {
        $TotalSize = 0
    }
    process {
        $TotalSize += $_.Length
    }
    end {
      $TotalSize /= 1000000
      "Gesamtgröße: {0:F1} MB" -f $TotalSize
    }
}
```

Ab Set-StrictMode -Version 2 beklagt die PowerShell wieder, dass nicht sicher ist, ob $_ eine Length-Eigenschaft hat. Bauen Sie den Code gegebenenfalls wie vorhin erläutert um.

filter ist eine Kurzschreibweise für eine Funktion, bei der der begin- und process-Block leer sind:

```
filter Process-Data {
    code
}
# gleichwertig
function Process-Data {
    begin   { }
    process { code }
    end     { }
}
```

In der Praxis ist filter nur in Ausnahmefällen zielführend. Ohne die in begin durchgeführte Initialisierung bzw. ohne abschließende Kommandos in end lässt sich nur schwer eine sinnvolle Funktion programmieren.

## 4.12 Module

In einer *.ps1-Datei gespeicherte Funktionen sind nur innerhalb dieses Scripts zugänglich, aber nicht außerhalb. Wenn Sie Ihre Funktionen wie CmdLets interaktiv in der Shell aufrufen oder in einem anderen Script nutzen möchten, müssen Sie den Code der Funktionen in einem Modul (Dateikennung *.psm1) speichern:


© Rheinwerk Verlag, Bonn 2024

```
# Datei MyModule.psm1
Function F1 { ... }
Function F2 { ... }
Function F3 { ... }
```

Bevor Sie eine der dort gespeicherten Funktionen aufrufen können, müssen Sie das Modul einmalig importieren:

```
> Import-Module .\MyModule.psm1
```

Bis zum Ende der aktuellen PowerShell-Sitzung können Sie nun `F1`, `F2` und `F3` wie Cmd-Lets aufrufen:

```
> F1
```

Wenn Sie Änderungen oder Korrekturen im Modul durchführen, reicht es nicht aus, `Import-Module` neuerlich auszuführen. Sie müssen das Modul zuerst explizit mit `Remove-Module` aus dem PowerShell-internen Speicher löschen und es dann wieder laden!

```
> Remove-Module MyModule
> Import-Module .\MyModule.psm1
```

Die in einem geladenen Modul definierten Funktionen können Sie mit `Get-Command` auflisten:

```
> Get-Command -Module MyModule

  CommandType      Name    Version     Source
  -----------      ----    -------     ------
  Function         F1      0.0         MyModule
  Function         F2      0.0         MyModule
  Function         F3      0.0         MyModule
```

### Modulverzeichnisse

Bei `Import-Module` müssen Sie immer den Pfad des Verzeichnisses angeben, in dem Sie die Moduldatei gespeichert haben. Diese Mühe können Sie sich sparen, wenn Sie das Modul in einem Verzeichnis speichern, das die PowerShell für Moduldateien vorsieht. Eine Liste dieser durch Strichpunkte getrennten Verzeichnisse ist in der Umgebungsvariablen `$Env:PSModulePath` enthalten. Im folgenden Kommando habe ich `-split` verwendet, um den Inhalt der Variable besser lesbar auszugeben:

```
> $Env:PSModulePath -split ';'

  C:\Users\kofler\Documents\PowerShell\Modules
  C:\Program Files\PowerShell\Modules
  C:\program files\powershell\7\Modules
```

Persönliches Exemplar für Thomas Bartholomäus

```
C:\Program Files\WindowsPowerShell\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

Wie Sie sehen, bietet sich das Verzeichnis `Documents\PowerShell\Modules` in Ihrem Heimatverzeichnis gut zur Speicherung eigener Module an. (Dieses Verzeichnis ist zwar standardmäßig in `$PSModulePath` vorgesehen, es existiert aber noch nicht. Sie müssen das Verzeichnis vor der ersten Verwendung mit `mkdir` erzeugen.)

Beachten Sie, dass Sie Ihr Modul nicht direkt in einem der `PSModulePath`-Verzeichnisse speichern dürfen, sondern dass Sie dort zuerst ein Unterverzeichnis einrichten müssen, das den gleichen Namen wie die Moduldatei hat (in diesem Beispiel also `MyModule`)!

```
> mkdir C:\Users\kofler\Documents\PowerShell\Modules\MyModule
> Copy-Item MyModule.psm1 `
  C:\Users\kofler\Documents\PowerShell\Modules\MyModule
```

Bei Bedarf können Sie `PSModulePath` in den Systemeinstellungen auch um zusätzliche Verzeichnisse erweitern.

### mkdir oder »New-Item«?

Das vorhin eingesetzte Kommando `mkdir` entspricht nicht dem typischen Aussehen von CmdLets. Tatsächlich lautet der quasi offizielle Weg zum Erzeugen eines Verzeichnisses `New-Item -ItemType "directory" -Name mydir`. Aber selbst Microsoft hat eingesehen, dass das zu weit geht. Deswegen steht mit `mkdir` eine Funktion zur Verfügung, die ihrerseits `New-Item` aufruft und dabei gleich die erforderlichen Optionen übergibt.

### Module automatisch laden

Wenn die von Ihnen programmierten Funktionen so wichtig sind, dass Sie sie in jeder PowerShell-Sitzung verwenden möchten, bauen Sie `Import-Module` in Ihre Profile-Datei ein. Diese Initialisierungsdatei der PowerShell öffnen Sie am einfachsten mit `<editor> $PROFILE` in Ihrem Lieblingseditor. Wenn Sie VSCode installiert haben, lautet das Kommando also `code $PROFILE`.

## Module für Profis

Damit sich Ihre Module möglichst exakt wie vollwertige CmdLets verhalten, bestehen noch viele Optimierungsmöglichkeiten, die ich hier aber nur ganz kurz andeute:


© Rheinwerk Verlag, Bonn 2024

- `Export-ModuleMember`: Standardmäßig stehen nach `Import-Module` alle in einer Moduldatei deklarierten Funktionen und Aliasse zur Verfügung. Mit der Anweisung `Export-ModuleMember` können Sie ausgewählte Funktionen öffentlich machen:

  ```
  # nur die Funktion F2 exportieren
  Export-ModuleMember -Function F2
  ```

  Beachten Sie, dass `Export-ModuleMember` *nach* der Deklaration der Funktionen angegeben werden muss, nicht am Beginn der Moduldatei! Weitere Details finden Sie hier:

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

- Manifest-Datei: Wenn Sie Ihr Modul öffentlich zum Download anbieten möchten, müssen Sie zusätzlich eine Manifest-Datei (Kennung `*.psd1`) einrichten. Diese Datei enthält eine exakte Beschreibung der Metadaten Ihres Moduls, also Ihren Namen, die Versionsnummer des Moduls, die vorausgesetzte PowerShell-Version usw.

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

- `CmdletBinding`: Mit dem Schlüsselwort `CmdletBinding` können Sie Attribute Ihrer Funktion näher beschreiben: Wo befindet sich ein Hilfetext? Sollen die Benutzer Ihrer Funktion die Möglichkeit bekommen, jede Aktion vorher zu bestätigen? (Das ist zweckmäßig, wenn Ihre Funktion Daten verändert oder gar löscht.) Sollen die Ausgaben der Funktion seitenweise angezeigt werden?

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

**Zusatzmodule installieren**

Anstatt Module selbst zu programmieren, können Sie auf ein großes Angebot kostenloser Module in der PowerShell Gallery zurückgreifen. Wie Sie solche Module installieren, verrate ich Ihnen in Abschnitt 7.8.

## 4.13 Fehlerabsicherung

Bevor ein Script ausgeführt wird, führt die PowerShell eine oberflächliche Syntaxkontrolle durch. Nur bei gravierenden Syntaxfehlern, etwa bei nicht geschlossenen Klammern oder Anführungszeichen, wird ein Script überhaupt nicht ausgeführt.

Auch in der Folge geht die PowerShell, wie viele andere Script-Sprachen, sehr tolerant mit Fehlern um. Wenn ein Kommando gar nicht existiert, auf eine nicht existierende

Persönliches Exemplar für Thomas Bartholomäus

Datei zuzugreifen versucht oder einen anderen Fehler auslöst, zeigt die PowerShell zwar eine Fehlermeldung an, setzt das Script dann aber mit der nächsten Anweisung fort. Dieser Abschnitt zeigt, wie Sie Ihre PowerShell-Scripts möglichst gut gegen Fehler absichern.

### Set-StrictMode

Das in diesem Kapitel bereits mehrfach erwähnte Kommando `Set-StrictMode` stellt ein, wie pingelig die PowerShell Fehler feststellen soll. `Set-StrictMode` unterstützt aktuell die Versionen 1.0, 2.0 und 3.0:

- `Set-StrictMode -Version 1.0` führt zu einer Fehlermeldung bei der Verwendung einer nicht initialisierte Variable. Ausgenommen sind Variablen in Zeichenketten, also `"text $UnknownVariable"`.
- `Set-StrictMode -Version 2.0` löst bei jeder nicht initialisierten Variable sowie bei der Verwendung nicht existierender Eigenschaften einen Fehler aus.
- `Set-StrictMode -Version 3.0` löst außerdem einen Fehler aus, wenn auf nicht existierende Array-Elemente zugegriffen wird.

Die durch `Set-StrictMode` erkannten Ungenauigkeiten führen zwar zu einer Fehlermeldung, das Script wird aber dennoch fortgesetzt! Anstelle einer Versionsnummer kann auch `Latest` angegeben werden. Das ist aber mit Vorsicht zu genießen: Wenn in einer zukünftigen PowerShell-Version noch mehr Validitätstests hinzugefügt werden, meldet Ihr Script womöglich einen Fehler, obwohl sich am Code nichts geändert hat.

### Scripts bei Fehlern abbrechen

Wie sich die PowerShell verhält, wenn sie einen Fehler bemerkt, hängt vom Inhalt der Variablen `$ErrorActionPreference` ab (siehe Tabelle 4.9). Das gewünschte Verhalten ändern Sie einfach, indem Sie der Variablen eine Zeichenkette zuweisen. Das ist überall im Script erlaubt. Sie können in verschiedenen Teilen des Programms eine unterschiedliche Reaktion auf Fehler bewirken.

| Einstellung | Wirkung |
|---|---|
| `"Continue"` | Fehlermeldung anzeigen, Script fortsetzen (gilt per Default) |
| `"Inquire"` | Fehlermeldung anzeigen und fragen, ob Script fortgesetzt werden soll |
| `"SilentlyContinue"` | Script ohne Fehlermeldung fortsetzen |
| `"Stop"` | Fehlermeldung anzeigen, Script beenden |

**Tabelle 4.9** Die wichtigsten Einstellungen für die ErrorActionPreference-Variable


© Rheinwerk Verlag, Bonn 2024

### Fehlermeldungen von anderen Ausgaben trennen

Wenn Sie ein Script interaktiv in der PowerShell ausführen, erscheinen gewöhnliche Ausgaben und Fehler gleichermaßen im Terminal. Wenn Sie die Ausgabe eines Scripts aber mit `> outfile.txt` in eine Datei umleiten, dann enthält diese Datei nur die regulären Ausgaben. Die Fehlermeldungen sind weiterhin am Bildschirm sichtbar. Dieses Verhalten hat damit zu tun, dass die PowerShell zwischen mehreren »Streams« differenziert, die für unterschiedliche Zwecke gedacht sind (siehe Tabelle 4.7). Hintergründe zu diesem Thema finden Sie in Abschnitt 4.8, »Ausgabeumleitung«.

### Fehlerabsicherung mit »try/catch«

Wie bei den meisten höheren Programmiersprachen (C#, Java usw.) besteht auch in der PowerShell die Möglichkeit, mehrere Anweisungen mit kritischem (fehleranfälligem) Code in einen `try`-Block zu verpacken. Tritt dort bei der Ausführung ein Fehler auf, wird der Code im `catch`-Block fortgesetzt. Die Variable `Error` gibt dort Aufschluss über den Fehler.

```
try {
    command1
    command2
    ...
} catch {
    # Code hier wird nur ausgeführt, wenn ein
    # Fehler aufgetreten ist
    Write-Host "An error occurred: $_"
}
```

Der entscheidende Vorteil von `try-catch`-Konstruktionen besteht darin, dass Sie Ihr Script auch bei Fehlern ohne Fehlermeldung fortsetzen und bei Bedarf ganz gezielt auf den jeweiligen Fehler reagieren können.

#### Ausgabe von Fehlermeldungen

Die PowerShell macht es nicht ganz einfach, Fehlermeldungen auszugeben. Im obigen Beispiel habe ich `Write-Host` verwendet. Die Fehlermeldung erscheint damit im Terminal, lässt sich aber nicht umleiten.

Nicht empfehlenswert ist normalerweise `Write-Output`, weil die Fehlermeldung damit Teil des Ergebnisses des Scripts oder der Funktion wird.

Nur in Ausnahmefällen zweckmäßig ist `Write-Error`, weil diese Ausgabe dann – je nach Einstellung von `$ErrorActionPreference` – neuerlich einen Fehler auslöst.

Persönliches Exemplar für Thomas Bartholomäus

Zur hier präsentierten `try-catch`-Syntax gibt es zwei wichtige Syntaxvarianten:

- Sie können mehrere `catch`-Blöcke für verschiedene Fehlertypen definieren. Die PowerShell berücksichtigt dann nur den `catch`-Code, der zum aufgetretenen Fehler passt.
- Im Anschluss an die `catch`-Blöcke kann es einen `finally`-Block geben, der für Aufräumarbeiten gedacht ist. Dort formulierter Code wird *immer* ausgeführt, egal ob ein Fehler auftritt oder nicht, und unabhängig davon, ob das Script oder die Funktion in einem `try`- oder `catch`-Block mit `return` verlassen wird.

Details zu diesen Syntaxvarianten können Sie hier nachlesen:

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

### Script-Ausführung vorzeitig beenden (Exit)

Mit `Exit` beenden Sie ein Script vorzeitig. Optional können Sie dabei einen Fehlercode zurückgeben: 0 bedeutet kein Fehler, jede andere Zahl ist ein Fehlercode, deren Bedeutung Sie frei wählen können.

```
Exit 1    # Script mit Fehlercode beenden
```

Nach der Ausführung eines Scripts können Sie dessen Exit-Code der Variable `$LASTEXITCODE` entnehmen. Beachten Sie, dass es Exit-Codes nur für herkömmliche Kommandos und für Scripts gibt. CmdLets haben keinen Exit-Code.


© Rheinwerk Verlag, Bonn 2024

# Kapitel 5
# Python

Python zählt aktuell zu den beliebtesten Programmiersprachen, und das erstaunlicherweise für meilenweit auseinanderliegende Einsatzzwecke: als *First Language* zum Programmierenlernen, für naturwissenschaftliche Aufgaben samt Statistik und Visualisierung, aber auch für die KI-Forschung (neuronale Netze, Bilderkennung etc.).

In diesem Buch konzentriere ich mich auf den Einsatz von Python als Script-Sprache. Im Vordergrund steht also die Entwicklung oft winziger Programme, um administrative Aufgaben zu erledigen oder zu automatisieren. Python eignet sich z. B. großartig, um Daten von einem Format in ein anderes umzuwandeln (CSV zu JSON), um Daten aus Datenbanken zu extrahieren oder dort zu speichern oder um Daten aus dem Netzwerk zu verarbeiten (Web Scraping, Requests für REST-APIs).

Wie ich in Kapitel 1, »Scripting: Do one thing ...«, schon ausgeführt habe, ist Python im Vergleich zu Bash und PowerShell zugänglicher: Die Syntax ist klar und konsistent, die im Web verfügbare Dokumentation großartig. In Python ist es vergleichsweise einfach, eine komplizierte Logik durch Funktionen nachzubilden. Python gelingt schließlich der Spagat über alle gängigen Betriebssysteme und ist nicht wie die Bash oder die PowerShell stark in einem bestimmten Betriebssystem verankert.

Bash und PowerShell haben deswegen keineswegs Ihre Daseinsberechtigung verloren. Diese Sprachen brillieren, wenn es um den Aufruf betriebssystemnaher Kommandos oder CmdLets geht. Ein Backup-Script, das primär ein paar externe Kommandos zu einem sinnvollen Ganzen vereint, lässt sich in der Bash am einfachsten realisieren. Analog gilt das für die PowerShell, wenn Sie ein ActiveDirectory administrieren möchten.

Dieses Kapitel gibt einen Schnelleinstieg in Python. Dabei konzentriere ich mich auf die Sprachmerkmale, die für das Scripting besonders wichtig sind. Sollten Sie sich in Python »verlieben« und mehr als nur kurze Scripts verfassen, finden Sie im Internet und am Buchmarkt eine Menge Literatur zur Vertiefung – darunter auch meinen »Python-Grundkurs«, der ebenfalls im Rheinwerk Verlag erschienen ist.

Persönliches Exemplar für Thomas Bartholomäus

## 5.1 Python installieren

Unter nahezu allen Linux-Distributionen ist Python standardmäßig installiert. Sie müssen nur noch feststellen, um welche Version es sich handelt. Dazu öffnen Sie ein Terminalfenster und führen das folgende Kommando aus:

```
python --version
  Python 3.11.1
```

Falls die Versionsnummer 2.7 angezeigt wird, sind bei Ihrer Distribution die veraltete Version 2 und die aktuelle Version 3 parallel installiert. Python 3 müssen Sie dann mit dem Kommando `python3` ausführen.

### Windows

Für Windows finden Sie auf der folgenden Webseite Links zu den gerade aktuellsten Python-Releases für Windows:

*https://python.org/downloads/windows*

Nachdem Sie die gewünschte Versionsnummer ausgewählt haben, müssen Sie sich noch für das richtige Installationsprogramm entscheiden. In der Regel ist das der *Windows installer (64-bit)*.

Achten Sie darauf, dass Sie im Installationsprogramm die Option ADD PYTHON.EXE TO PATH aktivieren (siehe Abbildung 5.1), bevor Sie auf INSTALL NOW klicken! Diese Option stellt sicher, dass Sie Python später unkompliziert aus `cmd.exe` oder der PowerShell starten können.

**Abbildung 5.1** Python-Installationsprogramm für Windows


© Rheinwerk Verlag, Bonn 2024

Nachdem die eigentliche Installation abgeschlossen ist, haben Sie die Möglichkeit, das Windows-typische Limit von 260 Zeichen für den Start von Kommandos aufzuheben. Wenn ein Python-Programm mit vielen Parametern aufgerufen wird, kann dieses Limit Fehler verursachen. Klicken Sie daher auf den Button DISABLE PATH LENGTH LIMIT. (Der Button wird nicht angezeigt, wenn das Limit schon bei einer früheren Installation oder auf einem anderen Weg deaktiviert wurde.)

Nach der Installation starten Sie `cmd.exe` oder die PowerShell und verifizieren, ob sich Python samt seinem Modulmanager `pip` starten lässt:

```
> python --version
  Python 3.11.1

> pip --version
  pip 22.3.1 from C:\Users\kofler\AppData\Local\Programs\Python\
                     Python311\Lib\site-packages\pip (python 3.11)
```

**Vermeiden Sie Mehrfachinstallationen!**

Die Installation von Python unter Windows ist kinderleicht – und genau da liegt das Problem: Der Microsoft Store, Entwicklungsumgebungen, Editoren usw. unterstützen Sie jederzeit gerne bei der Installation der gerade neuesten Python-Version. Das endet mit zwei oder drei parallel installierten Versionen. Auf den ersten Blick scheint alles zu funktionieren – bis Sie versuchen, ein Erweiterungsmodul mit `pip` zu installieren. `pip` meldet keinen Fehler, trotzdem kann Python das Modul nicht finden. Die Ursache ist banal: Die von Ihnen aufgerufene `pip`-Version stammt von einer anderen Python-Version als der, mit der Sie gerade arbeiten – höchstwahrscheinlich, weil die ältere Version in der `PATH`-Umgebungsvariable bevorzugt wird.

Der Versuch, die resultierenden Probleme zu lösen, hat mich (und noch öfter meine Studenten) schon zur Verzweiflung gebracht. Die beste Lösung besteht darin, systematisch nach allen Python-Versionen am Rechner zu suchen und jede davon zu deinstallieren. Danach entfernen Sie im Einstellungsprogramm *Systemeigenschaften* sämtliche `PATH`-Verzeichnisse, die mit Python zu tun haben. Ein Windows-Neustart kann sicher auch nicht schaden. Schließlich installieren Sie die aktuellste Python-Version, wobei Sie das Installationsprogramm von der oben erwähnten Webseite herunterladen. Gehen Sie dem Microsoft Store oder irgendwelchen anderen Installationsverfahren aus dem Weg!

### macOS

Den einfachsten Weg, Python 3 unter macOS zu installieren, bietet das grafische Installationsprogramm (`*.pkg`-Datei). Sie finden es auf der schon erwähnten Python-Downloadseite *https://python.org/downloads*.

Persönliches Exemplar für Thomas Bartholomäus

Während der Installation erscheint ein Finder-Fenster, in dem Sie per Doppelklick das Script `Install Certificates.command` ausführen müssen. Dieses Script installiert diverse SSL-Zertifikate. Damit erreichen Sie, dass Root-Zertifikate installiert werden, mit denen Python-Programme die HTTPS-Verschlüsselung verifizieren können. (Python verwendet nicht die Zertifikate von macOS. Der Schritt ist deswegen unbedingt erforderlich!)

Nach der Installation überzeugen Sie sich wie unter Linux in einem neu gestarteten Terminalfenster davon, dass alles geklappt hat. Beachten Sie, dass die Kommandos `python3` und `pip3` lauten, nicht `python` und `pip` wie unter Windows und bei aktuellen Linux-Distributionen!

```
$ python3 --version
  Python 3.11.1
$ pip3 --version
  pip 22.3.1 from /Library/Frameworks/Python.framework/Versions/
             3.11/lib/python3.11/site-packages/pip (python 3.11)
```

## 5.2 Python in einem Terminalfenster kennenlernen

Im Gegensatz zu den meisten anderen Programmiersprachen können Sie Python ausprobieren, *ohne* Code in eine Datei zu schreiben. Dazu öffnen Sie ein Terminalfenster und führen darin das Kommando `python` bzw. `python3` aus. Damit starten Sie eine interaktive Python-Umgebung. Die drei Zeichen `>>>` (der »Python-Prompt«) geben darin den Ort an, an dem Sie selbst Eingaben durchführen können. [↵] beendet die Eingabe, und Python zeigt daraufhin das Ergebnis an:

```
$ python
>>> 1+2
    3
```

Beim interaktiven Ausprobieren von Python können Sie zumeist auf `print` verzichten, weil Ergebnisse ohnedies automatisch angezeigt werden:

```
>>> x=5
>>> print(x + 7)
    12
>>> x + 7
    12
```

Sie können sogar mehrzeilige Anweisungen eingeben, z. B. `for`-Schleifen. Dabei sind zwei Dinge zu beachten: Zum einen müssen die Anweisungen innerhalb der Schleife durch Leerzeichen eingerückt werden, und zum anderen müssen Sie die gesamte


© Rheinwerk Verlag, Bonn 2024

Eingabe durch *zweimaliges* [↵] abschließen. Der Python-Interpreter stellt der ersten Zeile >>> voran, bei allen Folgezeilen erscheinen drei Punkte:

```
>>> for i in range(3):
...     print(i)
...
    0
    1
    2
```

Um Python zu beenden, drücken Sie [Strg]+[D].

**Interaktive Ausgaben versus Script-Ausgaben**

In den folgenden Abschnitten sind viele Python-Kommandos abgedruckt, die zum interaktiven Ausprobieren gedacht sind und mit dem Prompt >>> gekennzeichnet sind. Auch wenn die Beispiele trivial aussehen, sollten Sie sich die Mühe machen, diese selbst nachzuvollziehen und vielleicht noch ein paar weitere Experimente durchzuführen.

Beachten Sie, dass bei der interaktiven Verwendung von Python immer das Ergebnis eines Ausdrucks angezeigt wird. `x` und [↵] zeigt den Inhalt von `x`, die Eingabe `x + 3` berechnet die Summe und zeigt das Ergebnis an.

In Scripts ist das anders: Dort ist für Ausgaben `print` erforderlich. Sie müssen in Ihr Script also `print(x)` bzw. `print(x + 3)` einbauen, um `x` oder die Summe auszugeben.

## 5.3 Eigene Scripts programmieren

Natürlich wollen Sie Python nicht in erster Linie interaktiv verwenden, sondern Scripts programmieren, um eine bestimmte Aufgabe immer wieder zu erledigen. Es gibt unzählige Editoren und Entwicklungsumgebungen, die Sie dabei unterstützen.

Eine gute Wahl ist VSCode. Sobald Sie darin ein Python-Script bearbeiten, fragt VSCode, ob Sie die empfohlenen Python-Erweiterungen installieren möchten. Das sollten Sie unbedingt tun! VSCode bietet dann neben diversen Zusatzfunktionen die Möglichkeit, Ihr Script direkt aus dem Editor heraus zu starten.

Nachdem Sie in VSCode eine neue, noch leere Datei erzeugt haben, geben Sie die folgende Zeile ein und speichern die Datei als `HelloWorld.py`:

```
print('Hello, World!')
```

Sofern Sie die Python-Erweiterung installiert haben, können Sie das Programm mit dem RUN-Button starten. Unter Windows geht die eigentliche Ausgabe unter diversen

Persönliches Exemplar für Thomas Bartholomäus

Statusinformationen von VSCode und der hinter den Kulissen eingesetzten PowerShell leider etwas verloren (siehe Abbildung 5.2).

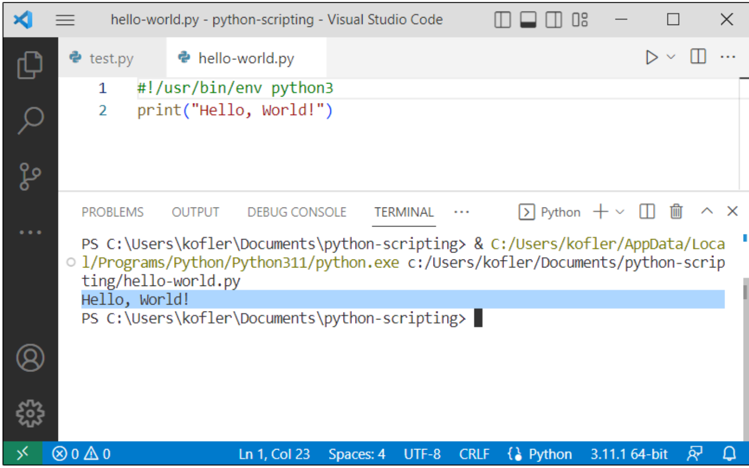



**Abbildung 5.2** Das Hello-World-Script im Editor »Visual Studio Code« unter Windows

## Shebang

Damit Sie Python-Scripts unter Linux und macOS auch ohne VSCode oder eine andere Entwicklungsumgebung ausführen können, müssen Sie das Script mit der folgenden Shebang-Zeile einleiten. Bei aktuellen Linux-Distributionen können Sie anstelle von python3 auch einfach python angeben, aber bei älteren Distributionen und unter macOS schafft nur python3 Klarheit, auf welche Python-Version Sie sich beziehen.

```
#!/usr/bin/env python3
```

Die Shebang-Syntax kennen Sie grundsätzlich schon von der Bash: Die Zeile gibt an, welches Programm (welcher Interpreter) den nachfolgenden Code ausführen soll. Neu ist hier aber env: Python wird je nach Betriebssystem bzw. Distribution in unterschiedliche Verzeichnisse installiert. env sucht in den gängigen Verzeichnissen und funktioniert deswegen zuverlässiger als die Angabe eines starren Pfads.

Jetzt müssen Sie Ihr Script mit chmod noch als ausführbar kennzeichnen und können es dann im Terminal starten:

```
$ chmod  +x hello-world.py         (Linux)
$ chmod a+x hello-world.py         (macOS)

$ ./hello-world.py

  Hello, World!
```


© Rheinwerk Verlag, Bonn 2024

Unter Windows ist die Shebang-Zeile nicht relevant, hier reicht die Kennung `*.py`. Die Shebang-Zeile stört aber nicht.

**Python-Scripts werden in einem eigenen Fenster ausgeführt**

Unter Windows kann es vorkommen, dass bei jedem Start eines Python-Scripts aus einem Terminal mit PowerShell ein neues Terminal-Fenster geöffnet wird. Das Script läuft dort, anschließend wird das Fenster geschlossen, ohne Ihnen eine Chance zu geben, das Ergebnis zu sehen.

Die Ursache für dieses Fehlverhalten liegt darin, dass die Kennung `.py` nicht in der Systemvariablen `PATHEXT` aufgelistet wird. Eigentlich sollte das nicht vorkommen (zumindest nicht nach einer Python-Neuinstallation), aber ich kann aus leidvoller Erfahrung bestätigen, dass es eben doch manchmal passiert. Windows eben!

Die Lösung besteht darin, dass Sie im Startmenü nach *Systemumgebungsvariablen bearbeiten* suchen. Es erscheint ein Dialog der Systemsteuerung, in welchem Sie auf UMGEBUNGSVARIABLEN klicken. Dann suchen Sie bei den Systemvariablen (untere Dialoghälfte) nach `PATHEXT` (nicht nach `Path`), bearbeiten diese und fügen einen Strichpunkt und die Endung `.py` hinzu. Nach einem Neustart des Terminals sollten Ihre Python-Scripts wieder *im* Terminal laufen.

## 5.4 Elementare Syntaxregeln

Python-Anweisungen sind normalerweise einzeilig. Sie werden im Gegensatz zu vielen anderen Programmiersprachen nicht durch einen Strichpunkt oder ein anderes Zeichen abgeschlossen.

Mehrzeilige Anweisungen sind erlaubt, wenn ihr Anfang und Ende durch Klammern eindeutig hervorgeht, z. B. bei Funktionsaufrufen. Alternativ können mehrzeilige Anweisungen auch mit dem Trennzeichen \ gebildet werden:

```
print("abc",
      "efg")
a = 1 + 2 + \
    3 + 4
```

Anweisungen dürfen mit einem Strichpunkt abgeschlossen werden. Normalerweise ist dieser Strichpunkt optional und hat keine Auswirkungen auf die Programmausführung. Strichpunkte erlauben es aber, mehrere Anweisungen in einer Zeile zu formulieren:

```
a = 1; b = 2; c = 3
```

Persönliches Exemplar für Thomas Bartholomäus

Die obige Dreifachzuweisung können Sie auch auf eine andere Art durchführen, indem Sie sowohl die Variablen als auch die Werte in Gruppen angeben, deren Bestandteile jeweils durch Kommas getrennt werden. Python-intern werden dabei Tupel gebildet. Beide Varianten sind *richtig*, aber die zweite Variante entspricht eher den Sprachkonzepten von Python.

```
a, b, c = 1, 2, 3
```

### Blockelemente

In Python gibt es wie in jeder anderen Programmiersprache einige Sprachelemente, die einen ganzen Block weiterer Anweisungen einleiten, z. B. Verzweigungen mit `if`, Schleifen mit `for` und `while` oder Funktionsdefinitionen mit `function`. In Python enden derartige Sprachelemente immer mit einem Doppelpunkt. Alle weiteren Anweisungen, die zum entsprechenden Block gehören, müssen eingerückt werden. Dafür entfallen die in anderen Sprachen üblichen Klammern. Also:

```
if xxx:
    anweisung1a
    anweisung1b
else:
    anweisung2a
    anweisung2b
    anweisung2c
```

Wenn die Bedingung `xxx` erfüllt ist, werden die Anweisungen 1a und 1b ausgeführt; ist sie nicht erfüllt, werden stattdessen die Anweisungen 2a, 2b und 2c ausgeführt. Mehr Details zu `if` und `else` folgen in Abschnitt 5.11, »Verzweigungen (if)«.

Die Codeeinrückung ist in Python nicht wie bei anderen Programmiersprachen optional, sondern Teil der Syntax! Für das Ausmaß der Einrückung gibt es keine starren Regeln: Ein Zeichen reicht, üblich sind vier Zeichen. Vorsicht ist bei Editoren angebracht, die Tabulatorzeichen zum Einrücken verwenden. Python nimmt an, dass sich die Tabulatorposition an Vielfachen von acht Zeichen befindet. Wenn Sie in Ihrem Editor eine andere Tabulatorbreite eingestellt haben und Tabulator- und Leerzeichen mischen, dann kann es passieren, dass Python bei der Interpretation der Einrückungen durcheinanderkommt. Am besten stellen Sie Ihren Editor so ein, dass dieser ausschließlich Leerzeichen und keine Tabulatoren verwendet.

Code darf auch direkt nach einem Blockelement angegeben werden. In einfachen Fällen lassen sich so einzeilige Bedingungen oder Schleifen formulieren:

```
if xxx: anweisung
```

Zur Not können Sie auf diese Weise mehrere Anweisungen in einer Zeile ausführen:

```
if xxx: anweisung1; anweisung2; anweisung3
```


© Rheinwerk Verlag, Bonn 2024

### Kommentare

Einfache Kommentare werden mit dem Zeichen # eingeleitet und reichen bis zum Ende der Zeile:

```
# ein Kommentar
print("abc")  # noch ein Kommentar
```

Mit """ können Sie mehrzeilige Kommentare bilden:

```
""" ein langer
    Kommentar """
```

## 5.5 Zahlen

In den folgenden Abschnitt erläutere ich Ihnen anhand von Beispielen den Umgang mit elementaren Python-Datentypen und -Datenstrukturen. Den Start machen ganze Zahlen und Fließkommazahlen. Die Grundrechenarten funktionieren in Python wie in jeder anderen Programmiersprache:

```
>>> 7+12
    19
>>> 3*6
    18
>>> 100-3*5
    85
```

Interessanterweise kennt Python für ganze Zahlen keine Grenzen. 2**100 berechnet $2^{100}$. Das Ergebnis ist aber keine Fließkommazahl mit nur 16 signifikanten Stellen, sondern weiterhin eine ganze Zahl ohne Genauigkeitsverlust. (Sie können auch $2^{10000}$ berechnen. Das Ergebnis hat über 3000 Stellen – kein Problem für Python!)

```
>>> 2 ** 100    # entspricht 2^100
    1267650600228229401496703205376
```

Divisionen werden in Python grundsätzlich in Fließkommaarithmetik durchgeführt. Wenn Sie explizit eine ganzzahlige Division wünschen, verwenden Sie den Operator //. Das Prozentzeichen liefert den Rest einer ganzzahligen Division.

```
>>> 17 / 4      # Python führt immer Fließkommadivisionen durch
    4.25
>>> 17 // 4     # Verwenden Sie // für ganzzahlige Divisionen
    4
>>> 17 % 4      # % liefert den Rest einer ganzzahligen Division
    1
```

Persönliches Exemplar für Thomas Bartholomäus

Sie können Zahlen in der Schreibweise `0xnnn` hexadezimal schreiben bzw. mit der Methode `hex` in eine Zeichenkette in hexadezimaler Schreibweise umwandeln:

```
>>> 0xff
    255
>>> hex(240)
    '0xf0'
```

### Fließkommazahlen

Wenn auch nur eine Zahl des Ausdrucks eine Fließkommazahl ist, führt Python die gesamte Berechnung in Fließkommaarithmetik durch. Die Genauigkeit beträgt dabei sechzehn Stellen:

```
>>> 12.0 - 2*3
    6.0
>>> 100 / 7
    14.285714285714286
```

Viele mathematische Funktionen und Konstanten befinden sich im Modul `math`, das vor seiner ersten Verwendung importiert werden muss:

```
>>> import math            # math-Modul laden
>>> math.sqrt(2)
    1.4142135623730951
>>> math.sin(math.pi / 8)
    0.3826834323650898
```

**Module**

Python besteht aus einem relativ kompakten Sprachkern und unzähligen Erweiterungsmodulen, die vor ihrer ersten Verwendung mit `import` geladen werden müssen. `import math` ist das erste Beispiel dafür, viele weitere werden noch folgen. Details zu diesem Konzept samt einer Beschreibung der unterschiedlichen `import`-Syntaxvarianten folgen in Abschnitt 5.17, »Module«.

### Zufallszahlen

Zufallszahlen können mit den Funktionen des `random`-Moduls erzeugt werden. `randrange(n)` liefert eine zufällige ganze Zahl von 0 bis n–1. `randrange(10, 20)` liefert eine Zufallszahl von 10 bis 19. Bei `randint` sind die Grenzen hingegen inklusive, d. h., `randint(10, 20)` ergibt eine Zufallszahl von 10 bis 20. `random()` liefert eine zufällige Fließkommazahl zwischen 0 und 1.


© Rheinwerk Verlag, Bonn 2024

```
>>> import random             # random-Modul laden
>>> random.randrange(100)     # Zufallszahl zwischen 0 und 99
    58
>>> random.random()           # Fließkommazahl zwischen 0 und 1
    0.26011495295431664
```

## 5.6 Zeichenketten

In Python können Zeichenketten wahlweise in einfache oder doppelte Apostrophe gestellt werden. Beide Schreibweisen sind gleichwertig und unterscheiden sich nur dadurch, dass der jeweils andere Apostroph unkompliziert in die Zeichenkette integriert werden kann. In den folgenden Zeilen werden drei Zeichenketten definiert. Sie werden in den drei Variablen `s`, `t` und `u` gespeichert.

```
>>> s = 'Python macht Spaß!'
>>> t = "Rheinwerk"
>>> u = '<img src="https://xxx">'
```

Unabhängig vom Apostrophtyp können Apostrophe mit \' oder \" in die Zeichenkette eingebaut werden. Außerdem gibt es einige weitere \-Zeichenkombinationen, die in Python Escape-Sequenzen genannt werden (siehe Tabelle 5.1). In der interaktiven Anwendung von Python verhalten sich Zeichenketten mit solchen Sonderzeichen unterschiedlich, je nachdem, ob sie direkt oder mit `print` ausgegeben werden:

```
>>> v='abc\nefg'
>>> v
    'abc\nefg'
>>> print(v)
    abc
    efg
```

Um mehrzeilige Zeichenketten zu bilden, verdreifachen Sie die Anführungszeichen:

```
>>> sql = """INSERT INTO mytable (name, date)
             VALUES ('Maria Müller',
                     '1988-12-31')"""
>>> print(sql)

INSERT INTO mytable (name, date)
             VALUES ('Maria Müller',
                     '1988-12-31')
```

Auch bei mehrzeiligen Zeichenketten haben Sie die Wahl zwischen beiden Apostrophtypen. Leerzeichen am Zeilenbeginn (im obigen Beispiel vor `INSERT`) sind Teil der Zeichenkette. Bei Bedarf können Sie unerwünschte Einrückungen mit `inspect.cleandoc` entfernen. Dabei bleiben Einrückungen relativ zur ersten eingerückten Zeile erhalten:

```
>>> import inspect
>>> print(inspect.cleandoc(sql))

INSERT INTO mytable (name, date)
VALUES ('Maria Müller',
        '1988-12-31')
```

| Zeichensequenz | Bedeutung |
|---|---|
| \a | Bell (Signalton) |
| \f | Formfeed (neue Seite) |
| \n | Zeilenumbruch |
| \r | Wagenrücklauf (für Windows-Textdateien) |
| \t | Tabulatorzeichen |
| \unnnn | Unicode-Zeichen mit dem Hexcode &xnnnn |
| \' | das Zeichen ' |
| \" | das Zeichen " |
| \\ | das Zeichen \ |

**Tabelle 5.1** Ausgewählte Escape-Sequenzen

### raw-Zeichenketten

Manche Zeichenketten enthalten das Zeichen \, ohne dass dieses als Escape-Sequenz ausgewertet werden soll. Das trifft z. B. bei Windows-Verzeichnisnamen oder LaTeX-Kommandos zu. Um solche Zeichenketten ohne eine ständige Verdoppelung von \ zu formulieren, stellen Sie der gesamten Zeichenkette den Buchstaben `r` (*raw*) voran:

```
windir = r'C:\Windows\System'
latex = r'\index{raw-Zeichenketten}'
```

### Zeichenketten verarbeiten

Mit den beiden Operatoren + und * werden Zeichenketten aneinandergefügt bzw. vervielfältigt:

```
>>> 'abc' + 'efg'
    'abcefg'
>>> 'abc' * 3
    'abcabcabc'
```


© Rheinwerk Verlag, Bonn 2024

Steht eine Zeichenkette einmal zur Verfügung, kann sie auf vielfältige Weise analysiert und verarbeitet werden:

```
>>> len(s)                # Anzahl der Zeichen
    18
>>> s.lower()             # in Kleinbuchstaben umwandeln
    'python macht spaß!'
>>> s.find('macht')       # ermittelt die Position, an der die
                          # Zeichenkette 'macht' in s vorkommt
    7
>>> s.count('a')          # zählt das Vorkommen von 'a'
    2
>>> s.replace('a', 'A')   # ersetzt 'a' durch 'A'
    'Python mAcht SpAß!'
>>> s.split(' ')          # zerlegt s in eine Liste
    ['Python', 'macht', 'Spaß!']
```

| Methode | Funktion |
|---|---|
| `len(s)` | ermittelt die Anzahl der Zeichen |
| `str(x)` | wandelt `x` in eine Zeichenkette um |
| `sub in s` | testet, ob `sub` in `s` vorkommt |
| `s.count(sub)` | ermittelt, wie oft `sub` in `s` vorkommt |
| `s.endswith(sub)` | testet, ob `s` mit `sub` endet |
| `s.expandtabs()` | ersetzt Tabulatorzeichen durch Leerzeichen |
| `s.find(sub)` | sucht `sub` in `s` und liefert die Startposition oder `-1` zurück |
| `s.isxxx()` | testet Eigenschaften von `s` (`islower()`, `isupper()` etc.) |
| `s.join(x)` | verbindet die Zeichenketten in `x` (Liste, Set, Tupel) |
| `s.lower()` | liefert `s` mit lauter Kleinbuchstaben zurück |
| `s.partition(sub)` | trennt `s` auf und liefert drei Teile als Tupel zurück |
| `s.replace(old, new)` | liefert `s` zurück, wobei `old` jeweils durch `new` ersetzt wird |
| `s.rfind(sub)` | wie `find`, aber beginnt die Suche am Ende der Zeichenkette |
| `s.split(sub)` | zerlegt `s` bei jedem Vorkommen von `sub`, liefert eine Liste |
| `s.startswith(sub)` | testet, ob `s` mit `sub` beginnt |
| `s.upper()` | liefert `s` mit lauter Großbuchstaben zurück |

**Tabelle 5.2** Ausgewählte String-Methoden und -Funktionen

Persönliches Exemplar für Thomas Bartholomäus

## Slicing

Mit der Schreibweise s[n] extrahieren Sie ein einzelnes Zeichen aus einer Zeichenkette. n gibt dabei die Position des Zeichens an, wobei 0 für das erste Zeichen steht. (Diese Regel gilt generell in Python: Bei jeder Art von Aufzählung bezeichnet 0 das erste Element!) Bei negativen Zahlen wird vom Ende der Zeichenkette gerechnet.

```
>>> s[0], s[1], s[2]       # liefert das erste, das zweite
                           # und das dritte Zeichen
    ('P', 'y', 't')
>>> s[-1], s[-2]           # liefert das letzte und das vorletzte
                           # Zeichen
    ('!', 'ß')
```

Um mehrere Zeichen auf einmal zu lesen, kennt Python die Syntax s[start:ende], wobei start den Beginn und ende das Ende des Teilstrings angibt. Beide Angaben sind optional. Die Startposition wird *inklusive*, die Endposition aber *exklusive* angegeben! s[:5] liefert also alle Zeichen bis einschließlich des fünften Zeichens. Die offizielle Bezeichnung für diese Art des Zeichenzugriffs lautet *Slicing*.

```
>>> s[:]                    # alles
    'Python macht Spaß!'
>>> s[:5]                   # alles bis zum fünften Zeichen
    'Pytho'
>>> s[5:]                   # alles ab dem sechsten Zeichen
    'n macht Spaß!'
>>> s[5:10]                 # vom sechsten bis zum zehnten Zeichen
    'n mac'
```

Die Start- und die Endposition können Sie auch durch negative Zahlen angeben – dann beginnt Python vom Ende der Zeichenkette her zu zählen:

```
>>> s[:-3]                 # alles außer den letzten drei Zeichen
    'Python macht Sp'
>>> s[-3:]                 # alles ab dem drittletzten Zeichen
    'aß!'
```

Durch einen optionalen dritten Parameter kann eine Art Schrittweite angegeben werden. In der Praxis ist hier am häufigsten der Wert -1 gebräuchlich, um die Reihenfolge einer Zeichenkette umzukehren:

```
>>> s[::2]                 # jedes zweite Zeichen
    'Pto ah pß'
>>> s[::-1]                # alles in umgekehrter Reihenfolge
    '!ßapS thcam nohtyP'
```


© Rheinwerk Verlag, Bonn 2024

### print und input

Beim Kennenlernen von Python ist die `print`-Funktion allgegenwärtig. Sie übergeben an die Funktion einen oder mehrere Parameter. `print` wandelt die Parameter in Zeichenketten um und gibt diese aus. Dabei wird zwischen den Parametern jeweils ein Leerzeichen und am Ende ein Zeilenumbruchzeichen gesetzt. `print` ist unkompliziert zu verwenden und kommt mit jeder Art von Python-Objekt zurecht, also z. B. auch mit Listen:

```
>>> print(1, 2, 3/4, 'abc', 2==3)
    1 2 0.75 abc False
>>> print('1/7 ist', 1/7)
    1/7 ist 0.14285714285714285
>>> lst = ['eine', 'Liste']
>>> print(lst)
    ['eine', 'Liste']
```

`print` kennt drei optionale Parameter:

- `sep` stellt die Zeichenkette ein, die zwischen den Parametern ausgegeben wird – standardmäßig `' '`.
- `end` definiert die Zeichenkette, die nach dem letzten Parameter ausgegeben wird – standardmäßig `'\n'`.
- `file` bestimmt, wo die Ausgabe durchgeführt wird. Normalerweise werden die Ausgaben zur Standardausgabe umgeleitet. `file` gibt Ihnen die Möglichkeit, die Ausgaben in eine Textdatei zu schreiben.

```
>>> print(1, 2, 3, sep='---')
    1---2---3
>>> print(1, 2, 3, sep=';', end='.\nEOF\n')
    1;2;3.
    EOF
>>> f = open('out.txt', 'w')
>>> print(1, 2, 3, file=f)
>>> f.close()
```

So wie Sie mit `print` Ausgaben in einem Terminalfenster durchführen können, verarbeitet `input` Texteingaben. `input` gibt zuerst den im optionalen Parameter angegebenen Text aus und erwartet dann eine Eingabe, die mit ↵ abgeschlossen werden muss. Leere Eingaben, also ein ↵ ohne Text, quittiert `input` mit einem Fehler.

```
name = input('Geben Sie Ihren Namen an:')
print('Ihr Name lautet:', name)
```

Persönliches Exemplar für Thomas Bartholomäus

`input` liefert immer eine Zeichenkette. Wenn Sie die Zeichenkette als Zahl interpretieren möchten (z. B. um danach eine Berechnung durchzuführen), müssen Sie `int` einsetzen:

```
number = input('Geben Sie eine Zahl ein:')
print(int(number) * 2)
```

### Ausgaben formatieren

Häufig müssen Sie aus Zahlen, Datums- und Zeitangaben etc. Zeichenketten bilden. Im einfachsten Fall verwenden Sie dazu die Funktionen `str(x)` oder `repr(x)`, die jedes beliebige Objekt als Zeichenkette darstellen. Die Funktion `repr` geht dabei so vor, dass die resultierende Zeichenkette mit `eval` wieder eingelesen werden kann. `str` bemüht sich hingegen, die Zeichenketten so zu formatieren, dass sie für Menschen gut leserlich sind.

Beide Funktionen geben keinen Einfluss auf die Formatierung. Wenn Sie Zahlen rechtsbündig formatieren oder mit Tausendertrennung darstellen möchten, dann benötigen Sie spezielle Formatierungsfunktionen. Unter Python haben Sie dabei die Wahl zwischen mehreren Verfahren. Am populärsten sind der `%`-Operator und die `format`-Methode:

- `formatzeichenkette % (daten, daten, daten)`: Hier wird die Formatzeichenkette in der Syntax der `printf`-Funktion der Programmiersprache C formuliert. Innerhalb dieser Zeichenkette geben `%`-Zeichen die Positionen an, an denen Daten einzusetzen sind.
- `formatzeichenkette.format(daten, daten, daten)`: Der Aufbau der Zeichenkette hat große Ähnlichkeiten mit dem Aufbau der gleichnamigen Methode des .NET-Frameworks von Microsoft. Innerhalb dieser Zeichenkette geben `{}`-Klammernpaare die Position der Parameter an.

Zuerst zwei Beispiele für das `%`-Verfahren:

```
>>>'1/7 mit drei Nachkommastellen: %.3f' % (1/7)
    '1/7 mit drei Nachkommastellen: 0.143'

>>> '<img src="%s" alt="%s" width="%d">' % ('foto.jpg',
         'Portrait', 200)
    '<img src="foto.jpg" alt="Portrait" width="200">'
```

Der größte Vorteil der neueren `format`-Methode besteht darin, dass Sie die Platzhalterreihenfolge durch `{n}` oder `{parametername}` frei wählen können. Die folgenden Beispiele illustrieren die sich daraus ergebenden Möglichkeiten:



© Rheinwerk Verlag, Bonn 2024

```
>>> '{} ist {} Jahre alt.'.format('Peter', 9)
    'Peter ist 9 Jahre alt.'

>>> '{1} ist {0} Jahre alt.'.format(9, 'Peter')
    'Peter ist 9 Jahre alt.'

>>> '{name} ist {alter} Jahre alt.'.format(alter=9, name='Peter')
    'Peter ist 9 Jahre alt.'

>>> '1/7 mit drei Nachkommastellen: {:.3f}'.format(1/7)
    '1/7 mit drei Nachkommastellen: 0.143'

>>> 'SELECT * FROM table WHERE id={:d}'.format(324)
    'SELECT * FROM table WHERE id=324'
```

Es gibt unzählige Codes zum Aufbau der Zeichenketten für die beiden Formatierungssysteme (siehe Tabelle 5.3 und Tabelle 5.4). Für eine vollständige Referenz fehlt hier der Platz. Es gibt im Internet genügend Seiten, die alle zulässigen Codes in allen denkbaren Varianten beschreiben. Die beiden folgenden Links verweisen auf die offizielle Python-Dokumentation:

*https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting*
*https://docs.python.org/3/library/string.html#format-string-syntax*

| Code | Bedeutung |
|---|---|
| %d | ganze Zahl (dezimal) |
| %5d | ganze Zahl mit fünf Stellen, rechtsbündig |
| %-5d | ganze Zahl mit fünf Stellen, linksbündig |
| %f | Fließkommazahl (*float*) |
| %.2f | Fließkommazahl mit zwei Nachkommastellen |
| %r | Zeichenkette, Python verwendet `repr`. |
| %s | Zeichenkette, Python verwendet `str`. |
| %10s | Zeichenkette mit zehn Zeichen, rechtsbündig |
| %-10s | Zeichenkette mit zehn Zeichen, linksbündig |
| %x | ganze Zahl hexadezimal ausgeben |

**Tabelle 5.3** Ausgewählte Codes für die %-Formatierung (printf-Syntax)

Persönliches Exemplar für Thomas Bartholomäus

| Code | Bedeutung |
|---|---|
| `{}` | Parameter, beliebiger Datentyp |
| `{0}, {1}, ...` | nummerierte Parameter |
| `{eins}, {zwei}, ...` | benannte Parameter |
| `{:d}` | ganze Zahl |
| `{:<7d}` | ganze Zahl mit sieben Stellen, linksbündig |
| `{:>7d}` | ganze Zahl mit sieben Stellen, rechtsbündig |
| `{:^7d}` | ganze Zahl mit sieben Stellen, zentriert |
| `{:f}` | Fließkommazahl |
| `{:.5f}` | Fließkommazahl mit fünf Nachkommastellen |
| `{:s}` | Zeichenkette |

**Tabelle 5.4** Ausgewählte Codes für die format-Methode

### Die Qual der Wahl

Was ist nun besser, das %-Verfahren oder die `format`-Methode? In der Regel kommen Sie mit beiden Verfahren zum Ziel. Wenn Ihnen die `printf`-Syntax vertraut ist, spricht nichts dagegen, beim %-Verfahren zu bleiben. Ansonsten sollten Sie für neuen Code `format` vorziehen – der resultierende Code ist zumeist besser lesbar.

## 5.7 Listen

Python bietet mit *Listen* ein ungemein flexibles Sprachkonstrukt zur Verwaltung größerer Datenmengen. Listen werden in eckigen Klammern formuliert. Listen können Elemente beliebigen Datentyps aufnehmen. Der Zugriff auf die Listenelemente erfolgt wie auf die Zeichen einer Zeichenkette durch »Slicing«, also in der Form `liste[start:ende]`. Außerdem stellt Python unzählige Funktionen und Methoden zur Bearbeitung von Listen zur Auswahl (siehe Tabelle 5.5).

```
>>> lst = [1, 2.3, 'abc', 'efg', 12]
>>> lst[2]                # das dritte Element
    'abc'
>>> lst[2:4]              # vom dritten bis zum vierten Element
    ['abc', 'efg']
>>> lst[::-1]             # umgekehrte Reihenfolge
    [12, 'efg', 'abc', 2.3, 1]
```


© Rheinwerk Verlag, Bonn 2024

```
>>> lst[0] = 3            # ändert das erste Listenelement
>>> lst
    [3, 2.3, 'abc', 'efg', 12]
```

Da als Listenelement jedes beliebige Python-Objekt erlaubt ist, sind auch verschachtelte Listen zulässig:

```
>>> lst = [[1, 2],
           [3, 4]]
>>> lst
    [[1, 2], [3, 4]]
```

Die Zeichen einer Zeichenkette können mit `list` in eine Liste umgewandelt werden:

```
>>> characters = list('Hello World!')
>>> characters
    ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!']
```

| Funktion/Methode | Bedeutung |
|---|---|
| `del l[start:ende]` | entfernt die angegebenen Listenelemente |
| `n = len(l)` | liefert die Anzahl der Elemente |
| `l.append(x)` | fügt das Element `x` am Ende der Liste hinzu |
| `l.clear()` | löscht die Liste (entspricht `l=[]`) |
| `n = l.count(x)` | ermittelt, wie oft das Element `x` in der Liste vorkommt |
| `l1.extend(l2)` | fügt die Liste `l2` am Ende von `l1` hinzu (also `l1 += l2`) |
| `iterator = filter(f, l)` | liefert die Elemente zurück, für die `f(element)==true` gilt |
| `n = l.index(x)` | ermittelt die erste Position von `x` in der Liste |
| `l.insert(n, x)` | fügt das Element `x` an der Position `n` in die Liste ein |
| `iterator = map(f, l)` | wendet die Funktion `f` auf alle Elemente an |
| `x = l.pop(n)` | liest das Element an der Position `n` und entfernt es |
| `l.remove(x)` | entfernt das Element `x` aus der Liste |
| `l.reverse()` | kehrt die Liste um (das erste Element zuletzt etc.) |
| `l.sort()` | sortiert die Liste |
| `iterator = zip(l1, l2)` | verbindet die Listenelemente paarweise zu Tupeln |

**Tabelle 5.5** Wichtige Funktionen und Methoden zur Bearbeitung von Listen

Persönliches Exemplar für Thomas Bartholomäus

### List Comprehension

Bei der sogenannten *List Comprehension* wird eine Anweisung in der Form `[ausdruck for x in liste]` gebildet. Python setzt nun der Reihe nach jedes Element der Liste in die Variable `x` ein und wertet dann den Ausdruck aus. Die Ergebnisse ergeben eine neue Liste. Im zweiten Beispiel ist jedes Ergebnis selbst eine aus zwei Elementen bestehende Liste, sodass die resultierende Liste verschachtelt ist.

```
>>> lst = [x for x in range(10, 101, 10)]
>>> lst
    [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>> [x * 2 + 1 for x in lst]
    [21, 41, 61, 81, 101, 121, 141, 181, 201]
>>> [ [x, x * x] for x in lst]
    [[10, 100], [20, 400], [30, 900], [40, 1600], [50, 2500],
     [60, 3600], [70, 4900], [90, 8100], [100, 10000]]
```

### map und filter

`map` wendet eine Funktion auf alle Elemente einer Liste an. `map` liefert aus Effizienzgründen nicht unmittelbar eine Ergebnisliste, sondern einen *Iterator*. Dieses Objekt kann z. B. in einer Schleife ausgewertet oder mit `list` in eine Liste umgewandelt werden. Das folgende Beispiel zerlegt zuerst einen Satz in eine Wortliste und ermittelt dann mit `map` die Länge aller Worte.

```
>>> lst = 'Python macht Spaß!'.split(' ')
    # lst enthält ['Python', 'macht', 'Spaß!']
>>> map(map(len, lst)
    <map object at 0x7f17bfbd1090>
>>> list(map(len, lst))
    [6, 5, 5]
```

`filter` wendet ähnlich wie `map` eine Funktion auf jedes Listenelement an. Das Ziel ist es diesmal aber nicht, die Funktionsergebnisse zurückzugeben, sondern vielmehr alle Listenelemente, bei denen die Filterfunktion `true` liefert. Es geht also darum, alle Elemente aus einer Liste herauszufiltern, die einer Bedingung genügen. Das Ergebnis ist wie bei `map` ein Iterator, der bei Bedarf mit `list` zu einer Liste ausgewertet werden kann.

Das folgende Beispiel filtert aus einer Liste alle geraden Zahlen heraus. Der Lambda-Ausdruck ergibt `true`, wenn bei einer Division durch 2 kein Rest bleibt. `lambda` definiert hier die Filter-Funktion ad hoc. Eine Erklärung zu `lambda` folgt in Abschnitt 5.13, »Funktionen«.

```
>>> lst = list(range(1,11)); lst
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```


© Rheinwerk Verlag, Bonn 2024

```
>>> even = filter(lambda x: x % 2 == 0, lst)
>>> list(even)
    [2, 4, 6, 8, 10]
```

Das gleiche Ergebnis können Sie auch mit *List Comprehension* erzielen, wenn Sie die Bedingung im Anschluss an die Liste formulieren:

```
>>> [ x for x in lst if x % 2 == 0 ]
    [2, 4, 6, 8, 10]
```

## 5.8 Tupel, Sets und Dictionaries

Listen sind in vielen Python-Scripts die dominierende Datenstruktur. Dennoch sollten Sie drei weitere elementare Python-Datenstrukturen kennen, nämlich *Tupel* (Sequenzen), *Sets* (Mengen) und *Dictionaries*.

Ein Tupel ist eine Sonderform einer unveränderlichen Liste. Tupel sind gewissermaßen die *primitivere* Datenstruktur. Ihre interne Verwaltung ist dafür mit geringem Overhead verbunden. Tupel werden in runden Klammern formuliert. Wenn es zu keinen syntaktischen Mehrdeutigkeiten kommen kann, ist es erlaubt, auf die Klammern zu verzichten.

```
>>> t = (12, 73, 3)
>>> t
    (12, 73, 3)
>>> t = 12, 73, 3
>>> t
    (12, 73, 3)
```

Eine sinnvolle Anwendung von Tupeln sind Funktionen, die nicht *ein* Ergebnis zurückgeben, sondern ein Wertepaar oder eine Kombination mehrerer Daten. So liefert das folgende Beispiel ein Koordinatenpaar zurück. (Wie Funktionen funktionieren, lernen Sie in Abschnitt 5.13, »Funktionen«.)

```
def myfunc():          # Funktion definieren
  x = 27
  y = 42
  return (x, y)

print(myfunc())        # Funktion aufrufen, Ausgabe: (27, 42)
```

### Sets (Mengen)

Ein *Set* ist eine ungeordnete Menge von Elementen ohne Doppelgänger. Es ist unmöglich, dass ein Set mehrfach dasselbe Objekt enthält. Anders als bei Listen merkt sich

Persönliches Exemplar für Thomas Bartholomäus

Python die Reihenfolge der Elemente nicht. Sie dürfen sich nicht darauf verlassen, dass die Elemente in derselben Reihenfolge verarbeitet werden, in der Sie sie eingefügt haben.

Sets werden in geschwungenen Klammern formuliert. Zeichenketten können mit `set` in ein Zeichen-Set umgewandelt werden, wobei Doppelgänger automatisch eliminiert werden.

```
>>> s = {1, 2, 3}
>>> s
    {1, 2, 3}
>>> s = set('Hello World!')
>>> s
    {'r', 'W', '!', ' ', 'e', 'd', 'H', 'l', 'o'}
```

Mit Sets können Sie Mengenoperationen durchführen, z. B. um herauszufinden, welche Objekte aus Set 1 auch in Set 2 enthalten sind. Ich habe die Ausgaben hier nur zur besseren Lesbarkeit geordnet dargestellt; wenn Sie die Tests selbst probieren, werden die Buchstaben in zufälliger Reihenfolge ausgegeben.

```
>>> x = set('abcdef')
>>> y = set('efgh')
>>> x | y     # Vereinigung
    {'a', 'c', 'b', 'e', 'd', 'g', 'f', 'h'}
>>> x - y     # Differenz
    {'a', 'c', 'b', 'd'}
>>> x & y     # Schnittmenge (gemeinsame Elemente)
    {'e', 'f'}
```

### Dictionaries

Bei Listen und Tupeln erfolgt der Zugriff auf einzelne Elemente über einen numerischen Index, also in der Form `liste[n]` oder `tupel[n]`. Dictionaries ermöglichen es hingegen, Elementaufzählungen mit einem beliebigen Schlüssel zu verwalten. In manchen Programmiersprachen (z. B. in der Bash) werden derartige Datenstrukturen als *assoziative Arrays* bezeichnet.

Dictionaries werden wie Sets in geschwungenen Klammern formuliert. Im Unterschied zu Sets werden aber immer Schlüssel-Wert-Paare gespeichert. Das folgende Beispiel speichert einige HTML-Farbcodes, wobei als Schlüssel der Name der Farbe verwendet wird:

```
>>> colors = {'red' : 0xff0000, 'green' : 0xff00, 'blue' : 0xff,
              'white' : 0xffffff}
```


© Rheinwerk Verlag, Bonn 2024

Dictionaries sind nicht nur syntaktisch, sondern auch funktionell mit Sets verwandt: Die Reihenfolge der Elemente bleibt nicht erhalten. Zum Zugriff auf die Elemente des Dictionarys können Sie nun den Schlüssel verwenden. hex wandelt die dezimal gespeicherten Zahlen in die hexadezimale Schreibweise um:

```
>>> colors['red']        # Zugriff auf ein Element
    16711680
>>> hex(colors['red'])
    '0xff0000'
```

Der Versuch, auf ein nicht vorhandenes Element zuzugreifen, führt zu einem *KeyError*. Um das zu vermeiden, können Sie vorweg mit key in dict testen, ob das Dictionary ein Element für einen bestimmten Schlüssel enthält:

```
>>> colors['yellow']
    KeyError: 'yellow'
>>> 'yellow' in colors
    False
```

len liefert wie bei Listen, Sets und Tupeln die Anzahl der Elemente. Mit der Anweisung dict[key]=value erweitern Sie das Dictionary:

```
>>> len(colors)          # ermittelt die Elementanzahl
    4
>>> colors['black'] = 0  # fügt ein Element hinzu
```

del dict[key] entfernt ein Element:

```
>>> del colors['red']    # löscht ein Element
>>> colors               # alle Key-Value-Paare ausgeben
    {'black': 0, 'blue': 255, 'white': 16777215, 'green': 65280}
```

Die Methoden keys und values liefern alle Schlüssel bzw. alle Werte des Dictionarys. Gegebenenfalls können Sie diese Daten mit list oder set in Listen oder Sets umwandeln:

```
>>> colors.values()
    dict_values([0, 255, 16777215, 65280])
>>> colors.keys()
    dict_keys(['black', 'blue', 'white', 'green'])
>>> set(colors.keys())
    {'black', 'blue', 'white', 'green'}
>>> list(colors.keys())
    ['black', 'blue', 'white', 'green']
```

Wenn Sie eine for-Schleife über ein Dictionary bilden, setzt Python in die Schleifenvariable alle Schlüssel ein:

Persönliches Exemplar für Thomas Bartholomäus

```
>>> for c in colors:
      print("Die Farbe", c, " hat den Farbcode", hex(colors[c]))

Die Farbe black hat den Farbcode 0x0
Die Farbe blue  hat den Farbcode 0xff
Die Farbe white hat den Farbcode 0xffffff
Die Farbe green hat den Farbcode 0xff00
```

In den obigen Beispielen wurden immer Zeichenketten als Schlüssel verwendet. Grundsätzlich ist aber jedes Python-Objekt als Schlüssel geeignet. Die einzige Voraussetzung besteht darin, dass die Schlüssel eindeutig sind.

**»Echte« Beispiele**

Auf den vorangegangenen Seiten habe ich mich bemüht, die Funktionsweise grundlegender Datenstrukturen mit möglichst einfachen Beispielen darzustellen. Sollte dabei der fälschliche Eindruck entstanden sein, dass Listen, Sets usw. nur eine mathematische Spielerei seien, blättern Sie vor zu Abschnitt 5.14, »Textdateien verarbeiten«, oder werfen Sie einen Blick in die Beispiele von Kapitel 10, »JSON XML und INI«!

## 5.9 Variablen

Für den Umgang mit Variablen gelten in Python einfache Regeln:

- **Zuweisung vor der Verwendung:** Jeder Variablen muss ein Startwert zugewiesen werden, bevor die Variable in einem Ausdruck ausgewertet werden kann. Es ist also nicht erlaubt, `x=x+1` auszuführen, wenn Sie nicht vorher eine erstmalige Zuweisung wie `x=0`, `x=1`, `x=n` durchgeführt haben.
- **Keine Typdeklaration:** In Python-Variablen können Objekte jedes beliebigen Typs gespeichert werden. Python merkt sich den Typ und weiß somit, auf welche Art von Daten eine Variable verweist.

  Im Gegensatz zu vielen anderen Programmiersprachen kann der Typ einer Variablen aber nicht festgelegt oder eingeschränkt werden. Es ist ohne Weiteres möglich, in derselben Variablen Daten unterschiedlichen Typs zu speichern, also z. B. zuerst eine Zahl (`x=1`), später eine Zeichenkette (`x='abc'`) und schließlich eine Liste (`x=[3,2,1]`).
- **Namen:** Variablennamen müssen mit Buchstaben oder einem Unterstrich beginnen. Allerdings ist der Unterstrich am Beginn von Variablennamen für Python-interne Daten vorgesehen, weswegen Sie ihn in eigenen Scripts möglichst nicht verwenden sollten. Die weiteren Zeichen dürfen auch Ziffern enthalten.


© Rheinwerk Verlag, Bonn 2024

```
a = 1
b = 'abc'
a = a + 1
a = c + 1               # Fehler: An c wurde nie etwas zugewiesen.
longName = 3            # OK
another_long_name = 4   # auch OK
long name = 5           # Fehler: Leerzeichen sind unzulässig.
long-name = 6           # Fehler: - ist unzulässig.
```

**Python kennt keine Konstanten**

Python bietet keine Möglichkeit, Konstanten zu definieren. Es ist üblich, den Namen einer Variablen, die wie eine Konstante verwendet werden soll, aus lauter Großbuchstaben zu bilden, also z. B. `MAXNO=10`. Sie können aber nicht verhindern, dass dieser Variablen später ein anderer Wert zugewiesen wird.

### Zuweisungen

Gewöhnliche Zuweisungen der Art `variable = ausdruck` gibt es in jeder Programmiersprache. Python kennt darüber hinaus einige ungewöhnliche Varianten. Beispielsweise können Sie mehreren Variablen auf einmal denselben Inhalt zuweisen. Python geht dabei von rechts nach links vor. Das heißt, im Beispiel unten werden die folgenden Zuweisungen durchgeführt: `c=16`, dann `b=c` und schließlich `a=b`. Damit verweisen alle drei Variablen auf dasselbe Objekt im Speicher, das die Zahl `16` repräsentiert.

```
a = b = c = 16        # drei Variablen denselben Wert zuweisen
```

Wenn Sie mehreren Variablen in kompakter Schreibweise unterschiedliche Werte zuweisen möchten, können Sie hierfür Listen oder Tupel verwenden. Beispielsweise funktionieren die folgenden Zuweisungen:

```
a, b = 2, 3
[e, f, g] = [7, 8, 9]
e, f, g = [7, 8, 9]     # gleichwertig
```

Diese Art der Zuweisung können Sie auch verwenden, um den Inhalt zweier Variablen zu vertauschen. Bei den meisten anderen Programmiersprachen würden Sie dazu eine dritte, temporäre Variable benötigen.

```
x, y = y, x             # x und y vertauschen
```

Eine spezielle Variante von Listen- und Tupel-Zuweisungen betrifft den Fall, dass es auf der linken Seite weniger Variablen gibt als auf der rechten Seite Listenelemente. In diesem Fall können Sie *einer* Variablen einen Stern voranstellen. Dieser Variablen werden dann alle überschüssigen Elemente als Liste zugewiesen:

Persönliches Exemplar für Thomas Bartholomäus

```
>>> a, *b, c = [1, 2, 3, 4, 5]
>>> a
    1
>>> b
    [2, 3, 4]
>>> c
    5
```

### Datentypen

Python kennt etliche vordefinierte Datentypen (siehe Tabelle 5.6).

| Datentyp | Funktion | Beispiel | veränderlich |
|---|---|---|---|
| `int` | ganze Zahlen | `x = 3` | nein |
| `float` | Fließkommazahlen | `x = 3.0` | nein |
| `bool` | boolesche Werte | `x = bool(1)` | nein |
| `str` | Zeichenketten | `x = 'abc'` | nein |
| `tuple` | Tupel | `x = (1, 2, 3)` | nein |
| `list` | Listen | `x = [1, 2, 3]` | ja |
| `set` | Sets | `x = {1, 2, 3}` | ja |
| `dict` | Dictionaries | `x = {1:'rot', 2:'blau'}` | ja |
| `bytearray` | Byte-Arrays | `x = bytearray(...)` | ja |
| `io.TextIOWrapper` | Dateien | `x = open('readme.txt')` | ja |
| ... | sonstige Klassen | ... | ja |

**Tabelle 5.6** Wichtige Python-Datentypen und -Klassen

Nur in wenigen Fällen kümmert sich Python selbstständig um die Typumwandlung. Wenn Sie beispielsweise eine ganze Zahl mit einer Fließkommazahl multiplizieren, wird die ganze Zahl automatisch in eine Fließkommazahl umgewandelt, sodass danach eine Fließkommamultiplikation möglich ist.

Von solchen Ausnahmen abgesehen, müssen Sie sich selbst um die Typumwandlung kümmern. Dazu verwenden Sie Funktionen, deren Namen mit dem jeweiligen Datentyp übereinstimmen. Um beispielsweise eine Zeichenkette und eine Zahl zu einer neuen, längeren Zeichenkette zusammenzusetzen, verwenden Sie die Funktion `str`:


© Rheinwerk Verlag, Bonn 2024

```
s = 'abc'
x = 3
s = s + str(x)  # Ergebnis 'abc3'
```

In die umgekehrte Richtung wandeln `int` und `float` eine Zeichenkette in eine Zahl um. Beachten Sie, dass dabei der Fehler *invalid literal* auftreten kann, z. B. wenn Sie versuchen, die Zeichenkette `'abc'` in eine Zahl umzuwandeln.

```
n = int('123')
f = float('123.3')
```

### Mutable oder immutable?

Was passiert bei `b=a`, also bei der Zuweisung einer Variablen an eine andere? Die Frage ist nicht so trivial, wie es den Anschein hat. Beginnen wir mit einem Beispiel mit ganzen Zahlen. Im folgenden Code wird zuerst in `a` der Wert 3 gespeichert. Bei der Zuweisung `b=a` wird `a` durch 3 ersetzt. Also wird auch in `b` die Zahl 3 gespeichert. Um es exakter zu formulieren: `a` und `b` sind nun zwei Variablen, die beide auf ein Objekt mit der ganzen Zahl 3 verweisen. Durch `a=4` wird `a` ein neuer Wert zugewiesen. Auf `b` hat dies keinen Einfluss. `a` und `b` sind unabhängig voneinander, `a` enthält nun den Wert 4, `b` den Wert 3.

```
a = 3
b = a          # b
a = 4
print(a, b)   # Ausgabe 4, 3
```

Der Code für das zweite Beispiel sieht ganz ähnlich aus. Allerdings werden hier in `a` und `b` keine einfachen Zahlen gespeichert, sondern Listen. Nach der Zuweisung `b=a` verweisen beide Variablen auf dieselbe Liste. Durch `a[0]=4` wird ein Element der Liste geändert. Wie der `print`-Aufruf beweist, gilt diese Änderung sowohl für `a` als auch für `b`! `a` und `b` sind also nicht wie im vorigen Beispiel unabhängig voneinander!

```
a = [1, 2, 3]
b = a          # b verweist auf dieselbe Liste wie a
a[0] = 4       # ändert das erste Listenelement
print(a, b)   # Ausgabe [4, 2, 3] [4, 2, 3]
```

Warum verhält sich Python bei zwei scheinbar ganz ähnlichen Programmen so unterschiedlich? Der Grund besteht darin, dass Python zwischen veränderlichen und unveränderlichen Datentypen unterscheidet (siehe Tabelle 5.6) – in der Fachsprache zwischen *mutable* und *immutable types*. Zahlen, Zeichenketten und Tupel sind *immutable*, d. h., eine Änderung ist unmöglich. Stattdessen wird jedes Mal, wenn ein Ausdruck neue Daten ergibt, auch ein neues Objekt erzeugt!

Persönliches Exemplar für Thomas Bartholomäus

Wenn Sie zuerst x=10 und dann x=x+1 ausführen, dann erzeugt Python zuerst ein Objekt mit der Zahl 10; x verweist auf dieses Objekt. Die Berechnung x+1 liefert die Zahl 11. Dafür wird noch ein Objekt im Speicher angelegt. Die Variable x wird nun so geändert, dass sie auf das neue Objekt 11 zeigt. Das ist auch im ersten Codebeispiel in der Zeile a=4 passiert: Python hat ein Objekt für die Zahl 4 erzeugt. a verweist auf dieses Objekt. Das hat keinen Einfluss auf b; b verweist weiterhin auf das Objekt für die Zahl 3.

Andere Datentypen sind hingegen *mutable*. Daher ist es möglich, die Elemente einer Liste zu verändern, *ohne* gleich ein neues Objekt zu erzeugen. Die Zuweisung a[0] ändert nicht die Liste als Ganzes, sondern nur ein Element der Liste. Im zweiten Beispiel verweisen a und b weiter auf dasselbe Objekt, dessen *Inhalt* sich geändert hat.

Wie gehen Sie vor, wenn Sie z. B. von einer Liste eine unabhängige Kopie benötigen, sodass zwei anfänglich gleichartige Listen über zwei Variablen unabhängig voneinander verändert werden können? In solchen Fällen verwenden Sie die Methoden copy oder deepcopy aus dem copy-Modul:

```
import copy
a=[1, 2, 3]
b=copy.copy(a)  # b verweist auf eine unabhängige Kopie von a.
a[0] = 4        # ändert das erste Listenelement von a,
                # b bleibt unverändert.
print(a, b)     # Ausgabe [4, 2, 3] [1, 2, 3]
```

Die copy-Methode erstellt eine Kopie des angegebenen Objekts. deepcopy geht noch einen Schritt weiter: Es erstellt auch Kopien aller veränderlichen Objekte, auf die das Ausgangsobjekt verweist. Im obigen Beispiel ist deepcopy überflüssig, weil die Liste nur drei ganze Zahlen enthält, also unveränderliche Objekte. Wenn die Liste aber selbst veränderliche Objekte enthält, dupliziert deepcopy den gesamten Objektbaum.

## 5.10 Operatoren

Python kennt im Wesentlichen dieselben Operatoren wie die meisten anderen Programmiersprachen. Auf einige Besonderheiten möchte ich aber hinweisen:

- **Division:** Der Operator / führt immer eine Fließkommadivision durch, auch wenn beide Operanden ganze Zahlen sind. Um eine ganzzahlige Division durchzuführen, müssen Sie den Operator // verwenden.
- **Zuweisung und Berechnung verbinden:** Zuweisungen können mit Grundrechenarten verbunden werden. Das heißt, a=a+1 kann auch in der Form a+=1 formuliert werden. Diese Kurzschreibweise ist nicht nur für die Grundrechenarten zulässig, sondern für fast alle Python-Operatoren. Im Gegensatz zu anderen Programmiersprachen sind a++ und a-- im Sinne von a=a+1 und a=a-1 aber *nicht* zulässig.


© Rheinwerk Verlag, Bonn 2024

- **Mehrfachvergleiche:** Bei den Vergleichsoperatoren `<`, `>` etc. sind auch Mehrfachvergleiche möglich. Beispielsweise testet `10 <= x <= 20`, ob `x` einen Wert zwischen 10 und 20 hat. Intern werden alle Vergleiche mit einem logischen Und verknüpft, d. h., `10 <= x <= 20` entspricht `10<=x and x<=20`.
- **Inhalte vergleichen:** `==` testet, ob zwei Ausdrücke denselben Inhalt haben, also beispielsweise, ob die Variable `x` den Wert 3 hat (`if x==3: ...`) oder ob die Zeichenkette `s` mit `'abc'` übereinstimmt (`if s=='abc': ...`).
- **Objekte vergleichen:** Im Gegensatz dazu überprüft der Operator `a is b`, ob die Variablen `a` und `b` auf das gleiche Objekt verweisen. Selbst wenn `a==b` gilt, kann es durchaus sein, dass `a` und `b` auf unterschiedliche Objekte mit demselben Inhalt verweisen. `a is b` entspricht also einer tiefergehenden Gleichheit als `a==b`.

| Operator | Funktion |
|---|---|
| `+ -` | Vorzeichen |
| `+ - * /` | Grundrechenarten |
| `//` | ganzzahlige Division (`20 // 6` ergibt 3.) |
| `%` | Rest der ganzzahligen Division (`20 % 6` ergibt 2.) |
| `**` | Exponentialfunktion bzw. Hochstellen (`2**8` ergibt 256.) |
| `+ *` | Zeichenketten verbinden bzw. vervielfachen (`'ab'*2` ergibt `'abab'`.) |
| `%` | Zeichenkette formatieren (`printf`-Syntax) |
| `=` | Zuweisung (`var = 3`) |
| `:=` | Zuweisung und Auswertung (`if x:=func()== value ...`) |
| `+=` | Zuweisung und Addition (`var+=3` entspricht `var = var + 3`.) |
| `-=` | Zuweisung und Subtraktion |
| `*=` | Zuweisung und Multiplikation |
| `/=` | Zuweisung und Division |
| `==` | Gleichheit testen (`if a==3: ...`) |
| `!=` | Ungleichheit testen |
| `< > <= >=` | kleiner, größer, kleiner/gleich, größer/gleich |
| `is` | testen, ob zwei Variablen auf dasselbe Objekt zeigen |
| `is not` | testen, ob zwei Variablen auf unterschiedliche Objekte zeigen |

**Tabelle 5.7** Rechen-, String- und Vergleichsoperatoren

Persönliches Exemplar für Thomas Bartholomäus

| Operator | Funktion |
|---|---|
| & \| | binäres Und und binäres Oder |
| ^ | binäres Exklusiv-Oder |
| ~ | binäres Nicht |
| << | binär nach links schieben (2<<4 ergibt 32.) |
| >> | binär nach rechts schieben (768>>2 ergibt 192.) |
| or | logisches Oder |
| and | logisches Und |
| not | logisches Nicht |

**Tabelle 5.8** Binäre und logische Operatoren

### Zuweisung und Auswertung kombinieren

Der Operator := steht erst seit Python 3.8 zur Verfügung. Er ermöglicht es, in *einer* Anweisung gleichzeitig eine Variablenzuweisung durchzuführen *und* das Ergebnis mit einem Wert zu vergleichen (*Assignment Expression*). Im folgenden Beispiel in einer while-Schleife ist dieser Vergleich gar nicht explizit sichtbar. Die Schleife wird so lange ausgeführt, wie line nicht leer ist, also das Ende der Datei erreicht wurde.

```
with open('readme.txt') as txtfile:
    # eine Zeile der Textdatei lesen, Schleife fortsetzen,
    # wenn ungleich False
    while line := txtfile.readline():
        print(line, end='')
```

## 5.11 Verzweigungen (if)

Die Syntax von if-Verzweigungen ist einfach. Vergessen Sie die Doppelpunkte nicht, die nach den Bedingungen und nach else zu setzen sind! Im Gegensatz zu anderen Programmiersprachen kennt Python keine switch- oder case-Konstruktionen.

```
if bedingung1:
    block1
elif bedingung2:
    block2
elif bedingung3:
    block3
else:
    block4
```


© Rheinwerk Verlag, Bonn 2024

## Bedingungen

Bedingungen werden in der Regel mit Vergleichsoperatoren gebildet, z. B. `x == 3` oder `a is b`. Mehrere Bedingungen können mit `and` bzw. `or` logisch verknüpft werden (siehe Tabelle 5.8).

```
if x > 0 and (y > 0 or z == 1):
  ...
```

Python optimiert wie nahezu alle anderen Programmiersprachen `and`- und `or`-Ausdrücke (*short-circuit*-Auswertung):

- Wenn bei einer `and`-Verknüpfung der erste Teilausdruck `False` ergibt, wird der zweite nicht mehr ausgewertet, weil das Ergebnis auf jeden Fall `False` lautet.
- Wenn bei einer `or`-Verknüpfung der erste Teilausdruck `True` ergibt, wird der zweite nicht mehr ausgewertet, weil das Ergebnis auf jeden Fall `True` lautet.

Anstelle von `a < x and x < b` ist die Kurzschreibweise `a < x < b` erlaubt. Bedingungen können auch ohne Vergleichsoperator formuliert werden, beispielsweise in dieser Form:

```
if x:
  ...
```

Diese Bedingung ist erfüllt, wenn:

- `x` eine Zahl ungleich 0 ist
- `x` eine nicht leere Zeichenkette ist
- `x` der boolesche Wert 1 (`True`) ist
- `x` eine Liste, ein Tupel oder ein Set mit mindestens einem Element ist
- `x` ein initialisiertes Objekt ist (nicht `None`)

## if-Kurzschreibweise

Mitunter benötigen Sie `if`-Konstruktionen nur, um eine Variable zuzuweisen:

```
if bedingung:
    x = wert1
else
    x = wert2
```

Für derartige Konstruktionen gibt es eine platzsparende Kurzschreibweise:

```
x = wert1 if bedingung else wert2
```

Persönliches Exemplar für Thomas Bartholomäus

## 5.12 Schleifen (for und while)

Schleifen werden in Python zumeist mit `for var in elements` gebildet. Die Schleifenvariable nimmt dabei der Reihe nach jedes der angegebenen Elemente an.

```
for var in mylst:
    anweisungen
```

Die Schleifenvariable `var` steht nach dem Ende der Schleife weiter zur Verfügung und enthält dann den zuletzt zugewiesenen Wert:

```
for i in [7, 12, 3]:
    anweisungen
print(i)     # Ausgabe 3
```

Alternativ können Schleifen mit `while` formuliert werden. Die eingerückten Anweisungen werden dann so lange ausgeführt, wie die Bedingung erfüllt ist.

```
while bedingung:
    anweisungen
```

### break, continue und else

`break` beendet `for`- und `while`-Schleifen vorzeitig:

```
for var in elemente:
    anweisung1
    if bedingung: break        # die Schleife abbrechen
    anweisung2
```

`continue` überspringt die restlichen Anweisungen für den aktuellen Schleifendurchgang, setzt die Schleife aber fort:

```
for var in elemente:
    anweisung1
    if bedingung: continue     # anweisung2 überspringen
    anweisung2
```

Python kennt auch einen `else`-Block für Schleifen und unterscheidet sich damit von den meisten anderen Programmiersprachen. Der `else`-Block wird ausgeführt, nachdem bei einer `for`-Schleife alle Elemente durchlaufen wurden bzw. wenn bei einer `while`-Schleife die Schleifenbedingung nicht mehr erfüllt ist.

```
for var in elemente:
    anweisung1
    anweisung2
else:
    anweisung3
```


© Rheinwerk Verlag, Bonn 2024

### Profi-Tipp: break und continue in verschachtelten Schleifen

Bei verschachtelten Schleifen gelten `break` und `continue` nur für die innerste Schleife. Um unkompliziert eine verschachtelte Schleife abzubrechen, verpacken Sie sie in eine Funktion und verlassen diese mit `return`. Eine andere Möglichkeit besteht darin, sämtliche Schleifen durch `try` abzusichern und zum Abbruch der Schleifen eine Exception auszulösen.

### Schleifen über Zahlenbereiche (range)

Für Schleifen über einen vorgegebenen Zahlenraum werden die Elemente in der Regel durch `range(start, ende)` erzeugt, wobei der Endwert exklusiv ist. Die folgende Schleife durchläuft daher die Werte von 1 bis 9 (nicht 10!). Die Option `end=' '` in `print` bewirkt, dass jeder Ausgabe ein Leerzeichen folgt (kein Zeilenumbruch).

```
for i in range(1, 10):
    print(i, end=' ')
# Ausgabe: 1 2 3 4 5 6 7 8 9
```

Bei `range` kann im optionalen dritten Parameter die Schrittweite angegeben werden:

```
for i in range(0, 20, 3): print(i, end=' ')
# Ausgabe: 0 3 6 9 12 15 18

for i in range(100, 0, -10):  print(i, end=' ')
# Ausgabe: 100 90 80 70 60 50 40 30 20 10
```

`range` kann nur für ganze Zahlen verwendet werden, nicht für Fließkommazahlen. Wenn Sie eine Schleife von 0 bis 1 mit einer Schrittweite von 0,1 bilden möchten, können Sie z. B. so vorgehen:

```
for i in range(0, 11):
    x = i / 10.0
    print(x, end=' ')
# Ausgabe 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Natürlich können Sie Schleifen über Zahlenbereiche auch mit `while` bilden. Das ist insbesondere dann vorteilhaft, wenn die Zahlen unregelmäßige Abstände aufweisen sollen.

```
i = 1
while i < 100000:
    print(i, end=' ')
    i += i*i
# Ausgabe: 1 2 6 42 1806
```

Persönliches Exemplar für Thomas Bartholomäus

### Schleifen über die Zeichen einer Zeichenkette

So können Sie eine Zeichenkette Zeichen für Zeichen verarbeiten:

```
for c in 'abc': print(c)
# Ausgabe: a
#          b
#          c
```

Unter Umständen ist es zweckmäßiger, die Zeichenkette mit `list` in eine Liste umzuwandeln, deren Elemente jeweils ein Zeichen enthalten. Dann können Sie die Elemente mit Listenfunktionen verarbeiten.

### Schleifen über Listen, Tupel und Sets

Die Elemente von Listen, Tupeln und Sets können Sie mühelos mit `for` verarbeiten:

```
for i in (17, 87, 4):
    print(i, end=' ')
# Ausgabe: 17 87 4

for s in ['Python', 'macht', 'Spaß!']:
    print(s)
# Ausgabe: Python
#          macht
#          Spaß!
```

Oft ist das Ziel solcher Schleifen, eine neue Liste, ein neues Tupel oder ein neues Set zu bilden. Dann ist es zumeist eleganter und effizienter, die in Abschnitt 5.7, »Listen«, schon erwähnte *List/Tuple/Set Comprehension* zu verwenden. Dabei wird die `for`-Schleife *in* eckigen, runden oder geschwungenen Klammern ausgeführt und liefert als Ergebnis eine Liste, ein Tupel oder ein Set. Die folgenden Beispiele beziehen sich alle auf Listen.

In der einfachsten Form lautet die Syntax `[ ausdruck for var in liste ]`. Dabei werden alle Listenelemente in die Variable eingesetzt. Die ausgewerteten Ausdrücke ergeben eine neue Liste. Optional kann mit `if` eine Bedingung für die Schleifenvariable angehängt werden: Dann werden nur die Listenelemente berücksichtigt, für die die Bedingung zutrifft.

```
>>> l = [1, 2, 3, 10]
>>> [ x*x for x in l ]              # Quadrat aller Listenelemente
                                    # bilden
    [1, 4, 9, 100]
>>> [ x*x for x in l if x%2==0 ]    # nur gerade Zahlen
                                    # berücksichtigen
    [4, 100]
```


© Rheinwerk Verlag, Bonn 2024

Der Ausdruck darf dabei natürlich selbst eine Liste, ein Set oder ein Tupel sein – dann ist das Ergebnis ein verschachtelter Ausdruck:

```
>>> [ [x, x*x] for x in l ]
    [[1, 1], [2, 4], [3, 9], [10, 100]]
```

Um aus einer Liste, einem Tupel oder einem Set ein Dictionary zu bilden, verwenden Sie die Schreibweise `{ k:v for x in liste }`, wobei `k` und `v` Ausdrücke für den Schlüssel (Key) und den Wert (Value) jedes Dictionary-Elements sind.

```
>>> { x:x*x for x in l }
    {1: 1, 2: 4, 3: 9, 10: 100}
```

### Schleifen über Dictionaries

Die vorhin beschriebene Comprehension-Syntax kann auch für Dictionaries verwendet werden. In der einfachsten Form wird dabei eine Schleife über die Schlüssel der Dictionary-Elemente gebildet. Je nachdem, ob Sie den Ausdruck in eckige oder geschwungene Klammern stellen, ist das Ergebnis eine Liste oder ein Set.

```
>>> d = {'a':12, 'c':78, 'b':3, 'd':43}
>>> { x for x in d }
    {'a', 'c', 'b', 'd'}
>>> [ x for x in d ]
    ['a', 'c', 'b', 'd']
```

Wenn Sie in der Schleife zwei Variablen für das Schlüssel-Wert-Paar benötigen, verwenden Sie die `items`-Methode:

```
>>> { k for k,v in d.items() }
    {'a', 'c', 'b', 'd'}
>>> { v for k,v in d.items() }
    {43, 3, 12, 78}
```

Damit das Ergebnis des Ausdrucks selbst wieder ein Dictionary wird, bilden Sie den Ergebnisausdruck in der Form `schlüssel:wert`:

```
>>> { k:v*2 for k,v in d.items() }
    {'a': 24, 'c': 156, 'b': 6, 'd': 86}
```

### Schleifen über die Parameter eines Scripts

Wenn Sie an ein Python-Script Parameter übergeben, können Sie sie im Script mit der Anweisung `sys.argv` auswerten. `sys` ist ein Modul; es muss vorweg mit `import` gelesen werden. `argv` ist eine Liste, deren erstes Element der Dateiname des Scripts ist. Die weiteren Listenelemente enthalten die übergebenen Parameter. Da der Script-Name selten benötigt wird, wird er mit dem Ausdruck `[1:]` eliminiert.

Persönliches Exemplar für Thomas Bartholomäus

```
# Beispieldatei script-parameters.py
import sys
if(len(sys.argv) <= 1):
    print("Es wurden keine Parameter übergeben.")
else:
    for x in sys.argv[1:]:
        print("Parameter:", x)
```

Ein möglicher Aufruf des Scripts könnte nun so aussehen:

```
./script-parameters.py a b
Parameter: a
Parameter: b
```

Im Terminalfenster läuft normalerweise die Bash. Sie wertet Ausdrücke wie `*.txt` sofort aus und übergibt an das aufgerufene Kommando dann die Liste der gefundenen Dateien. Wenn im aktuellen Verzeichnis also die drei Dateien `readme.txt`, `copyright.txt` und `gpl.txt` gespeichert sind, dann werden mit `./script-parameters.py *.txt` die Namen aller drei Dateien übergeben:

```
./script-parameters.py *.txt
Parameter: copyright.txt
Parameter: gpl.txt
Parameter: readme.txt
```

### Globbing unter Windows

Wenn Sie unter macOS oder Linux `myscript.py *.txt` ausführen, werden die Namen aller Textdateien des aktuellen Verzeichnisses ans Script übergeben und können mit `sys.argv[:1]` ausgewertet werden. Unter Windows funktioniert das nicht. Das Script erhält `*.txt` als einzigen Parameter und muss sich selbst darum kümmern, die Textdateien zu finden. Dabei hilft die Funktion `glob` aus dem gleichnamigen Modul:

```
import glob from glob
filenames = glob('*.txt')
```

Wenn Sie sämtliche Parameter, die an ein Script übergeben werden, auf diese Weise auswerten möchten, formulieren Sie den `glob`-Aufruf so:

```
filenames = []
for arg in sys.argv[1:]:
    filenames.extend(glob(arg))
```

Diese Vorgehensweise funktioniert in allen Betriebssystemen selbst dann, wenn mehrere Dateimuster übergeben werden (z. B. `myscript.py *.jpg *.png`). Ein Anwendungsbeispiel dieser Technik finden Sie in Abschnitt 16.3, »EXIF-Metadaten in SQL-Kommandos umwandeln«.


© Rheinwerk Verlag, Bonn 2024

### Schleifen über die Zeilen einer Textdatei

Es kommt recht oft vor, dass Sie eine Textdatei zeilenweise verarbeiten müssen. Python akzeptiert daher file-Objekte in for-Schleifen und übergibt jeweils eine Zeile an die Schleifenvariable. (Die open-Funktion stelle ich Ihnen in Abschnitt 5.14, »Textdateien verarbeiten«, näher vor.)

Bei der print-Anweisung verhindert end='', dass nach jeder ausgegebenen Zeile eine Leerzeile erscheint. Die in die Schleifenvariable eingesetzten Zeichenketten enthalten bereits die Zeilenumbruchcodes aus der Textdatei.

```
f = open('readme.txt', 'r')
cnt = 0
for line in f:
      cnt+=1
      print("Zeile ", cnt, ": ", line, sep='', end='')
f.close()
```

### Schleifen über alle Dateien eines Verzeichnisses

Die Funktion os.listdir liefert eine ungeordnete Liste aller Dateien und Unterverzeichnisse in einem Verzeichnis. Mit for können Sie unkompliziert eine Schleife darüber bilden.

Bei der Verarbeitung der Dateien müssen Sie aber beachten, dass listdir die Dateinamen ohne Pfadinformationen liefert, also z. B. readme.txt, nicht /home/kofler/readme.txt. Zur Weiterverarbeitung müssen Sie deswegen zumeist mit os.path.join den vollständigen Dateinamen bilden. Das folgenden Listing gibt hierfür ein Beispiel:

```
# Beispieldatei list-directory.py
import os
startdir = os.path.expanduser('~')
print('All files and directories in', startdir)

for filename in os.listdir(startdir):
    fullname = os.path.join(startdir, filename)

    if os.path.isfile(fullname):
        print("File: ", fullname)
    elif os.path.isdir(fullname):
        print('Directory: ', fullname)
```

Im obigen Beispiel liefert os.path.expanduser('~') den vollständigen Pfad des Heimatverzeichnisses. Wenn Sie stattdessen nur an *.pdf-Dateien in diesem Verzeichnis interessiert sind, können Sie wie folgt vorgehen:

Persönliches Exemplar für Thomas Bartholomäus

```
for filename in os.listdir(startdir):
    if not filename.lower().endswith('.pdf'): continue
    fullname = os.path.join(startdir, filename)
    if os.path.isfile(fullname):
        print('PDF file: ', fullname)
```

## 5.13 Funktionen

Python bietet zur Definition eigener Funktionen zwei Möglichkeiten: Gewöhnliche Funktionen leiten Sie mit dem Schlüsselwort `def` ein. Alternativ können Sie mit minimalem Overhead sogenannte *Lambda-Funktionen* definieren und sofort anwenden. Auf diese Variante gehe ich am Ende dieses Abschnitts ein.

Die Definition von Funktionen beginnt mit dem Schlüsselwort `def`. Dem folgt der Funktionsname, für den dieselben Regeln wie für Variablennamen gelten. Die Parameter müssen in runde Klammern gesetzt werden.

```
def myfunc(para1, para2, para3):
    code
```

Für die Programmierung und Anwendung von Funktionen gelten einige Regeln:

- Funktionen müssen definiert werden, bevor sie verwendet werden können. Deswegen ist es üblich, zuerst alle Funktionen zu definieren und erst im Anschluss daran den restlichen Code anzugeben. Die Codeausführung beginnt somit in der ersten Zeile, die *nicht* zu einer Funktionsdefinition gehört.
- Funktionen können vorzeitig mit `return` verlassen werden. Die Verwendung von `return` ist optional.
- Mit `return` kann die Funktion ein Ergebnis zurückgeben, auch in Form von Listen, Tupeln etc. Auf diese Weise kann eine Funktion ganz einfach *mehrere* Werte zurückgeben.

Im folgenden Miniprogramm werden zuerst zwei Funktionen definiert und dann aufgerufen, um den prinzipiellen Umgang mit Funktionen zu zeigen:

```
def f1(x, y):              # Funktion ohne Ergebnis
  print('Parameter 1:', x)
  print('Parameter 2:', y)

def f2(x, y):              # Funktion mit Ergebnis
  return x + y

# hier beginnt die eigentliche Code-Ausführung
```


© Rheinwerk Verlag, Bonn 2024

```
f1(2, 3)      # gibt die Parameter aus
# Ausgabe: Parameter 1: 2
#          Parameter 2: 3

n = f2(4, 5)
print("Ergebnis:", n)
# Ausgabe: Parameter 1: 4
#          Parameter 2: 5
#          Ergebnis: 9
```

### Lokale und globale Variablen

Funktionen können Variablen lesen (aber nicht verändern), die außerhalb der Funktion definiert sind:

```
def f1():
  print(x)

x = 3
f1()  # Ausgabe 3
```

Umgekehrt gelten Variablen, die in einer Funktion initialisiert werden – die also auf der linken Seite einer Zuweisung stehen –, als *lokal*: Sie können nur innerhalb der Funktion verwendet werden. Diese Regel gilt auch dann, wenn eine Variable in einer Funktion denselben Namen hat wie eine Variable außerhalb der Funktion.

```
def f1():
  y = 5
  print(y)    # Ausgabe 5

f1()
print(y)      # Fehler, y ist nicht definiert!
```

Wenn Sie in einer Funktion eine Variable verändern möchten, die außerhalb der Funktion initialisiert wurde, dann müssen Sie diese Variable in der Funktion als `global` kennzeichnen. Genau genommen sagen Sie damit, dass die Funktion `z` nicht als lokale Variable betrachten soll, sondern als eine Variable aus dem globalen Gültigkeitsbereich (*Global Scope*) des Programms.

```
def f1():
  global z
  z = z + 3

z = 3
f1()
print(z)      # Ausgabe 6
```

Persönliches Exemplar für Thomas Bartholomäus

In der Praxis sollten Sie das Schlüsselwort `global` vermeiden. Es führt zu unübersichtlichem Code. Besser ist es, das Funktionsergebnis mit `return` zurückzugeben und dann zu speichern:

```
def f1(x):
  return x + 3

z = 3
z = f1(z)
print(z)   # Ausgabe 6
```

### Parameter

Parameter dienen dazu, Daten an eine Funktion zu übergeben. Intern verhalten sich die Parameter wie lokale Variablen. Ein Parameter mit dem Namen `x` ist somit vollkommen unabhängig von einer gleichnamigen Variablen, die außerhalb der Funktion definiert ist.

Für die Übergabe von Daten in den Parametern einer Funktion gelten dieselben Regeln wie bei Variablenzuweisungen (siehe Abschnitt 5.9, Stichwort »Mutable oder immutable«). Das bedeutet, dass bei unveränderlichen Datentypen eine Änderung der Daten durch die Funktion ausgeschlossen ist (siehe das vorige Beispiel).

Bei veränderlichen Datentypen sind Änderungen am *Inhalt* eines Objekts, einer Liste, eines Sets etc. möglich. Im folgenden Beispiel wird in der Funktion ein Element zur Liste hinzugefügt.

```
def f1(x):
  x.append(3)

x = [1, 2]
f1(x)
print(x)   # Ausgabe [1, 2, 3]
```

### Optionale Parameter

Mit `para=default` kann für einen Parameter ein Defaultwert definiert werden. Dieser Parameter ist damit gleichzeitig optional. Alle optionalen Parameter müssen am Ende der Parameterliste angegeben werden.

Damit der Aufruf von Funktionen mit vielen optionalen Parametern übersichtlicher ist, können die Funktionsparameter auch in der Schreibweise `name=wert` übergeben werden. Der Funktionsaufruf mit benannten Parametern hat zudem den Vorteil, dass Sie sich nicht an die Reihenfolge der Parameter halten müssen und der Code unter Umständen besser lesbar wird.


© Rheinwerk Verlag, Bonn 2024

```
def f(a, b, c=-1, d=0):
  print(a, b, c, d)

f(6, 7, 8, 9)      # Ausgabe 6 7  8 9
f(6, 7, 8)         # Ausgabe 6 7  8 0
f(a=6, b=7, d=9)   # Ausgabe 6 7 -1 9
f(d=9, b=7, a=6)   # Ausgabe 6 7 -1 9
f(6, 7)            # Ausgabe 6 7 -1 0
f(6, 7, d=3)       # Ausgabe 6 7 -1 3
f(6)               # Fehler, b fehlt
f(b=6, c=7)        # Fehler, a fehlt
```

### Variable Parameteranzahl

Wenn Sie einen Parameter in der Form `*para` oder `**para` definieren, nimmt dieser Parameter beliebig viele Werte entgegen. Bei `*para` stehen diese Parameter anschließend als Tupel zur Verfügung, bei `**para` als Dictionary. `**para`-Argumente *müssen* als benannte Parameter übergeben werden.

```
def f(a, b, *c):
  print(a, b, c)

f(1, 2, 3)            # Ausgabe 1 2 (3)
f(1, 2, 3, 4)         # Ausgabe 1 2 (3, 4)
f(1, 2, 3, 4, 5, 6)   # Ausgabe 1 2 (3, 4, 5, 6)
```

Wenn die Daten, die Sie an eine Funktion übergeben wollen, in einer Liste, einem Tupel oder einer anderen aufzählbaren Datenstruktur vorliegen, ist beim Funktionsaufruf auch die Schreibweise `function(*liste)` erlaubt. Damit werden die Elemente der Liste automatisch auf die Parameter verteilt:

```
l = [1, 2, 3, 4, 5, 6]
f(*l)                 # Ausgabe 1 2 (3, 4, 5, 6)
```

### Lambda-Funktionen

Eigene Funktionen verwenden Sie aus zwei Gründen: einerseits, um komplexen Code in überschaubare Teile zu zerlegen, oder andererseits, weil Sie eine bestimmte Aufgabe an verschiedenen Stellen im Code erledigen möchten und so redundanten Code vermeiden möchten. Mitunter gibt es aber eine dritte Variante: Sie brauchen an *einer* Stelle im Code – z. B. in `map`, `filter` oder zur Formulierung einer Callback-Funktion – eine oft recht einfache Funktion. Der herkömmliche Weg ist dann umständlich: Sie müssen die Funktion zuerst mit `def` definieren, um sie dann nur ein einziges Mal zu verwenden.

Persönliches Exemplar für Thomas Bartholomäus

In solchen Fälle sind *Lambda-Funktionen* eine platzsparende und vor allem übersichtlichere Alternative: Die Funktion wird ad hoc definiert *und* gleichzeitig an dieser Stelle im Code sofort verwendet. Die Funktion bekommt gar keinen Namen, weswegen Lambda-Funktionen auch als *anonyme Funktionen* bezeichnet werden.

Die folgende Zeile zeigt die Syntax zur Definition einer Lambda-Funktion. Dem Schlüsselwort `lambda` folgen zuerst die Parameter der Funktion und nach einem Doppelpunkt der Funktionsausdruck. Eine wesentliche Einschränkung im Vergleich zu anderen Funktionen besteht darin, dass Lambda-Funktionen nur aus einem einzigen Ausdruck bestehen dürfen.

```
lambda var1, var2, var3, ...: ausdruck
```

Das folgende Miniprogramm zeigt den Einsatz von gleich zwei Lambda-Funktionen: Der erste Lambda-Ausdruck erkennt durch 3 teilbare Zahlen und verwendet dieses Kriterium, um Elemente in `lst2` aufzunehmen. Der zweite Lambda-Ausdruck wird auf alle Elemente von `lst2` angewandt, um sie ganzzahlig durch 3 zu dividieren. Die resultierende Liste landet in `lst3`:

```
lst1 = [1, 2, 3, 9, 345, 36, 33]
# lst2 enthält alle durch 3 teilbaren Elemente von lst1
lst2 = list(filter(lambda x: x % 3 == 0, lst1))
print(lst2)    # Ausgabe [3, 9, 345, 36, 33]
# alle Elemente von lst2 durch 3 dividieren
lst3 = list(map(lambda x: x // 3, lst2))
print(lst3)    # Ausgabe [1, 3, 115, 12, 11]
```

## 5.14 Textdateien verarbeiten

Bevor Sie eine Textdatei lesen oder schreiben können, müssen Sie die Datei mit `open` öffnen. Dabei übergeben Sie im ersten Parameter den Dateinamen und im zweiten den Zugriffsmodus (siehe Tabelle 5.9). Mit dem zusätzlichen Buchstaben `b` können Sie auch binäre Dateien verarbeiten, was hier aber nicht behandelt wird.

| Modus | Bedeutung |
|---|---|
| `'r'` | Datei lesen (gilt standardmäßig) |
| `'w'` | Datei schreiben; existierende Dateien werden überschrieben! |
| `'a'` | an das Ende einer vorhandenen Datei schreiben (*append*) |
| `'r+'` | Datei lesen und schreiben |

**Tabelle 5.9** Zugriffsmodi der open-Methode


© Rheinwerk Verlag, Bonn 2024

open liefert ein file-Objekt, das nun mit diversen Methoden bearbeitet werden kann (siehe Tabelle 5.10). close() beendet den Zugriff auf die Datei und gibt sie wieder für andere Programme frei. In kurzen Scripts können Sie auf close() verzichten – zum Programmende werden in jedem Fall alle geöffneten Dateien geschlossen.

| Methode | Bedeutung |
|---|---|
| s = f.read() | liest die gesamte Datei |
| s = f.read(n) | liest n Bytes und liefert sie als Zeichenkette |
| s = f.readline() | liest eine Zeile der Datei |
| f.write(s) | schreibt die Zeichenkette s in die Datei |
| n = f.tell() | gibt die aktuelle Lese-/Schreibposition an |
| f.seek(n, offset) | verändert die Lese-/Schreibposition |
| close() | schließt die Datei |

**Tabelle 5.10** Wichtige Methoden für file-Objekte

readline() liefert immer auch das Zeilenendezeichen jeder Zeile mit, unter macOS und Linux also \n. Oft ist das nicht erwünscht. Bei Bedarf eliminiert line.rstrip() den Whitespace (also Leer-, Tabulator und Zeilenumbruchzeilen) am Ende der Zeile.

Wenn das Ende der Datei erreicht ist, gibt readline() eine leere Zeichenkette zurück. Da leere Zeilen *innerhalb* der Datei zumindest aus \n bestehen, gibt es hier keine Doppeldeutigkeiten.

Sofern die Struktur Ihres Codes es zulässt, sollten Sie den gesamten Code zur Bearbeitung einer Datei mit with open() as f formulieren. Damit stellen Sie sicher, dass die Datei beim Auftreten eines Fehlers oder beim Verlassen der aktuellen Code-Blocks (z. B. einer Funktion) automatisch geschlossen wird. Sie brauchen sich also keine Gedanken über close zu machen. (with funktioniert auch für andere Python-Objekte, die nach ihrer Verwendung geschlossen werden sollen, z. B. für Netzwerk- und Datenbankverbindungen.)

```
# Textdatei zeilenweise lesen und ausgeben
with open('readme.txt') as f:
    for line in f:
        print(line, end='')
```

Wenn Sie Textdateien schreiben, vergessen Sie nicht, \n in die Zeichenketten einzubauen, die Sie ausgeben möchten. write() kümmert sich nicht um Zeilenenden! Beachten Sie auch, dass write im Gegensatz zu print nur einen Parameter erwartet und dass dieser Parameter wirklich eine Zeichenkette sein muss. Wenn Sie Zahlen

Persönliches Exemplar für Thomas Bartholomäus

oder andere Daten in der Textdatei speichern möchten, müssen Sie sie zuerst in Zeichenketten umwandeln.

```
# schreibt 'Zeile 1' bis 'Zeile 10' in eine Textdatei
with open('result.txt', 'w') as out:
    for i in range(1, 11):
        out.write('Zeile %d\n' % i)
```

### Beispiel: CSV-Datei auswerten

Als Ausgangspunkt für das folgende Beispiel dient die Datei `employees.csv` mit fiktiven Firmendaten:

```
emp_no,first_name,last_name,birth_date,salary,hire_date
87461,Moni,Decaestecker,1952-02-01,67914,1989-05-31
237571,Ronghao,Schaad,1952-02-01,59355,1994-06-14
406121,Supot,Remmele,1952-02-01,76470,1989-01-27
91374,Eishiro,Kuzuoka,1952-02-01,42250,1999-07-08
...
```

Das erste Ziel des Scripts ist es, Vorname, Nachname und Geburtsdatum ordentlich formatiert am Bildschirm anzuzeigen:

```
first_name          last_name           birth_date
Moni                Decaestecker        1952-02-01
Ronghao             Schaad              1952-02-01
Supot               Remmele             1952-02-01
```

Dazu reichen vier Zeilen Code aus:

```
# Beispieldatei employees.py
with open('employees.csv', 'r') as f:
    for line in f:
        columns = line.split(',')
        print('%-20s %-20s %s' % tuple(columns[1:4]))
```

Die Schleife liest jede Zeile der Datei ein. `split` zerlegt die Zeile in ihre Spalten. Dabei ist zu beachten, dass die letzte Spalte jeweils `\n` enthält, weil die `for`-Schleife die Zeilenumbruchzeichen inkludiert. `columns[1:4]` extrahiert die Spalten 2 bis 4. (Vergessen Sie nicht, die Zählung beginnt bei 0!) `tuple` verwandelt die Liste in einen Tupel, wie der Formatoperator `%` dies erwartet.

### Beispiel: SQL-Kommandos erzeugen

Schon etwas anspruchsvoller ist die zweite Aufgabe: Die Angestellten der Firma sollen in der Tabelle `employees` gespeichert werden. Das Script soll die erforderlichen SQL-


© Rheinwerk Verlag, Bonn 2024

Kommandos erstellen und in einer neuen Datei `employees.sql` speichern, wobei die Spaltennamen der CSV-Datei mit denen der Tabelle übereinstimmen:

```
INSERT INTO employees (emp_no, first_name, last_name,
                       birth_date, salary, hire_date)
VALUES ('87461', 'Moni', 'Decaestecker', '1952-02-01',
        '67914', '1989-05-31');
INSERT INTO ...
```

Der Code ist nicht viel länger, aber doch ein wenig komplexer. `with` öffnet hier gleich zwei Dateien, eine zum Lesen, die andere zum Schreiben. Die erste Zeile enthält die Spaltennamen und könnte unverändert übernommen werden. Nur aus kosmetischen Gründen wird nach jedem Komma ein Leerzeichen eingebaut (`replace`) und außerdem das Zeilenumbruchzeichen am Ende der Zeile entfernt.

In der `for`-Schleife werden dann alle weiteren Zeilen gelesen und mit `split` in Spalten zerlegt. Jedes Element der Liste in `data` wird nun in Anführungszeichen gestellt und dann wieder zur Zeichenkette `values` zusammengesetzt. Die Ausgabe in die SQL-Datei erfolgt mit `write`.

```
# Beispieldatei employees.py
sqlcmd = 'INSERT INTO employees (%s)\nVALUES (%s);\n'
with open('employees.csv', 'r') as csv,\
     open('employees.sql', 'w') as sql:

    # Spaltennamen aus der ersten Zeile lesen
    columnnames = csv.readline().rstrip().replace(',', ', ')

    # Schleife über alle weiteren Zeilen
    for line in csv:
        columns = line.rstrip().split(',')
        # Spalten in Anführungszeichen setzen
        data = ["'" + c + "'" for c in columns]
        # die Liste in data zu Zeichenkette verbinden
        values = ', '.join(data)
        # Ausgabe in der SQL-Datei
        sql.write(sqlcmd % (columnnames, values))
```

## 5.15 Fehlerabsicherung

Bei Fehlern löst Python wie viele moderne Programmiersprachen eine *Exception* aus. Wenn Ihr Programm keine Absicherung gegen derartige Exceptions vorsieht, dann endet es mit einer unschönen Fehlermeldung. Ebendas können Sie vermeiden, wenn Sie Ihren Code mit `try/except` absichern.

Persönliches Exemplar für Thomas Bartholomäus

### try/except

Die Syntax für `try/except` ist einfach:

```
try:
    # fehleranfälliger Code
except someError:
    # Reaktion auf einen bestimmten Fehler
except:
    # Reaktion auf alle anderen Fehler
finally:
    # wird immer ausgeführt
```

Im Anschluss an `try` muss zumindest ein `except`- oder `finally`-Block folgen. Alle anderen Teile der `try`-Konstruktion sind optional. Tritt ein Fehler auf, sucht Python die erste auf den Fehler zutreffende `except`-Anweisung. `except` ohne einen Fehlernamen gilt dabei als Defaultanweisung, die auf jede Art von Fehler zutrifft.

Sofern es einen zutreffenden `except`-Block gibt, wird der dort angegebene Code ausgeführt. Anschließend gilt der Fehler als erledigt. Das Programm wird unterhalb der `try`-Konstruktion fortgesetzt.

Code im `finally`-Block wird *immer* ausgeführt, sogar wenn sich die `try`-Konstruktion in einer Schleife oder Funktion befindet, die Sie vorzeitig mit `break` oder `return` beenden. `finally` ist der richtige Ort für Aufräumarbeiten.

Wenn Sie die `except`-Anweisung in der Form `except xxxError as e` formulieren, enthält `e` ein `Exception`-Objekt. Bei dessen Auswertung ist vor allem `e.args` von Interesse. Damit erhalten Sie ein Tupel mit allen Parametern, die beim Auslösen des Fehlers übergeben wurden. Wenn ein `Exception`-Objekt in eine Zeichenkette umgewandelt wird – wahlweise explizit durch `str(e)` oder implizit in der `print`-Funktion –, dann wird der Inhalt von `args` automatisch in eine Zeichenkette umgewandelt.

```
try:
    n = 1 / 0
except ZeroDivisionError as e:
    print(e)  # Ausgabe: 'division by zero'
```

## 5.16 Systemfunktionen

In diesem Abschnitt stehen einige grundlegende Systemfunktionen im Vordergrund, von denen sich viele im Modul `sys` befinden. Dieses Modul muss also mit `import` eingebunden werden:

```
import sys
```


© Rheinwerk Verlag, Bonn 2024

### Zugriff auf die Programmparameter

`sys.argv` enthält die Liste aller Parameter, die an ein Python-Script übergeben wurden. Dabei müssen Sie beachten, dass das erste Listenelement den Programmnamen enthält, den Sie in der Regel nicht auswerten wollen. Auf die restlichen Elemente greifen Sie dann am einfachsten in der Form `sys.argv[1:]` zu.

### Zugriff auf Standardeingabe und Standardausgabe

`sys.stdin` und `sys.stdout` enthalten `file`-Objekte, die Sie zur Ausgabe von Daten an die Standardausgabe bzw. zum Einlesen von Daten aus der Standardeingabe verwenden können. Dabei stehen Ihnen dieselben Funktionen zur Verfügung wie bei gewöhnlichen Dateien (siehe Abschnitt 5.14, »Textdateien verarbeiten«). Fehler- und Logging-Meldungen senden Sie am besten an `sys.stderr`.

### Programm beenden

Normalerweise endet ein Python-Programm mit der Ausführung der letzten Anweisung oder wenn ein nicht behandelter Fehler auftritt. Wenn Sie ein Programm vorzeitig beenden möchten, führen Sie `sys.exit()` aus. Mit `sys.exit(n)` können Sie dabei auch einen Fehlercode zurückgeben, dessen Bedeutung wie bei Bash-Scripts ist (siehe Tabelle 3.11).

Alternativ können Sie an die `exit`-Methode eine Zeichenkette übergeben, die dann als Fehlermeldung angezeigt wird. Als Fehlercode kommt in diesem Fall automatisch 1 zur Anwendung.

#### exit produziert eine Exception!

Beachten Sie, dass die `exit`-Methode intern eine `SystemExit`-Exception auslöst. Das Programmende kann daher durch `try/except` verhindert werden.

### Linux-Kommandos aufrufen

Mit `subprocess.run` führen Sie aus einem Python-Script heraus ein anderes Programm oder Kommando aus. Im einfachsten Fall übergeben Sie an `run` eine Liste mit dem Kommandonamen und den dazugehörenden Optionen. Die folgenden Zeilen rufen das Kommando `ls` mit der Option `-l` aus, das Details zu den Dateien im aktuellen Verzeichnis anzeigt. `run` liefert als Ergebnis ein Objekt der Klasse `CompletedProcess` zurück (siehe Tabelle 5.11).

```
import subprocess
result = subprocess.run(['ls', '-l'])
```

Persönliches Exemplar für Thomas Bartholomäus

| Eigenschaft | Bedeutung |
|---|---|
| args | das ausgeführte Kommando als Zeichenkette oder Liste |
| returncode | der Rückgabecode (0 = OK, sonst Fehler) |
| stdout | Standardausgabe des Kommandos |
| stderr | Fehlermeldungen des Kommandos |

**Tabelle 5.11** Eigenschaften eines Objekts des Typs »CompletedProcess«

### Ergebnisse verarbeiten

Wenn run wie im vorigen Listing ausgeführt wird, erfolgen die Ausgaben des Kommandos direkt im Terminal, in dem das Python-Script ausgeführt wird. stdout und stderr des Ergebnisobjekts bleiben leer.

Wenn Sie die Ausgaben selbst verarbeiten möchten, übergeben Sie an run den Parameter capture_output=True. Nach der Kommandoausführung liefern die Eigenschaften stdout und stderr die Ausgabe bzw. die Fehlermeldungen des Kommandos als Byte-Strings. Mit decode('utf-8') machen Sie daraus einen gewöhnlichen Unicode-String. Falls Sie die Zeichenkette zeilenweise verarbeiten möchten, verwenden Sie am einfachsten splitline:

```
import subprocess
result   = subprocess.run(["ls", "-l"], capture_output=True)
output   = result.stdout
errormsg = result.stderr
for line in output.decode('utf-8').splitlines():
    print('Ergebnis:', line)
```

### Kommando durch die Shell ausführen

Anstatt ein Kommando direkt durch Python auszuführen, können Sie es auch über die Standard-Shell leiten. Bei vielen Linux-Distributionen ist das die Bash. Dazu übergeben Sie an run den zusätzlichen Parameter shell=True. Das hat zwei Vorteile: Zum einen können Sie nun das oder die auszuführenden Kommandos in einer einfachen Zeichenkette angeben, wobei auch das Pipe-Zeichen | funktioniert, also beispielsweise:

```
import subprocess
result = subprocess.run('dmesg | grep -i eth',
                        capture_output=True, shell=True)
print('Shell:\n', result.stdout.decode('utf-8'))
```



© Rheinwerk Verlag, Bonn 2024

Zum anderen wertet die Shell die Jokerzeichen * und ? aus, womit Sie unkompliziert Dateien verarbeiten können, die einem bestimmten Muster entsprechen:

```
result = subprocess.run('ls -l *.py',
                        capture_output=True, shell=True)
```

Die Verwendung der Shell verursacht allerdings einen höheren Overhead. Wenn es Ihnen darum geht, viele Kommandos möglichst schnell auszuführen, sollten Sie auf `shell=True` nach Möglichkeit verzichten. Auf der folgenden Website finden Sie Tipps, wie gängige Aufgabenstellungen ohne `shell=True` gelingen:

*https://docs.python.org/3/library/subprocess.html*

### Fehler beim Kommandoaufruf

Wie `run` auf Fehler reagiert, hängt davon ab, ob Sie das Kommando direkt oder über die Shell ausführen und welche Art von Fehler auftritt. Wenn Ihnen beispielsweise beim Kommandonamen ein Tippfehler passiert, dann kommt es ohne `shell=True` zu einer `OSError`-Exception. Mit `shell=True` erhalten Sie lediglich einen Rückgabewert ungleich 0.

Empfehlenswert ist der zusätzliche Parameter `check=True`. Damit erreichen Sie, dass bei jedem Fehler eine Exception ausgelöst wird. Der Exception-Typ ist dabei `CalledProcessError`. Diese Fehlerklasse müssen Sie aus dem `subprocess`-Modul importieren.

### Warten (sleep)

`sleep` aus dem Modul `time` wartet die angegebene Zeit in Sekunden ab, ohne dabei die CPU für andere Aufgaben zu blockieren:

```
import time
print("200 ms warten")
time.sleep(0.2)
print("Programmende")
```

## 5.17 Module

Für Einsteigerinnen und Einsteiger wirkt die Sprache Python oft umfangreicher, als sie in Wirklichkeit ist. Tatsächlich ist die Anzahl der unmittelbar in Python implementierten Funktionen überschaubar. Dafür sind alle erdenklichen Zusatzfunktionen in Form von Zusatzmodulen implementiert. Diese Module müssen vor ihrer Verwendung importiert werden.

Persönliches Exemplar für Thomas Bartholomäus

### import

Für das `import`-Kommando gibt es diverse Syntaxvarianten. Die wichtigsten drei sehen so aus:

- `import modulname`: Diese Anweisung liest das Modul. Anschließend können Sie alle darin definierten Funktionen in der Schreibweise `modulname.funktionsname()` nutzen. Mit `import m1, m2, m3` können Sie auch mehrere Module auf einmal importieren.
- `import modulname as m`: Bei dieser Variante können die im Modul definierten Funktionen in der Form `m.funktionsname()` verwendet werden. Bei langen Modulnamen minimiert das den Tippaufwand und macht den Code übersichtlicher.
- `from modulname import f1, f2`: Bei dieser Variante können Sie die Funktionen `f1` und `f2` ohne das Voranstellen des Modulnamens verwenden.

Python-intern bewirkt `import`, dass die Datei `modulname.py` gelesen und ausgeführt wird. Viele Module enthalten einfach die Definition diverser Funktionen; damit sind diese Funktionen Python nun bekannt und können genutzt werden. Module können aber auch Code enthalten, der sofort ausgeführt wird, beispielsweise um Initialisierungsarbeiten durchzuführen.

Es ist üblich, `import`-Anweisungen immer an den Anfang eines Python-Scripts zu setzen. Module können selbst weitere Module importieren. Python merkt sich, welche Module es bereits eingelesen hat, und vermeidet so einen neuerlichen Import bereits aktivierter Module.

Mehr Details zum Umgang mit Modulen können Sie in der Python-Dokumentation nachlesen:

*https://docs.python.org/3/reference/simple_stmts.html#import*

### Eigene Scripts über mehrere Dateien verteilen

Sie können den Modulmechanismus auch dazu verwenden, umfangreiche Scripts über mehrere Dateien zu verteilen. Die betreffenden Dateien importieren Sie einfach mit `import myname` in Ihr Haupt-Script. Damit wird die Datei `myname.py` aus dem lokalen Verzeichnis geladen. Die dort definierten Funktionen stehen jetzt im Haupt-Script zur Verfügung.

Es ist zweckmäßig, in eigenen Moduldateien ausschließlich Funktionen zu definieren (oder Klassen, aber darauf gehe ich in diesem Buch nicht ein). Unmittelbar auszuführender Code sollte sich nur in der Hauptdatei Ihres Scripts befinden.


© Rheinwerk Verlag, Bonn 2024

### Namenskonflikte mit Standardmodulen

Dieser praktische Mechanismus kann zu schwer nachvollziehbaren Fehlern führen. Beispielsweise wollen Sie in Ihrem Script das Python-Modul `csv` nutzen. Sollte sich nun im lokalen Verzeichnis Ihre eigene Datei `csv.py` befinden, dann wird diese anstelle des Python-Moduls geladen.

Vermeiden Sie es also, eigene Script-Dateien gleich zu benennen wie Python-Module! Diese Empfehlung ist leichter ausgesprochen als durchgeführt, weil Sie ja unmöglich die Namen aller Python-Module kennen können. Behalten Sie auf jeden Fall diese Fehlermöglichkeit im Hinterkopf.

## 5.18 Zusatzmodule installieren mit »pip«

Unter Python stehen mehrere Hundert Module standardmäßig zur Auswahl. Diese Module können ohne weitere Vorbereitungsarbeiten mit `import` aktiviert werden. Das ist aber nur der Anfang! Auf der Plattform *https://pypi.org* (*Python Package Index*) stehen außerdem Dateien von über 400.000 Projekten zum Download zur Verfügung.

Zur Installation externer Module sieht Python das Kommando `pip` (Windows, aktuelle Linux-Distributionen) bzw. `pip3` (macOS, ältere Linux-Distributionen) vor. Überzeugen Sie sich von der Existenz dieses Kommandos in einem Terminalfenster!

```
> pip --version
  pip 22.3.1 from C:\Users\kofler\AppData\Local\Programs\Python\
                    Python311\Lib\site-packages\pip (python 3.11)
```

### »pip« installieren

Unter Windows und macOS ist `pip` bzw. `pip3` ein integraler Bestandteil von Python. Bei den meisten Linux-Distributionen befindet sich `pip` dagegen in einem eigenen Paket, das extra installiert werden muss – unter Ubuntu beispielsweise so:

```
$ sudo apt update
$ sudo apt install python3-pip
```

Leider macht `pip` oft auch dann Ärger, wenn es korrekt installiert ist. Das gilt insbesondere unter Windows, wobei zwei Fehlerursachen dominieren:

- `pip` ist installiert, aber nicht im `PATH`: Damit `pip` im `cmd.exe` oder im Terminal ausgeführt werden kann, muss sein Speicherort (z. B. `C:\Users\<name>\AppData\Local\Programs\Python\Python<nnn>\Scripts`) in der Umgebungsvariable `PATH` enthalten sein.

Persönliches Exemplar für Thomas Bartholomäus

Das Python-Setup-Programm sieht eine Option vor, PATH automatisch anzupassen (siehe Abbildung 5.1). Wenn Sie diese Option übersehen haben, müssen Sie die Installation entweder wiederholen oder den Pfad zu pip.exe selbst zu PATH hinzufügen. Suchen Sie im Windows-Menü nach *Systemumgebungsvariablen bearbeiten*!

- Es sind mehrere Python-Versionen parallel installiert: Wenn Python mehrfach installiert ist, funktioniert pip womöglich, installiert die Module aber nicht für die Python-Version, die Sie gerade verwenden.

  Aus meiner Erfahrung besteht die sicherste Lösung darin, zuerst *sämtliche* Python-Versionen zu deinstallieren und auch die Einstellungen für PATH entsprechend aufzuräumen. Danach installieren Sie Python neu, wobei Sie exakt nach der Anleitung in Abschnitt 5.1, »Python installieren«, vorgehen.

### »pip« anwenden

Sind die Installationshürden einmal genommen, ist die Anwendung von pip kinderleicht: Um beispielsweise die matplotlib zu installieren, mit der Sie Diagramme erstellen können, führen Sie im Terminal eines der beiden folgenden Kommandos aus (pip unter Windows und bei neuen Linux-Distributionen, pip3 unter macOS bzw. bei älteren Linux-Distributionen):

```
> pip install --user matploblib
$ pip3 install --user matplotlib

  Collecting matplotlib
  Downloading matplotlib-3.6.3-cp311-cp311-win_amd64.whl
  Downloading contourpy-1.0.7-cp311-cp311-win_amd64.whl
  ...
  Successfully installed contourpy-1.0.7 ... matplotlib-3.6.3
```

Bei manchen Paketen – so wie hier bei der matplotlib – installiert pip nicht nur das eigentliche Paket, sondern einige weitere Pakete mit Funktionen, die das Hauptpaket benötigt.

Unter Linux und macOS führen Sie pip bzw. pip3 ohne root-Rechte und ohne sudo aus. Die Option --user, die zumeist per Default gilt, stellt sicher, dass das betreffende Paket lokal in das jeweilige Benutzerverzeichnis installiert wird (üblicherweise in .local/lib/python<n>/site-packages) und nur diesem Benutzer zur Verfügung steht.

Wenn Sie zu einem späteren Zeitpunkt ein installiertes Modul aktualisieren möchten, führen Sie pip install mit der Option --upgrade aus:

```
> pip install --upgrade <name>
```


© Rheinwerk Verlag, Bonn 2024

Erstaunlicherweise gibt es kein einfaches Kommando, um *alle* Module zu aktualisieren. StackOverflow gibt einige Tipps, wie Sie diese Einschränkung bei Bedarf umgehen können:

*https://stackoverflow.com/questions/2720014*

### requirements.txt

Um zu dokumentieren, welche Module Ihr Script benötigt, können Sie im Projektverzeichnis die Datei `requirements.txt` anlegen. Diese Datei hält fest, welches Modul in welcher Version eingesetzt wird. Die folgenden Zeilen illustrieren die einfache Syntax dieser Datei:

```
beautifulsoup4==4.12.0
requests==2.28.2
requests_html==0.10.0
```

Anstatt die Datei manuell zu warten, können Sie auch das Kommando `pipreqs` einsetzen. Es wertet die `import`-Anweisungen aller Python-Dateien in einem Verzeichnis aus, ermittelt, welche Versionen der Module aktuell installiert sind, und erzeugt dann die entsprechende Datei. `pipreqs` ist selbst ein Modul, das vor der ersten Nutzung mit `pip install pipreqs` installiert werden muss.

```
$ pipreqs code/directory
```

Wenn `requirements.txt` einmal existiert, ist es ganz einfach, alle darin aufgezählten Module zu installieren:

```
$ pip install -r requirements.txt
```

### pipenv und virtualenv

Wenn Sie auf Ihrem Rechner diverse Python-Projekte entwickeln, die unterschiedliche Zusatzmodule benötigen, führt `pip` mit etwas Pech direkt ins Chaos. Rasch wird unklar, welche Module für welches Script erforderlich sind. Das merken Sie spätestens, wenn Sie versuchen, Ihr Script auf einem anderen Rechner auszuführen.

In seltenen Fällen führt die parallele Installation von Modulen bzw. die Durchführung von Modul-Updates für mehrere Projekte sogar zu Konflikten: Ein Script, das zuletzt einwandfrei funktioniert hat, meldet plötzlich seltsame Fehler.

Derartigen Ärger vermeiden Sie, wenn Sie Ihr Projekt in einem eigenen Verzeichnis organisieren und zur Verwaltung der benötigten Module `pipenv` einsetzen. Bevor Sie dieses Tool verwenden können, brauchen Sie absurderweise noch einmal `pip`:

```
$ pip install --user pipenv
```

Persönliches Exemplar für Thomas Bartholomäus

Im Verzeichnis Ihres Projekts verwenden Sie nun `pipenv` anstelle von `pip`, um die erforderlichen Module zu installieren:

```
$ cd my-project
$ pipenv install requests beautifulsoup4

  Creating a virtualenv for this project...
  Pipfile: /home/kofler/my-project/Pipfile
  Creating virtual environment, virtualenv location:
  /home/kofler/.local/share/virtualenvs/py-project-6zwWqRDz
  ...
  To activate this project's virtualenv, run 'pipenv shell'.
  Alternatively, run a command inside the virtualenv with
  'pipenv run'.
```

Die im Beispiel genannten Module `requests` und `beautifulsoup4` stelle ich Ihnen in Kapitel 17, »Web Scraping«, näher vor. Um nun ein Script auszuführen, das diese beiden Module nutzt, müssen Sie `pipenv run` verwenden:

```
$ pipenv run ./myscript.py
```

Alternativ können Sie mit `pipenv shell` eine neue Shell starten. Innerhalb dieser Shell können Sie Ihr Script wie üblich mit `./myscript.py` ausführen. Gleichzeitig können Sie dort Python auch interaktiv starten und alle mit `pipenv` installierten Module nutzen. Wenn Sie die Shell nicht mehr benötigen, verlassen Sie diese mit [Strg]+[D] oder `exit`.

Der größte Vorteil von `pipenv` besteht darin, dass es in Ihrem Projektverzeichnis ein `Pipfile` einrichtet. Dieses fasst zusammen, welche Module Ihr Projekt nutzt:

```
# Datei Pipfile (gekürzt)
...
[packages]
beautifulsoup4 = "*"
requests = "*"

[requires]
python_version = "3.10"
```

Wollen Sie nun Ihr Script auf einen anderen Rechner portieren, kopieren Sie den Code sowie das `Pipfile` dorthin und führen einmal `pipenv install` ohne weitere Parameter aus. Damit werden alle Module installiert. Fertig!

`pipenv` greift auf `virtualenv` zurück, ein Python-Tool, um auf einem Rechner mehrere voneinander getrennte Python-Umgebungen einzurichten. `virtualenv` bietet zwar mehr Funktionen als `pipenv`, ist dafür aber etwas schwieriger anzuwenden (siehe *https://virtualenv.pypa.io*). Für die hier skizzierte Aufgabenstellung reicht `pipenv` meist aus. Weitere Alternativen zur Python-Projektverwaltung sind die in Python


© Rheinwerk Verlag, Bonn 2024

integrierten Funktionen des Moduls `venv` (dabei handelt es sich eine Minimalversion von `virtualenv`) sowie `pip-tools`.

### Nachteile und Einschränkungen

`pipenv` ist nicht unumstritten. Ein Problem besteht darin, dass Module für jedes Projekt getrennt installiert werden. Bei umfangreichen Modulen wie der Matplotlib oder Pandas kostet das unnötig Platz. Außerdem ist die Script-Ausführung mittels `pipenv run` etwas umständlicher.

Chris Warrick zählt in seinem Blog weitere Mängel auf. Seine Fundamentalkritik ist fundiert, aber nicht mehr in allen Punkten aktuell. Insbesondere die Geschwindigkeitsprobleme von `pipenv` sind mittlerweile behoben.

*https://chriswarrick.com/blog/2018/07/17/pipenv-promises-a-lot-delivers-very-little*

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

# TEIL II

# Arbeitstechniken und Werkzeuge

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

# Kapitel 6
# Linux-Toolbox

Bevor Sie eigene Bash-Scripts mit der Bash entwickeln, benötigen Sie einen »Grundwortschatz« elementarer Kommandos. Damit erzeugen Sie Dateien, kopieren oder löschen Verzeichnisse, erstellen komprimierte Archive, installieren Zusatzkommandos, wechseln in den root-Modus usw. Diese Kommandos können Sie sowohl interaktiv in einem Terminal als auch automatisiert in Scripts verwenden.

Dieses Kapitel ist eine Art Crashkurs für Linux-Neulinge. Wenn Sie schon häufig im Linux-Terminal gearbeitet haben, können Sie dieses Kapitel getrost überspringen.

Der Großteil dieses Kapitels gilt auch für macOS. Zwar basiert macOS nicht auf Linux, sondern auf BSD (einem anderen Unix-Derivat), aber grundlegende Kommandos wie `cp` oder `mkdir` funktionieren gleich wie unter Linux. Vereinzelt gibt es leider funktionelle Abweichungen bei einzelnen Optionen. Das kann zu Kompatibilitätsproblemen führen, wenn Ihre Scripts auf beiden Plattformen laufen sollen.

Viele der hier vorgestellten Kommandos funktionieren sogar unter Windows! Das setzt aber voraus, dass Sie in der Git Bash arbeiten. Diese Shell wird normalerweise zusammen mit Git installiert – siehe auch Kapitel 14, »Git und GitHub«.

### Voraussetzungen für dieses Kapitel

Linux-Kommandos können Sie auch ohne Bash-Vorwissen anwenden. Aber bevor Sie diese Kommandos in eigenen Scripts aufrufen können, brauchen Sie Grundkenntnisse zur Bash-Programmierung (siehe Kapitel 3).

### Kurz ein bisschen Werbung

Eine auch nur annähernd vollständige Referenz aller wichtigen Linux-Kommandos ist an dieser Stelle unmöglich. Ich möchte aber nicht verheimlichen, dass ich – ebenfalls im Rheinwerk Verlag – die »Linux-Kommandoreferenz« veröffentlicht habe. Darin ist die Funktionsweise von ca. 500 Linux-Kommandos zusammengefasst. Weil sich kein Mensch all die Optionen merken kann (auch ich nicht), ist die Linux-Kommandoreferenz dasjenige meiner Bücher, in dem ich selbst am häufigsten Details nachlese.

Persönliches Exemplar für Thomas Bartholomäus

## 6.1 Verzeichnisse und Dateien

Der Umgang mit Verzeichnissen ist einfach: Mit `cd` wechseln Sie in das gewünschte Verzeichnis. Wenn Sie an `cd` keinen Parameter übergeben, gelangen Sie in Ihr Heimatverzeichnis. Mit `cd ..` gelangen Sie in das übergeordnete Verzeichnis (*parent directory*), mit `cd -` in das zuletzt aktive Verzeichnis.

`pwd` zeigt den Pfad des gerade aktuellen Verzeichnisses an. `mkdir` erzeugt ein neues Verzeichnis. Besonders praktisch ist die Option `-p`, mit der Sie eine ganze Kette von Verzeichnissen anlegen können, z. B. `mkdir -p dir/subdir/subsubdir`. Die Option vermeidet außerdem Fehlermeldungen, wenn Verzeichnisse bereits existieren.

`rmdir` löscht ein Verzeichnis – aber nur, wenn es leer ist. Verzeichnisse samt Inhalt können Sie mit `rm -r` löschen – aber passen Sie auf! Dieses Kommando arbeitet rekursiv und löscht auch alle Unterverzeichnisse. Generell kann das Löschen von Dateien und Verzeichnissen nicht rückgängig gemacht werden!

```
$ cd                    # in das Heimatverzeichnis wechseln
$ pwd
/home/kofler
$ mkdir subdir          # neues Verzeichnis erzeugen
$ cd subdir             # dorthin wechseln
$ cd ..                 # zurück ins Heimatverzeichnis
$ rmdir subdir          # subdir wieder löschen
```

Linux-Einsteiger müssen erst lernen, sich im verzweigten Verzeichnisbaum zu orientieren (siehe Tabelle 6.1). Beachten Sie, dass Sie zur Veränderung von Systemdateien (z. B. im Verzeichnis `/etc`) root-Rechte benötigen – siehe Abschnitt 6.4!

| Verzeichnis | Inhalt |
|---|---|
| `/boot` | Systemdateien für den Boot-Prozess |
| `/dev` | Device-Dateien für Hardware-Komponenten |
| `/etc` | Systemeinstellungen |
| `/home` | Benutzerverzeichnisse |
| `/mnt` | externe Dateisysteme und Datenträger |
| `/proc` | Prozessinformationen (virtuelles Dateisystem) |
| `/usr` | Programme und Bibliotheken |
| `/tmp` | temporäre Dateien (gehen bei Reboot verloren) |
| `/var` | variable Daten (Webdateien, Datenbanken, Mails) |

**Tabelle 6.1** Wichtige Linux-Verzeichnisse


© Rheinwerk Verlag, Bonn 2024

### Dateien auflisten, kopieren, verschieben und löschen

`ls` listet Dateien auf. Mit der Option `-l` zeigt das Kommando außer dem Dateinamen eine Menge Zusatzinformationen an, z. B. die Größe in Byte und den Zeitpunkt der letzten Änderung. Mit der Option `-a` berücksichtigt `ls` auch sogenannte »verborgene« Verzeichnisse und Dateien, deren Name einfach mit einem Punkt beginnt (z. B. `.ssh` mit den Konfigurationsdateien des `ssh`-Kommandos).

`touch` erzeugt eine neue leere Datei. Sollte die Datei schon existieren, wird nur der Zeitpunkt der letzten Änderung aktualisiert.

Das Kommando `cp old new` kopiert eine Datei. `cp file1 file2 file3 dir` kopiert mehrere Dateien in ein anderes Verzeichnis. In der Form `cp -r olddir newdir` kopiert das Kommando einen ganzen Verzeichnisbaum (`r` wie *recursive*). `cp -a olddir newdir` erfüllt den gleichen Zweck, belässt aber die Zugriffsrechte und den Zeitpunkt der letzten Änderung unverändert.

`mv` funktioniert ähnlich wie `cp`: `mv old new` benennt die Datei um. `mv file1 file2 dir` verschiebt die Dateien an einen neuen Ort.

`rm` löscht die angegebenen Dateien unwiderruflich. Mit der bereits erwähnten Option `-r` arbeitet das Kommando rekursiv, löscht also auch Unterverzeichnisse mit deren Inhalt. Vorsicht!

```
$ mkdir tst                 # erzeugt ein neues Verzeichnis
$ cd tst                    # wechselt dorthin
$ date > now.txt            # speichert Datum und Uhrzeit in now.txt
$ cp now.txt copy.txt       # erstellt eine Kopie von now.txt
$ mv copy.txt backup.txt    # gibt der Kopie einen anderen Namen
$ ls *.txt                  # listet alle *.txt-Dateien auf
  backup.txt now.txt
$ rm *.txt                  # löscht alle *.txt-Dateien
```

## 6.2 Dateien suchen

Mit `find` durchsuchen Sie ein Verzeichnis samt allen Unterverzeichnissen nach Dateien, die bestimmte Merkmale aufweisen. Mögliche Suchkriterien sind z. B. der Name, die Dateigröße oder das Datum der letzten Änderung. `find` berücksichtigt aber nie den Inhalt der Datei!

An das Kommando muss das Ausgangsverzeichnis für die Suche übergeben werden (oder `.` für das aktuelle Verzeichnis). Die Suchkriterien werden mit zahllosen Optionen formuliert, wobei sämtliche Kriterien gleichzeitig zutreffen müssen (logische Und-Verknüpfung). Die folgenden Beispiele deuten die Möglichkeiten von `find` nur an. Viel mehr Details können Sie mit `man find` nachlesen. Einigen `find`-Kommandos ist `sudo` vorangestellt. Damit wird die Suche im root-Modus ausgeführt

Persönliches Exemplar für Thomas Bartholomäus

(siehe Abschnitt 6.4, »Arbeiten mit root-Rechten«). Das ist notwendig, weil Sie als gewöhnlicher Benutzer viele Dateien in den Verzeichnissen /var und /etc nicht einmal lesen dürfen.

```
# Dateien in /var/log, die größer als 250 kByte sind
$ sudo find /var/log -size +250k

# Dateien in /etc, die in den letzten 28 Tagen verändert wurden
# mtime = modify time
$ sudo find /etc -mtime -28

# alle Unterverzeichnisse in /etc (-type d = directory)
$ sudo find /etc -type f

# PDF-Dateien in Downloads, die größer als 1 MByte sind
# und seit mindestens 60 Tagen nicht mehr gelesen wurden
# (atime = access time)
$ find Downloads -name '*.pdf' -size +1M -atime +60
```

### Textsuche mit grep

Das Kommando `grep pattern file.txt` filtert aus einer Textdatei alle Zeilen heraus, die dem Suchmuster `pattern` entsprechen. Im folgenden Beispiel sollen aus einer Logging-Datei des Webservers Apache alle Zeilen herausgefiltert werden, die eine bestimmte IP-Adresse enthalten. Weil das Suchmuster als regulärer Ausdruck verarbeitet wird (siehe Kapitel 9, »Reguläre Ausdrücke«), muss das Zeichen ».« durch einen Backslash gekennzeichnet werden. (In regulären Ausdrücken ist ».« ein Platzhalter für ein beliebiges Zeichen.)

```
$ sudo grep '175\.55\.93\.123' /var/log/apache2/access.log
```

#### Textauswertung und -analyse

In seiner Funktion als Filterkommando gehe ich in diesem Buch noch viel ausführlicher auf `grep` ein (siehe Kapitel 8, »Textauswertung mit Filtern und Pipes«). In diesem Kapitel stelle ich Ihnen weitere Kommandos zur Textauswertung vor, unter anderem `wc` (*word count*), `head` und `tail` (die ersten/letzten Zeilen eines Texts anzeigen), `sort` (Text sortieren) und `uniq` (Doppelgänger entfernen).

Mit ein paar Optionen wird aus dem Filterkommando `grep` ein mächtiges Suchwerkzeug. Das folgende Kommando durchsucht das Verzeichnis /etc rekursiv (`-r`) nach Dateien, die in beliebiger Groß- und Kleinschreibung (`-i`) den Suchbegriff `password` enthalten. Die resultierenden Dateien werden aufgelistet (`-l`):

```
$ sudo grep -r -i -l password /etc
```


© Rheinwerk Verlag, Bonn 2024

## 6.3 Dateien komprimieren und archivieren

Um eine einzelne Datei zu komprimieren, verwenden Sie am einfachsten `gzip`. Das Kommando nennt die Datei gleich um, d. h. aus `name` wird `name.gz`. Um die Datei wieder auszupacken, verwenden Sie `gunzip`. Dabei wird die Endung `.gz` wieder entfernt.

```
$ gzip somefile        # komprimiert die Datei -> somefile.gz
$ gunzip somefile.gz   # dekomprimiert die Datei -> somefile
```

Zu den Standardkommandos `gzip` und `gunzip` gibt es eine ganze Reihe von Alternativen. `xz` komprimiert deutlich besser, erfordert aber auch viel mehr CPU-Leistung. Umgekehrt arbeitet `lz4` viel effizienter als `gzip`, liefert dafür aber nicht so kleine Dateien.

### Dateien archivieren mit tar

Das Kommando `tar` steht eigentlich für *tape archive*, stammt also aus einer Zeit, als Bandlaufwerke noch allgegenwärtig waren. Heute wird `tar` meist zum Erstellen komprimierter Dateiarchive verwendet. `tar`-Dateien spielen unter Linux eine ähnlich wichtige Rolle wie `zip`-Dateien unter Windows.

Bei `tar` legt die erste Option (`-c`, `-r`, `-x` oder `-t`) die Aktion fest, die `tar` durchführen soll. Alle weiteren Optionen beeinflussen die Funktionsweise (siehe Tabelle 6.2).

| Option | Bedeutung |
|---|---|
| `-c` | neues Archiv erzeugen (*create*) |
| `-r` | vorhandenes Archiv erweitern (*replace*) |
| `-x` | Archiv auspacken (*extract*) |
| `-t` | Inhalt auflisten (*list*) |
| `-v` | visuelles Feedback anzeigen (*verbose*) |
| `-z` | mit `gzip` komprimieren/dekomprimieren |
| `-j` | mit `bzip2` komprimieren/dekomprimieren |
| `-J` | mit `xz` komprimieren/dekomprimieren |
| `-f file.tar` | Dateiname des Archivs (*file*) |
| `-f -` | Standardeingabe/Standardausgabe verwenden |
| `-C directory` | in diesem Verzeichnis arbeiten |

**Tabelle 6.2** tar-Optionen

Persönliches Exemplar für Thomas Bartholomäus

Besonders wichtig ist `-f filename`: Sie legt den Dateinamen der Archivdatei fest. Sollten Sie diese Option vergessen, versucht `tar`, auf ein Bandlaufwerk zuzugreifen. Da dieses auf den meisten Rechnern nicht existiert, kommt es zu einem Fehler.

Mit `-f -` schreibt `tar` das Archiv in die Standardausgabe bzw. liest es von dort. Das ist zweckmäßig, wenn Sie `tar` mit einem weiteren Kommando verbinden möchten, z. B. um ein Archiv mit `tar cf - | gpg ...` gleich zu verschlüsseln.

Es ist üblich, die an `tar` übergebenen Optionen zu einem Buchstabenblock ohne Minuszeichen zusammenzufassen. Aus `-c -z -f` oder der Kurzschreibweise `-czf` wird einfach `czf`.

Das erste der folgenden Kommandos erzeugt die komprimierten Datei `backup.tar.gz`, die sämtliche Dateien innerhalb des `images`-Verzeichnisses enthält. (`tar` arbeitet standardmäßig rekursiv, berücksichtigt also auch Unterverzeichnisse.)

Die weiteren Kommandos zeigen das Inhaltsverzeichnis des Archivs an und packen es schließlich wieder aus. Statt `.tar.gz` ist auch die verkürzte Kennung `.tgz` gebräuchlich.

```
$ tar czf backup.tar.gz images    # Archiv erzeugen

$ tar tzf backup.tar.gz           # Inhalt anzeigen

$ cd other-directory              # Archiv auspacken
$ tar xzf backup.tar.gz
```

**Backups**

Weitere Anwendungsbeispiele zu `tar` folgen in Kapitel 15, »Backups«, in dem ich auf verschiedene Techniken zur Automatisierung von Backups eingehe.

### ZIP-Dateien

Vielleicht müssen Ihre Archive kompatibel zur Windows-Welt sein, oder Sie möchten eine ZIP-Datei auspacken. In solchen Situationen setzen Sie die Kommandos `zip`, `zipinfo` und `unzip` ein, deren Syntax ich hier nur anhand von drei Beispielen erläutere.

Die Beispiele beziehen sich wieder auf das Verzeichnis `images`, das gesichert werden soll. Beachten Sie, dass `zip` standardmäßig nicht rekursiv arbeitet. Abhilfe schafft bei Bedarf die Option `-r`.

```
$ zip backup.zip -r images        # Archiv erzeugen
$ zipinfo backup.zip              # Archiv anzeigen
$ unzip backup.zip                # Archiv auspacken
```


© Rheinwerk Verlag, Bonn 2024

## 6.4 Arbeiten mit root-Rechten

Normalerweise können Sie in Linux und macOS nur Ihre eigenen Dateien verändern und nur ausgewählte Dateien anderer Benutzer lesen. Für das alltägliche Arbeiten reicht das aus – nicht aber für Scripts, die administrative Aufgaben erledigen sollen.

Abhilfe schafft in solchen Fällen sudo. Damit können Sie ein einzelnes Kommando oder auch ein ganzes Script mit Administratorrechten (oder, wie es unter Linux heißt, mit »root-Rechten«) ausführen. Das erste der beiden folgenden Kommandos führt eine Software-Installation durch (siehe Abschnitt 6.6), das zweite führt ein Script im lokalen Verzeichnis aus:

```
$ sudo apt install somepackage
  [sudo] Password for <accountname>: ********
$ sudo ./myscript.sh
```

### sudo-Rechte nur für ausgewählte Accounts

sudo darf nur von ausgewählten Benutzerinnen und Benutzern verwendet werden. Normalerweise handelt es sich dabei um diejenige Person, die die Linux-Installation durchgeführt und deren Account zuerst eingerichtet wurde.

sudo-Benutzer müssen sich mit ihrem eigenen Passwort authentifizieren. Die Authentifizierung bleibt dann für einige Minuten gültig. Bei weiteren sudo-Aufrufen innerhalb dieser Zeitspanne ist deswegen keine neuerliche Passworteingabe notwendig.

Wer in sudo welche Rechte hat, wird durch die Datei /etc/sudoers gesteuert. Änderungen an dieser relativ komplexen Datei sind zum Glück nur selten notwendig. Wenn Sie einem weiteren Account sudo-Rechte zuweisen möchten, führen Sie das folgende Kommando aus (das natürlich voraussetzt, dass Sie selbst sudo-Rechte haben):

```
$ sudo usermod -a -G sudo <account>     # Debian, Ubuntu ...
$ sudo usermod -a -G wheel <account>    # Fedora, RHEL ...
```

Das obige usermod-Kommando fügt den Benutzer <account> der Gruppe sudo (für Debian, Ubuntu und dazu kompatible Distributionen) bzw. der Gruppe wheel hinzu (für Fedora und RHEL-kompatible Distributionen). Alle Mitglieder dieser beiden Gruppen haben automatisch sudo-Rechte.

### Probleme bei der Ausgabeumleitung

Die folgenden beiden Kommandos versuchen jeweils eine Datei im /etc-Verzeichnis zu verändern. Das erste Kommando funktioniert, das zweite führt zu einem Fehler. Warum?

```
$ sudo touch /etc/new-file              # funktioniert
$ sudo echo "hello" > /etc/new-file     # Fehler: keine Berechtigung
```

Persönliches Exemplar für Thomas Bartholomäus

Das Problem ist die Ausgabeumleitung. Beim zweiten Beispiel führt `sudo` zwar `echo "hello"` mit `root`-Rechten aus. Um den Rest des Kommandos, also um die Umleitung der Ausgabe in die Datei `/etc/new-file`, kümmert sich aber die Bash – und die läuft mit normalen Rechten. Dabei tritt der Fehler ein.

Sobald die Ursache einmal klar ist (ich stolpere selbst in regelmäßigen Abständen in diese Falle), ist die Lösung nicht schwer. Anstatt nur `echo` (oder welches Kommando auch immer) per `sudo` auszuführen, übergeben Sie an `sudo` einen neuen Bash-Prozess, der seinerseits das restliche Kommando inklusive der Umleitung ausführt. Die Option `-c` wie *command* ermöglicht es Ihnen, an den Bash-Prozess das auszuführende Kommando zu übergeben:

```
$ sudo bash -c 'echo "bla" > /etc/new-file'
```

## Zugriffsrechte

Wer welche Dateien unter Linux oder macOS lesen, schreiben und ausführen darf, wird durch ein dreistufiges System gesteuert. Zusammen mit jeder Datei werden neun Zugriffsbits gespeichert, `r` (*read*), `w` (*write*) und `x` (*execute*) für den Besitzer der Datei, für alle Mitglieder der Gruppe, der diese Datei zugeordnet ist, und schließlich für alle anderen Accounts.

Die Systemdatei `/etc/shadow`, die unter Linux Hashcodes für Passwörter enthält, gehört beispielsweise dem Benutzer `root` und ist der Gruppe `shadow` zugeordnet. `root` darf die Datei lesen und verändern, Mitglieder der `shadow`-Gruppe dürfen sie lesen, alle anderen Benutzer haben keine Rechte. All diese Informationen können Sie mit `ls -l` ermitteln:

```
$ ls -l /etc/shadow

-rw-r-----  root shadow  ... shadow

 ^^^                                Rechte für Besitzer
    ^^^                             Rechte für Gruppenmitglieder
       ^^^                          Rechte für andere Benutzer
            ^^^^                    zugeordneter Besitzer
                 ^^^^^^             zugeordnete Gruppe
                         ^^^        Zeit (nicht dargestellt)
                             ^^^^^^ Dateiname
```

Mit den Kommandos `chown` (*change owner*), `chgrp` (*change group*) und `chmod` (*change mode*) können Sie die Zuordnung einer Datei zu ihrem Besitzer, der zugeordneten Gruppe sowie die Zugriffsrechte verändern. All diese Daten können auch für Verzeichnisse geändert werden, wobei das Execute-Bit `x` bei Verzeichnissen eine andere Bedeutung hat: Es erlaubt die Benutzung dieses Verzeichnisses (also `cd verzeichnis`).


© Rheinwerk Verlag, Bonn 2024

Die folgenden drei Zeilen zeigen einfache Beispiele für die Anwendung von chown, chgrp und chmod: Das erste Kommando macht den Account peter zum Besitzer der Datei /var/www/html/index.html. Das zweite Kommando weist dieser Datei die Gruppe www-data zu. Das dritte Kommando setzt die Read- und Write-Bits für den Besitzer (u für *user*) und die Gruppe, deaktiviert diese Bits aber für alle anderen (o wie *other*). Damit kann die Datei index.html von peter sowie von allen Mitgliedern der Gruppe www-data gelesen und verändert werden – aber von niemandem sonst. (www-data ist unter Debian und Ubuntu die Gruppe, die dem Webserver zugeordnet ist.)

```
$ sudo chown peter       /var/www/html/index.html
$ sudo chgrp www-data    /var/www/html/index.html
$ sudo chmod ug+rw o-rw  /var/www/html/index.html
```

Die Ausführung aller drei Kommandos erfordert in den meisten Fällen root-Rechte. Ausgenommen sind Änderungen für eigene Dateien, die nur Ihren eigenen Benutzer-Account bzw. Ihnen zugeordnete Gruppen betreffen. Derartige Modifikationen – z. B. chmod +x myscript.sh, um ein Script ausführbar zu machen – gelingen ohne sudo. (Die macOS-Variante von chmod unterscheidet sich minimal von der Linux-Variante. Dort müssen Sie chmod a+x ausführen.)

Es fehlt hier der Platz, um auf noch mehr Details einzugehen, z. B. auf die vielen Syntaxvarianten von chmod oder auf die oktale Darstellung von Zugriffsrechten. Eine ausführlichere Beschreibung der Linux/Unix-Zugriffsrechte finden Sie in jedem guten Linux-Buch sowie auf zahlreichen Webseiten, beispielsweise hier:

*https://wiki.ubuntuusers.de/Rechte*

## 6.5 Prozesse verwalten

Das Kommando ps listet ohne Optionen die aus dem Terminal gestarteten Prozesse auf. Schon erheblich ist das Ergebnis von ps ax: Damit erhalten Sie eine Liste aller auf dem Computer oder in der virtuellen Maschine laufenden Prozesse samt detaillierten Informationen (Prozessnummer, Parameter usw.). Noch informativer ist ps axu: Jetzt verrät das Kommando auch noch, wer welchen Prozess gestartet hat.

Das Ergebnis von ps kann gut mit anderen Kommandos weiterverarbeitet werden. In den folgenden Beispielen zählt wc -l (*word count, lines*) die Ausgabezeilen und damit die aktiven Prozesse. grep filtert die Zeilen aus dem Ergebnis, die den Suchbegriff ssh enthalten.

```
$ ps ax | wc -l      # Anzahl der Prozesse zählen
$ ps ax | grep ssh   # nur SSH-Prozesse anzeigen
```

Persönliches Exemplar für Thomas Bartholomäus

`top` zeigt die laufenden Prozesse geordnet nach ihrem CPU-Bedarf an. Die Anzeige wird alle drei Sekunden erneuert, bis Sie das Kommando mit `Q` beenden.

### top-Alternativen

Zu `top` gibt es einige interessante Alternativen. Während `htop` durch seine übersichtliche Darstellung brilliert, verwenden `iotop`, `iftop` oder `powertop` andere Sortierkriterien – nämlich die I/O-Belastung, die Netzwerkleistung oder den (geschätzten) Stromverbrauch. Diese Kommandos müssen zumeist extra installiert werden.

## Prozesse beenden

Um ein außer Kontrolle geratenes oder in einer Endlosschleife befindliches Script zu beenden, das Sie im Terminal gestartet haben, drücken Sie einfach `Strg`+`C`. Andere Prozesse, die Ihrem Account zugeordnet sind, können Sie mit `kill pid` stoppen. Anstelle von `pid` müssen Sie die Prozessnummer angeben, die Sie wahlweise mit `ps ax` oder mit `top` ermitteln können.

Trotz seines dramatischen Namens versendet `kill` eigentlich nur Signale an andere Prozesse. Ohne weitere Optionen wird das Signal `SIGTERM` gesendet. Das ist quasi eine höfliche Aufforderung, dass der Prozess beendet werden soll. Der Prozess kann dieses Signal aber ignorieren. In solchen Fällen führt meist `kill -9 pid` zum Ziel. Dabei gibt `-9` die Signalnummer an. Das Signal 9 hat den Namen `SIGKILL` und beendet jeden Prozess sofort – es sei denn, es wird gerade eine länger andauernde Systemfunktion ausgeführt. Darauf hat auch `SIGKILL` keinen Einfluss.

Sie werden es sich schon gedacht haben: Normalerweise können Sie mit `kill` nur eigene Prozesse beenden. Fremde Prozesse darf nur `root` stoppen. Dazu kombinieren Sie `kill` mit `sudo`, also z. B. `sudo kill -9 pid`.

Es ist ein wenig lästig, dass Sie vor der Ausführung von `kill` zuerst die Prozessnummer herausfinden müssen. Bequemer ist `killall n`, wobei `n` der Name des zu stoppenden Prozesses ist. Passen Sie aber auf: `sudo killall n` kann weitreichende Folgen haben, weil es *alle* gleichnamigen Prozesse beendet, egal, wer diese gestartet hat.

## Hintergrundprozesse und Systemdienste

Mit dem Hochfahren eines Linux-Systems werden eine Menge Hintergrundprozesse automatisch gestartet. Diese Prozesse sind für diverse Systemaufgaben und Server-Dienste zuständig. Mit dem Kommando `systemctl` können Sie derartige Systemdienste stoppen, neu starten, in Zukunft automatisch starten usw.


© Rheinwerk Verlag, Bonn 2024

```
$ sudo systemctl                    # alle Dienste auflisten
$ sudo systemctl stop name          # Dienst 'name' stoppen
$ sudo systemctl start name         # den Dienst starten
$ sudo systemctl restart name       # den Dienst neu starten
$ sudo systemctl reload name        # Konfigurationsdateien einlesen
$ sudo systemctl enable name        # den Dienst in Zukunft
                                    # automatisch starten
$ sudo systemctl disable name       # den Dienst in Zukunft
                                    # nicht mehr starten
```

`systemctl` ist vor allem dann wichtig, wenn Sie unter Linux einen neuen Server-Dienst installiert haben. Je nach Distribution (Fedora, Red Hat) werden Dienste nach der Installation *nicht* automatisch gestartet. Sie müssen sich zuerst um die Konfiguration kümmern und den Dienst dann mit `systemctl start` manuell starten. Wenn alles funktioniert, aktivieren Sie den Dienst mit `systemctl enable` dauerhaft.

Bei anderen Distributionen werden neu installierte Dienste zwar mit einer Defaultkonfiguration sofort gestartet (Debian, Ubuntu), aber auch dort sind meist Änderungen an der Konfiguration erforderlich. Damit Ihre Änderungen wirksam werden, führen Sie `systemctl reload` aus. (Bei sehr grundlegenden Änderung ist sogar `systemctl restart` erforderlich.)

### Logging-Dateien

Die unter Linux laufenden Dienste protokollieren eine Menge Daten. Bei einigen Programmen werden die Protokolldateien im Verzeichnis `/var/log` als Textdateien gespeichert – z. B. bei den meisten Mail- und Webservern. Das folgende Kommando listet alle Logging-Meldungen des Mail-Servers aus, die eine bestimmte E-Mail-Adresse betreffen:

```
$ sudo grep customer@somecompany.com /var/log/mail.log
```

Kleinere Server-Prozesse verwenden zumeist den zentralen Syslog. Die Nachrichten können Sie mit `journalctl` lesen:

```
$ journalctl                 # alle Syslog-Nachrichten lesen
$ journalctl -u sshd         # nur Meldungen des SSH-Servers lesen
```

### Freien Speicherplatz ermitteln

`free -h` ermittelt den freien Arbeitsspeicher. Die Option `-h` steht dabei für *human readable*. Die Zahlenangaben werden daher mit passenden Einheiten (M für MByte, G für GByte usw.) versehen. Das Ergebnis ist allerdings nicht ganz einfach zu interpretieren. Der freie Speicher ist in der letzten Spalte *available* angegeben und beträgt am Testrechner rund 15 GByte. Laut der Spalte *free* wäre es scheinbar weniger als ein

Persönliches Exemplar für Thomas Bartholomäus

GByte, aber dabei wird eine Menge Pufferspeicher mitberücksichtigt, die Linux jederzeit freigeben kann.

```
$ free -h

        total     used      free    shared  buff/cache  available
  Mem:    30Gi     14Gi     938Mi    478Mi        15Gi       15Gi
  Swap:     0B       0B        0B
```

Den freien Speicher auf der Festplatte oder SSD können Sie mit `df -h` (*disk free*) ergründen. Standardmäßig listet das Kommando alle Dateisysteme von Linux auf, auch diverse interne Dateisysteme. Mit `-x tmpfs` eliminieren Sie zumindest die temporären Dateisysteme und mindern so die Informationsflut ein wenig. Im folgenden Beispiel sind die zweite und dritte Zeile am interessantesten. Im Root-Dateisystem sind noch 159 GByte frei, im Dateisystem `/home` für die Benutzerverzeichnisse noch 195 GByte.

```
$ df -h -x tmpfs

  Filesystem                  Size  Used  Avail  Use% Mounted on
  dev                          16G     0    16G    0% /dev
  /dev/mapper/vgcrypt-root    196G   28G   159G   15% /
  /dev/mapper/vgcrypt-home    590G  366G   195G   66% /home
  /dev/nvme0n1p1              2.0G   80M   1.9G    4% /boot
  /dev/sdb1                   1.7T  1.1T   570G   66% /run/media/
    kofler/p1-backup2
```

Eine großartige Alternative zu `df` ist das Kommando `duf`, das aber extra installiert werden muss. Es liefert zwar prinzipiell dieselben Informationen, formatiert das Ergebnis aber viel übersichtlicher.

### Sonstige Systeminformationen ermitteln

Wollen Sie wissen, welche Distribution in einer virtuellen Maschine oder auf einem Server läuft? Werfen Sie einen Blick in die Datei `/etc/os-release`:

```
$ cat /etc/os-release

  NAME="Arch Linux"
  BUILD_ID=rolling
  ...
```

Die Version des laufenden Kernels ermitteln Sie mit `uname`:

```
$ uname -r

  6.1.1-arch1-1
```


© Rheinwerk Verlag, Bonn 2024

Fehler-, Warn-, Debug- und Statusmeldungen des Kernels können Sie mit `dmesg` lesen. Am besten leiten Sie das Ergebnis von `dmesg` an `less`, damit Sie durch die seitenlangen Informationen scrollen können. Bei den meisten aktuellen Distributionen dürfen die Kernelmeldungen nur mit `root`-Rechten gelesen werden.

```
$ sudo dmesg |  less
```

## 6.6 Software-Installation

Linux-Distributionen enthalten standardmäßig bereits eine gute Grundausstattung von Kommandos. Je nachdem, in welche Richtung sich Ihre Scripts entwickeln, kann es aber sein, dass genau ein Spezialkommando fehlt. Besonders oft tritt dieser Fall auf, wenn Sie – z. B. mit Docker – in Containern arbeiten. Bei Containern ist die Basisausstattung aus Platzgründen auf ein Minimum reduziert.

Ein fehlendes Kommando stellt selten ein unüberwindbares Problem dar: Die meisten Linux-Distributionen sind mit einem riesigen Archiv von Software-Paketen verbunden. Sie müssen aus diesen Paketquellen lediglich das richtige Paket installieren, das die von Ihnen benötigten Kommandos enthält.

#### apt versus dnf versus zypper versus brew

Bei der Paketverwaltung gibt es grundlegende Unterschiede zwischen den Linux-Distributionen. In diesem Kapitel konzentriere ich mich auf das Kommando `apt`, das unter Debian, Ubuntu und verwandten Distributionen zum Einsatz kommt.

In der Red-Hat-Welt, also z. B. auch unter Fedora oder AlmaLinux, verwenden Sie stattdessen `dnf`. Es funktioniert ganz ähnlich wie `apt`. Bei SUSE-Distributionen kommt stattdessen `zypper` zum Einsatz.

Unter macOS gibt es standardmäßig gar keine Paketverwaltung. Es gibt aber externe Tools wie `brew` (siehe *https://brew.sh*), die diese Aufgabe übernehmen und eine reiche Auswahl von Open-Source-Tools für macOS zugänglich machen. Einmal installiert funktioniert auch das Kommando `brew` ähnlich wie `apt` oder `dnf`.

### Software aktualisieren

Mit nur zwei Kommandos können Sie sämtliche unter Debian oder Ubuntu installierten Pakete auf den neuesten Stand bringen:

```
$ sudo apt update
$ sudo apt full-upgrade
```

Persönliches Exemplar für Thomas Bartholomäus

Sie werden sich wahrscheinlich fragen, warum zwei, noch dazu ähnlich klingende, Kommandos notwendig sind und nicht eines ausreicht. `apt update` aktualisiert nur die Informationen darüber, welche Pakete in den externen Paketquellen zur Verfügung stehen. Das Programm aktualisiert also das Verzeichnis verfügbarer Software, ohne aber die vorhandenen Pakete anzurühren. Erst `apt full-upgrade` lädt nach einer Rückfrage die aktualisierten Pakete herunter und installiert diese. Wenn das Update grundlegende Funktionen wie den Linux-Kernel betrifft, muss im Anschluss ein Neustart durchgeführt werden, damit das Update wirksam wird.

### Zusatzpakete installieren

Auch die Installation neuer Software beginnt mit `apt update` (es sei denn, Sie haben dieses Kommando gerade im Rahmen eines Updates ausgeführt). `apt install` lädt dann das gewünschte Software-Paket herunter, im folgenden Beispiel diverse im Paket `openssl` gesammelte Verschlüsselungswerkzeuge:

```
$ sudo apt update
$ sudo apt install openssl
```

In Linux ist Software oft über mehrere Pakete verteilt. Relativ oft passiert es, dass zur Ausführung der Kommandos aus Paket A Bibliotheken erforderlich sind, die sich im Paket B befinden. `apt` erkennt derartige Abhängigkeiten und installiert nach einer Rückfrage alle erforderlichen Zusatzpakete. Seien Sie also nicht überrascht, wenn `apt install xy` meldet, dass es nicht ein Paket installiert, sondern gleich fünf oder zehn.

**Pakete suchen**

Wenn Sie für Ihr Script ein bestimmtes Kommando benötigen, ist es oft schwierig zu erraten, in welchem Paket sich das Kommando befindet. Paket- und Kommandoname stimmen nicht immer überein, insbesondere, wenn ein Paket mehrere Kommandos enthält. Manchmal (aber leider nicht immer!) führt das Kommando `apt search xy` zum Ziel, das den Suchbegriff `xy` in der Paketbeschreibung sucht. Besser funktionieren im Internet verfügbare Paketsuchmaschinen, für Ubuntu z. B. hier:

*https://packages.ubuntu.com*

## 6.7 Sonstige Kommandos

Zum Abschluss dieses Kapitels stelle ich Ihnen ganz kurz einige weitere, oft benötigte Kommandos vor:

- `alias` definiert Abkürzungen. `alias ll='ls -lh'` ermöglicht es, das häufig benötigte Kommando `ls -l -h` (Dateien mit Detailinformationen auflisten und die


© Rheinwerk Verlag, Bonn 2024

Dateigröße *human readable* präsentieren) in der Kurzschreibweise `ll` auszuführen. Abkürzungen werden am besten in der Datei `.bashrc` angegeben, damit sie dauerhaft zur Verfügung stehen.

- `cat file` gibt eine Textdatei aus. Das Kommando wird oft mit einer Ein- oder Ausgabeumleitung kombiniert. Beispielsweise fügt `cat text1 text2 text3 > result` drei Textdateien aneinander und speichert das Ergebnis in einer neuen Datei.

  Ausgesprochen praktisch ist auch `cat > newfile`. Damit können Sie eine kurze neue Textdatei erzeugen, ohne einen Editor zu starten. Weil an das Kommando keine Datei zum Lesen übergeben wurde, erwartet es die Eingabe im Terminal (also aus der Standardeingabe). Strg+D beendet die Eingabe:

  ```
  $ cat > newfile
  Zeile 1
  Zeile 2
  <Strg>+<D>
  ```

- `date` liefert das aktuelle Datum samt Uhrzeit. Eine mit + beginnende Zeichenkette steuert optional das Ausgabeformat. Beispielsweise erzeugt `date "+%Y-%m-%d"` eine Zeichenkette mit dem Datum im ISO-Format (also z. B. `2023-12-31`).

- `history` listet alle zuletzt ausgeführten Kommandos auf.

- `ip` ermittelt oder verändert die Netzwerkkonfiguration. Die folgenden Beispiele zeigen besonders oft benötigte Kommandos:

  ```
  $ ip addr        # listet alle IP-Adressen des Rechners auf
  $ ip link        # listet die Netzwerkadapter auf
  $ ip route       # zeigt die Routing-Tabelle an
  ```

- `less file` zeigt wie `cat` eine Textdatei an. Das Kommando eignet sich für längere Dateien, weil Sie in Ruhe mit den Cursortasten durch den Text blättern können. Q beendet das Programm.

- `ln file link` erzeugt einen Link auf eine schon existierende Datei. Mit der Option `-s` wird anstelle eines *Hard Links* ein symbolischer Link erzeugt. Hintergründe zu Links können Sie bei Bedarf unter *https://wiki.ubuntuusers.de/ln* nachlesen.

  Falls Sie unsicher sind, in welcher Reihenfolge die Parameter anzugeben sind: Alle wichtigen Linux-Kommandos erwarten zuerst die Quelle, dann das Ziel. Das gilt für `cp`, `mv`, `ln` usw.

- `man command` zeigt den Hilfetext zu einem Kommando an. Wie bei `less` blättern Sie mit den Cursortasten durch den oft seitenlangen Text. Mit / können Sie einen Suchbegriff angeben. Q beendet die Hilfe.

- `ping hostname/ipaddress` sendet ICMP-Pakete an einen anderen Rechner und zeigt an, wie lange die Antwort dauert. Das Kommando läuft normalerweise endlos, bis

Persönliches Exemplar für Thomas Bartholomäus

es mit Strg+C beendet wird. ping -c <n> sendet nur *n* Pakete und endet dann selbstständig.

- wc file zählt die Anzahl der Zeilen, Wörter und Zeichen eine Textdatei. Mit den Optionen -c (*count bytes*), -w (*words*) oder -l (*lines*) kann die Ausgabe auf eine Zahl reduziert werden.
- which command ermittelt den Ort, an dem das Kommando gespeichert ist. Beispielsweise liefert which cp das Ergebnis /usr/bin/cp.



© Rheinwerk Verlag, Bonn 2024

# Kapitel 7
# CmdLets für die PowerShell

In diesem Kapitel präsentiere ich Ihnen eine kompakte Zusammenstellung der allerwichtigsten CmdLets – also gleichsam einen PowerShell-Grundwortschatz für den interaktiven Betrieb und für die Script-Programmierung. Wenn Sie schon seit Jahren Erfahrungen in der PowerShell gesammelt haben, können Sie dieses Kapitel getrost überspringen.

Neben den vorinstallierten CmdLets stehen im Internet unzählige PowerShell-Erweiterungen zur Auswahl. Bei der Installation dieser Zusatzmodule hilft der Paketmanager NuGet, den ich Ihnen in Abschnitt 7.8 vorstelle.

Das Kapitel endet mit einer Referenz der wichtigsten Aliasse. Diese Kürzel sparen bei der interaktiven Verwendung der PowerShell eine Menge Tipparbeit und ermöglichen eine ähnliche Effizienz wie unter Linux.

**Voraussetzungen für dieses Kapitel**

Naturgemäß macht die Lektüre dieses Kapitels nur Sinn, wenn Sie mit den in Kapitel 4 präsentierten Grundkonzepten der PowerShell vertraut sind – z. B. mit der Verkettung von CmdLets durch den Pipe-Operator `|`.

## 7.1 Verzeichnisse und Dateien

Mit den folgenden Kommandos wechseln Sie in Ihr Heimatverzeichnis (»Eigene Dateien«), erzeugen dort ein neues Verzeichnis und wechseln dorthin. Danach verlassen Sie das neue Verzeichnis wieder und löschen dieses.

```
> Set-Location                 # in das Heimatverzeichnis wechseln
> Get-Location                 # dessen Ort anzeigen

  Path
  ----
  C:\Users\kofler
```

Persönliches Exemplar für Thomas Bartholomäus

```
> New-Item -ItemType "directory" subdir   # neues Verz. erzeugen
> Set-Location subdir         # dorthin wechseln
> Set-Location ..             # zurück ins Heimatverzeichnis
> Remove-Item subdir          # subdir wieder löschen
```

Für sämtliche CmdLets aus dem vorigen Listing gibt es Aliasse oder Funktionen, die die Erledigung dieser Aktionen mit weniger Tippaufwand erlauben (siehe Tabelle 7.1). Da es empfehlenswert ist, in eigenen Scripts auf diese Kurzschreibweisen zu verzichten, verwende ich in allen weiteren Beispielen dieses Kapitels immer den originalen CmdLet-Namen.

```
> cd              # cd  ist ein Alias für Set-Location
> pwd             # pwd ist ein Alias für Get-Location
> md subdir       # md  ist ein Alias für die Funktion mkdir
> cd subdir
> cd ..
> rmdir subdir    # rmdir ist ein Alias für Remove-Item
```

Mit `New-Item` können Sie auch eine ganze Gruppe verschachtelter Verzeichnisse erzeugen. Obwohl Verzeichnisse unter Windows durch \ getrennt werden, akzeptieren CmdLets auch das Verzeichnistrennzeichen /.

```
> New-Item -ItemType "directory" images\2023\12
> New-Item -ItemType "directory" images/2023/12  # gleichwertig
```

### Dateien auflisten, kopieren, verschieben und löschen

Im folgenden Listing wird zuerst das Verzeichnis `tst` erstellt. Dort wird der aktuelle Zeitpunkt in der Datei `now.txt` gespeichert. Diese Datei wird zuerst kopiert, die Kopie dann umbenannt. `Get-ChildItem` listet die nun vorhandenen Textdateien auf. `Remove-Item` räumt zum Schluss wieder auf.

```
                                # erzeugt das Verzeichnis tst
> New-Item -ItemType "directory" tst
> Set-Location tst              # wechselt dorthin
> Get-Date > now.txt            # speichert Datum und Uhrzeit
                                # in now.txt
> Copy-Item now.txt copy.txt    # erstellt eine Kopie von now.txt
> Move-Item copy.txt backup.txt # benennt die Kopie um
> Get-ChildItem *.txt           # listet alle *.txt-Dateien auf

  Mode         LastWriteTime   Length  Name
  ----         -------------   ------  ----
  -a---     24.01.2023 12:29       40  backup.txt
  -a---     24.01.2023 12:29       40  now.txt

> Remove-Item *.txt             # löscht alle *.txt-Dateien
```


© Rheinwerk Verlag, Bonn 2024

Beachten Sie, dass mit `Remove-Item` gelöschte Dateien oder Verzeichnisse nicht im Papierkorb landen, sondern unwiderruflich gelöscht werden!

Um zu testen, ob eine Datei existiert, verwenden Sie `Test-Path`. Das CmdLet liefert `True` oder `False` zurück.

```
> Test-Path file1.txt
  False
```

Sie können mit `Copy-Item` oder `Move-Item` mehrere Dateien auf einmal verarbeiten. Dabei müssen Sie aber beachten, dass die Quelldateien entweder in Form eines Musters oder durch Kommata getrennt angegeben werden müssen. Nicht erlaubt ist hingegen die Aufzählung mehrerer Dateien ohne Kommata! Mit `-Recurse` können Sie auch ganze Verzeichnisbäume kopieren. Falls das Zielverzeichnis noch nicht existiert (z. B. `dir2` im letzten Beispiel), wird es automatisch erstellt.

```
> Copy-Item *.txt another-directory     # OK
> Copy-Item file1.txt, file2.txt a-d    # auch OK
> Copy-Item file1.txt file2.txt a-d     # Fehler!
> Copy-Item -Recurse dir1 dir2          # OK
```

`Get-ChildItem` bzw. dessen Kurzformen `dir`, `gci` und `ls` (nur unter Windows) listet Dateien und Verzeichnisse im angegebenen bzw. im gerade aktuellen Verzeichnis auf. Mit der Option `-Recurse` berücksichtigt das CmdLet auch alle Unterverzeichnisse. Mit der Option `-File` ermittelt `Get-ChildItem` nur Dateien, mit `-Directory` nur Verzeichnisse.

Standardmäßig zeigt das `Get-ChildItem` außer dem Dateinamen den Modus, den Zeitpunkt der letzten Änderung und die Dateigröße an. Hinter den Kulissen liefert `Get-ChildItem` aber Objekte mit wesentlich umfassenderen Daten. Es liegt an Ihnen, diese Daten auszuwerten. Im folgenden Beispiel berücksichtigt `Where-Object` nur `*.exe`-Dateien im Downloads-Verzeichnis, die größer als 1 MByte sind. `Select-Object` wählt aus den Ergebnisobjekten den Dateinamen, die Größe und den Zeitpunkt des letzten Zugriffs aus. Die Option `-gt` steht dabei für *greater than*.

```
> Get-ChildItem Downloads\*.exe |
  Where-Object {$_.Length -gt 1000000} |
  Select-Object Name, Length, LastAccessTime

  Name                             Length   LastAccessTime
  ----                             ------   --------------
  emacs-28.1-installer.exe       47972467   22.01.2023 08:09:58
  npp.8.4.4.Installer.x64.exe     4533912   24.01.2023 08:44:01
  VSCodeUserSetup-x64-1.69.2.exe 81424833    2.02.2023 12:47:04
  ...
```

Persönliches Exemplar für Thomas Bartholomäus

**Universelle »Items«**

In diesem Abschnitt habe ich Ihnen `New-Item`, `Copy-Item` usw. als Kommandos zur Manipulation von Dateien und Verzeichnissen präsentiert. Tatsächlich sind diese CmdLets wesentlich flexibler und können je nach Option bzw. Kontext auch ganz andere Objekte bearbeiten, z. B. Registry-Einträge oder Zertifikate.

### Text lesen und schreiben

`Get-Content` (Alias `cat`) gibt die angegebene Datei vollständig im Terminal aus.

```
> Get-Content todo.md
```

Bei längeren Dateien wäre es praktisch, mit den Cursor-Tasten durch den Text zu scrollen. Unter Linux zählt `less` sicher zu den am häufigsten aufgerufenen Kommandos. Umso erstaunlicher ist es, dass die PowerShell diesbezüglich nur rudimentäre Funktionen bietet: Mit `Out-Host -Paging` können Sie zwar mit ↵ eine Zeile und mit Leertaste eine Seite weiter scrollen, aber es gibt keinen Weg zurück, geschweige denn eine Suchfunktion.

```
> Get-Content tst.csv | Out-Host -Paging
```

Mit `Out-Host` (Alias `oh`) können Sie auch umfangreiche CmdLet-Ergebnisse seitenweise studieren:

```
> Get-Process | Out-Host -Paging
```

Immerhin ist es recht einfach, nur die ersten oder letzten Ergebnisobjekte eines CmdLets zu betrachten:

```
# die ersten fünf / die letzten Prozesse ansehen
> Get-Process | Select-Object -First 5
> Get-Process | Select-Object -Last 10
```

Um die Ergebnisse eines CmdLets in einer Textdatei zu speichern, greifen Sie einfach auf die Ausgabeumleitung zurück. Gespeichert werden dabei nicht die zugrunde liegenden Objekte, sondern nur der Text, den Sie üblicherweise im Terminal sehen.

```
> Get-Process > processes.txt
```

Mit `Set-Content` können Sie eine Zeichenkette in eine Textdatei schreiben. Falls die Datei schon existiert, wird sie überschrieben. Achten Sie auf die korrekte Reihenfolge der Parameter oder verwenden Sie Optionen!

```
> Set-Content out.txt "some text"
> Set-Content -Value "some text 2" out.txt  # gleichwertig
```

Um Text einer vorhandenen Datei hinzuzufügen, verwenden Sie `Add-Content`:


© Rheinwerk Verlag, Bonn 2024

```
> Add-Content out.txt 'second line'
```

Clear-Content löscht den Inhalt einer Datei. Die Datei selbst bleibt aber erhalten (Dateigröße: 0 Byte). Clear-Content löst einen Fehler aus, wenn die angegebene Datei nicht existiert.

```
> Clear-Content out.txt
```

### Text ein- und ausgeben

Texteingaben in einem Script führen Sie mit Read-Host durch. Beachten Sie, dass das CmdLet dem übergebenen Text automatisch einen Doppelpunkt hinzufügt. Um eine Zeichenkette (mit oder ohne Variablen) wieder auszugeben, verwenden Sie Write-Output (Alias echo):

```
> $firstname = Read-Host "First name"

  First name: Michael

> Write-Output $firstname

  Michael

> Write-Output "Hello, $firstname!"

  Hello, Michael!
```

## 7.2 Dateien suchen

Um Dateien zu suchen, die bestimmte Eigenschaften haben, gehen Sie wie im vorigen Beispiel vor: Sie erstellen also mit Get-ChildItem eine Ergebnisliste und wählen dann mit Where-Object die Objekte aus, die Ihren Vorstellungen entsprechen. Die Bedingung formulieren Sie in geschwungenen Klammern. Darin bezieht sich $_ auf das gerade verarbeitete Objekt.

Im folgenden Listing wird zuerst eine Variable erstellt, die das Datum vor 60 Tagen enthält. Get-ChildItem durchsucht das aktuelle Verzeichnis und alle Unterverzeichnisse nach allen *.pdf-Dateien. Where-Object, das sich sehr elegant mit ? abkürzen lässt, filtert die PDF-Dokumente heraus, die in den letzten zwei Monaten erzeugt oder verändert wurden. (-ge steht für *greater or equal*.)

```
> $twoMonthsAgo = (Get-Date).AddDays(-60)
> Get-ChildItem -Recurse *.pdf |
  ? {$_.LastWriteTime -ge $twoMonthsAgo}
```

Persönliches Exemplar für Thomas Bartholomäus

Sie können auch mehrere Bedingungen kombinieren. Das nächste Kommando durchsucht das Downloads-Verzeichnis nach Dateien, die größer als 1 MByte sind und die schon seit ca. zwei Monaten nicht mehr gelesen wurden:

```
> Get-ChildItem -Recurse Downloads |
  ? {$_.Length -gt 100000 -and
     $_.LastAccessTime -lt $twoMonthsAgo}
```

Wenn Sie mehrere Dateikennungen berücksichtigen wollen, verwenden Sie am besten die Option `-Include` und übergeben – durch Kommata getrennt – die gewünschten Dateitypen:

```
> Get-ChildItem -Recurse -Include *.png, *.jpg, *.jpeg, *.tif ...
```

Alternativ können Sie mit `-Exclude` auch Dateien, die dem jeweiligen Muster entsprechen, aus dem Ergebnis ausschließen. Beachten Sie, dass `-Include` und `-Exclude` nur in Kombination mit `-Recurse` funktionieren.

### Textdateien durchsuchen

`Get-ChildItem` in Kombination mit `Where-Object` berücksichtigt ausschließlich die Metadaten von Dateien, also Name, Typ, Größe usw., aber nicht den Inhalt. Mit `Select-String` können Sie Textdateien durchsuchen. Im einfachsten Fall können Sie `Select-String` direkt auf Dateien anwenden. Das folgende Kommando findet alle Logging-Dateien im aktuellen Verzeichnis, die den Text »error« in beliebiger Groß- und Kleinschreibung enthalten:

```
> Select-String error *.log
```

Anders als `Get-ChildItem` bietet `Select-String` aber keine Möglichkeiten zur rekursiven Suche nach Dateien. Deswegen werden die beiden Kommandos oft kombiniert. Im folgenden Beispiel sucht `Get-ChildItem` im aktuellen Verzeichnis und in allen Unterverzeichnissen nach *.txt-Dateien. `Select-String` sieht sich den Inhalt dieser Dateien an und listet alle Vorkommnisse auf:

```
> Get-ChildItem -Recurse -Include *.txt | Select-String license

  init/readme.txt:6:This work is licensed under ..., see
  init/readme.txt:7:https://creativecommons.org/licenses/by/4.0/
```

Oft sind Sie nicht an den Trefferdetails interessiert, sondern wollen nur eine Liste der gefundenen Dateien. Jedes `Select-String`-Ergebnis ist ein `MatchInfo`-Objekt. Dessen Eigenschaft `Path` enthält den Dateinamen. Indem Sie Gruppen solcher `MatchInfo`-Objekte bilden und dann den Namen jeder Gruppe anzeigen, erhalten Sie nur die Dateinamen:


© Rheinwerk Verlag, Bonn 2024

```
> Get-ChildItem -recurse *.txt | Select-String license |
  Group-Object Path | Select-Object Name

  init/readme.txt
  other/gpl.txt
  ...
```

Das Verhalten von `Select-String` kann durch diverse Optionen beeinflusst werden. Ich nenne hier nur die wichtigsten vier:

- `-CaseSensitive` beachtet die Groß- und Kleinschreibung im Suchmuster.
- `-List` zeigt nur die erste Übereinstimmung des Suchmusters. Das ergibt bei vielen Übereinstimmungen übersichtlichere Ergebnisse. Es beschleunigt auch die Auswertung, wenn wie im vorigen Beispiel nur die Dateien aufgelistet werden sollen, die einen Treffer enthalten.
- `-NotMatch` liefert Treffer (ohne `-List` jede Zeile!), die das Suchmuster *nicht* enthalten.
- `-Quiet` liefert nur `True` oder `False` zurück, je nachdem, ob das Muster erkannt wurde.

`Select-String` ist nicht in der Lage, die Anzahl der Übereinstimmungen zu zählen. Bei einer einzelnen Datei können Sie die `Matches`-Eigenschaft des resultierenden Objekts auswerten. Die folgende Anweisung zählt, wie oft »error« in der Datei `access.log` vorkommt:

```
> (Select-String error access.log).Matches.Count
```

Schwieriger ist es, die Trefferanzahl für mehrere Dateien aufzulisten. Der vielleicht nahe liegende Weg, `Select-String` mit `Matches.Count` in einer Pipeline anzuwenden, ist nicht zulässig und führt zu einer Fehlermeldung:

```
> Get-ChildItem *.log | (Select-String error).Matches.Count

  ParseError:
  Expressions are only allowed as the first element of a pipeline
```

Zum Ziel führt `ForEach-Object`: Damit können Sie für jede gefundene Datei eine Aktion ausführen. Im folgenden Beispiel gibt `Write-Host` den Namen der Datei aus. Danach ermittelt `Select-String` die Anzahl der »error«-Texte:

```
> Get-ChildItem *.log | ForEach-Object -Process {
    Write-Host -NoNewline $_.Name " -> "
    (Select-String error $_).Matches.Count
  }

  error.log  -> 231
  ...
```

Das an `Select-String` übergebene Suchmuster wird als regulärer Ausdruck ausgewertet. Details dazu folgen in Kapitel 9, »Reguläre Muster«.

#### Text versus Objekte

Anders als das Linux-Kommando `grep` können Sie `Select-String` nicht dazu verwenden, um ganz allgemein die Ergebnisse beliebiger CmdLets zu filtern. Der Grund dafür ist einfach: CmdLets liefern Objekte zurück, keinen Text. (Im Terminal wird schon Text angezeigt, aber nur, weil die PowerShell die Objekte vorher automatisch in Textform umwandelt.)

Wenn Sie also z. B. wissen möchten, welchen Alias das CmdLet `Start-Process` hat, bleibt das folgende Kommando wirkungslos. Es erkennt im `Get-Alias`-Ergebnis keinen Text:

```
> Get-Alias | Select-String Start-Process
```

Nun können Sie mit `Out-String` das `Get-Alias`-Ergebnis in eine Zeichenkette umwandeln und diese durchsuchen. Prinzipiell funktioniert das, `Select-String` findet auch die Übereinstimmung. Allerdings macht `Out-String` aus dem Ergebnis *eine* Zeichenkette. Anstatt nur die Alias-Definition von `Start-Process` anzuzeigen, liefert das Kommando die komplette Liste aller Aliasse, also letztlich das originale Ergebnis von `Get-Alias`. Damit ist nichts gewonnen.

```
> Get-Alias | Out-String | Select-String Start-Process

  CommandType     Name
  -----------     ----
  Alias           ? -> Where-Object
  Alias           % -> ForEach-Object
  Alias           cd -> Set-Location
  ...
```

Erst `Out-String -Stream` führt zum Ziel. Mit dieser Zusatzoption liefert das CmdLet jede Zeile als einzelne Zeichenkette. Endlich werden die gesuchten Aliasse `saps` und `start` angezeigt!

```
> Get-Alias | Out-String -Stream | Select-String Start-Process

  Alias           saps -> Start-Process
  Alias           start -> Start-Process
```

`command | Out-String -Stream | Select-String ...` ist aber selten eine gute Idee. CmdLets liefern Objekte. Ergründen Sie den Ergebnisdatentyp mit `Get-Member` oder `Format-List` und formulieren Sie dann eine entsprechende Bedingung zur Auswertung:


© Rheinwerk Verlag, Bonn 2024

```
> Get-Alias | ? {$_.Definition -eq 'Start-Process'}

  Alias             saps -> Start-Process
  Alias             start -> Start-Process
```

Noch zielführender ist es, hin und wieder `Get-Help` zu konsultieren. Mit der Option `-Definition` ermittelt `Get-Alias` ohne Umwege den Alias für das gewünschte Kommando:

```
> Get-Alias -Definition Start-Process

  Alias             saps -> Start-Process
  Alias             start -> Start-Process
```

## 7.3 Dateien komprimieren und archivieren

Mit `Compress-Archive` packen Sie ein Verzeichnis oder einzelne Dateien in eine komprimierte Archivdatei. Prinzipiell ist die Anwendung des Kommandos denkbar einfach:

```
> Compress-Archive mypictures\ mypictures.zip
> Compress-Archive *.txt archive.zip
```

Mit `-Update` können Sie ein vorhandenes Archiv erweitern bzw. ändern, mit `-Force` eine eventuell schon vorhandene ZIP-Datei überschreiben. `-CompressionLevel` gibt Ihnen die Möglichkeit, auf den Komprimieralgorithmus Einfluss zu nehmen. Zulässige Einstellungen sind `Optimal` (gilt per Default), `Fastest` und `NoCompression` (z. B. für schon komprimierte Bild- oder PDF-Dateien).

Zum Auspacken verwenden Sie `Expand-Archive`:

```
> Expand-Archive my.zip      # im aktuellen Verzeichnis auspacken
> Expand-Archive my.zip -DestinationPath dir   # in dir auspacken
```

Verblüffenderweise gibt es keine Möglichkeit, den Inhalt einer Archiv-Datei anzusehen bzw. aufzulisten. Gegebenenfalls müssen Sie dazu auf Kommandos von einem der zahlreichen ZIP-Erweiterungsmodule zurückgreifen, die im Internet zur Auswahl stehen.

### Suchergebnisse komprimieren

`Compress-Archive` kann prinzipiell auch Ergebnisse von `Get-ChildItem` verarbeiten, sofern Sie den Dateinamen des Archivs mit der Option `-DestinationPath` festlegen. Allerdings ist das Verhalten von `Compress-Archive` gewöhnungsbedürftig. Nehmen wir an, es liegt die folgende Verzeichnisstruktur vor:

```
dirA\
  file1
  file2
  dirB\
     file3
```

`Get-ChildItem` wird nun den Inhalt von `dirA` rekursiv durchlaufen und an `Compress-Archive` weitergegeben:

```
> Get-ChildItem -Recurse dirA |
  Compress-Archive -DestinationPath my.zip
> Expand-Archive my.zip -DestinationPath dirC
```

Im Verzeichnis `dirC` befinden sich nun folgende Dateien und Verzeichnisse:

```
dirC\
  file1
  file2
  file3 (!)
  dirB\
     file3
```

Das liegt daran, dass `Compress-Archive` von `Get-ChildItem` einmal das Verzeichnis `dirB` und einmal die Datei `file3` erhalten hat. `dirB` wurde korrekt verarbeitet, aber `file3` ohne Berücksichtigung des Verzeichnisses. Die Datei wurde deswegen doppelt in das Archiv eingefügt. Die Dokumentation weist auf diesen Sonderfall hin, das Verhalten wird also offenbar nicht als Fehler betrachtet. Insofern sollten Sie `Compress-Archive` nur dann zur Weiterverarbeitung von Dateien verwenden, wenn sich alle Dateien im selben Verzeichnis befinden.

## 7.4 Prozessverwaltung

Mit `Start-Process` können Sie einen neuen Prozess starten – für erste Tests z. B. den Editor *Notepad*.

```
> Start-Process notepad
```

`Start-Process` startet den Prozess im Hintergrund. Wenn Sie das nicht möchten, übergeben Sie die Option `-Wait`. Das Terminal bzw. Ihr Script ist damit blockiert, bis der Prozess endet.

Normalerweise hat der neue Prozess dieselben Rechte wie das Terminal bzw. Ihr Script. Um einen neuen Prozess mit Administrator-Rechten auszuführen, verwenden Sie `-Verb RunAs`. Allerdings erscheint nun ein Warndialog, in dem der Benutzer bzw. die Benutzerin des Computers bestätigen muss, dass die App Änderungen am Computer vornehmen kann.


© Rheinwerk Verlag, Bonn 2024

`Get-Process` liefert eine Liste aller auf dem Computer laufenden Prozesse. Wenn Sie als Parameter einen Prozessnamen angeben, werden nur gleichnamige Prozesse ermittelt. Mit `-IncludeUserName` zeigt das CmdLet auch den zugeordneten Benutzer an:

```
# alle Prozesse
> Get-Process

  NPM(K)  PM(M)  WS(M)  CPU(s)    Id  SI ProcessName
  ------  -----  -----  ------    --  -- -----------
       6   1,09   6,14    0,05  3888   0 AggregatorHost
      24   9,11  34,42    0,52  7492   1 ApplicationFrameHost
      10   1,79   8,25    0,00  2916   0 blnsvr
                                         ...
# nur notepad-Prozesse anzeigen, mit Benutzernamen
> Get-Process -IncludeUserName notepad

     WS(M)   CPU(s)      Id UserName        ProcessName
     -----   ------      -- --------        -----------
     60,01     0,50    1212 KVMWIN\kofler   Notepad
     59,88     0,47    8640 KVMWIN\kofler   Notepad
```

Sofern Sie über ein geeignetes Prozessobjekt verfügen und den Prozess selbst gestartet haben, können Sie Prozesse mit `Stop-Process` auch wieder beenden. Passen Sie aber auf: Das folgende Kommando beendet nicht nur den vorhin gestarteten Prozess, sondern *alle* laufenden *Notepad*-Instanzen! (Nur mit Administratorrechten können Sie auch fremde Prozesse beenden.)

```
> Get-Process notepad | Stop-Process
```

### Andere PowerShell-Scripts starten

Wenn Sie – ausgehend von einem bereits laufenden Script – ein anderes PowerShell-Script ausführen möchten, gehen Sie am besten so vor:

```
> Start-Process powershell.exe `
  -ArgumentList "-file C:\directory\myscript.ps1", "arg1", "arg2"
```

`Start-Process` startet also einen neuen PowerShell-Prozess, an den Sie den Speicherort Ihres Scripts und die gegebenenfalls notwendigen Parameter übergeben.

Falls Sie ein zweites Script nicht als neuen Prozess starten, sondern lediglich seinen Code an der aktuellen Position einlesen (importieren) möchten, gelingt das mit dem Source-Operator ».«:

```
. other-file.ps1
```

Persönliches Exemplar für Thomas Bartholomäus

Bei komplexen Aufgabenstellungen, die viel Code erfordern, sollten Sie Ihr Script in Module zerlegen und diese mit `Import-Module` einlesen.

### Kommandos im Hintergrund ausführen

Nicht immer wollen Sie gleich ein ganzes Script im Hintergrund ausführen – oft reichen auch ein paar Kommandos. Diese können Sie an `Start-Job` übergeben. Im folgenden Beispiel wird der Inhalt eines Verzeichnisses in einer ZIP-Datei gespeichert. Wenn die Anweisung in einem Script steht, wird das Script in der Zwischenzeit fortgesetzt.

```
> Start-Job { Compress-Archiv mydir\ archive.zip }
```

Falls Sie später im Script auf die Fertigstellung der ZIP-Datei angewiesen sind, müssen Sie sich die Job-ID merken. Damit können Sie später mit `Receive-Job` die Ausgabe des Vorgangs auslesen bzw. mit `Wait-Job` auf die Fertigstellung warten:

```
$myjob = Start-Job { Compress-Archiv mydir\ archive.zip }
# first do something else ...
# then wait for Compress-Archiv to finish
Wait-Job $myjob
```

### Dienste verwalten

»Dienste« (auch Systemdienste, englisch *Services*) sind Hintergrundprozesse, die für den Betrieb von Windows und seiner Funktionen erforderlich sind. Welche Dienste auf Ihrem System laufen, können Sie mit dem gleichnamigen Programm feststellen – oder mit dem Kommando `Get-Service`. Es listet sämtliche Dienste auf. Sie können an das CmdLet auch ein Muster übergeben, um gezielt nach Diensten zu suchen:

```
> Get-Service wl*

  Status   Name     DisplayName
  ------   ----     -----------
  Stopped  WlanSvc  Automatische WLAN-Konfiguration
  Running  wlidsvc  Anmelde-Assistent für Microsoft-Konten
  Stopped  wlpasvc  Dienst-Assistent für lokale Profile
```

Wie viele CmdLets zeigt `Get-Service` weniger Daten an, als intern tatsächlich zur Verfügung stehen. Wenn Sie wissen möchten, welcher dieser Dienste automatisch gestartet wird, zeigen Sie mit `Select-Object` auch die Eigenschaft `StartType` an:

```
> Get-Service wl* | Select-Object Name, StartType
```


© Rheinwerk Verlag, Bonn 2024

```
Name      StartType
----      ---------
WlanSvc     Manual
wlidsvc     Manual
wlpasvc     Manual
```

Mit `Start-`, `Restart-` und `Stop-Service` können Sie den angegebenen Dienst starten bzw. beenden. `Suspend-Service` deaktiviert den Dienst vorübergehend; er bleibt dabei im Speicher, reagiert aber nicht mehr. `Resume-Service` reaktiviert den Dienst wieder. (Beachten Sie aber, dass nicht alle Dienste Suspend und Resume unterstützen.)

```
> Start-Service winrm
```

Alle gerade aufgezählten Kommandos wirken unmittelbar. Um das Startverhalten für die Zukunft einzustellen, müssen Sie dagegen `Set-Service` mit der Option `-StartupType` verwenden.

```
> Set-Service -StartupType Automatic winrm
```

Eigene Dienste können Sie bei Bedarf mit `New-Service` einrichten und mit `Remove-Service` wieder entfernen.

### Auf anderen Rechnern arbeiten

Grundsätzlich werden CmdLets immer auf dem lokalen Rechner ausgeführt. Mit `Invoke-Commando` können Sie Kommandos auch auf anderen Rechnern ausführen. Das setzt unter anderem voraus, dass auf diesem Rechner das *Windows Remote Management* aktiv ist (Dienst `winrm`). Die beiden folgenden Beispiele führen zuerst das Kommando `Get-Process` und dann `myscript.ps1` auf dem Rechner `otherhost` aus:

```
> Invoke-Command otherhost { Get-Process }
> Invoke-Command otherhost -FilePath C:\myscript.ps1
```

Anstatt nur einzelne Kommandos auszuführen, können Sie mit `Enter-PSSession hostname` eine Verbindung zu einem anderen Rechner herstellen und dort eine PowerShell-Session öffnen. Gegebenenfalls geben Sie mit `-Credential name` den gewünschten Login-Namen an, wenn dieser von Ihrem eigenen abweicht. Sie müssen sich natürlich mit einem Passwort authentifizieren. Eine aktive Session beenden Sie mit `Exit-PSSession`.

```
>  Enter-PSSession otherhost
[otherhost]: PS C:\Users\username> Get-Process

  ...

[otherhost]: PS C:\Users\username> Exit-PSSession
```

Persönliches Exemplar für Thomas Bartholomäus

Eine Alternative zu `Enter-PSSession` ist das CmdLet `New-PSSession`. Es erzeugt ebenfalls eine Verbindung zu einem anderen Rechner. Das Session-Objekt können Sie in der Folge als Parameter an andere CmdLets übergeben, z. B. an `Copy-Item`, um Dateien auf einen anderen Rechner zu kopieren. `New-PSSession` ist aber nicht für die interaktive Anwendung gedacht.

**Remoting funktioniert nicht**

`Enter-PSSession` und `InvokeCommand` scheitern oft mit wenig aussagekräftigen Fehlermeldungen (z. B. »Zugriff verweigert« ohne weitere Begründung). Abhilfe schafft in der Regel das Kommando `Enable-PSRemoting`, das auf dem Zielrechner ausgeführt wird. Diverse Fehlerursachen samt Tipps zu deren Behebung zählt der folgende Artikel auf:

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

**SSH statt Remoting**

SSH bietet vergleichbare Funktionen wie `Invoke-Command` und `Enter-PSSession`, funktioniert aber plattformübergreifend. SSH setzt allerdings auf dem Zielrechner einen laufenden SSH-Server voraus, was gerade unter Windows noch eher unüblich ist.

Seit Version 6 unterstützt die PowerShell sogar *SSH Remoting*, also die Verwendung der Kommandos `Enter-PSSession` oder `New-PSSession` über eine SSH-Verbindung. Bei meinen Tests hat sich die Konfiguration aber als ähnlich fehleranfällig wie für das Windows Remote Management erwiesen. Mehr Details zur direkten Anwendung von SSH sowie zum SSH Remoting folgen in Kapitel 12.

## 7.5 Registrierdatenbank und Systeminformationen

Viele wichtigen Einstellungen befinden sich in der Registrierdatenbank. Zum Lesen der Einträge verwenden Sie das bereits bekannte Kommando `Get-ChildItem`. Das folgende Kommando liefert eine Liste der eigenen Einträge (`HKEY_CURRENT_USER`) in der ersten Ebene. Mit der zusätzlichen Option `-Recurse` durchläuft das CmdLet alle Ebenen der Registry.

```
> Get-ChildItem HKCU:\

  Name            Property
  ----            --------
  Console         ColorTable00                : 789516
                  ...
```


© Rheinwerk Verlag, Bonn 2024

```
  Environment     OneDrive           : C:\Users\kofler\OneDrive
                  OneDriveConsumer : C:\Users\kofler\OneDrive
                  ...
```

Mit Get-ItemProperty und Get-ItemPropertyValue können Sie gezielt den Inhalt eines Registry-Verzeichnisses bzw. eines einzelnen Eintrags ansehen:

```
# alle Einträge von HKEY_CURRENT_USER\Environment zeigen
> Get-ItemProperty HKCU:\Environment\

  OneDrive         : C:\Users\kofler\OneDrive
  OneDriveConsumer : C:\Users\kofler\OneDrive
  Path             : C:\Users\kofler\AppData\Local\Programs\...
  ...

# nur den Path-Eintrag zeigen
> Get-ItemProperty HKCU:\Environment\ | Select-Object Path

  C:\Users\kofler\AppData\Local\Programs\Python\Python311\...

# gleiches Ergebnis
> Get-ItemPropertyValue "HKCU:\Environment" -Name Path

  C:\Users\kofler\AppData\Local\Programs\Python\Python311\...
```

Natürlich können Sie die Registry auch verändern. New-Item erzeugt einen neuen Key, Set-ItemProperty speichert einen Eintrag. Passen Sie aber auf, dass Sie keine wichtigen Einstellungen verändern!

```
# erzeugt in zwei Schritten das Verzeichnis \michael\mykey
> New-Item HKCU:\michael
> New-Item HKCU:\michael\mykey

# speichert dort version=1.0
> Set-ItemProperty HKCU:\michael\mykey -Name version -Value 1.0

# liest die Daten wieder aus
> Get-ChildItem HKCU:\michael\

  Hive: HKEY_CURRENT_USER\michael

  Name    Property
  ----    --------
  mykey   version : 1

> Get-ItemPropertyValue "HKCU:\michael\mykey" -Name version
  1
```

Persönliches Exemplar für Thomas Bartholomäus

Remove-Item löscht alle in diesem Beispiel angelegten Registry-Verzeichnisse und Einträge. Passen Sie auf, dass Sie bei der Ausführung den korrekten Pfad angeben! Mit Remove-Item -Recurse können Sie viel Schaden anrichten, unabhängig davon, ob Sie nun Dateien oder Registry-Einträge löschen.

```
> Remove-Item -Recurse HKCU:\michael\
```

## Systeminformationen ermitteln

Get-ComputerInfo liefert eine lange Liste von Versionsnummern und Zeichenketten zur Identifizierung der Windows-Version. Mit Select-Object können Sie die davon relevanten Informationen herausfiltern:

```
> Get-ComputerInfo | Select-Object OsName, OSType, OsProductType

  OsName                     OsType   OsProductType
  ------                     ------   -------------
  Microsoft Windows 11 Pro    WINNT     WorkStation

> Get-ComputerInfo | Select-Object OsVersion, WindowsVersion

  OsVersion  WindowsVersion
  ---------  --------------
  10.0.22621 2009
```

Die sogenannte *Display Version* für installierte Funktions-Updates können Sie allerdings nur der Registry entnehmen:

```
> $key = "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion"
> (Get-ItemProperty $key).DisplayVersion
  22H2
```

Oft sind auch Sprache, Zeitzone und Installationsort von Interesse:

```
> Get-ComputerInfo | select OsLocale,TimeZone,OsWindowsDirectory

  OsLocale TimeZone                          OsWindowsDirectory
  -------- --------                          ------------------
  de-DE    (UTC+01:00) Amsterdam, Berlin, ... C:\Windows
```

Get-ComputerInfo verrät bei Bedarf auch die Bezeichnung der CPU, die Anzahl der Cores, die Größe des Arbeitsspeichers usw. Den freien Speicherplatz auf den Laufwerken (hier nur für C:, Angaben in Byte) ermittelt Get-PSDrive:

```
> Get-ComputerInfo | Select-Object CsProcessors

  {Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, ...}
```


© Rheinwerk Verlag, Bonn 2024

```
> Get-PSDrive C | Select-Object Used, Free

       Used         Free
       ----         ----
40943972352 173030539264
```

### Logging-System auswerten

Windows protokolliert alle erdenklichen Ereignisse in einem Logging-System. `Get-EventLog -List` verrät die Namen der Logs. Mit `Get-EventLog <name> -Newest n` können Sie die aktuellsten *n* Einträge auslesen.

```
> Get-EventLog Security -Newest 5 |
  Select-Object TimeGenerated, Message

  TimeGenerated          Message
  -------------          -------
  30.01.2023 15:47:30    Einer neuen Anmeldung wurden ...
  30.01.2023 15:47:30    Ein Konto wurde erfolgreich angemeldet.
  30.01.2023 15:41:52    Einer neuen Anmeldung wurden ...
  30.01.2023 15:41:52    Ein Konto wurde erfolgreich angemeldet.
  30.01.2023 15:13:48    Einer neuen Anmeldung wurden ...
```

## 7.6 CmdLet-Ergebnisse verarbeiten

CmdLets liefern Objekte zurück, nicht einfach Text. Der im Terminal angezeigte Text entsteht, weil die PowerShell die Objekte standardmäßig in eine Textform umwandelt. Um eine bessere Lesbarkeit zu erzielen, werden dabei viele Details entfernt.

Zu den elementarsten PowerShell-Aufgaben zählt es, Ergebnisse mit dem Pipe-Operator an ein zweites CmdLet weiterzuleiten. Damit können Sie die Daten auf vielfältige Weise filtern, weiterverarbeiten oder formatieren. Einige CmdLets zu diesem Zweck haben Sie ja schon kennengelernt – z. B. `Select-Object` oder `ForEach-Item`. Dieser Abschnitt fasst die oft benötigten Grundfunktionen dieser und einiger weiterer CmdLets zusammen.

### Select-Object und Sort-Object

Mit `Select-Object` (Alias `select`) können Sie die Ausgabe auf einzelne Eigenschaften reduzieren bzw. nur die ersten/letzten Ergebnisse auswählen. Diese Funktion ist vor allem dann zweckmäßig, wenn Sie die Ergebnisse vorher mit `Sort-Object` (Alias `sort`) nach einem Kriterium ordnen. `-Descending` sortiert abfallend. Die folgenden Beispiele

Persönliches Exemplar für Thomas Bartholomäus

verdeutlichen die wichtigsten Anwendungsvarianten. Machen Sie sich die Mühe, und probieren Sie die Kommandos in einem Verzeichnis mit vielen Dateien aus!

```
> Get-ChildItem    # alle Dateien/Standardeigenschaften anzeigen
> Get-ChildItem | Select-Object Name         # nur Name anzeigen
> Get-ChildItem | Select-Object Name, Size   # Name und Größe

# die drei kleinsten/größten Dateien anzeigen
> Get-ChildItem | Sort-Object Size | Select-Object -First 3
> Get-ChildItem | Sort-Object Size | Select-Object -Last 3

# die alphabetisch ersten fünf Dateien, nur Name und Größe
> Get-ChildItem | Sort-Object Name |
  Select-Object -First 10 Name, Size

# zuletzt geändert/erzeugt, Name, Größe und Änderungszeit
> Get-ChildItem | Sort-Object LastWriteTime |
  Select-Object -Last 10 Name, Size, LastWriteTime
```

`Select`- und `Sort-Object` kann auch auf Eigenschaften angewendet werden, die standardmäßig gar nicht angezeigt werden, z. B. auf `LastAccessTime` bei `Get-ChildItem`-Ergebnissen. Wenn Sie die Eigenschaften nicht kennen, wenden Sie `Get-Member` auf das erste Ergebnisobjekt an:

```
# Eigenschaften des ersten Ergebnisses von cmdlet anzeigen
> cmdlet | Select -First 1 | Get-Member -MemberType Properties
```

`Select-Object` kann auch die ersten *n* Objekte überspringen (`-Skip n`), die letzten Objekte überspringen (`-SkipLast n`), Doppelgänger eliminieren (`-Unique`) oder bestimmte Objekte auswählen (`-Index 2, 3, 7` für das 3., 4. und 8. Element; die Zählung beginnt bei 0).

### Where-Object

Mit `Where-Object` (Alias `?`) wählen Sie die Objekte aus, die ein bestimmtes Kriterium erfüllen. Die Bedingung wird in geschwungenen Klammern formuliert. Das jeweilige Objekt wird mit `$_` angesprochen.

```
# Dateien > 1 MByte anzeigen, nur Name und Größe
> Get-ChildItem | ? {$_.Size -gt 1000000} | Select Name, Size
```

### Group-Object

`Group-Object` (Alias `group`) bildet Gruppen, bei denen eine Eigenschaft übereinstimmt. Das Ergebnis enthält für jede Gruppe die Anzahl der Objekte, den Wert der Gruppeneigenschaft und eine Aufzählung der Elemente.


© Rheinwerk Verlag, Bonn 2024

```
> Get-ChildItem | Group-Object -Property Extension

  Count   Name    Group
  -----   ----    -----
     73   .jpg    {C:\Users\kofler\img_2234.jpg, ...}
      5   .jpeg   {C:\Users\kofler\tree.jpeg, ...}
    334   .png    {C:\Users\kofler\screenshot.jpeg, ...}
      3   .tif    {C:\Users\kofler\figure-23.tif, ...}

# die größten Gruppen zuerst, nur drei Gruppen
> Get-ChildItem | Group-Object -Property Extension |
  Sort-Object -Property Count -Descending |
  Select-Object -First 3

# Prozesse mit mehr als 5 Teilprozessen (Threads)
> Get-Process | Group-Object -Property ProcessName |
  Where-Object { $_.Count -gt 5 }
```

### ForEach-Item

Mit `ForEach-Item` (Alias %) können Sie jedes der ermittelten Objekte weiterverarbeiten. Welche Art der Weiterverarbeitung möglich ist, hängt stark von der Art der Objekte an. Dateien können Sie z. B. kopieren oder löschen, Prozesse beenden, offene Datenbankverbindungen schließen, installierte Pakete aktualisieren. Die auszuführende Aktion wird in geschwungenen Klammern formuliert, der Objektzugriff erfolgt wieder durch `$_`. Die folgende Kommandosequenz verschiebt alle Dateien aus dem aktuellen Verzeichnis, die älter als ein Jahr sind (`-lt` für *less than*), in das Unterverzeichnis `old`.

```
> $oneYear = (Get-Date).AddDays(-365)
> mkdir old
> Get-ChildItem | ? {$_.LastWriteTime -lt $oneYear} |
  ForEach-Object { Move-Item $_ old\ }
```

### Measure-Object

`Measure-Object` (Alias `measure`) wendet Aggregatfunktionen auf die angegebenen Objekte an. Ohne weitere Parameter zählt es die Objekte ganz einfach (Ergebniseigenschaft `Count`). Je nach Option können auch die Summe, der Durchschnitt, das Minimum oder Maximum einer Eigenschaft ermittelt werden.

```
# Anzahl der Dateien, Größe der kleinsten und größten Datei
> Get-ChildItem | Measure-Object  -Property Size -Min -Max |
  Select-Object Count, Minimum, Maximum
```

Persönliches Exemplar für Thomas Bartholomäus

```
  Count    Minimum       Maximum
  -----    -------       -------
    431   4096,000   3263035,000

# Zeilen, Wörter und Zeichen in der todo-Datei
> Get-Content todo.md | Measure-Object -Line -Word -Character

  Lines  Words  Characters
  -----  -----  ----------
    195    899        6396
```

Wenn Sie lediglich die Anzahl der Objekte zählen möchten, können Sie einfach die `Count`-Eigenschaft auf das Ergebnis anwenden. Das folgende Kommando ermittelt die Anzahl der PDF-Dateien im aktuellen Verzeichnis und allen Unterverzeichnissen:

```
> (Get-ChildItem *.pdf -Recurse).Count
```

### Formatierung und Export von CmdLet-Ergebnissen

Das, was Sie beim Ausführen eines Kommandos im Terminal sehen, ist zumeist eine Kurzform der zur Verfügung stehenden Daten. Für die Default-Formatierung ist das CmdLet `Out-Default` zuständig. Es berücksichtigt unter anderem XML-Dateien, die für wichtige Klassen (z. B. für `System.IO.DirectoryInfo`) festlegen, welche Eigenschaften wie ausgegeben werden sollen.

Wenn Sie mit der Default-Formatierung nicht zufrieden sind, können Sie darauf Einfluss nehmen:

- Mit der schon bekannten Konstruktion `cmdlet | Select-Object prop1, prop2, prop3` legen Sie fest, welche Eigenschaften Sie sehen möchten.
- Mit `cmdlet | Format-List` geben Sie die wichtigsten Eigenschaften jedes Objekts in Listenform an. Wenn Sie wirklich alle Eigenschaften sehen wollen, verwenden Sie `Format-List -Property *`. Passen Sie aber auf, dass Sie nicht von den Daten erschlagen werden! `Get-Process | Format-List -Property *` liefert eine Tausende Zeilen lange Ausgabe. In meiner Arbeitsumgebung waren es ca. 11.000 Zeilen, die sich so messen lassen:

  ```
  > Get-Process | Format-List -Property * | Out-String |
    Measure-Object -Line
  ```

- `cmdlet | Format-Table` führt eine tabellarische Ausgabe durch, wie dies für viele Klassen standardmäßig der Fall ist. Sie können die Ausgabe aber durch Optionen beeinflussen. Wenn Sie einen breiten Monitor haben, können Sie z. B. `Get-Process | Format-Table -Property * -AutoSize` ausprobieren. (`-AutoSize` berücksichtigt die Daten, um die optimale Spaltenbreite zu ermitteln.)


© Rheinwerk Verlag, Bonn 2024

- `cmdlet | Format-Size` versucht, die wichtigste Eigenschaft (z. B. den Dateinamen) über mehrere Spalten zu verteilen. `Get-ChildItem | Format-Wide -AutoSize` liefert z. B. eine mehrspaltige Liste der Namen aller Dateien im aktuellen Verzeichnis.
- `cmdlet | Out-Gridview` zeigt das Ergebnis in einem eigenen grafischen Fenster an (siehe Abbildung 7.1). Die Daten können dort neu sortiert und gefiltert werden, was gerade bei umfangreichen Ergebnissen äußerst hilfreich ist.

Get-Process | Out-GridView

Filter

Add criteria

| NPM(K) | PM(M) | WS(M) | CPU(s) | Id | SI | ProcessName |
|---|---|---|---|---|---|---|
| 298 | 538,53 | 186,90 | 0,00 | 3.500 | 0 | mysqld |
| 12 | 3,52 | 11,29 | 0,00 | 532 | 0 | NisSrv |
| 56 | 23,38 | 91,25 | 5,91 | 2.280 | 1 | OneDrive |
| 11 | 2,14 | 9,29 | 0,09 | 6.344 | 1 | OpenConsole |
| 84 | 61,41 | 131,77 | 2,38 | 6.120 | 1 | PhoneExperienceHost |
| 100 | 62,19 | 136,92 | 0,00 | 6.360 | 0 | pwsh |
| 67 | 34,81 | 93,82 | 4,41 | 1.580 | 1 | pwsh |
| 12 | 2,19 | 9,66 | 0,00 | 3.108 | 0 | qemu-ga |
| 12 | 4,44 | 67,98 | 0,00 | 100 | 0 | Registry |
| 11 | 2,12 | 10,95 | 0,06 | 7.404 | 1 | RuntimeBroker |
| 12 | 2,32 | 13,64 | 0,06 | 8.700 | 1 | RuntimeBroker |
| 12 | 2,55 | 11,86 | 0,08 | 4.624 | 1 | RuntimeBroker |

**Abbildung 7.1** Grafische Ausgabe von CmdLet-Ergebnissen

- `cmdlet | Export-Csv` speichert das CmdLet-Ergebnis in einer CSV-Datei. Standardmäßig enthält die erste Zeile als Spaltenbeschriftung die Namen aller Eigenschaften. In den weiteren Zeilen werden alle Texte in Anführungszeichen gesetzt und durch Kommata getrennt. Durch diverse Optionen können Sie Einfluss auf die Formatierungsdetails der Datei nehmen. Das Gegenstück zu `Export-Csv` ist `Import-Csv`. Dieses Kommando stelle ich Ihnen in Kapitel 8, »Textauswertung mit Filtern und Pipes«, vor.
- Mit `Test-NetConnection` können Sie eine Netzwerkverbindung überprüfen. Im Prinzip ist das CmdLet eine moderne Ping-Alternative, wobei Sie aber auch eine Port-Nummer übergeben können:

```
> Test-NetConnection kofler.info

  ComputerName           : kofler.info
  RemoteAddress          : 168.119.33.110
  InterfaceAlias         : Ethernet-Instanz 0
  SourceAddress          : 192.168.122.162
  PingSucceeded          : True
  PingReplyDetails (RTT) : 29 ms
```

Persönliches Exemplar für Thomas Bartholomäus

## 7.7 Sonstige CmdLets

In diesem Abschnitt stelle ich Ihnen ganz kurz einige weitere CmdLets vor, die oft benötigt werden, aber inhaltlich zu keinem der vorigen Abschnitte passen:

- `Get-Command <name>` liefert Informationen zu einem CmdLet oder Kommando. Beim CmdLets verrät die Eigenschaft `Source` das zugrunde liegende Modul. Bei herkömmlichen Kommandos können Sie dagegen die `Path`-Eigenschaft auswerten – dann wissen Sie, wo das Kommando installiert ist (vergleichbar mit `which` unter Linux):

  ```
  > (Get-Command Get-ChildItem).Source
    Microsoft.PowerShell.Management
  > (Get-Command ping.exe).Path
    C:\Windows\system32\PING.EXE
  ```

- `Get-History` liefert eine Liste aller zuletzt ausgeführten Kommandos.
- `Get-Random` liefert eine Zufallszahl zwischen 0 und 2.147.483.647. Mit `-Minimum` und `-Maximum` können Sie den Zahlenbereich einschränken. Wenn Sie `Get-Random` auf ein Array anwenden, liefert das CmdLet ein zufälliges Element zurück.
- `Measure-Command { expr }` ermittelt, wie lange die Ausführung des angegebenen Ausdrucks dauert.
- `Start-Sleep n` (Alias `sleep`) hält den laufenden Prozess bzw. das Script für *n* Sekunden an.
- `Get-Clipboard` liest den Inhalt der Zwischenablage aus. Zwar gibt es kein vergleichbares CmdLet, um die Zwischenablage mit Daten zu füllen. Sie können für diesen Zweck aber auf das Kommando `clip.exe` zurückgreifen. Beispielsweise kopiert `Get-Date | clip` das aktuelle Datum als Text in die Zwischenablage.

## 7.8 Zusatzmodule installieren

Standardmäßig sind unter Windows etliche PowerShell-Module mit CmdLets installiert. Im Internet gibt es unzählige Erweiterungsmodule. Als zentrale Quelle für derartige Erweiterungen hat sich die *PowerShell Gallery* (kurz PSGallery) etabliert. Dort werden mehr als 10.000 Module zum Download angeboten:

*https://powershellgallery.com*

Zur Installation von Modulen aus der PSGallery steht standardmäßig das Modul `PowerShellGet` zur Verfügung. Es enthält CmdLets wie `Install-Module` und `Find-Module`.


© Rheinwerk Verlag, Bonn 2024

**PowerShellGet-Update für PowerShell 5.1**

Zusammen mit PowerShell 7.n wird eine aktuelle Version von `PowerShellGet` ausgeliefert. Wenn Sie aber mit PowerShell 5.1 arbeiten, müssen Sie das Modul `PowerShellGet` zuerst aktualisieren. Die dafür notwendigen Schritte (Installation eines aktuellen .NET-Frameworks, Aktivierung von TLS 1.2 etc.) sind hier zusammengefasst:

*https://learn.microsoft.com/en-us/powershell/scripting/gallery/installing-psget*

Um ein Modul aus der PSGallery zu installieren, führen Sie einfach `Install-Module` aus:

```
> Install-Module PSReadExif

  Untrusted repository
  You are installing the modules from an untrusted repository.
  Are you sure you want to install the modules from 'PSGallery'?
  [Y] Yes  [A] Yes to All  [N] No  [L] No to All  ...: y
```

Bereits installierte Module können Sie mit `Update-Module` aktualisieren:

```
> Update-Module SQLServer
```

### Module, Kommandos und Paketquellen auflisten

In der PowerShell gibt es zwei Kommandos, um Module aufzulisten:

- `Get-Module` listet die Module auf, die in der aktuellen PowerShell-Instanz geladen wurden. Wenn Sie das Terminalfenster frisch geöffnet haben, sind das nur recht wenige. Da Module automatisch geladen werden, sobald ein darin enthaltenes Kommando ausgeführt wird, vergrößert sich die Liste je nachdem, welche Kommandos Sie verwenden.

  `Get-Module -ListAvailable` zählt sämtliche Module auf, die auf Ihrem Rechner zur Verfügung stehen.

- `Get-InstalledModule` ist Teil des `PowerShellGet`-Moduls und listet nur solche Module auf, die Sie mit `Install-Module` eingerichtet haben. Wenn Sie `Install-Module` noch nie verwendet haben, ist das Ergebnis leer.

  ```
  > Get-InstalledModule

    Version    Name        Repository   Description
    -------    ----        ----------   -----------
    1.0.2      PSReadExif  PSGallery    Read EXIF metadata ...
  ```

Persönliches Exemplar für Thomas Bartholomäus

Wenn Sie wissen möchten, welche Kommandos ein bestimmtes Modul zur Verfügung stellt, führen Sie `Get-Command` mit der Option `-Module` aus:

```
> Get-Command -Module PSReadExif

  CommandType    Name           Version    Source
  -----------    ----           -------    ------
  Function       Add-ExifData   1.0.2      PSReadExif
  Function       Get-ExifData   1.0.2      PSReadExif
  Function       Get-ExifTag    1.0.2      PSReadExif
```

Um herauszufinden, welche Paketquellen aktuell eingerichtet sind, führen Sie `Get-PackageProvider` aus. Normalerweise sind `PowerShellGet` und `NuGet` vorkonfiguriert, selbst dann, wenn Sie die PowerShell unter Linux oder macOS verwenden.

```
> Get-PackageProvider

  Name            Version   DynamicOptions
  ----            -------   --------------
  NuGet           3.0.0.1   Destination, ExcludeVersion, ...
  PowerShellGet   2.2.5.0   PackageManagementProvider, ...
```

### NuGet und winget

`PowerShellGet` ist der Paketmanager für die PowerShell. Unter Windows gibt es aber diverse andere Paketmanager, z. B. *NuGet*, *winget* und *Chocolatey*. Ich gehe an dieser Stelle nur auf die von Microsoft entwickelten Tools NuGet und winget ein. Diese Tools haben eine ganz andere Zielrichtung als `PowerShellGet`:

- **NuGet** ist ein Paketmanager für Software-Entwickler. Sie können damit alle erdenklichen Software-Komponenten, .NET-Bibliotheken und Tools verwalten. Unter *https://nuget.org* stehen über 300.000 Pakete zum kostenlosen Download zur Verfügung. NuGet wird seit vielen Jahren zusammen mit der Entwicklungsumgebung Visual Studio ausgeliefert.

  Aus PowerShell-Perspektive ist NuGet in zweierlei Hinsicht von Interesse: Einerseits können Sie NuGet durch PowerShell-Kommandos wie `Find-Package`, `Install-Package` und `Uninstall-Package` steuern. Andererseits kommt es manchmal vor, dass PowerShell-Module auf andere Tools oder Bibliotheken angewiesen sind. Diese können Sie unkompliziert mit NuGet installieren.

- **winget** ist ein neues Tool zur Paketverwaltung. Im Gegensatz zu NuGet, das sich nur an Entwickler richtet, hilft `winget` bei der Installation von Anwender-Software. `winget` bietet dieselben Funktionen und nutzt auch die gleichen Quellen wie der Microsoft Store, lässt sich aber effizient im Textmodus bedienen:

  ```
  > winget install Mozilla.Firefox
  ```


© Rheinwerk Verlag, Bonn 2024

Das Kommando `winget` steht unter Windows 11 sowie bei aktuellen Versionen von Windows 10 standardmäßig zur Verfügung. Sie können `winget` in PowerShell-Scripts verwenden, um Programme zu installieren oder zu aktualisieren. Allerdings ist `winget` aktuell nur als traditionelles Kommando implementiert. Ein PowerShell-Modul mit CmdLets wie `Install-WinGetPackage` ist immerhin schon in Arbeit:

*https://github.com/microsoft/winget-cli/tree/master/src/PowerShell/Microsoft.WinGet.Client*

## 7.9 Standard-Aliasse

Im Vergleich zu Linux, wo die Namen wichtiger Kommandos selten länger als vier Zeichen sind, sind die Namen häufig benötigter CmdLets sehr ausschweifend. Immerhin gibt es für PowerShell-Profis zu häufig eingesetzten Kommandos Abkürzungen (siehe Tabelle 7.1).

| Alias | CmdLet | Alias | CmdLet |
|---|---|---|---|
| ? | Where-Object | measure | Measure-Object |
| % | ForEach-Object | mv | Move-Item |
| cat | Get-Content | ni | New-Item |
| cd | Set-Location | oh | Out-Host |
| cpi | Copy-Item | ps | Get-Process |
| curl | Invoke-WebRequest | pwd | Get-Location |
| del | Remove-Item | rm | Remove-Item |
| dir | Get-ChildItem | select | Select-Object |
| echo | Write-Output | sleep | Start-Sleep |
| ft | Format-Table | sls | Select-String |
| gci | Get-ChildItem | sort | Sort-Object |
| gm | Get-Member | tee | Tee-Object |
| ls | Get-ChildItem | write | Write-Output |
| md | mkdir | | |

**Tabelle 7.1** Die Aliasse der wichtigsten CmdLets und Funktionen

Persönliches Exemplar für Thomas Bartholomäus

Viele Aliasse ergeben sich einfach aus den Anfangsbuchstaben des Verbs und des Substantivs des CmdLets – beispielsweise `mi` für `Move-Item`. Andere Aliasse wie `cat`, `dir` oder `ls` entsprechen dagegen gängigen DOS- und Linux-Kommandos.

Eigene Aliasse können Sie mit `Set-Alias` definieren und in `Documents/profile.ps1` dauerhaft speichern. Eine vollständige Referenz aller aktuell definierten Aliasse liefert das Kommando `Get-Alias`.

### Vermeiden Sie Aliasse in Scripts!

Sobald Sie sich im interaktiven Betrieb an die wichtigsten Aliasse gewöhnt haben, ist die Versuchung groß, diese auch in Scripts einzusetzen. Das ist keine gute Idee. Je nach Konfiguration oder Betriebssystem können unterschiedliche Kürzel aktiv sein. Beispielsweise ist `ls` unter Windows ein Alias für `Get-ChildItem`; unter Linux und macOS ruft `ls` dagegen das Unix-typische `ls`-Kommando auf!

In diesem Buch angegebene Aliasse gelten immer für Windows.

### Aliasse versus Funktionen

Oft ist es auf ersten Blick schwer zu sagen, ob eine Kurzschreibweise ein Alias oder eine Funktion ist. Wenn Sie neugierig sind, rufen Sie `Get-Command` auf. Dieses CmdLet verrät beispielsweise, dass `mkdir` eine Funktion ist, `rmdir` hingegen ein Alias.


© Rheinwerk Verlag, Bonn 2024

# Kapitel 8
# Textauswertung mit Filtern und Pipes

Zum Alltag von Systemadministratorinnen und -administratoren zählt die Auswertung von Logging-Dateien. Das Extrahieren relevanter Daten aus oft Millionen von Zeilen langen Dateien gleicht aber der sprichwörtlichen Suche nach der Nadel im Heuhaufen. Durch die geschickte Kombination von Kommandos wie `grep`, `sort`, `cut` oder `uniq` können Sie sich das Leben leichter machen. Mit etwas Übung lassen sich mit einzeiligen Kommandos wahre Wunder vollbringen.

Was hat dies mit den Begriffen »Filter« und »Pipe« aus der Überschrift zu tun?

- Als Filter wird im Shell-Kontext ein Kommando bezeichnet, das Text als Eingabe erwartet und neuen Text als Ausgabe liefert. Im Abschnitt 8.1, »grep, sort, cut und uniq«, stelle ich Ihnen die wichtigsten Filter-Kommandos vor, die Linux und macOS standardmäßig zur Verfügung stellen.
- Der Pipe-Operator `|` sollte Ihnen aus Kapitel 3, »Bash und Zsh«, vertraut sein. Er leitet die Ausgabe eines Kommandos zur Weiterverarbeitung an ein zweites Kommando weiter. Beispielsweise bewirkt `sort my.txt | uniq`, dass die Zeilen von `my.txt` zuerst sortiert und dann im zweiten Schritt alle Doppelgänger eliminiert werden.

Naturgemäß beschränkt sich die Anwendung von Filtern und Pipes keineswegs auf Logging-Dateien. Sie können damit auch Adressdaten, Benutzerverzeichnisse, CSV-Dateien etc. auswerten.

### Voraussetzungen für dieses Kapitel

Dieses Kapitel ist stark Bash-lastig. Die Bash ist die ideale Umgebung, um Kommandos wie `sort` oder `grep` anzuwenden. Dementsprechend setze ich in diesem Kapitel Bash-Grundwissen voraus. Erst in Abschnitt 8.5, »CSV-Dateien«, kommen auch Python und die PowerShell kurz zum Einsatz.

Die PowerShell verfügt auch über einen Pipe-Operator `|`. Dieser leitet allerdings Objekte anstelle von Text weiter. Daraus ergeben sich im Vergleich zur Bash großartige Verarbeitungsmöglichkeiten – aber eben nur wenn es sich bei den zugrunde liegenden Daten *nicht* um Text handelt. Aus diesem Grund spielt die PowerShell in diesem Kapitel nur eine untergeordnete Rolle – sorry!

Persönliches Exemplar für Thomas Bartholomäus

## 8.1 grep, sort, cut und uniq

In diesem Abschnitt stelle ich Ihnen die in der Überschrift genannten Kommandos vor – und einige weitere. Alle vorgestellten Kommandos gehören unter macOS und Linux zum Standardrepertoire, müssen also nicht extra installiert werden.

Um den Abschnitt so praxisnah wie möglich zu machen, zeige ich Ihnen die Anwendung sämtlicher Kommandos anhand der Textdatei `employees.txt`. Sie enthält fiktive Daten von 500 Angestellten in den USA. Das folgende Listing zeigt die ersten drei Zeilen, wobei diese jeweils umbrochen wurden:

```
FirstName;LastName;DateOfBirth;Street;Zip;City;
  State;Gender;Email;Job;Salary

Ruthanne;Summers;1977-06-04;4 Dewy Turnpike;27698;Clifton Hill;
  NJ;F;ruthanne_ferguson5693@fastmail.cn;Engineer;5201.45

Lorie;Warner;1972-04-27;59 Hickory Way;78509;Simsbury Center;
  MA;F;lorie7961@mailforce.net;Climatologist;3794.34
```

Sie finden `employees.txt` in den Beispieldateien zu diesem Buch.

**Beispiele unter Windows ausprobieren**

Unter Windows gibt es zum Teil gleichnamige Kommandos (manche stammen aus DOS-Zeiten!) bzw. PowerShell-Aliasse, Funktionsweise und Optionen weichen aber von Linux/macOS ab. Wenn Sie die hier präsentierten Beispiele unter Windows nachvollziehen möchten, verwenden Sie am besten WSL oder die Git Bash.

### grep

`grep` ist nach `ls` vermutlich das Kommando, das ich im Terminal am häufigsten ausführe. In seiner Grundform filtert es aus einer Textdatei die Zeilen, die den Suchbegriff enthalten. Das erste Beispiel filtert alle Fehlermeldungen aus `log.txt`, das zweite alle Angestellten aus Kalifornien (Kürzel `CA`):

```
$ grep error log.txt
$ grep CA employees.txt

  Elissa;David;...;CA;...
  Sammie;van Luyn;...;CA;...
  Dinorah;Mcgee;...;CA;...
```

Das Verhalten von `grep` kann durch diverse Optionen beeinflusst werden. Ich nenne hier nur die wichtigsten:


© Rheinwerk Verlag, Bonn 2024

- `-i` ignoriert bei der Suche die Groß- und Kleinschreibung (*ignore case*).
- `-r` durchsucht alle Dateien im aktuellen bzw. im angegebenen Verzeichnis sowie in allen Unterverzeichnissen (*recursive*).
- `-c` zeigt die gefundenen Zeilen nicht an, sondern nur deren Anzahl (*count*).
- `-l` listet nur die Dateien auf, die Treffer enthalten (*list*). Die Optionen `-l` und `-c` sind vor allem dann praktisch, wenn Sie mehrere Dateien auf einmal verarbeiten, z. B. `grep -l myfunction *java`.
- `-v` invertiert die Suche, berücksichtigt also nur Zeilen, die den Suchbegriff *nicht* enthalten.

`grep` wendet das Suchmuster immer auf die ganze Zeile an, nicht auf Spalten. Passen Sie daher auf, dass Ihr Suchmuster eindeutig ist. Angenommen, Sie wollen alle Personen auflisten, die 1985 geboren sind:

```
$ grep 1985 employees.txt

  Dorthea;Mckinney;1985-06-02;915 Field Divide;...
  Kandi;Meester;1985-01-27;424 Smith Vale;...
  ...
  Kanesha;Matthews;1968-06-24;P.O. Box 19858;...
```

Die ersten Treffer stimmen, aber die letzte Zeile ist falsch. *Kanesha* ist 1986 geboren. Ihre Adresse enthält aber blöderweise die Postfachnummer 19858. Durch eine genauere Formulierung des Suchmusters können Sie den Fehler in diesem Fall eliminieren:

```
$ grep ';1985-' employees.txt
```

**Reguläre Muster**

Der im ersten Parameter übergebene Suchbegriff wird als reguläres Muster ausgewertet. Solange Sie nach gewöhnlichem Text suchen, spielt das keine Rolle. Wenn Ihr Suchbegriff aber Sonderzeichen enthält, müssen Sie aufpassen und den Sonderzeichen \ voranstellen – etwa `help\.com`, um nach `help.com` zu suchen. Was reguläre Muster sind und welche Anwendungsmöglichkeiten sich daraus im Allgemeinen und speziell für `grep` ergeben, erfahren Sie in Abschnitt 9, »Reguläre Muster«.

### wc

`wc` (*word count*) zählt die Zeilen, Wörter und Zeichen einer Datei.

```
$ wc employees.txt

  501  2628 61102 employees.txt
```

Persönliches Exemplar für Thomas Bartholomäus

`wc` lässt sich ausgezeichnet mit anderen Kommandos verknüpfen. Wie viele Mitarbeiterinnen wohnen im Bundesstaat Colorado (Kürzel CO)?

```
$ grep ';CO;' employees.txt | grep ';F;' | wc -l
  4
```

Das erste `grep`-Kommando liefert alle Colorado-Zeilen. Anstatt diese auszugeben, werden sie mit dem Pipe-Operator an das zweite `grep` weitergeleitet. Es filtert aus dem Zwischenergebnis die Frauen heraus (Geschlecht `F`). `wc -l` muss dann nur noch die resultierenden Zeilen zählen.

## cut

`cut` (nicht zu verwechseln mit `cat`!) schneidet Spalten aus einem Text. Für die meisten Anwendungen reicht es aus, wenn Sie die folgenden drei Optionen kennen:

- `-d` gibt das Trennzeichen an. Als Defaulteinstellung gilt das Tabulatorzeichen. Sonderzeichen müssen oft durch `\` »quotiert« werden oder in Anführungszeichen gestellt werden, z. B. `-d\;` oder `-d ';'`.
- `-f` wählt die gewünschten Spalten aus. Beispielsweise extrahiert `-f 1,4` die erste und vierte Spalte, `-f 4-7` die Spalten vier bis sieben.
- `-s` ignoriert Zeilen, in denen das Trennzeichen nicht vorkommt.

Wenn Sie nur die Namen und die E-Mail-Adressen der Angestellten sehen wollen, gehen Sie so vor:

```
$ cut -d ';' -f 1,2,9 employees.txt

  FirstName;LastName;Email
  Ruthanne;Ferguson;ruthanne_ferguson5693@fastmail.cn
  Lorie;Warner;lorie7961@mailforce.net
  Kerri;Tsioupra;kerri2377@emailuser.net
  ...
```

Die nächsten beiden Kommandos ermitteln die E-Mail-Adressen der Angestellten aus dem Bundesstaat New York. Zuerst werden die Adressen im Terminal angezeigt, wobei Sie dank `less` in Ruhe mit den Cursortasten durch das Ergebnis blättern können; danach werden die Resultate in `email.txt` gespeichert.

```
$ grep ';NY;' employees.txt | cut -d ';' -f 9 | less

  nakesha_lambrinou6390@speedymail.org
  vancapelle1740@fastmailbox.net
  noble7595@internetemails.net
  ...

$ grep ';NY;' employees.txt | cut -d ';' -f 9 > emails.txt
```


© Rheinwerk Verlag, Bonn 2024

### sort

Im einfachsten Fall bewirkt `sort my.txt`, dass die Zeilen alphabetisch sortiert am Bildschirm ausgegeben werden können. Anstatt eine Datei zu verarbeiten, können Sie mit dem Pipe-Operator aber auch die Ergebnisse eines anderen Kommandos sortieren. Wie üblich gibt es ein paar Optionen, um das Sortierverhalten zu steuern:

- `-n` sortiert numerisch, also 1, 2, 3, 10 usw., nicht 1, 10, 2, 3.
- `-r` sortiert abfallend statt aufsteigend (*reverse*).
- `-k <n>` gibt an, ab welcher Spalte sortiert werden soll. Die Option wird zumeist in Kombination mit `-t` verwendet.
- `-t` gibt das Trennzeichen für Spalten an, z. B. `-t ';'`. Ohne die Option betrachtet `sort` den Übergang von einem Leerzeichen zu einem Textzeichen als Beginn einer neuen Spalte.

Das folgende Kommando erstellt zwei nach Namen sortierte Mitarbeiterlisten:

```
# nach Vornamen sortieren (also ab dem Beginn der Zeile)
$ sort employees.txt

# nach Nachnamen sortieren (zweite Spalte)
$ sort -t ';' -k 2 employees.txt
```

**Setzen Sie »sort« überlegt ein!**

500 Zeilen zu sortieren, ist für einen modernen Computer keine Herausforderung. Vorsicht ist beim Einsatz von `sort` aber bei größeren Datenmengen geboten. Wenn Sie eine Logging-Datei mit Millionen von Zeilen sortieren, erfordert das eine Menge Zeit und Speicherplatz. Zudem kann `sort` bei `cmd | sort` erst mit der Arbeit beginnen, wenn `cmd` vollständig abgeschlossen ist.

Soweit möglich, sollten Sie umfangreiche Textdateien zuerst mit `grep` filtern und vielleicht mit `cut` die Spaltenanzahl reduzieren, bevor Sie `sort` anwenden.

### head und tail

Gerade im Zusammenhang mit `sort` kommt es häufig vor, dass Sie nur die ersten oder letzten *n* Ergebnisse sehen wollen. Hier kommen `head` und `tail` zu Hilfe:

```
# die drei Angestellten mit dem niedigsten Gehalt (Spalte 11)
$ sort -t ';' -k 11 -n employees.txt | cut -d ';' -f 1,2,11 | \
  head -n 3

  Mohamed;Mcbride;1804.67
  Gabrielle;Melendez;1807.37
  Benedict;Oconnor;1812.30
```

Persönliches Exemplar für Thomas Bartholomäus

```
# die drei Angestellten mit dem höchsten Gehalt (fehlerhaft)
$ sort -t ';' -k 11 -n employees.txt | cut -d ';' -f 1,2,11 | \
  tail -n 3

  Estella;Guerra;5586.85
  Dolly;Leonard;5600.00
  FirstName;LastName;Salary
```

Beachten Sie aber, dass das zweite Ergebnis fehlerhaft ist und auch die Beschriftungszeile enthält! Von `employees.txt` sollten eigentlich nur die Zeilen 1 bis 501 sortiert werden, nicht aber Zeile 1 mit der Spaltenbeschriftung. Um die erste Zeile zu überspringen, verwenden Sie `tail -n +2`. Diese gewöhnungsbedürftige Formulierung bedeutet, dass alle Zeilen ab der zweiten Zeile bis zum Ende der Datei ausgegeben werden sollen.

```
# die drei Angestellten mit dem höchsten Gehalt (korrekt)
$ tail -n +2 employees.txt | sort -t ';' -k 11 -n | \
  cut -d ';' -f 1,2,11 | tail -n 3

  Arden;Lit;5584.95
  Estella;Guerra;5586.85
  Dolly;Leonard;5600.00
```

### uniq

`uniq my.txt` eliminiert aufeinanderfolgende gleichlautende Zeilen der Textdatei und gibt das Ergebnis im Terminal aus. `uniq` erkennt allerdings keine Doppelgänger, wenn diese nicht unmittelbar aufeinanderfolgen. Diese Einschränkung umgehen Sie, indem Sie den Text zuvor sortieren.

Im folgenden Listing ermittelt das erste Kommando eine alphabetische Liste aller Bundesstaaten, in denen Angestellte wohnhaft sind. `tail` eliminiert die Spaltenbeschriftung, `cut` schneidet die siebte Spalte mit dem Bundesstaat aus. Das zweite Kommando beweist, dass es offenbar Mitarbeiterinnen und Mitarbeiter in allen 50 Staaten der USA gibt.

```
$ tail -n +2 employees.txt | cut -d ';' -f 7 | sort | uniq
  AK
  AL
  AR
  ...

$ tail -n +2 employees.txt | cut -d ';' -f 7 | sort | \
  uniq | wc -l

  50
```


© Rheinwerk Verlag, Bonn 2024

Jetzt würde uns noch interessieren: Wie viele Mitarbeiter und Mitarbeiterinnen gibt es in jedem Bundesstaat? Auch diese Frage kann `uniq` beantworten, wenn Sie die Option `-c` (*count*) übergeben:

```
$ tail -n +2 employees.txt | cut -d ';' -f 7 | sort | uniq -c
     11 AK
     12 AL
      8 AR
    ...
```

Wenn wir nun die fünf Bundesstaaten mit den meisten Angestellten sehen wollen, sortieren wir das `uniq`-Ergebnis nochmals. Die Option `-n` stellt sicher, dass `sort` numerisch korrekt sortiert. `-r` dreht die Sortierordnung um, zeigt also die größten Ergebnisse zuerst an. `head` zeigt nur die ersten fünf Zeilen des Ergebnisses:

```
$ tail -n +2 employees.txt | cut -d ';' -f 7 | sort | uniq -c | \
  sort -n -r | head -n 5

     21 OH
     19 MN
     16 MS
     14 OK
     14 NV
```

### tr

`tr` (für *translate*) ersetzt in einem Text einzelne Zeichen durch andere. `tr` arbeitet rein zeichenbasiert, kann also keine Wörter ersetzen. Mit der Option `-d` (*delete*) werden die betreffenden Zeichen eliminiert. Die folgenden drei Beispiele verdeutlichen die Funktionsweise:

```
# . durch , ersetzen
$ tr '.' ',' < in.txt > out.txt

# Kleinbuchstaben durch Großbuchstaben ersetzen
$ tr "a-zäöü" "A-ZÄÖÜ" < in.txt > out.txt

# deutsche Sonderzeichen ersatzlos eliminieren
$ tr -d "äöüßÄÖÜ" < in.txt > out.txt
```

Natürlich kann auch `tr` als Filter eingesetzt werden. Das folgende Kommando liefert die Vor- und Nachnamen sowie E-Mail-Adressen aller Mitarbeiterinnen und Mitarbeiter aus Texas, wobei die Spalten durch das Zeichen `:` getrennt werden. Zuvor wurde durch ein `;` getrennt:

Persönliches Exemplar für Thomas Bartholomäus

```
$ grep ';TX;' employees.txt |  cut -d ';' -f 1,2,9 | tr ';' ':'

  Mae:van Diedenhoven:mae5561@emailgroups.net
  Nancee:Coleman:nancee_coleman4689@150ml.com
  Mitch:Matse:matse2138@theinternetemail.com
  ...
```

### awk und sed

Neben den hier vorgestellten Kommandos, von denen jedes für sich einfach anzuwenden ist und die ihre Vielseitigkeit erst in der Kombination entfalten, stellt das UNIX-Ökosystem auch wesentlich komplexere Kommandos zur Auswahl. Das von den Entwicklern Aho, Weinberger und Kernigham entwickelte Tool `awk` ist eine eigene Script-Sprache zur Bearbeitung und Analyse von Texten. Mindestens ebenso vielseitig ist der Stream-Editor `sed`. Auch wenn in diesem Buch der Platz für eine detaillierte Behandlung dieser Programme nicht ausreicht, möchte ich Ihnen die `awk` und `sed` zumindest anhand eines Beispiels kurz vorstellen.

Erinnern Sie sich an das `grep`-Beispiel `grep 1985 employees.txt`? Es filtert alle Zeilen aus der Textdatei, die an irgendeiner Stelle die Zeichenkette `1985` enthalten. Wir wollten aber eigentlich nur solche Treffer, bei denen das Geburtsdatum 1985 war, die also 1985 in der dritten Spalte enthalten. Mit `grep` lässt sich eine derartige Suchbedingung nur schwer formulieren. Für `awk` ist das dagegen kein Problem. Hier gibt die Option `-F` das Spaltentrennzeichen an. `$3` bezieht sich auf die dritte Spalte. Diese muss den zwischen // angegebenen Suchbegriff enthalten:

```
$ awk -F ';' '$3 ~/1985/' employees.txt

  Dorthea;Mckinney;1985-06-02;915 Field Divide;...
  Kandi;Meester;1985-01-27;424 Smith Vale;...
  ...
```

Wenn Sie häufig derartige Aufgaben erledigen müssen, lohnt es, die `awk`-Syntax zu lernen. Es gibt im Internet diverse Tutorials. Empfehlenswert und beispielorientiert ist diese Seite:

*https://linuxhandbook.com/awk-command-tutorial*

Ähnliche Funktionen wie `awk` erfüllt auch `sed`. Ein Beispiel dazu folgt im nächsten Kapitel. Dort zeige ich Ihnen, wie Sie mit `sed` unter der Anwendung regulärer Ausdrücke nach Mustern in einer Textdatei suchen und diese durch andere Ausdrücke ersetzen.


© Rheinwerk Verlag, Bonn 2024

## 8.2 Beispiel: Automatische Coronazahlenauswertung

Vermutlich haben Sie sich bei den Beispielen im vorigen Abschnitt gedacht: »Wenn ich ein paar Kontaktdaten sortieren muss, verwende ich dazu Excel. Das ist einfacher.« Natürlich, so mache ich das auch. Wobei es mir noch lieber ist, wenn die Daten in einer Datenbank gespeichert sind und ich SQL verwenden kann. Die Beispiele hatten ausschließlich den Sinn, Ihnen mit einfach zu verstehenden Daten die Anwendungsmöglichkeiten von `grep`, `cut`, `sort`, `uniq` usw. zu veranschaulichen.

Ab jetzt geht es um Beispiele, die näher am Scripting-Alltag sind. Ich beginne hier mit der Auswertung öffentlich zugänglicher Gesundheitsdaten. So stellt die österreichische Regierung einmal täglich einen CSV-Datensatz mit den aktuellen Corona-Fallzahlen zur Verfügung:

*https://covid19-dashboard.ages.at/data/CovidFaelle_Timeline.csv*

Eine Beschreibung der Daten gibt es hier:

*https://www.data.gv.at/katalog/dataset/ef8e980b-9644-45d8-b0e9-c6aaf0eff0c0*

Die folgenden Zeilen zeigen einen Ausschnitt der Daten. Täglich kommt für jedes Bundesland sowie für die gesamtösterreichischen Daten jeweils eine Zeile hinzu:

```
08.02.2023 00:00:00;Vorarlberg;8;403203;0;256751;599;
  148,5604;0;681;111;255728
08.02.2023 00:00:00;Wien;9;1951354;0;1328827;7141;
  365,951;0;4609;423;1309493
08.02.2023 00:00:00;Österreich;10;9027999;1;5812695;22469;
  248,8813;0;21773;2091;5754729
```

Die siebte Spalte enthält die Anzahl der neuen Fälle innerhalb der letzten sieben Tage. In der achten Spalte wird diese Zahl in das Verhältnis zu den Einwohnern Österreichs bzw. des Bundeslands gesetzt (Siebentages-Inzidenz pro 100.000 Einwohner). Wenn Sie die Datei also unter `covid.csv` gespeichert haben, können Sie die Auswertung so durchführen:

```
$ grep Österreich covid.csv | cut -d ';' -f 1,8 | tail -n 5

  04.02.2023 00:00:00;292,7338
  05.02.2023 00:00:00;294,6943
  06.02.2023 00:00:00;298,2831
  07.02.2023 00:00:00;298,7816
  08.02.2023 00:00:00;248,8813
```

Vergleichbare öffentliche Daten gibt es natürlich auch von den meisten anderen europäischen Ländern.

Persönliches Exemplar für Thomas Bartholomäus

### Ein hoffentlich veraltetes Beispiel

Als ich an diesem Buch gearbeitet habe, sah es so aus, als würde die Corona-Epidemie endlich zu Ende gehen. Daher kann es sein, dass die obigen Links bis zum Erscheinen des Buchs nicht mehr funktionieren oder nur noch veraltetes Zahlenmaterial liefern.

## Auswertung per Script automatisieren

Um den Download und die Auswertung der Daten zu automatisieren, können Sie ein kleines Script schreiben. An dieses Script übergeben Sie einen optionalen Parameter mit dem Namen des Bundeslands. Fehlt dieser Parameter, werden automatisch die gesamtösterreichischen Daten angezeigt.

Kurz noch einige Informationen zu `curl`. Dieses Kommando lädt eine Datei aus dem Internet herunter und zeigt sie am Bildschirm an. Die Option `-s` (*silent*) verhindert Statusausgaben während des Downloads. Im folgenden Script wird die `curl`-Ausgabe direkt an `grep` weitergeleitet.

```
#!/bin/bash
# Beispieldatei: covid.sh
# Anwendung:     ./covid.sh  <country>
url="https://covid19-dashboard.ages.at/data/\
     CovidFaelle_Timeline.csv"
if [ $# -ge 1 ]; then
  country=$1
else
  country="Österreich"
fi
curl -s $url | grep $country | cut -d ';' -f 1,8 | tail -n 5
```

Wenn Sie die Entwicklung der 7-Tages-Inzidenz für Tirol während der letzten Tage wissen möchten, rufen Sie das Script so auf:

```
$ ./covid.sh Tirol

  04.02.2023 00:00:00;241,7507
  05.02.2023 00:00:00;243,7077
  06.02.2023 00:00:00;250,7528
  07.02.2023 00:00:00;267,7132
  08.02.2023 00:00:00;227,3996
```


© Rheinwerk Verlag, Bonn 2024

## 8.3 Beispiel: ping-Auswertung

Mit dem Kommando `ping` können Sie die Netzwerkverbindung zu einem anderen Rechner überprüfen. Die Option `-c` gibt an, wie viele ICMP-Pakete dabei versendet werden sollen, d. h. nach wie vielen Versuchen `ping` endet. Ohne die Option läuft `ping` unter Linux und macOS so lange, bis Sie das Kommando mit Strg+C beenden. Das folgende Listing zeigt ein typisches `ping`-Ergebnis.

```
$ ping -c 4 google.com

  PING google.com(bud02s41-in-x0e.1e100.net
     (2a00:1450:400d:802::200e)) 56 Datenbytes
  64 Bytes von bud02s41-...: icmp_seq=1 ttl=117 Zeit=15.4 ms
  64 Bytes von bud02s41-...: icmp_seq=2 ttl=117 Zeit=17.4 ms
  64 Bytes von bud02s41-...: icmp_seq=3 ttl=117 Zeit=17.3 ms
  64 Bytes von bud02s41-...: icmp_seq=4 ttl=117 Zeit=15.6 ms

  --- google.com ping-Statistik ---
  4 Pakete übertragen, 4 empfangen, 0% packet loss, time 3006ms
  rtt min/avg/max/mdev = 15.434/16.431/17.382/0.914 ms
```

Beachten Sie, dass manche Server (vermeintlich aus Sicherheitsgründen) so konfiguriert sind, dass sie gar keine Antwort auf `ping`-Requests senden. Ein prominentes Beispiel ist Microsoft (also `microsoft.com`).

Nehmen Sie an, Sie wollen aus diesem Ergebnis nur die durchschnittliche Antwortzeit extrahieren, im obigen Listing also den Wert 16.431 Millisekunden. Dazu filtern Sie zuerst mit `grep` die letzte Zeile aus dem Ergebnis. Mit `cut` filtern Sie nun die fünfte Spalte aus dem Ergebnis, wobei Sie / als Trennzeichen betrachten:

```
$ ping -c 4 google.com | grep avg | cut -d '/' -f 5

  16.431
```

Sofern Sie bei `cut` die zusätzliche Option `-s` angeben, können Sie sogar auf `grep` verzichten. `-s` bewirkt, dass `cut` nur solche Ergebniszeilen berücksichtigt, in denen das Spaltentrennzeichen (hier also /) vorkommt.

```
$ ping -c 4 google.com | cut -s -d '/' -f 5
```

### ping-Aufruf per Script

Jetzt können Sie den `ping`-Aufruf in ein Script verpacken und den Hostnamen des Rechners übergeben. Das Script überprüft, dass genau ein Parameter übergeben wird, und gibt das Ergebnis ordentlich formatiert aus:

Persönliches Exemplar für Thomas Bartholomäus

```
# Beispieldatei ping-avg.sh
if [ $# -ne 1 ]; then
  echo "usage: ./ping-avg.sh <hostname>"
  exit 2
else
  hostname=$1
fi
avg=$(ping -c 4 $hostname | cut -s -d '/' -f 5)
echo "Average ping time for $hostname is $avg ms"
```

Bei meinem Testaufruf reagierte der Server von `python.org` etwas schneller als der von Google:

```
$ ./ping-avg.sh python.org

  Average ping time for python.org is 12.796 ms
```

Eine denkbare Erweiterung des Scripts wäre die Verarbeitung von mehreren Hostnamen. Das Script erwartet dann also beliebig viele Parameter, zumindest aber einen:

```
#!/bin/bash
# Beispieldatei ping-avg.sh
if [ $# -lt 1 ]; then
  echo "usage: ./ping-avg.sh <hostnames>"
  exit 2
fi
for hostname in $*; do
  avg=$(ping -c 4 $hostname | cut -s -d '/' -f 5)
  echo "Average ping time for $hostname is $avg ms"
done
```

So sieht ein Testaufruf aus:

```
$ ./ping-avg.sh apple.com dell.com lenovo.com

  Average ping time for apple.com is 22.675 ms
  Average ping time for dell.com is 259.528 ms
  Average ping time for lenovo.com is 12.251 ms
```

## 8.4 Beispiel: Apache-Log-Analyse

Als Ausgangspunkt für dieses Beispiel dient eine Logging-Datei des Webservers Apache im sogenannten *Combined-Logging*-Format. Die ersten Zeilen dieser Datei sehen wie folgt aus (hier aus Platzgründen umbrochen):


© Rheinwerk Verlag, Bonn 2024

```
65d3:f5b9:e9e5:4b1c:331b:29f3:97c1:c18f - - [05/Feb/2023:00:00:22
    +0100]
  "POST /consumer/technology HTTP/1.1" 200 5166
  "https://example.com/land/raise/authority/him"
  "WordPress/6.1.1; https://example.com"

221.245.9.91 - - [05/Feb/2023:00:00:21 +0100] "GET /explain/out
    HTTP/1.1"
  200 31222 "https://example.com/add/maybe/person"
  "FreshRSS/1.20.2 (Linux; https://freshrss.org)"

37e3:498f:44d8:c471:3ba3:9902:17e7:2be8 - -
  [05/Feb/2023:00:00:24 +0100]
  "GET /by/financial/within/benefit HTTP/1.1" 200 97598
  "https://example.com/defense/friend/race/protect"
  "Mozilla/5.0 (iPhone; CPU iPhone OS 16_3 like Mac OS X)
   AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.3
   Mobile/15E148 Safari/604.1"
```

Die Spalten dieser Datei geben an, von welcher IP-Adresse der Request erfolgte, welcher Benutzer (falls bekannt) den Request durchgeführt hat, um welche Art von Request es sich handelt (GET, POST etc.), welche Adresse des Webservers angefordert wurde, welcher HTTP-Code zurückgegeben wurden (z. B. 200, also »OK«), welche Datenmengen übertragen wurden, von welcher Seite der Request ausgelöst wurde (Referrer-Adresse) und um welchen Client es sich handelt.

Um die folgenden Beispiele nachzuvollziehen, brauchen Sie `access.log` aus den Beispieldateien. `wc` zeigt, dass die Datei über 170.000 Zeilen umfasst.

```
$ wc -l access.log

  172607 access.log
```

### Anonymisierte Daten

Bei `access.log.gz` handelt es sich um eine echte Logging-Datei, die aber aus Datenschutzgründen anonymisiert wurde. Sämtliche IP-Adressen, URLs und Benutzernamen wurden durch zufällige Daten ersetzt. Diese Anonymisierung ist natürlich auch durch ein Script erfolgt. Wenn Sie sich für den Code interessieren, sollten Sie einen Blick auf `anonymize-log.py` werfen, das intensiven Gebrauch von regulären Mustern macht. Falls Sie Zugang zu »echten« Logging-Dateien von einem eigenen Webserver haben, sollten Sie damit arbeiten.

### IP-Adressen extrahieren

Mit `cut` können Sie aus der Logging-Datei die IP-Adressen extrahieren:

```
$ cut -d ' ' -f 1 access.log | less

  65d3:f5b9:e9e5:4b1c:331b:29f3:97c1:c18f
  221.245.9.91
  37e3:498f:44d8:c471:3ba3:9902:17e7:2be8
  135.84.251.52
  ...
```

Mit `grep` können Sie herausfinden, wie viele Zugriffe auf den Webserver über IPv4 bzw. IPv6 erfolgen:

```
$ cut -d ' ' -f 1 access.log | grep ':' | wc -l     # IPv6
  65405
$ cut -d ' ' -f 1 access.log | grep -v ':' | wc -l  # IPv4
  107202
```

Eine geordnete Liste aller IPv4-Adressen liefern `sort` und `uniq`:

```
$ cut -d ' ' -f 1 access.log | grep -v ':' | sort  |  uniq

  0.117.133.93
  0.125.223.169
  0.164.206.211
  ...
```

Während manche IP-Adressen nur ein einziges Mal im Log auftauchen, sind andere besonders häufig. `sort` sortiert alle IP-Adressen, `uniq -c` zählt die Adressen, `sort -n -r` sortiert deren Häufigkeit abfallend, und `head -n 5` liefert extrahiert schließlich die Top-5-Ergebnisse:

```
$ cut -d ' ' -f 1 access.log | sort | uniq -c | sort -n -r | \
  head -n 5

  6166 65d3:f5b9:e9e5:4b1c:331b:29f3:97c1:c18f
  6048 3547:0b26:4c84:4411:0f66:945e:7741:d887
  5136 186.107.89.128
  4620 d741:a4ea:f6e1:6a17:78b1:1694:f518:c480
  2362 168.81.233.22
```

### Beliebte Seiten erkennen

Im Combined Format befinden sind die Request-Daten in Anführungszeichen eingeschlossen. Mit `cut -d '"'` können Sie die Requests von den restlichen Daten trennen:


© Rheinwerk Verlag, Bonn 2024

```
$ cut -d '"' -f 2 access.log | less

  POST /consumer/technology HTTP/1.1
  GET /explain/out HTTP/1.1
  GET /by/financial/within/benefit HTTP/1.1
  GET /action/but HTTP/1.1
  ...
```

Uns interessieren nur die GET-Requests, und bei diesen wiederum nur die Adresse:

```
$ cut -d '"' -f 2 access.log | grep GET | cut -d ' ' -f 2 | less

  /explain/out
  /by/financial/within/benefit
  /action/but
  /place/paper
  ...
```

Die bewährte Kombination aus `sort`, `uniq -c` und nochmals `sort` ermittelt die beliebtesten URLs:

```
$ cut -d '"' -f 2 access.log | grep GET | cut -d ' ' -f 2 | \
  sort | uniq -c | sort -n -r | head

  25423 /explain/out
   5822 /international/hundred/can
   4979 /goal/Congress/short/peace
   4874 /rate/victim/detail
    ...
```

Wenn Sie dieses Kommando auf die originale Logging-Datei anwenden (die ich leider nicht weitergeben kann), ergibt sich das folgende Ergebnis:

```
$ cut -d '"' -f 2 access.log.orig | grep GET | \
  cut -d ' ' -f 2 | sort | uniq -c | sort -n -r | head

  25424 /feed/
   5822 /
   4979 /wp-content/plugins/wp-spamshield/js/jscripts-ftr2-min.js
   4874 /favicon.ico
   4347 /blog/feed/
   3352 /wp-content/uploads/fonts/49
    ff7721659f0bc7d77a59e1de422e07/font.css?v=1664309579
   3328 /wp-content/themes/twentyfourteen/genericons/genericons.
    css?ver=3.0.3
    ...
```

Persönliches Exemplar für Thomas Bartholomäus

Der Webserver hostet also eine WordPress-Website. Die häufigsten Zugriffe betreffen eine Feed-Seite, die Startseite (/), ein Spam-Schutz-Plugin, das Website-Icon, eine weitere Feed-Seite.

## 8.5 CSV-Dateien

CSV steht für *Comma Separated Values*. Es handelt sich also um ein Textformat, bei dem die Spalten durch Kommata getrennt sind. Länderspezifisch sind auch andere Trennzeichen gebräuchlich, im deutschen Sprachraum insbesondere das Semikolon. Es hilft bei der eindeutigen Differenzierung zwischen Spalten und dem Nachkommaanteil von Zahlen. Grundsätzlich ist jedes Trennzeichen erlaubt, also auch ein Doppelpunkt oder ein Tabulatorzeichen (`\t`).

Die Auswertung von CSV-Dateien wird deutlich komplizierter, wenn das Trennzeichen auch *in* den Spalten vorkommen darf. Die Spalteninhalte werden dann in Anführungszeichen gestellt, normalerweise mit ' oder ".

Ausgangspunkt für die folgenden Beispiele ist die Datei `2022_population.csv` (Quelle: *https://www.kaggle.com/datasets/rsrishav/world-population*). Die Datei enthält Daten zur Bevölkerung der Länder der Erde und hat den folgenden Aufbau:

```
iso_code,country,2022_last_updated,2021_population,area_sq_km,
  land_area_sq_km,density_/sq_km,growth_rate,world_%,rank

CHN,China,"1,425,849,077","1,425,893,465",9.7M,9.4M,151,0.00%,
  17.88%,1

IND,India,"1,422,076,805","1,407,563,842",3.3M,3M,477,0.68%,
  17.77%,2

...
```

Das Kommando `cut` zur Spaltenextraktion ist mit diesen Daten überfordert. Wenn Sie in der Bash arbeiten möchten, können Sie das `csvkit` installieren. Dabei handelt es sich um eine Sammlung von Kommandos zur Verarbeitung von CSV-Dateien (siehe *https://csvkit.readthedocs.io*). In diesem Abschnitt möchte ich stattdessen aber kurz auf die CSV-Features von Python und der PowerShell eingehen.

### CSV-Dateien mit Python verarbeiten

In einem Python-Script können Sie CSV-Daten im einfachsten Fall zeilenweise aus einer Textdatei lesen und die Zeilen dann mit `split` in ihre Spalten zerlegen. Das folgende Script wertet nochmals die Datei `access.log` aus Abschnitt 8.4, »Beispiel: Apache-Log-Analyse«, aus und ermittelt die häufigsten IP-Adressen. Im Unterschied


© Rheinwerk Verlag, Bonn 2024

zu den äquivalenten Bash-Kommandos ist das Python-Script vor allem bei großen Log-Dateien effizienter, weil keine Notwendigkeit besteht, sämtliche IP-Adressen zu sortieren.

```
# Beispieldatei analyze-access-log.py
counters = {}   # Dictionary mit Zählern für alle IP-Adressen

# Schleife über alle Zeilen von access.log
with open('access.log') as f:
    for line in f:
        # die IP-Adresse befindet sich in der ersten Spalte
        ip = line.split()[0]
        if ip in counters:
            counters[ip] += 1
        else:
            counters[ip] = 1

# geordnete Liste der Dictionary-Elemente erzeugen;
# jedes Listenelement ist selbst eine Liste: [ip, cnt]
sortedIps = sorted(counters.items(),
                   key = lambda x: x[1], reverse = True)

# Top-5-IP-Adressen anzeigen
for i in range(5):
    print('%6d: %s' % (sortedIps[i][1], sortedIps[i][0]))
```

Die oben eingesetzte Methode `split` zur Extraktion der Spalten scheitert allerdings bei komplexen CSV-Dateien wie `2022_population.csv`. Um solche Dateien einzulesen, verwenden Sie am besten das Modul `csv`. Das folgende Listing zeigt eine einfache Anwendung des Moduls.

Die Dokumentation zum `csv`-Modul empfiehlt, die CSV-Datei mit der Option `newline=''` zu öffnen, um allfälligen Problemen mit der Zeilenendekennzeichnung unter Windows (also `\r\n`) aus dem Weg zu gehen. Beim Öffnen des CSV-Readers geben Sie mit den Optionen `delimiter` und `quotechar` an, welche Zeichen Spalten bzw. Zeichenketten kennzeichnen. Anschließend können Sie mit dem Reader die CSV-Datei in einer Schleife zeilenweise auslesen. Die Schleifenvariable enthält jeweils eine Liste mit den Spalten dieser Zeile. Die erste Zeile mit der Spaltenbeschriftung wird mit `next` übersprungen.

```
# Beispieldatei analyze-population.py
import csv
total = 0
with open('2022_population.csv', newline = '') as f:
    reader = csv.reader(f, delimiter = ',', quotechar = '"')
```

Persönliches Exemplar für Thomas Bartholomäus

```
    next(reader)             # Spaltenbeschriftung überspringen
    for columns in reader:   # zeilenweise lesen, columns enthält
        print(columns)       # Liste mit Spalten
        total += int(columns[2].replace(',', ''))

print("Earth population:", total)
```

Um die gesamte Erdbevölkerung zu errechnen, werden die Einwohnerzahlen der dritten Spalte in Zahlen umgewandelt und summiert. `replace` eliminiert die zur Tausendertrennung verwendeten Kommas.

### CSV-Dateien in der PowerShell verarbeiten

In der PowerShell verwenden Sie `Import-Csv` zum Einlesen von CSV-Dateien. Standardmäßig erwartet `Import-Csv`, dass die Spalten durch Kommata getrennt sind. Mit `-Delimiter` kann bei Bedarf ein anderes Zeichen eingestellt werden. Zeichenketten, die in doppelten Anführungszeichen stehen, werden korrekt verarbeitet. Es gibt aber keine Möglichkeit, ein anderes Zeichen zur Kennzeichnung von Zeichenketten einzustellen.

Als Ergebnis liefert das CmdLet `PSCustomObject`-Objekte zurück, deren Eigenschaften von der Spaltenbeschriftung der CSV-Datei übernommen werden. Sollte die CSV-Datei keine Spaltenbeschriftungszeile haben, müssen Sie die Namen der Spalten mit der Option `-Header` angeben, z. B. so:

```
> Import-Csv -Header colA, colB, colC file.csv
```

Passen Sie auf, dass Sie bei `-Header` nicht zu wenige Spalten benennen: Es werden nur so viele Spalten verarbeitet, wie Sie Namen angeben!

Um wie im vorigen Beispiel die Erdbevölkerung aus `2022_population.csv` zu ermitteln, können Sie so vorgehen:

```
# Beispieldatei analyze-population.ps1
$population = Import-CSV "2022_population.csv"
$total = 0
$population | ForEach-Object {
    $total += [long]($_.'2022_last_updated'.Replace(',', ''))
}
Write-Output "World population: $total"
```

Innerhalb der Schleife werden in der Zeichenkette der Eigenschaft `2022_last_updated` die Kommata entfernt. `[long]` wandelt die Zeichenkette in eine Zahl um.

Anstatt selbst zu summieren, können Sie das Ergebnis auch mit `Measure-Object` errechnen. Wegen der benötigten Umwandlung der `2022_last_updated`-Spalte in eine Zahl wird der Code dadurch aber weder kürzer noch effizienter:


© Rheinwerk Verlag, Bonn 2024

```
$total = $population | ForEach-Object {
    [long]($_.'2022_last_updated'.Replace(',', ''))
} | Measure-Object -Sum
Write-Output "World population: $($total.Sum)"
```

Mit `Export-Csv` bietet die PowerShell eine großartige Möglichkeit, CmdLet-Ergebnisse in einer CSV-Datei zu speichern. Die resultierende CSV-Datei zählt in der ersten Zeile zur Spaltenbeschriftung sämtliche Eigenschaften der Ergebnisobjekte auf. In den weiteren Zeilen folgen die Werte der betreffenden Eigenschaften, wobei sämtliche Ausgaben durch Kommata getrennt und in Anführungszeichen gestellt werden:

```
> Get-Process | Export-Csv processes.csv

> Get-Content processes.csv

  "Name","SI","Handles","VM","WS","PM","NPM","Path",...
  "AggregatorHost","0","92","2203375112192","2568192",...
  "ApplicationFrameHost","1","378","2203582570496",...
  ...
```

### Performance-Probleme

Bei meinen Tests dauerte die Ausführung von `Get-Process | Export-Csv` unbegreiflich lange – auf einer Windows-11-Desktop-Installation im Leerlauf ca. 10 Sekunden. `Get-Process` alleine wird in wenigen Millisekunden ausgeführt, daran kann es eigentlich nicht liegen. Die resultierende Datei ist mit ca. 130 kByte winzig. Selbst in einer unter Linux ausgeführten PowerShell dauerte die Ausführung des Kommandos mehrere Sekunden lang.

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

# Kapitel 9
# Reguläre Muster

Als »reguläres Muster« (*Regular Expression*, kurz *regex*) wird eine Zeichenkette bezeichnet, die ein Suchmuster für Text beschreibt. Dabei müssen spezielle, wenig intuitive Regeln eingehalten werden. Um gleich mit einem Beispiel zu beginnen: Das folgende Muster erkennt ein Datum im US-Format (also mm/dd/yyyy):

```
(0[1-9]|1[0-2])\/(0[1-9]|1\d|2\d|3[01])\/(19|20)\d{2}
```

Das Muster akzeptiert Monate zwischen 01 und 12, Tage zwischen 01 und 31 sowie Jahreszahlen zwischen 1900 und 2099. Eine Verifizierung, ob das Datum wirklich korrekt ist, bleibt Ihnen aber nicht erspart – das Muster akzeptiert auch das Datum 02/30/2023, obwohl der Februar nie 30 Tage haben kann.

Reguläre Muster haben eine relativ hohe Einstiegshürde. Sie müssen zuerst die Syntax lernen und üben. Das kostet ein paar Stunden Zeit. Aber wenn Sie einmal mit regulären Mustern vertraut sind, gibt es schier unendlich viele Anwendungsmöglichkeiten – und zwar in allen gängigen Programmiersprachen. Ja, manche Aufgabenstellungen lassen sich auch ohne reguläre Muster lösen. Aber wenn Sie mit der Syntax vertraut sind, führen reguläre Muster schneller und zuverlässiger zum Ziel und vermeiden Spaghetti-Code!

Dieses Kapitel beginnt mit den Syntaxregeln, wobei ich die wichtigsten Syntaxvarianten berücksichtige (POSIX, POSIX extended und PCRE). Sie hatten doch nicht erwartet, dass für jedes Tool bzw. für jede Programmiersprache dieselben Regeln gelten, oder? Keine Angst, die Grundregeln sind immer gleich, die Abweichungen zwischen den RegEx-Varianten überschaubar. In der Folge zeige ich Ihnen anhand diverser Beispiele, wie Sie RegEx in Bash-Scripts (speziell in den Kommandos `grep` und `sed`), in Python und der PowerShell anwenden.

### Voraussetzungen für dieses Kapitel

Große Teile dieses Kapitels gelten unabhängig davon, welches Ihre Lieblings-Script-Sprache ist. Erst in den sprachspezifischen Abschnitten benötigen Sie Grundkenntnisse in der Bash, der PowerShell oder in Python. Eine gute logische Fortsetzung nach der Lektüre dieses Kapitels ist Kapitel 17, »Web Scraping«. Dort finden Sie nochmals einige Anwendungsbeispiele, aber auch Hinweise auf die Grenzen regulärer Muster.

Persönliches Exemplar für Thomas Bartholomäus

## 9.1 Syntaxregeln für reguläre Ausdrücke

Die ersten Konzepte für reguläre Ausdrücke wurden schon in den 1950er-Jahren formuliert. Im Laufe der Zeit entwickelten sich mehrere Dialekte bzw. Syntaxvarianten:

- **POSIX:** Erstmalig standardisiert wurden reguläre Ausdrücke im Unix-Umfeld als *POSIX Regular Expression*, wobei POSIX wiederum für *Portable Operating System Interface* steht.
- **POSIX Extended Regular Expressions (ERE):** In dieser Erweiterung wurden unter anderem Bezüge auf schon verarbeitete Muster (*back references*) sowie Codes für Zeichenklassen (Ziffern, Buchstaben) definiert.
- **Perl Compatible Regular Expressions (PCRE):** Die Script-Sprache Perl hat ihren Zenit zwar schon überschritten, aber die in Perl gültige Syntax für reguläre Ausdrücke hat sich als Maßstab etabliert. Die meisten populären Programmiersprachen (Python, .NET-Sprachen inklusive der PowerShell, JavaScript etc.) sind bis auf wenige Details PCRE-kompatibel.

  Um das Ganze auf die Spitze zu treiben, gibt es *zwei* PCRE-Varianten, einfach »PCRE« (seit 1997) sowie »PCRE 2« (seit 2015).

In diesem Kapitel stelle ich reguläre Muster ohnedies nur verkürzt dar, verzichte also auf die Diskussion einiger fortgeschrittener Features. Soweit ich nicht explizit auf andere Dialekte hinweise, beziehe ich mich dabei auf PCRE oder den großen gemeinsamen Nenner aller Regex-Varianten.

Lassen Sie sich von den verschiedenen Regex-Dialekten nicht abschrecken: Die vergleichsweise wenigen Unterschiede betreffen überwiegend fortgeschrittene Features. In vielen Fällen ist es möglich, ein für einen bestimmten Regex-Dialekt formuliertes Muster an einen anderen Dialekt anzupassen.

#### Vergleichstabelle

Für den Großteil der Anwendungsfälle sollte das in diesem Abschnitt vermittelte Basiswissen ausreichen. Nur bei der Nutzung von Spezialfunktionen bzw. von sprach- oder frameworkspezifischen Zusatzfunktionen bleibt Ihnen ein Blick in die jeweilige Dokumentation nicht erspart. Wenn Sie sich partout für die Unterschiede interessieren, suchen Sie im Internet nach dem *Regular Expression Engine Comparison Chart*. Die Adresse dieser ausgezeichneten Vergleichsseite ist ebenso abschreckend wie ihr Inhalt, der dafür aber exzellent recherchiert ist:

*https://gist.github.com/CMCDragonkai/6c933f4a7d713ef712145c5eb94a1816*

Die Unterschiede zwischen PCRE und PCRE 2 sind gut auf StackOverflow zusammengefasst:

*https://stackoverflow.com/questions/70273084*


© Rheinwerk Verlag, Bonn 2024

### Zeichen

In regulären Ausdrücken gibt es diverse Möglichkeiten, um ein einzelnes Zeichen bzw. ein Zeichen aus einer ganzen Gruppe auszudrücken (siehe Tabelle 9.1). Grundsätzlich differenzieren reguläre Muster zwischen Groß- und Kleinschreibung, wobei dieses Verhalten je nach Tool oder Programmiersprache oft durch Optionen deaktiviert werden kann. Das Muster `[Hh][ae]lle` trifft somit auf die Zeichenketten »Hallo«, »Hello«, »hallo« oder »hello« zu, aber nicht auf »HALLO«. Das Muster `[0-9a-fA-F]` erkennt eine hexadezimale Ziffer.

Beachten Sie, dass `a-z` zwar alle Kleinbuchstaben des ASCII-Zeichensatzes erfasst, nicht aber ä, ö, ü oder ß. Diese Buchstaben müssen Sie gegebenenfalls extra hinzufügen, z. B. `[a-zA-ZäöüßÄÖÜ]`. Wenn der Bindestrich selbst ein erlaubtes Zeichen sein soll, geben Sie diesen zum Schluss an, z. B. `[a-zA-Z-]` (ASCII-Klein- und Großbuchstaben plus Bindestrich).

| Muster | Bedeutung |
|---|---|
| `.` | ein beliebiges Zeichen |
| `[aeiou]` | a, e, i, o oder u |
| `[^aeiou]` | nicht a, e, i, o oder u |
| `[a-g]` | a bis g |
| `[0-9]` | 0 bis 9 |
| `\.` | ein Punkt; generell muss \ diversen Regex-Sonderzeichen vorangestellt werden, damit diese ihre Spezialbedeutung verlieren |

**Tabelle 9.1** Codes für Zeichen

PCRE kennt diverse Kürzel für Zeichenklassen (siehe Tabelle 9.2).

| Muster | Bedeutung |
|---|---|
| `\d` | Ziffer, gleichwertig zu `[0-9]` |
| `\D` | keine Ziffer, gleichwertig zu `[^0-9]` |
| `\s` | Whitespace (Leerzeichen, Tabulator etc.) |
| `\S` | kein Whitespace |
| `\w` | Buchstabe oder Ziffer, entspricht `[a-zA-Z0-9_]` |
| `\W` | nicht Buchstabe/Ziffer |

**Tabelle 9.2** PCRE-Codes für Zeichenklassen

Persönliches Exemplar für Thomas Bartholomäus

In den POSIX Regular Expressions gelten abweichende Codes für Zeichenklassen (siehe Tabelle 9.3). Beachten Sie, dass diese Codes selbst wieder in eckigen Klammern verwendet werden, also z. B. `[[:upper:]]`! Da ist es platzsparender, einfach `[a-z]` zu schreiben. Das Muster `[[:lower:]äöüß]` erkennt Kleinbuchstaben inklusive der deutschen Buchstaben ä, ö, ü und ß.

| Muster | Bedeutung |
|---|---|
| `[:upper:]` | Großbuchstaben, entspricht `[A-Z]` |
| `[:lower:]` | Kleinbuchstaben, entspricht `[a-z]` |
| `[:alpha:]` | Buchstaben, entspricht `[A-Za-z]` |
| `[:alnum:]` | Buchstaben und Ziffern, entspricht `[A-Za-z0-9]` |
| `[:word:]` | Buchstaben, Ziffern und Unterstrich, wie `\w` |
| `[:digit:]` | Ziffern, entspricht `[0-9]` bzw. `\d` |
| `[:xdigit:]` | hexadezimale Ziffern, entspricht `[0-9a-fA-F]` |
| `[:punct:]` | Punktationszeichen (».,:;« usw.) |
| `[:blank:]` | Leerzeichen und Tabulator |
| `[:space:]` | Leerzeichen, Tabulator, Zeilenumbruchzeichen |
| `[:cntrl:]` | Steuerzeichen wie Ctrl-C |
| `[:graph:]` | Zeichen mit grafischer Entsprechung (entspricht `[^[:cntrl:]]`) |

**Tabelle 9.3** POSIX-Codes für Zeichenklassen, müssen innerhalb von »[...]« eingesetzt werden

## 9.2 Gruppen und Alternativen

Mit `()` bilden Sie innerhalb des Musters eine Gruppe (siehe Tabelle 9.4). Innerhalb des Musters können Sie mit `|` alternative Muster formulieren, von denen wahlweise eines zutrifft. In der Folge können Sie je nach Regex-Dialekt mit `\1`, `\2` usw. oder mit `$1`, `$2` usw. auf den Inhalt erkannter Gruppen zugreifen. Dafür gibt es diverse Anwendungen, z. B. die Erkennung von Wiederholungen oder die Ausführung von Suchen-und-Ersetzen-Operationen.

Die folgenden drei Beispiele verdeutlichen die Anwendung von Gruppen. Dabei muss ich allerdings mit `+` und `{1,2}` auf den nächsten Abschnitt vorgreifen, in dem es um Quantifizierer geht. `+` bedeutet, dass das vorangestellte Muster einmal oder öfter


© Rheinwerk Verlag, Bonn 2024

vorkommen muss. {1,2} bedeutet, dass das vorangestellte Muster exakt ein- oder zweimal vorkommen muss.

- Reguläres Muster: ([1-9]|10|11|12)

  Dieses Muster trifft auf die Zahlen 1, 2, 3 bis 12 zu, aber nicht auf 0, nicht auf 01 und auch nicht auf 13.

- Reguläres Muster: ([0-9]+) \1 bzw. ([0-9]+) $1 bei PCRE 2

  Der erste Teil dieses Musters trifft auf jede ganze Zahl zu, z. B. auf 0, 1, 34 oder 789123. Im zweiten Teil des Musters wird auf diese Gruppe nochmals Bezug genommen, d. h., dieselbe Zahl muss nach einem Leerzeichen noch einmal angegeben werden.

  Das Muster trifft auf diese Zeichenketten zu: 123 123, 7 7, 456 456 und abc1 1abc, wobei im letzten Fall nur die Teilzeichenkette 1 1 erkannt wird.

  Das Muster trifft aber nicht zu auf 123123 (kein Leerzeichen) oder auf 123   123 (zu viele Leerzeichen).

- Reguläres Muster: #([0-9a-fA-F]{1,2})\1\1

  Dieses Muster erkennt CSS-Farbcodes für Grauwerte. Solche Farbcodes werden mit dem Zeichen # eingeleitet und dürfen drei oder sechs hexadezimale Stellen umfassen. Damit es ein Grauwert ist, müssen Rot-, Grün- und Blauanteil jeweils gleich sein. Dazu dient die zweifache Wiederholung der ersten Gruppe.

  Das Muster trifft z. B. auf die folgenden Farb-Codes zu: #000, #222222, #ababab, #777 und #777777.

  Es trifft nicht zu auf #aabbcc, #abc, #0000 (falsche Stellenanzahl) oder 000 (hier fehlt das einleitende Zeichen #).

  Das Muster versagt leider bei #aAa. Das ist ein korrekter Grauwert, aber a (erkannte Gruppe) und A (erste Wiederholung) stimmen bezüglich der Groß- und Kleinschreibung nicht überein.

| Muster | Bedeutung |
|---|---|
| (...) | alles zwischen ( und ) bildet eine Gruppe |
| (abc) | das Muster abc als Gruppe |
| (abc\|efg) | Alternative: Muster abc oder efg |
| \1, \2 | Verweis auf den Inhalt der 1. oder 2. Gruppe (POSIX, PCRE) |
| $1, $2 | Verweis auf den Inhalt der 1. oder 2. Gruppe (PCRE 2) |

**Tabelle 9.4** Regex-Gruppen

Persönliches Exemplar für Thomas Bartholomäus

### Quantifizierer

Mit Quantifizierern drücken Sie aus, wie oft ein Muster vorkommen darf bzw. muss (siehe Tabelle 9.5). Sie können Quantifizierer auf Zeichenbereiche oder Gruppen anwenden. Vorsicht: abc+ bedeutet zuerst ab, dann beliebig oft c (aber mindestens einmal). Wenn dagegen abc beliebig oft vorkommen darf, müssen Sie das Muster mit (abc)+ ausdrücken.

| Muster | Bedeutung |
|---|---|
| x? | null oder ein Mal die durch x beschriebenen Zeichen |
| x* | null Mal oder öfter (beliebig oft) |
| x+ | ein Mal oder öfter (aber nicht null Mal!) |
| x{3} | genau drei Mal |
| x{3,5} | drei bis fünf Mal |
| x{3,} | mindestens drei Mal |

**Tabelle 9.5** Quantifizierer für Regex-Muster

Wie die folgenden Beispiele zeigen, sind Quantifizierer einfach anzuwenden:

- [a-f]: Ohne Quantifizierer verlangt dieses Muster genau ein Zeichen a bis f.
- [a-f]?: Mit ? wird eines der Zeichen oder gar keines erwartet.
- [a-f]+: Mit + sind beliebig viele Zeichen zwischen a und f zulässig – aber zumindest ein Zeichen muss angegeben werden. a, aaa, abcdef sind in Ordnung. abcx trifft auch zu, aber das Muster umfasst nur die ersten drei Buchstaben, nicht den Buchstaben x.
- .*: Dieses sehr offene Muster erlaubt beliebig viele (auch null) beliebige Zeichen.
- (abc|efg){4}: Bei diesem Muster muss genau vier Mal abc oder efg angegeben werden. abcabcefgabc wäre passend, abcefg aber nicht.

### Über die Gier (Greedy vs. Lazy)

Nein, dieser Abschnitt beschreibt weder menschliche Eigenschaften noch die von Wirtschaftssystemen. Vielmehr geht es darum, wie groß die durch ein Muster erfasste Zeichenkette sein soll. Standardmäßig erfolgt die Auswertung von regulären Mustern »gierig« (*greedy*), d. h., die Regex-Funktion gibt möglichst große passende Zeichenketten zurück.


© Rheinwerk Verlag, Bonn 2024

Nehmen wir an, das Muster lautet <.+>. Es soll also ein HTML-Tag erkannt werden, das mit < beginnt und mit > endet. Dazwischen dürfen beliebig viele Zeichen stehen (mindestens eines).

Wenn Sie dieses Muster auf die Zeichenkette <html><body><p>lorem ipsum<p>dolores est anwenden, lautet das Ergebnis <html><body><p>lorem ipsum<p>. Das wird in den meisten Fällen nicht Ihren Erwartungen entsprechen.

Glück haben Sie, wenn die Regex-Funktionen Ihrer Tools PCRE-konform sind. Dann geben Sie einfach nach dem Quantifizierer ein Fragezeichen an. Der Ausdruck wird nun *lazy* oder *non-greedy* ausgewertet, also so minimalistisch wie möglich. Das folgende Listing verdeutlicht das:

```
Muster:          <.+?>

Text:            <html><body><p>lorem ipsum<p>dolores est

Erster Treffer:  <html>
Zweiter Treffer:       <body>
Dritter Treffer:             <p>
Vierter Treffer:                          <p>
```

Im Bash-Umfeld ist die PCRE-Syntax leider selten eine Option. Viele Unix-Tools sind lediglich POSIX-konform. In solchen Fällen müssen Sie das Muster exakter formulieren. In unserem Beispiel beschreiben Sie das Innere des HTML-Tags statt durch .+? mit [^>]+. Im Inneren sind somit alle Zeichen außer > erlaubt. Das gesamte Muster lautet dann <[^>]+>.

### Faul ist effizient ...

Sie haben es vielleicht immer schon vermutet, aber Übereifer zahlt sich selten aus. Ganz so allgemeingültig lässt sich das für reguläre Ausdrücke zwar nicht sagen, aber in der Praxis sind *lazy* formulierte Muster oft effizienter. Das gilt insbesondere dann, wenn ein komplexes Muster auf eine lange Zeichenkette angewendet werden soll. Die Anzahl der möglichen Fälle und der Umfang des auszuwertenden Texts sinken.

### Reguläre Muster und HTML: Eine schlechte Kombination!

Widerstehen Sie der Versuchung, HTML-Code mit regulären Mustern auszuwerten. Das geht meistens schief, weil HTML keine (im wissenschaftlichen Sinn) »reguläre« Sprache ist. Zur Auswertung von HTML-Dokumenten sollten Sie Parser verwenden, die HTML-Code verstehen und daraus im Idealfall ein *Document Object Model* machen. In Python erfüllt z. B. das Modul BeautifulSoup diese Aufgabe. Beispiele folgen in Kapitel 17, »Web Scraping«.

Persönliches Exemplar für Thomas Bartholomäus

### Alpha und Omega

Bis jetzt haben wir den Ort ignoriert, wo ein Muster im Text erkannt wird. Mit den Zeichen ^ und $ können Sie zum Ausdruck bringen, dass das Muster am Beginn oder Ende des Texts erkannt wird (siehe Tabelle 9.6).

| Muster | Bedeutung |
|---|---|
| x | Das Muster x darf irgendwo im Text auftreten. |
| ^x | Das Muster muss am Beginn des Texts auftreten. |
| x$ | Das Muster muss am Ende des Texts auftreten. |
| ^x$ | Das Muster muss dem gesamten Text entsprechen. |

**Tabelle 9.6** Position regulärer Muster

> **Mehrzeiliger Text**
>
> Wenn reguläre Muster auf längere Texte oder Textdateien angewendet werden, dann wird üblicherweise jede Zeile für sich betrachtet. Manche Regex-Werkzeuge oder -Funktionen kennen Optionen, um die Mustererkennung zeilenübergreifend durchzuführen. Führen Sie aber unbedingt Performance-Tests durch, bevor Sie eine derartige Option leichtfertig anwenden!

### Übungsseiten und Cheatsheets

Bevor Sie reguläre Muster einigermaßen sicher anwenden können, müssen Sie üben! Es gibt im Internet gleich mehrere großartige Seiten, wo Sie eine Reihe von Übungen mit zunehmender Komplexität lösen können – samt Auflösung und Erklärung. Investieren Sie eine Stunde auf einer dieser Seiten – ich verspreche Ihnen, es lohnt sich!

*https://regexone.com*
*http://regextutorials.com*

Für alle, die schon etwas Routine mit regulären Mustern haben, gibt es Webseiten, in denen Sie reguläre Muster eingeben und unmittelbar auf Testdaten anwenden können (siehe Abbildung 9.1). Wenn ich komplizierte Muster entwickeln muss, verwende ich *immer* eine derartige Seite, bevor ich den dann funktionierenden Code in mein Script einbaue! Beachten Sie aber, dass beide Seiten nur PCRE-Dialekte unterstützen.

*https://regex101.com*
*https://regexr.com*


© Rheinwerk Verlag, Bonn 2024

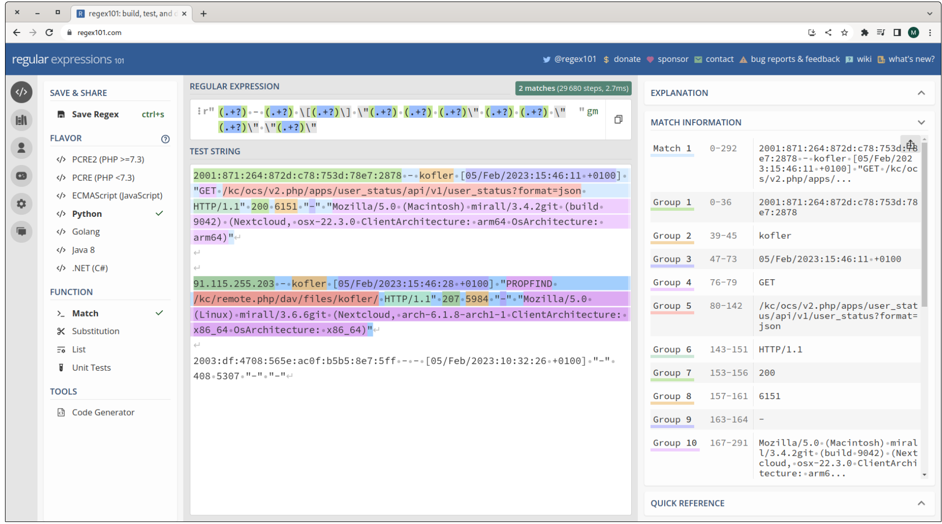



**Abbildung 9.1** Test eines regulären Musters zur Verarbeitung von Apache-Log-Dateien

Die allerwichtigsten Regex-Zeichen habe ich im Laufe der Jahre verinnerlicht, aber bei den Details muss ich doch immer wieder nachlesen. Praktisch sind sogenannte »Cheatsheets«, die eine Referenz der wichtigsten Codes auf einer Seite zum Ausdrucken zusammenfassen:

*https://github.com/shadowbq/Cheat-Sheets/raw/master/regex/regular-expressions-cheat-sheet-v2.pdf*
*https://jamesbachini.com/resources/REGEXCheatSheet.pdf*

### Reguläre Ausdrücke mit künstlicher Intelligenz zusammensetzen

Auch KI-Tools wie ChatGPT verstehen reguläre Ausdrücke. Die Eingabe »Kannst du mir ein reguläres Muster zur Erkennung von E-Mail-Adressen zeigen?« führt direkt zum Ziel – samt Erläuterungen, welche Bedeutung die einzelnen Zeichen des Musters haben. Überprüfen Sie besonders bei komplexen Aufgaben aber, ob das Ergebnis wirklich Ihren Anforderungen entspricht! ChatGPT präsentiert nämliche gerne mit großem Selbstvertrauen Lösungen, die dann doch nicht alle Matches abdecken oder zu viel filtern.

## Beispiel: Datum erkennen

Nehmen wir an, Ihr Script soll ein Datum im ISO-Format (yyyy-mm-dd) verarbeiten, sei es als Eingabe oder aus einer Datei. Ein einfaches reguläres Muster, das prinzipiell funktioniert, sieht so aus: `\d{4}-\d{2}-\d{2}`

Persönliches Exemplar für Thomas Bartholomäus

Das Datum besteht aus acht Ziffern, getrennt von zwei Bindestrichen. Allerdings akzeptiert das Muster auch vollkommen irre Daten, z. B. `4375-52-68`. Mit nur wenig Mühe können Sie die zulässigen Wertbereiche ganz gut abgrenzen, hier auf Jahreszahlen zwischen 0000 und 2999, auf Monate zwischen 01 und 12 sowie auf Tage zwischen 0 und 31. Das ist nicht perfekt, verhindert aber zumindest komplette Fehleingaben.

```
[012]\d{3}-(0[1-9]|10|11|12)-([012]\d|30|31)
```

### Beispiel: IPv4-Adressen erkennen

Auch bei IPv4-Adressen können Sie mehr oder weniger exakt vorgehen. Um IP-Adressen aus einer Logging-Datei zu extrahieren, die ohnedies nur korrekte Adressen enthalten wird, reicht das folgende Muster:

```
(\d{1,3}\.){3}\d{1,3}
```

- `\d{1,3}` verlangt eine ein- bis dreistellige Zahl.
- `\.` erfordert einen Punkt. Vergessen Sie nicht das vorangestellte `\`-Zeichen. Einfach nur `.` erlaubt jedes beliebige Zeichen!
- `(...){3}` bewirkt eine Verdreifachung dieses Musters, würde bei der IP-Adresse 1.2.3.4 also »1.2.3.« erfassen. Beachten Sie, dass `{4}` falsch wäre, weil die IP-Adresse dann mit einem Punkt enden müsste.
- Jetzt fehlt noch `\d{1,3}` für die letzte Stelle.

Exaktere Muster, die für jede Stelle nur den Zahlenbereich von 0 bis 255 akzeptieren, finden Sie hier:

*https://stackoverflow.com/questions/5284147*

### Beispiel: E-Mail-Adressen erkennen

Zur groben Kontrolle, ob eine E-Mail-Adresse korrekt aussieht, können Sie das folgende Muster verwenden:

```
[a-zA-Z0-9+_.-]+@[a-zA-Z0-9.-]+
```

Auch in diesem Fall finden Perfektionisten auf StackOverflow vollständig unlesbare Muster, die dafür aber alle erdenklichen Regeln und Sonderfälle berücksichtigen.

## 9.3 Reguläre Muster in der Bash (grep, sed)

In Grundzügen sollten Sie reguläre Muster nun verstehen. Ich weise noch einmal darauf hin, dass ich im vorigen Abschnitt nur die wichtigsten Syntaxregeln skizziert


© Rheinwerk Verlag, Bonn 2024

habe. Es gibt hundertseitige Bücher, die sich ausschließlich diesem Thema widmen und alle erdenklichen Varianten behandeln.

In diesem und den folgenden Abschnitten wenden wir uns der Anwendung regulärer Muster zu. Ich beginne hier mit der Bash, wobei ich überwiegend auf die Kommandos `grep` und `sed` fokussieren werde.

### Bash-Vergleichsoperator für reguläre Muster

Auf elementare Weise können die Bash und die Zsh ohne externe Tools reguläre Muster verarbeiten. Dazu verwenden Sie den Vergleichsoperator `=~`, den Sie in doppelten eckigen Klammern anwenden müssen.

```
if [[ string =~ pattern ]] then;
  echo "string matches pattern"
fi
```

Dabei kommen POSIX Extended Regular Pattern zum Einsatz. Sie müssen daher anstelle der Kurzschreibweisen `\d` für *Digit* oder `\w` für *Word* POSIX-konforme Ausdrücke wie `[:digit:]` oder `[:alnum:]` verwenden oder Zeichenbereiche mit `[0-9]` direkt formulieren.

Der Operator `=~` kann beispielsweise dazu verwendet werden, um Eingaben zu validieren. Im folgenden Beispiel erwartet das Script die Eingabe eines Datums in der Form `yyyy-mm-dd`. Wenn schon die Form der Zeichenkette nicht dieser Form entspricht, muss die Eingabe wiederholt werden.

Das Muster erwartet Jahreszahlen im Bereich zwischen 1900 und 2099, Monate zwischen 01 und 12 sowie Tage zwischen 00 und 31. Beachten Sie, dass das Muster einige ungültige Daten akzeptiert, z. B. `2000-01-00` oder `2023-04-31`.

```
# Beispieldatei input-date.sh
pattern='^(19|20)[0-9]{2}-(0[1-9]|10|11|12)-([012][0-9]|30|31)$'
while true; do
    read -p "Enter a date in ISO format (yyyy-mm-dd): " date
    if [[  $date =~ $pattern ]]; then
        break
    else
        echo "Invalid date, please try again."
    fi
done
echo "Valid date: $date"
```

### Text filtern mit »grep«

Das Kommando `grep` ist Ihnen aus Kapitel 6, »Linux-Toolbox«, und Kapitel 8, »Textauswertung mit Filtern und Pipes«, bereits vertraut. Bisher haben wir dieses Kom-

Persönliches Exemplar für Thomas Bartholomäus

mando als simplen Textfilter verwendet. Tatsächlich wird der an `grep` übergebene erste Parameter aber als reguläres Muster ausgewertet, standardmäßig in der einfachen POSIX-Syntax. Das Kommando kommt aber auch mit den Regex-Dialekten POSIX ERE und PCRE zurecht:

- `-e`: POSIX Extended Regular Expressions (ERE) aktivieren
- `-P`: PCRE aktivieren
- `-o` bzw. `--only-matching`: nur den Ausdruck ausgeben, der dem Muster entspricht (also nicht die ganze Zeile anzeigen)

Um zu vermeiden, dass die diversen Regex-Spezialzeichen von der Bash ausgewertet werden, sollten Sie das Muster in Anführungszeichen übergeben.

Das folgende Kommando filtert alle IPv4-Adressen aus der Logging-Datei `access.log` und zeigt nur die Adressen an. (Sie finden die Datei in den Beispieldateien zum vorigen Kapitel.)

```
$ grep -o -P '(\d{1,3}\.){3}\d{1,3}' access.log

  221.245.9.91
  135.84.251.52
  160.85.252.207
  ...
```

Wenn Sie analog alle IPv6-Adressen extrahieren möchten, können Sie `grep` mit dem folgenden Muster aufrufen:

```
$ grep -o -P '([\da-f]*:){1,7}\d{1,4}' access.log
```

Das Suchmuster funktioniert für die meisten »gewöhnlichen« IPv6-Adressen. Die Syntax von IPv6 erlaubt aber diverse Sonderfälle, deren korrekte Verarbeitung wesentlich komplexere (und langsamere!) Muster erfordert. Werfen Sie gegebenenfalls einen Blick auf die folgende StackOverflow-Webseite:

*https://stackoverflow.com/questions/53497*

### Der Stream Editor »sed«

Der *Stream Editor* `sed` ist ein universelles Kommando zur Textverarbeitung. Sie übergeben im ersten Parameter ein Kommando, das `sed` auf die Standardeingabe bzw. an die im nächsten Parameter angegebene Datei anwendet. Der verarbeitete Text wird an die Standardausgabe geschrieben.

Bei der unter Linux üblichen GNU-Variante von `sed` bewirkt die Option `-i` (*in place*), dass die Ursprungsdatei direkt verändert wird. `-i.bak` erzeugt ein Backup der Ursprungsdatei mit der Kennung `.bak`.


© Rheinwerk Verlag, Bonn 2024

### GNU versus BSD

Bei der BSD-Variante von `sed`, die z. B. unter macOS installiert ist, hat die Option `-i` eine andere Bedeutung. Generell gibt es zwischen GNU-`sed` und BDS-`sed` einige weitere kleine Unterschiede. Installieren Sie gegebenenfalls mit `brew` die GNU-Variante von `sed`!

Das folgende Einführungsbeispiel zeigt das Delete-Kommando `d`. Es liest die Datei `myfile.txt`, löscht die Zeilen 3 bis 5 und schreibt alle anderen Zeilen in die Datei `out.txt`:

```
$ sed 3,5d myfile.txt > out.txt
```

Wenn Sie an `sed` mehrere Kommandos übergeben wollen, verwenden Sie `-e`:

```
$ sed -e 'cmd1' -e 'cmd2' in.txt > out.txt
```

Alternativ können Sie die Kommandos in einer eigenen Datei speichern und diese mit `-f` verarbeiten:

```
$ sed -f sedcmds.txt in.txt > out.txt
```

`sed` kann auch reguläre Muster verarbeiten. Standardmäßig kommen dabei POSIX Regular Expressions zum Einsatz. Mit den gleichwertigen Optionen `-E` oder `-r` unterstützt `sed` auch die POSIX Extended Regular Expressions. Anders als bei `grep` gibt es leider keine Option für PCRE.

Beim folgenden Kommando sucht `sed` nach Zeilen, die mit einer IPv4-Adresse beginnen. Diese Zeilen werden gelöscht. Alle anderen Zeilen werden in `ipv6.log` gespeichert. Das Suchmuster wird dabei üblicherweise zwischen zwei Schrägstrichen formuliert. Für den Fall, dass Sie den Schrägstrich selbst im Muster verwenden möchten, können Sie aber auch ein beliebiges anderes Zeichen verwenden, das im Muster nicht vorkommt. Der ganze Ausdruck muss nun aber mit einem Backslash beginnen. Die beiden folgenden Kommandos sind somit gleichwertig:

```
$ sed -E  '/^([0-9]{1,3}\.){3}[0-9]{1,3}/d' access.log > ipv6.log
$ sed -E '\@^([0-9]{1,3}\.){3}[0-9]{1,3}@d' access.log > ipv6.log
```

### Suchen und ersetzen mit »sed«

Zu den wichtigsten `sed`-Kommandos zählt `s/pattern/replace/`. Damit werden in jeder Zeile die durch `pattern` erkannten Ausdrücke durch die `replace`-Zeichenkette ersetzt. Innerhalb von `\replace` können Sie mit `\1`, `\2` auf die durch `pattern` erkannten Gruppen zugreifen.

Auch bei diesem Kommando sind anstelle von / andere Trennzeichen erlaubt, die im Muster nicht vorkommen, also z. B. `s:pattern:replace:`. Das Trennzeichen wird aber

Persönliches Exemplar für Thomas Bartholomäus

nicht extra durch einen Backslash markiert. Dem dritten Trennzeichen können zwei Buchstaben folgen (auch in Kombination):

- `s/ptrn/rpl/g` bedeutet, dass gegebenenfalls mehrere Suchen-und-Ersetzen-Operationen pro Zeile ausgeführt werden (`g` wie *global replace*).
- `s/ptrn/rpl/i` bedeutet, dass die Auswertung des Musters ohne Rücksicht auf die Groß- und Kleinschreibung erfolgt (*case insensitive*).

Um ein Muster bzw. ein `sed`-Kommando zu testen, kombinieren Sie am besten `echo` und `sed`:

```
$ echo "abc ABC" | sed 's/abc/efg/gi'
  efg efg
```

### sed-Beispiel: Pfade von Bilddateien manipulieren

Persönlich setze ich `sed` intensiv für den Satz meiner Bücher ein. Ich verfasse die meisten Texte in einer Markdown-Syntax. Diverse Scripts machen daraus LaTeX-, HTML- und PDF-Dateien. Unter anderem muss ich bei der Erstellung der Druckdateien meine Screenshots und Fotos (Kennungen `*.png` und `*.jpg`) durch vom Verlag optimierte PostScript-Dateien ersetzen (Kennung `*.eps`), die sich außerdem in einem anderen Verzeichnis befinden. Die ursprüngliche Markdown-Datei enthält also beispielsweise Bildreferenzen in dieser Form:

```
![Relaisboard \mylabel{b-rb}](images-chris/relayboard.png)
![Audioplayer \mylabel{b-mpd}](images-michael/mpd.png)
![L298N-Pins \mylabel{b-l298}](images-charly/L298_Pins.png)
```

Bevor ich die Druckdatei erzeuge, muss der Markdown-Code umgebaut werden:

```
![Relaisboard \mylabel{b-rb}](images-final/relayboard.eps)
![Audioplayer \mylabel{b-mpd}](images-final/mpd.eps)
![L298N-Pins \mylabel{b-l298}](images-final/L298_Pins.eps)
```

Um diese lästige Arbeit kümmert sich natürlich ein Script, das unter anderem das folgende `sed`-Kommando enthält:

```
# Beispieldatei preprocess-markdown.sh
cmd="s,\(images-.*\/(.*)\.(jpg|png)\),(images-final/\1.eps),"
sed -E "$cmd" text.md > text-final.md
```

Weil innerhalb des `sed`-Kommandos das Zeichen / vorkommt, habe ich das Kommando in der Form `s,pattern,replace,` aufgebaut. Das reguläre Muster erkennt Ausdrücke, die in der Form `(images-<aname>/<iname>.jpg|png)` aufgebaut sind, wobei `aname` der Autorenname und `iname` der Bildname ist. Dieses Muster wird ersetzt durch `(images-final/<iname>.eps)`.



© Rheinwerk Verlag, Bonn 2024

## 9.4 Reguläre Muster in der PowerShell

Aufgrund des objektorientierten Ansatzes der PowerShell ist die Auswertung von Text seltener erforderlich als in Bash-Scripts. Oft führt das simple Auslesen von Eigenschaften zum Ziel. Unter Windows sind Textdateien zur Konfiguration oder mit Logging-Daten zudem eher die Ausnahme als die Regel.

Dennoch stellt auch die PowerShell diverse Regex-Features zur Auswahl, die bei der Verarbeitung von Text helfen. In der PowerShell gilt die Regex-Syntax des .NET-Frameworks, die weitgehend kompatibel zu PCRE ist:

*https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference*

### Die Operatoren »match« und »replace«

Am einfachsten können Sie reguläre Ausdrücke mit den diversen `match`-Operatoren anwenden: `txt -match pattern` testet, ob im Text das angegebene Muster erkannt werden kann. Das Resultat ist `True` oder `False`.

PowerShell-spezifisch (oder sollte man sagen: Microsoft-typisch) wird die Groß- und Kleinschreibung dabei ignoriert – außer Sie verwenden `-cmatch` (*case sensitive match*). `-notmatch` bzw. `-cnotmatch` haben einfach die inverse Bedeutung zu `-match`, testen Sie also, ob das Muster *nicht* enthalten ist. Die `match`-Operatoren lassen sich interaktiv ganz unkompliziert ausprobieren. Das ist auch dann praktisch, wenn Sie rasch testen wollen, ob Ihr reguläres Muster syntaktisch korrekt ist und so funktioniert, wie Sie es sich vorstellen.

```
> "I love PowerShell" -match "p.*l"
  True

> "I love PowerShell" -cmatch "p.*l"
  False

> "connection from 127.0.0.1" -match "(\d{1,3}\.){3}\d{1,3}"
  True
```

Sie können `-match` auch auf Datei- oder Verzeichnisnamen anwenden. Das folgende Kommando durchsucht `C:\Windows` rekursiv nach Verzeichnissen in der Form `xx-xx`, wobei `x` ein Buchstabe oder eine Ziffer ist. Sie erhalten damit eine Menge Verzeichnisse mit Namen wie `de-DE` oder `en-US` mit Lokalisierungsdateien (also Dateien, die Zeichenketten in einer bestimmten Sprache enthalten).

```
> Get-ChildItem C:\windows -Recurse -Directory |
  Where-Object { $_.name -match "^\w\w-\w\w$" }
```

Persönliches Exemplar für Thomas Bartholomäus

Mit dem Operator replace können Sie Suchen- und Ersetzen-Operationen für Zeichenketten durchführen.

```
> $txt = "connection from 127.0.0.1 to 192.168.74.104"
> $txt -replace "(\d{1,3}\.){3}\d{1,3}", "-hidden-"
  connection from -hidden- to -hidden-
```

### Select-String

Wie grep wertet auch Select-String den Suchtext als reguläres Muster aus. Ich habe Ihnen dieses CmdLet in Abschnitt 7.2, »Dateien suchen«, ja schon ausführlich vorgestellt. Insofern sollte hier ein Beispiel im Regex-Kontext ausreichen.

Ich werfe dabei einen Blick in das Verzeichnis C:\Windows\diagnostics\system: Dieses enthält diverse PowerShell-Scripts, die vom Troubleshooting-Assistenten der Systemeinstellungen aufgerufen werden (also System • Problembehandlung • Andere Problembehandlungen). Das Ziel des folgenden Kommandos besteht darin, eine Referenz aller in diesen Dateien definierten Funktionen zu ermitteln. Das gelingt relativ einfach, allerdings ist das Ergebnis unübersichtlich:

```
> Set-Location C:\Windows\diagnostics\system
> Get-ChildItem -Filter *.ps1 -Recurse |
  Select-String '^function '

  Apps\RC_WSReset.ps1:7:function Get-FaultyAppsFromEventLogs([...
  Apps\Utils_Apps.ps1:17:Function Write-ExceptionTelemetry($Fu...
  Apps\VF_WSReset.ps1:14:function Get-AppContainerEvents([Syst...
  ...
```

Im zweiten Versuch habe ich ein komplizierteres reguläres Muster entwickelt, das in einer Gruppe function <name> und in der Folge die Parameter dieser Funktion erfasst. Nun will ich aber nicht die gesamte Ergebniszeile sehen, sondern nur function <name>. Dazu müssen die von Select-String zurückgegebenen Match-Objekte ausgewertet werden. Groups[0] liefert dann die ganze Zeile, Groups[1] den Inhalt der erste Regex-Gruppe:

```
> Get-ChildItem -Filter *.ps1 -Recurse |
  Select-String '^(function [^(]*)\(.*\)$' |
  ForEach-Object { $_.Matches.Groups[0].Value }

  function Get-FaultyAppsFromEventLogs
  function Get-CompletedTroubleshooterSessions
  Function Write-ExceptionTelemetry
  ...
```


© Rheinwerk Verlag, Bonn 2024

### Beispiel: Eingabevalidierung

Das folgende Mini-Script erwartet die Eingabe eines Farbcodes in der Form `#ccc` oder `#cccccc`, wobei `c` hexadezimale Stellen sind. Eine korrekte Eingabe wäre also z. B. `#ff0000` für Rot. Fehlerhafte Eingaben müssen wiederholt werden.

```
# Beispieldatei input-css-color.ps1
do {
    $color = Read-Host "Enter a CSS color (#ccc or #cccccc)"
} until ($color -match "^#([0-9a-fA-F]{3}){1,2}$")
Write-Host "Valid color code: $color"
```

### Beispiel: Abbildungsverzeichnis

In einer Markdown-Datei sind Abbildungen in der folgenden Form enthalten:

```
![Abbildungsbeschriftung ...](images-michael/dateiname.png)
```

Die Abbildungen haben die Formate PNG und JPEG. Das Script soll die Dateinamen aller Abbildungen extrahieren:

```
.\extract-images.ps1 text.md

  images-chris/relayboard.png
  images-michael/mpd.png
  images-charly/L298_Pins.jpg
```

Wenn Sie möchten, können Sie die Ausgabe auch in eine Textdatei umleiten.

```
.\extract-images.ps1 text.md > out.txt
```

Das Script ist kurz:

```
# Beispieldatei extract-images.ps1
# das Script erwartet Dateinamen als Parameter
Param( [string[]] $markdownfiles)

$pattern = '\]\((images-\w+[\/\\][\w-]+\.(png|jpg|jpeg))\)'

# Schleife über alle Dateien
foreach ($file in $markdownfiles) {
    # Muster erkennen, Ausgabe des Inhalts der Mustergruppe
    Select-String $pattern $file |
      ForEach-Object { $_.Matches.Groups[1].Value }
}
```

Das reguläre Muster sucht nach `](images-<aname>/<iname>.png|jpg|jpeg)`, wobei `<aname>` der Autoren- und `<iname>` der Bildname sind.

Persönliches Exemplar für Thomas Bartholomäus

Einige Details des Musters sind erklärungsbedürftig:

- `[\/\\]` erlaubt als Verzeichnistrennzeichen sowohl / als auch \. Zur Einhaltung der Regex-Syntax müssen beiden Zeichen Backslashes vorangestellt werden.
- `[\w-]+` akzeptiert alle Buchstaben, Ziffern, den Unterstrich _ und den Bindestrich - im Dateinamen. Beachten Sie, dass der Bindestrich im Zeichenvorrat von `\w` nicht enthalten ist!
- Wie in der PowerShell üblich, unterscheidet `Select-String` nicht zwischen Groß- und Kleinschreibung. Das ist hier ein Vorteil, weil somit auch Kennungen in Großbuchstaben (z. B. `*.PNG`) akzeptiert werden.

Die obige erste Variante des Scripts liefert Textausgaben. Vielleicht fragen Sie sich, wo die Ausgabe durchgeführt wird: Der Ausdruck

```
$_.Matches.Groups[1].Value
```

liefert eine Zeichenkette, die direkt ausgegeben wird. Der folgende Ausdruck innerhalb der `ForEach-Object`-Konstruktion wäre gleichwertig:

```
Write-Output $_.Matches.Groups[1].Value
```

Scripts, die Text ausgeben, entsprechen der Bash-Denkweise, machen vom objektorientierten Ansatz der PowerShell aber keinen richtigen Gebrauch. Bei diesem Beispiel wäre es nahe liegender, wenn das Script `FileInfo`-Objekte zurückgeben würde.

Genau diese Aufgabe erfüllt die Variante `extract-images2.ps1`. Geändert habe ich abermals nur die Zeile in der `ForEach-Object`-Konstruktion. Die resultierende Zeichenkette wird nun an `Get-ChildItem` übergeben. Dieses CmdLet liest die Datei und gibt ein `FileInfo`-Objekt zurück.

```
Get-ChildItem $_.Matches.Groups[1].Value
```

Wenn Sie einen Test mit der Beispieldatei `text.md` durchführen, liefert das Script allerdings nur Fehlermeldungen:

```
.\extract-images2.ps1 text.md

  Cannot find path '...\images-michael\mpd.png' because
  it does not exist.
  ...
```

Das liegt daran, dass die Beispieldateien nur die Markdown-Datei umfassen, nicht aber die dazugehörenden Bilddateien.


© Rheinwerk Verlag, Bonn 2024

## 9.5 Reguläre Muster in Python (re-Modul)

Zur Verwendung regulärer Ausdrücke in Python muss das Standardmodul `re` importiert werden. Es stellt diverse Funktionen zur Verfügung, von denen ich die wichtigsten hier kurz vorstelle:

- `re.search(pattern, str)` liefert entweder ein `Match`-Objekt mit dem Ort des ersten Treffers oder `None`. Die im Muster definierten Gruppen können über die `group`-Eigenschaft ausgewertet werden (siehe den Mustercode im Anschluss an diese Aufzählung).
- `re.match(pattern, str)` funktioniert wie `search`, berücksichtigt das Muster nur am Anfang der Zeichenkette.
- `re.fullmatch(pattern, str)` funktioniert auch wie `search`, allerdings muss das Muster jetzt die gesamte Zeichenkette beschreiben, also vom Anfang bis zum Ende.
- `re.findall(pattern, str)` liefert eine Liste mit *allen* gefundenen Übereinstimmungen. Im Unterschied zu `search`, `match` und `fullmatch` gibt `findall` einfache Zeichenketten mit den Treffern zurück, keine `Match`-Objekte.

  Wenn das Muster *eine* Gruppe enthält, dann gibt die Ergebniszeichenkette den Inhalt dieser Gruppe an. Enthält das Muster dagegen mehrere Gruppen, erhalten Sie ein Tupel mit Zeichenketten für jede Gruppe.
- `re.sub(pattern, replace, str)` ersetzt alle übereinkommenden Muster durch neuen Text.

Die vollständige Dokumentation des `re`-Moduls können Sie hier nachlesen:

*https://docs.python.org/3/library/re.html*

In Python drücken Sie das Suchmuster am besten als Raw-String in der Form `r'pattern'` aus. Sie vermeiden damit durch Backslashes verursachte Probleme. Das folgende Script zeigt die Anwendung von `search`, zuerst für ein Suchmuster ohne Gruppen und dann für ein Suchmuster mit Gruppen.

```
# Beispieldatei hello-re.py, Suchmuster ohne Gruppen
import re
pattern = r'\d{4}-\d{2}-\d{2}'
txt = 'Easter 2024 is on 2024-03-31.'
if result := re.search(pattern, txt):
    print(result.group())        # 2024-03-31
```

Bei der zweiten Variante liefert die `group`-Methode ohne Parameter das gesamte erkannte Muster. Mit `len(result.groups())` kann ermittelt werden, wie viele Mustergruppen es gibt. Diese können nun mit `group(n)` ausgewertet werden. Beachten Sie, dass `group(0)` gleichwertig zu `group()` ist und das Gesamtergebnis liefert.

Persönliches Exemplar für Thomas Bartholomäus

```
# Beispieldatei hello-re.py, Suchmuster mit Gruppen
pattern = r'(\d{4})-(\d{2})-(\d{2})'
if result := re.search(pattern, txt):
    print(result.group())          # 2024-03-31
    print(len(result.groups()))  # 3
    year = result.group(1)
    month = result.group(2)
    day = result.group(3)
    print(year, month, day)       # 2024 12 31
```

### Beispiel: MAC-Adresse verifizieren

Im folgenden Script werden Sie zur Eingabe einer MAC-Adresse (*Media Access Control Address* zur Identifizierung von Netzwerkgeräten) aufgefordert. Dabei handelt es sich um eine 12-stellige hexadezimale Zahl. Das Script akzeptiert Eingaben in der Form `00-80-41-ae-fd-7e` oder `00:80:41:ae:fd:7e`, also mit Bindestrichen oder Doppelpunkten zwischen den Zweiergruppen. `a` bis `f` werden auch als Großbuchstaben akzeptiert.

Kurz eine Erläuterung zum Muster: Auf exakt zwei hexadezimale Stellen muss wahlweise das Zeichen `-` oder `:` folgen. Diese Gruppe muss im Muster fünf Mal auftreten. Danach folgen noch zwei hexadezimale Stellen.

```
# Beispieldatei input-mac.py
import re
ok = False
pattern = r'^([a-fA-F0-9]{2}[:-]){5}[a-fA-F0-9]{2}$'
while not ok:
    mac = input('Enter a MAC address: ')
    ok = re.match(pattern, mac)
    if not ok:
        print('Not valid, please try again:')

print('Valid address:', mac)
```

### Beispiel: Logging-Datei anonymisieren

Im vorigen Kapitel habe ich Ihnen eine Menge Anwendungsbeispiele für `grep`, `sort` und `uniq` präsentiert. Die dort eingesetzte Beispieldatei `access.log` ist eine echte Apache-Logging-Datei eines Webservers, die ich aber aus Datenschutzgründen anonymisiert habe. Die Logging-Datei verwendet das Apache Combined Log Format:

*https://httpd.apache.org/docs/2.4/logs.html*

Das nächste Listing enthält Auszüge aus dem Script `anonymize-log.py`, das Sie in den Beispieldateien im Verzeichnis zum vorigen Kapitel finden. Das reguläre Muster besteht aus zehn Gruppen zur Erkennung der Bestandteile einer Zeile im Apa-


© Rheinwerk Verlag, Bonn 2024

che Combined Log Format (siehe auch *https://httpd.apache.org/docs/2.4/logs.html*). Obwohl pro Zeile nur ein Match zu erwarten ist (`result[0]`), habe ich hier `findall` eingesetzt. Das vereinfacht die weitere Auswertung der Ergebnisse, die `findall` als Tupel zur Verfügung stellt. Dieses Tupel wird in eine Liste umgewandelt. Dann werden einige Komponenten ersetzt, und zuletzt wird die Logging-Zeile wieder ausgegeben.

```
# Beispieldatei anonymize-log.py
# (im Verzeichnis zum vorigen Kapitel)
# Verwendung: ./anonymize-log.py in.log > out.log
import random, re, string, sys

def randomIPv4(): ...     # Funktionen, um zufällige IP-Adressen
def randomIPv6(): ...     # URLs und Namen zu generieren
def randomUrl(): ...
def randomUsername(): ...

# Dictionary, ordnet den echten IP-Adressen zufällige Adressen zu
ips = {}
# Pattern mit 10 Gruppen für das Apache Combined Log Format
pattern = r'(.+?) - (.+?) \[(.+?)\] \"(.+?) (.+?) (.+?)\"
             (.+?) (.+?) \"(.+?)\" \"(.+?)\"'

# im ersten Parameter übergebene Datei öffnen
with open(sys.argv[1], 'r') as f:
    # Schleife über alle Zeilen
    for line in f:
        result = re.findall(pattern, line)
        if len(result):
            groups = list(result[0])  # Liste -> Tupel
            # IP-Adresse durch zufällige Adresse ersetzen
            # und in Dictionary ips speichern
            ip = groups[0]
            if ip not in ips:
                if ':' in ip:
                    ips[ip] = randomIPv6()
                else:
                    ips[ip] = randomIPv4()
            groups[0] = ips[ip]

            # analoger Code für Namen, URLs etc. ...

            # Ausgabe der Zeile (stdout)
            print('%s - %s [%s] "%s %s %s" %s %s "%s" "%s"' %
                  tuple(matches))
```

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

# Kapitel 10
# JSON, XML und INI

Die *JavaScript Object Notation* hat sich im letzten Jahrzehnt als dominierendes Format etabliert, wenn Programme oder Webdienste (z. B. REST-APIs) Daten austauschen sollen.

Der Hype der Nuller-Jahre rund um die *Extensible Markup Language* (XML) ist inzwischen verflogen: Die vergleichsweise komplexe Syntax mit vielen Sonderfällen ist nicht nur für Menschen schwer leserlich, sondern auch per Code mühsam auszuwerten. Aber selbst wenn XML bei neuen Projekten immer seltener zum Einsatz kommt, gibt es einen riesigen Altbestand von Programmen und Web-Services, die auf XML basieren.

Es muss aber nicht immer JSON oder XML sein: Um ein paar Konfigurationseinstellungen zu speichern, reicht das INI-Format vollständig aus.

In diesem Kapitel zeige ich Ihnen Funktionen, die die PowerShell und Python zur Erzeugung bzw. Verarbeitung derartiger Dateien bieten. Zur Not gelingen solche Aufgaben auch in der Bash. Dort müssen Sie aber auf externe Kommandos zurückgreifen, z. B. auf `jq` oder `xmllint`.

**Voraussetzungen für dieses Kapitel**

Dieses Kapitel berücksichtigt wie üblich alle drei Script-Sprachen dieses Buchs, aber der Fokus liegt diesmal bei der PowerShell und Python: Diese beiden Sprachen bieten eine sehr gute Unterstützung für die Formate JSON und XML. Eine gute logische Fortsetzung zu diesem Kapitel ist Kapitel 18, »REST-APIs nutzen«.

## 10.1 JSON in der PowerShell

In der PowerShell machen zwei ebenso simple wie mächtige Funktionen die Verarbeitung von JSON-Daten zum Kinderspiel:

- `ConvertTo-Json` wird zumeist in der Form `cmdlet | ConvertTo-Json` eingesetzt und wandelt die vom CmdLet gelieferten Ergebnisobjekte in eine Zeichenkette im

JSON-Format um. Die Eigenschaften der Objekte dienen als Schlüssel für die Key-Value-Paare im JSON-Dokument. Per Ausgabeumleitung kann das Dokument in einer Datei gespeichert werden.

- Genau umgekehrt funktioniert `ConvertFrom-Json`: Das CmdLet verwandelt die als Parameter oder per Pipe übergebene JSON-Zeichenkette zurück in `PSObjects` oder `Hashtables`, die dann mit `Select-Object`, `ForEach-Object` etc. weiterverarbeitet werden können.

Die folgenden Beispiele zeigen einige Anwendungen der beiden CmdLets.

### Beispiel: Event-Log-Einträge im JSON-Format speichern

Das folgende Kommando liest aus dem Event-Log die letzten zehn sicherheitsspezifischen Einträge und speichert diese in einer JSON-Datei:

```
> Get-EventLog Security -Newest 10 |
  ConvertTo-Json > security-events.json
```

Die resultierende Datei `security-events.json` sieht wie folgt aus:

```
[
  {
    "EventID": 4799,
    "PSComputerName": "localhost",
    "Index": 308255,
    "Message": "Eine sicherheitsaktivierte lokale
                Gruppenmitgliedschaft wurde ...",
    "Source": "Microsoft-Windows-Security-Auditing",
    ...
  },
  {
    "EventID": 4799,
    "PSComputerName": "localhost",
    "Index": 308254,
     ...
  }
]
```

### Beispiel: Domain-Abfragen auswerten

Das zweite Beispiel dreht sich um die Auswertung eines Web-Requests. Die Website *https://domainsdb.info* macht eine ziemlich große (aber keineswegs vollständige!) Datenbank mit öffentlichen Informationen über registrierte Domains zugänglich. Die PowerShell bietet mit `Invoke-WebRequest` einen unkomplizierten Weg, Get-Requests durchzuführen. (`Invoke-WebRequest` ist in etwa mit den Linux-Kommandos


© Rheinwerk Verlag, Bonn 2024

`curl` oder `wget` vergleichbar.) Um Domains zu finden, in denen die Zeichenkette *rheinwerk* vorkommt, gehen Sie so vor:

```
> $url =
  "https://api.domainsdb.info/v1/domains/search?domain=rheinwerk"
> Invoke-WebRequest $url

  StatusCode        : 200
  StatusDescription : OK
  Content           : {"domains": [{"domain": "rheinwerk-
                        office.com", "create_date": ...
  ...
```

Der spannende Teil des Ergebnisses ist über die Eigenschaft `Content` zugänglich und liegt im JSON-Format vor. Das Dokument sieht formatiert (stark gekürzt) so aus:

```
{
  "domains": [
    {
      "domain": "rheinwerk-office.com",
      "create_date": "2022-12-29T13:21:38.809015",
      "A": [
        "87.106.63.234"
      ],
      ...
    },
    {
      "domain": "rheinwerk-publishing.com",
      ...
    }, ... ]
}
```

Ziel des Beispiels ist es, aus dem recht umfangreichen JSON-Dokument nur die Eigenschaften `domain` und `A` (also den A-Record-Eintrag) herauszufiltern. Nach einer Umwandlung der JSON-Daten in PowerShell-Objekte gelingt das mühelos:

```
> (Invoke-WebRequest $url | ConvertFrom-Json).domains |
  Select-Object domain, A

  domain                   A
  ------                   -----------
  rheinwerk-office.com     {87.106.63.234}
  rheinwerk-publishing.com {46.235.24.150}
  rheinwerk-verlag.com     {46.235.24.168}
  ...
```

Persönliches Exemplar für Thomas Bartholomäus

**REST-APIs**

Eine Alternative zur Kombination aus `Invoke-WebRequest` und `ConvertFrom-Json` ist das Kommando `Invoke-RestMethode`, das ich Ihnen in Kapitel 18, »REST-APIs nutzen«, vorstellen werde. Es wandelt JSON-Dokumente direkt in PowerShell-Objekte um.

### Beispiel: Umwandlung zwischen CSV und JSON

Geradezu ein Kinderspiel ist es, eine CSV-Datei in das JSON-Format umzuwandeln. Die einzige Voraussetzung besteht darin, dass die CSV-Datei in der ersten Zeile eine Beschriftung der Spalten enthält. Als Ausgangspunkt für dieses Beispiel dient die Datei `employees.csv`, die wie folgt aussieht:

```
FirstName;LastName;DateOfBirth;Street;Zip;City;...
Ruthanne;Summers;1977-06-04;4 Dewy Turnpike;27698;Clifton ...
...
```

`Import-Csv` wandelt die Datei in PowerShell-Objekte um, `ConvertTo-Json` macht daraus ein JSON-Dokument:

```
> Import-Csv -Delimiter ';' employees.csv |
  ConvertTo-Json > employees.json
```

Standardmäßig kümmert sich `ConvertTo-Json` um eine lesefreundliche Einrückung des JSON-Dokuments. Wenn die Lesbarkeit der resultierenden Datei keine Rolle spielt, übergeben Sie die zusätzliche Option `-Compress`. Das nachfolgende `Get-ChildItem` beweist, dass die Platzersparnis durchaus spürbar ist.

```
> Import-Csv -Delimiter ';' employees.csv |
  ConvertTo-Json -Compress > employees-compressed.json

> Get-ChildItem *.json | Select-Object Name, Length

  Name                      Length
  ----                      ------
  employees-compressed.json 122528
  employees.json            165030
```

Ebenso einfach ist der Rückumwandlung von JSON zu CSV:

```
> Get-Content .\employees-compressed.json | ConvertFrom-Json |
  Export-Csv employees2.csv
```


© Rheinwerk Verlag, Bonn 2024

## 10.2 JSON und Python

Python und JSON sind ein Dream-Team. Zur Verarbeitung von JSON-Dokumenten importieren Sie das json-Modul. Es wird mit Python standardmäßig ausgeliefert und muss nicht extra installiert werden. Das Modul stellt die folgenden Funktionen zur Verfügung:

- load(filehandle) liest eine zuvor mit open geöffnete Textdatei und liefert die darin enthaltenen JSON-Daten in Form von Python-Dictionaries und -Listen zurück.
- loads(str) (im Sinne von *load string*) erwartet das JSON-Dokument in der als Parameter übergebenen Zeichenkette.
- dump(obj, filehandle) speichert das im ersten Parameter übergebene Python-Objekt als JSON-Zeichenkette in der angegebenen Datei. Mit einigen zusätzlichen Parametern können Sie Einfluss auf das resultierende JSON-Dokument nehmen. indent=2 gibt die gewünschte Einrücktiefe pro Verschachtelungsebene an. ensure_ascii=False stellt die korrekte Verarbeitung von UTF8-Zeichen sicher.
- dumps(obj) funktioniert wie dump, gibt das JSON-Dokument aber als Zeichenkette zurück.

Das folgende Listing zeigt die Anwendung der vier Methoden:

```
# Beispieldatei hello-json.py
import json

# JSON-Datei lesen
with open('employees.json', 'r') as f:
    employees = json.load(f)

# JSON-Zeichenkette verarbeiten
txt = '{"key1": "value1", "key2": "value2"}'
data = json.loads(txt)

# Auswertung der Daten
print(data['key2'])   # Ausgabe: value2

# Python-Objekt (Liste, Dictionary) als JSON-Datei speichern
with open('otherfile.json', 'w') as f:
    json.dump(data, f, indent=2, ensure_ascii=False)

# JSON-Zeichenkette ausgeben
print(json.dumps(data, indent=2, ensure_ascii=False))
```

Persönliches Exemplar für Thomas Bartholomäus

### Beispiel: Geburtstage sammeln

Als Ausgangspunkt für das Script `birthdays.py` dient die Datei `employees.json`. Sie hat den folgenden Aufbau:

```
[
  {
    "FirstName": "Ruthanne",
    "LastName": "Ferguson",
    "DateOfBirth": "1977-06-04",
    ...
  }, ...
]
```

Das Script `birthdays.py` liest die JSON-Datei ein und verarbeitet die Angestellten in einer Schleife. Dabei wird ein Dictionary gebildet. Als Schlüssel dienen Tag und Monat des Geburtstags (z. B. `06-24`). Der eigentliche Eintrag enthält eine Liste der Mitarbeiternamen, die an diesem Tag Geburtstag haben.

```
# Beispieldatei birthdays.py
import json
with open('employees.json') as f:
    employees = json.load(f)

birthdates = {}   # Geburtstags-Dictionary
for employee in employees:
    # [5:] überspringt die ersten fünf Zeichen, nur Monat und Tag
    birthdate = employee['DateOfBirth'][5:]
    name = employee['FirstName'] + ' ' + employee['LastName']
    if birthdate in birthdates:
        # Name zur vorhandenen Liste hinzufügen
        birthdates[birthdate].append(name)
    else:
        # neuen Dictionary-Eintrag mit Liste erzeugen
        birthdates[birthdate] = [name]

# Test: alle Geburtstagskinder am 24.1.
# Ausgabe: ['Nannette Ramsey', 'Allena Hoorenman',
#           'Arden Lit', 'Duncan Noel']
print(birthdates['01-24'])
```

### Beispiel: Feiertage ermitteln

Auf der Website *https://calendarific.com* erhalten Sie nach einer kostenlosen Registrierung einen API-Key, mit dem Sie die Feiertage und alle erdenklichen sonstigen Gedenktage für ein bestimmtes Land und das angegebene Jahr ermitteln können, beispielsweise so:


© Rheinwerk Verlag, Bonn 2024

```
$ curl 'https://calendarific.com/api/v2/holidays\?
             api_key=1234&country=DE&year=2023'
```

Das Ergebnis ist ein JSON-Dokument, das wie folgt aufgebaut ist:

```
{
    "meta": {
        "code": 200
    },
    "response": {
        "holidays": [
            {
                "name": "Name of holiday,
                "description": "Description of holiday,
                "date": {
                    "iso": "2023-12-31",
                    "datetime": { ... }
                },
                "type": [ ... ]
            }, ...
        ]
    }
}
```

Das Beispiel-Script `holidays.py` erwartet zwei optionale Parameter, einen Ländercode und eine Jahreszahl. Wenn diese Angaben fehlen, verwendet das Script standardmäßig `'US'` und das aktuelle Jahr. Das Script führt dann einen Request durch, wertet das resultierende Ergebnis aus und liefert das Ergebnis in der folgenden Form:

```
$ ./holidays.py DE 2023
  Holidays for DE in 2023
  2023-01-01: New Year's Day
    New Year's Day, which is on January 1, ...
  2023-01-06: Epiphany
    Epiphany on January 6 is a public holiday in 3 German states
    and commemorates the Bible story of the Magi's visit to
    baby Jesus.
  ...
```

### Einschränkungen

Die Nutzung von *https://calendarific.com* ist nach einer einfachen Registrierung zwar kostenlos, unterliegt aber diversen Einschränkungen. Eine kommerzielle Nutzung ist nur bei Bezahlung einer monatlichen Gebühr erlaubt.

Persönliches Exemplar für Thomas Bartholomäus

Das Script beginnt mit dem Import diverser Module und der Initialisierung der Variablen api_key, country und year. Eine Schleife wertet alle an das Script übergebenen Parameter aus und überschreibt gegebenenfalls year bzw. country.

```
# Beispieldatei holidays.py
import datetime, json, sys, urllib.request

# Bitte verwenden Sie einen eigenen Key, den Sie
# kostenlos auf https://calendarific.com erhalten.
api_key = "xxx"

# Defaulteinstellungen
country = 'US'
year = datetime.datetime.now().year

# Script-Parameter auswerten, year und country einstellen
for arg in sys.argv[1:]:
    if arg.isdigit():
        year = arg
    else:
        country = arg

print("Holidays for", country, "in", year)
```

Für den Web-Request habe ich das Python-Modul urllib verwendet, das standardmäßig zur Verfügung steht, also nicht mit pip installiert werden muss. Die Anwendung dieses Moduls ist umständlich: Zuerst erzeugen Sie ein Request-Objekt. Dieses übergeben Sie an die Methode urlopen und erhalten ein Response-Objekt. Dessen read-Methode liefert Ihnen die vom Server zurückgesendeten Daten in binärer Form, die Sie schließlich mit decode in eine UTF-8-Zeichenkette umwandeln.

Bei meinen Tests hat sich herausgestellt, dass *https://calendarific.com* Requests von Python verweigert. (Ein mit curl ausgeführter Request mit der gleichen URL funktioniert dagegen.) Vermutlich wollen die Calendarific-Betreiber, dass Sie das Modul python-calendarific einsetzen, was ich hier aus didaktischen Gründen aber vermeiden wollte. Stattdessen habe ich mit dem headers-Parameter den Default-Header überschrieben und so Calendarific überlistet.

Aus der Sicht dieses Kapitels sind die letzten Zeilen des Scripts am spannendsten. json.loads(txt) macht aus dem JSON-Dokument einen Python-Objektbaum. data['response']['holidays'] liefert eine Liste von holiday-Dictionaries, die dann in einer Schleife ausgewertet werden. Sie sehen, wenn einmal die Hürden des Downloads überwunden sind, lässt sich die JSON-Auswertung sehr einfach durchführen.


© Rheinwerk Verlag, Bonn 2024

```
# Beispieldatei holidays.py (Forts.)
# Web-Request durchführen
query = "https://calendarific.com/api/v2/
         holidays?api_key=%s&country=%s&year=%s"
url = query % (api_key, country, year)
req = urllib.request.Request(url,
                             headers={"User-Agent": "curl"})
response = urllib.request.urlopen(req)
txt = response.read().decode("utf-8")

# Auswertung der JSON-Daten
data = json.loads(txt)
for holiday in data['response']['holidays']:
    name = holiday['name']
    date = holiday['date']['iso']
    descr = holiday['description']
    print('%s: %s' % (date, name))
    print('  %s' % (descr))
```

**requests statt request**

Anstelle des `request`-Moduls gibt es in Python auch das Modul `requests` (mit Plural-S). Es muss extra installiert werden, ist dafür aber wesentlich komfortabler anzuwenden. Beispiele für die Anwendung dieses Moduls finden Sie in Kapitel 17, »Web Scraping«, sowie in Kapitel 18, »REST-APIs nutzen«.

## 10.3 JSON in der Bash

Die Bash enthält keine eingebauten JSON-Funktionen. Dafür steht in den meisten Linux-Distributionen das Kommando `jq` (für *JSON Query*) zur Verfügung. Sollte die Ausführung des Kommandos mit der Fehlermeldung *command not found* enden, müssen Sie das Kommando installieren. Unter Ubuntu gelingt dies mit `sudo apt install jq`.

`jq` liest das JSON-Dokument aus der Standardeingabe oder aus den als Parameter übergebenen Dateien. Es wendet dann einen Filterausdruck auf das Dokument an und schreibt das Ergebnis an die Standardausgabe. Die einfachste Anwendung besteht darin, JSON-Dokumente lesbar zu formatieren. Dazu geben Sie als Filterausdruck einfach einen Punkt an:

Persönliches Exemplar für Thomas Bartholomäus

```
$ echo '{"key1":"123","key2":"456"}' | jq .

  {
    "key1": "123",
    "key2": "456"
  }
```

Basis für die folgenden Beispiele ist die ca. 1400 Zeilen lange Datei `commits.json`. Dabei handelt es sich um Daten aus dem GitHub-Event-Log, der öffentlich im JSON-Format zugänglich ist. Als Datengrundlage habe ich ein Repository des Docker-Buchs verwendet, das ich zusammen mit Bernd Öggl verfasst habe:

```
$ curl 'https://api.github.com/repos/docker-compendium/\
        samples/commits' > commits.json
```

Die JSON-Datei besteht einfach aus einem Array von Commit-Daten. In `jq` greifen Sie mit `.[]` auf das ganze Array zu, mit `.[0]` auf das erste Element (den neuesten Commit).

```
$ jq '.[0]' commits.json

  {
    "sha": "8b17e24e278cd594456db77abbbefe4190ac1d88",
    "node_id": "C_kwD",
    "commit": {
      "author": {
        "name": "Michael Kofler",
  ...
```

Mit `.keyname` greifen Sie auf einzelne Datenelemente zu:

```
$ jq '.[0].commit.author.name' commits.json
  "Michael Kofler"
```

Wer hat wie viele Commits durchgeführt?

```
$ jq '.[].commit.author.name' commits.json | sort | uniq -c

  11 "Bernd Oeggl"
   7 "Michael Kofler"
```

Das nächste Kommando listet alle Commit Messages auf:

```
$ jq '.[].commit.message' commits.json

  "Update README.md"
  "add k8s sample code files"
  "add swarm sample code"
  ...
```


© Rheinwerk Verlag, Bonn 2024

Innerhalb des Filter-Ausdrucks ist der Pipe-Operator | erlaubt. Im folgenden Ausdruck extrahieren Sie aus jedem Listenelement zwei Eigenschaften, geben diesen neue Namen und machen daraus neue JSON-Elemente:

```
$ jq '.[] | {date: .commit.author.date,
             msg: .commit.message}' commits.json

  {
    "date": "2023-01-22T06:40:14Z",
    "msg": "Update README.md"
  }
  {
    "date": "2023-01-21T20:50:32Z",
    "msg": "add k8s sample code files"
  }
  ...
```

Das Ergebnis ist allerdings selbst kein gültiges JSON-Dokument. Dazu müssen Sie es mit eckigen Klammern in ein Array verpacken:

```
$ jq '[ .[] | {date: .commit.author.date,
               msg: .commit.message} ]' commits.json

  [
    {
      "date": "2023-01-22T06:40:14Z",
      "msg": "Update README.md"
    },
    ...
```

Mit `select` können Sie Bedingungen formulieren. Das folgende Kommando erzeugt ein neues JSON-Dokument mit den Commits von Bernd Öggl:

```
$ jq '[ .[] | select (.commit.author.name=="Bernd Oeggl") ]' >
  bernds-commits.json
```

Weitere Anwendungsbeispiele der `jq`-Syntax finden Sie im Tutorial, eine umfassende Referenz der zahlreichen Funktionen im Handbuch:

*https://stedolan.github.io/jq/tutorial*
*https://stedolan.github.io/jq/manual*

### JSON-Dateien mit »fx« interaktiv ansehen

Natürlich können Sie mit `less` durch JSON-Dateien blättern. Wesentlich mehr Komfort bietet das Kommando `fx`, das Sie allerdings vorher installieren müssen:

*https://github.com/antonmedv/fx*

Persönliches Exemplar für Thomas Bartholomäus

Anschließend können Sie mit `fx myfile.json` oder `cmd | fx` ein JSON-Dokument öffnen, mit [◄] und [►] JSON-Gruppen öffnen und schließen, mit \ nach Texten suchen usw. Das JSON-Dokument wird ordentlich formatiert, Schlüssel und Einträge werden farbig hervorgehoben (siehe Abbildung 10.1). Kurz und gut, das Lesen oder Analysieren großer JSON-Dateien im Terminal wird mit `fx` zum Vergnügen. [Q] beendet das Programm.

```
fx commits.json
[
  {
    "sha": "8b17e24e278cd594456db77abbbefe4190ac1d88",
    "node_id": "C_kwDOIDroUNoAKDhiMTdlMjRlMjc4Y2Q1OTQ0NTZkYjc3YWJiYmVmZTQxOTBhYzFkODg",
    "commit": {
      "author": {
        "name": "Michael Kofler",
        "email": "MichaelKofler@users.noreply.github.com",
        "date": "2023-01-22T06:40:14Z"
      },
      "committer": {"name": "GitHub", …},
[0].commit.author                                                    commits.json
/michael/i   found: [1/136]
```

**Abbildung 10.1** JSON-Dateien im Terminal mit »fx« lesen

## 10.4 XML in der PowerShell

Microsoft hat für Windows voll auf XML gesetzt. Deswegen tummeln sich in einem Windows-System zahlreiche XML-Dateien. Auf meinem Windows-11-Testrechner bin ich alleine in `C:\Windows` auf ca. 1500 Dateien mit der Kennung `*.xml` gestoßen. Selbst ganz »normale« Office-Dateien (z. B. `*.docx`) verwenden XML-Inhalte. (Genau genommen handelt es sich bei Office-Dateien um ZIP-Archive, die wiederum XML-Dateien enthalten.) Zusammengefasst – XML ist unter Windows ein allgegenwärtiges Format.

### XML-Datentyp

Die PowerShell unterstützt XML durch einen eigenen Datentyp. Beim Einlesen einer XML-Datei müssen Sie der Variable den Datentyp in eckigen Klammern voranstellen, damit der PowerShell klar ist, dass Sie die Datei nicht als gewöhnlichen Text betrachten möchten, sondern im XML-Format.

```
> [xml]$tzm = Get-Content `
    "C:\Windows\Globalization\Time Zone\timezoneMapping.xml"
```

Die im obigen Beispiel verwendete Datei enthält Informationen über Zeitzonen. (Falls Sie nicht unter Windows arbeiten, finden Sie eine Kopie der Datei in den Beispieldateien.) Die Datei ist wie folgt aufgebaut:


© Rheinwerk Verlag, Bonn 2024

```
<?xml version="1.0" encoding="utf-8"?>
<TimeZoneMapping GeneratedAt="2022-11-08T09:34:04.6992291+05:30">
  <MapTZ TZID="Etc/GMT+12" WinID="Dateline Standard Time"
         Region="001" Default="true" StdPath="GMT+12/standard"
         DltPath="GMT+12/daylight" />
  <MapTZ TZID="Etc/GMT+12" WinID="Dateline Standard Time"
         Region="ZZ" ... />
  ...
```

Sie können nun über Eigenschaften auf die Elemente der XML-Struktur zugreifen:

```
> $tzm.TimeZoneMapping.MapTZ[0]

  TZID     : Etc/GMT+12
  WinID    : Dateline Standard Time
  Region   : 001
  Default  : true
  StdPath  : GMT+12/standard
  DltPath  : GMT+12/daylight

> $tzm.TimeZoneMapping.MapTZ[57].WinID
  Mountain Standard Time
```

Das folgende Kommando durchläuft die ersten fünf Zeitzonen in einer Schleife und liefert Objekte zurück, deren Eigenschaften den Attributen `TZID` und `StdPath` entsprechen:

```
> $tzm.TimeZoneMapping.MapTZ | Select-Object -First 5 |
  ForEach-Object {
    [PSCustomObject]@{ 'id' = $_.WinID; 'stdpath'=$_.StdPath}
  }

  id                     stdpath
  --                     -------
  Dateline Standard Time GMT+12/standard
  Dateline Standard Time GMT+12/standard
  UTC-11                 GMT+11/standard
  UTC-11                 Samoa/standard
  UTC-11                 Niue/standard
```

Mit `GetElementsByTagName` können Sie auf alle XML-Elemente mit einem bestimmten Namen zugreifen, unabhängig davon, in welcher Ebene des XML-Dokuments sich die Elemente befinden.

```
> $tzm.GetElementsByTagName("MapTZ")
```

Persönliches Exemplar für Thomas Bartholomäus

### Select-Xml

Select-Xml wendet XPath-Ausdrücke auf XML-Dokumente an. XPath ist eine standardisierte Abfragesprache für XML, die ich in diesem Buch nicht weiter behandle. Sie finden gegebenenfalls in der Wikipedia eine gute Zusammenfassung der Syntax und im Internet unzählige Anwendungsbeispiele.

Die XML-Dokumente können wahlweise aus einer Datei gelesen (Option -Path), in einer Zeichenkette (-Content) oder als XML-Objekt (-Xml) übergeben werden. Die drei folgenden Select-Xml-Beispiele sind daher gleichwertig und ergeben jeweils die Anzahl der MapTZ-Tags in der Datei. Beachten Sie, dass beim Einlesen der XML-Datei in eine Zeichenkette Get-Content mit der Option -Raw verwendet werden muss. Die Option bewirkt, dass die gesamte Datei als ein Objekt und nicht als ein Array von Zeilen verarbeitet wird.

```
> [xml]$tzm = Get-Content `
    "C:\Windows\Globalization\Time Zone\timezoneMapping.xml"

> $str = Get-Content -Raw `
    "C:\Windows\Globalization\Time Zone\timezoneMapping.xml"

> $xpath = "/TimeZoneMapping/MapTZ"

> (Select-xml -XPath $xpath -Path .\timezone-mapping.xml).Count
> (Select-Xml -XPath $xpath -Content $tmz ).Count
> (Select-Xml -XPath $xpath -Xml $tmz).Count
  616
```

Das zweite Beispiel ermittelt den Text des Attributs WinID vom 58. MapTZ-Tag. (Die Zählung beginnt mit 0.) Select-Xml liefert SelectXmlInfo-Objekte zurück, deren Defaultdarstellung in der Ausgabe nicht besonders hilfreich ist. Erst Select-Object -ExpandProperty Node liefert den eigentlichen Text des Attributs.

```
> Select-Xml "/*/MapTZ[57]/@WinID" -Xml $tmz

  Node  Path        Pattern
  ----  ----        -------
  WinID InputStream /TimeZoneMapping/MapTZ[57]/@WinID

> Select-Xml -Xml $tmz -XPath $xpath |
  Select-Object -ExpandProperty Node

  #text
  -----
  Mountain Standard Time
```


© Rheinwerk Verlag, Bonn 2024

Das dritte Beispiel bezieht sich auf *.vbox-Dateien, die virtuelle Maschinen des Programms *VirtualBox* beschreiben. Trotz der Kennung *.vbox handelt es sich um gewöhnliche XML-Dateien.

```
<?xml version="1.0"?>
<VirtualBox xmlns="http://www.virtualbox.org/" version="...">
  <Machine uuid="{c197c895-7b0c-4fa3-a05b-165377b33232}" ...>
    <MediaRegistry>
      <HardDisks>
        <HardDisk uuid="..." location="kali-english.vdi"
            format="VDI" type="Normal"/>
            ...
```

Die ersten Versuche, die Dateien auszuwerten, scheitern aber kläglich. Select-Xml liefert einfach ein leeres Ergebnis – ohne Warnung oder Fehlermeldung:

```
> Select-Xml  './/HardDisk/@location' kali.vbox
```

Die Ursache besteht darin, dass die XML-Datei auf einen Namespace verweist. In diesem Fall gibt es nur einen Default-Namespace (xmlns), manche XML-Dateien verwenden sogar mehrere Namespaces (xmlns:name). Damit Sie solche Dateien verarbeiten können, müssen Sie an Select-Xml eine Hashtable übergeben, die jedem Namespace einen Prefix zuordnet. Für den Default-Namespace wird üblicherweise ns verwendet, es ist aber jede beliebige Zeichenkette mit der Ausnahme von xmlns erlaubt. Den Prefix müssen Sie in der Folge auch im XPath-Ausdruck angeben. Das folgende Kommando extrahiert den Namen der Image-Datei einer virtuellen Maschine.

```
> $vbnamespace = @{ns = "http://www.virtualbox.org/"}

> Select-Xml './/ns:HardDisk/@location' -Namespace $vbnamespace `
  kali.vbox | Select-Object -ExpandProperty Node

  kali-install.vdi
```

Das zweite Beispiel durchsucht alle virtuellen Maschinen nach aktiven Netzwerkadaptern und liefert Objekte mit dem *.vbox-Dateinamen und der verwendeten MAC-Adresse. Das CmdLet Split-Path mit der Option -Leaf wird hier verwendet, um den Dateinamen aus dem Pfad zu extrahieren.

```
> Select-Xml './/ns:Network/ns:Adapter' `
     -Namespace $vbnamespace *.vbox |
  Where-Object { $_.Node.enabled -eq 'true' } |
  ForEach-Object {
    [PSCustomObject]@{path = Split-Path $_.Path -Leaf;
                      mac = $_.Node.MACAddress}
  }
```

Persönliches Exemplar für Thomas Bartholomäus

```
path                 mac
----                 ---
kali-2022.vbox       080027DB966A
kali.vbox            0800278EB571
metasplaitable2.vbox 080027E770F0
ubuntu.vbox          0800278556EA
```

### ConvertTo-Xml, Export-Clixml und Import-Clixml

Zuletzt stelle ich Ihnen ganz kurz drei CmdLets zur Umwandlung zwischen PowerShell-Objekten und XML-Dokumenten vor:

- `ConvertTo-Xml` wandelt ein PowerShell-Objekt in den XML-Datentyp um:

  ```
  $myxmlvar = Get-Date | ConvertTo-Xml
  ```

  Beachten Sie, dass *kein* offizielles CmdLet `ConvertFrom-Xml` mit der Umkehrfunktion existiert. Wenn Sie im Internet nach `ConvertFrom-Xml` suchen, werden Sie aber auf diverse CmdLets aus der Community stoßen, die aus einem XML-Dokument ein möglichst gut entsprechendes `PSCustomObject` zusammensetzen.

- `Export-Clixml` speichert ein PowerShell-Objekt in einer XML-Datei, die dem Standard *Common Language Infrastructure* entspricht:

  ```
  Get-Date | Export-Clixml 'date.xml'
  ```

- Das Gegenstück `Import-Clixml` liest eine derartige Datei wieder ein und macht daraus das entsprechende PowerShell-Objekt:

  ```
  $olddate = Import-Clixml 'date.xml'
  ```

## 10.5 XML und Python

Es gibt diverse offizielle und noch mehr externe XML-Bibliotheken für Python. Die Module sind für unterschiedliche Anwendungszwecke optimiert, z. B. für die besonders effiziente Verarbeitung großer Dateien, für die möglichst sichere Auswertung von XML-Dateien oder für das Erstellen eigener XML-Dokumente. Einen Überblick geben die beiden folgenden Seiten:

*https://docs.python.org/3/library/xml.html*
*https://realpython.com/python-xml-parser*

Ich konzentriere mich in diesem Abschnitt auf das Modul `xml.etree.ElementTree`, das standardmäßig mit Python ausgeliefert wird. Es ist einfach anzuwenden und gut geeignet, um XML-Dokumente auszuwerten. Die folgenden Zeilen zeigen, wie Sie eine XML-Zeichenkette bzw. eine XML-Datei einlesen und auf das Root-Element des XML-Dokuments zugreifen:


© Rheinwerk Verlag, Bonn 2024

```
import xml.etree.ElementTree as ET
# XML-Datei lesen ...
root = ET.parse('filename.xml').getroot()

# ... oder XML aus Zeichenkette verarbeiten
s = '<?xml ...>...'
root = ET.fromString(s)
```

`root` ist ein Objekt der `Element`-Klasse. Es hat die folgenden Eigenschaften:

- `tag`: Name des aktuellen XML-Tags
- `text`: Text im Tag (aber keine Subelemente)
- `attrib`: Dictionary mit Schlüsseln des XML-Tags

Alle Subelemente eines `Elements` können direkt in Schleifen durchlaufen werden. Die Schleifenvariable enthält selbst wieder `Element`-Objekte:

```
for sub in e:  # Schleife über alle Sub-Elemente
```

Alternativ können Sie auch direkt auf das *n*-te Element zugreifen.

```
sub = e[3]     # 4. Subelement (die Zählung beginnt mit 0)
```

Wenn Sie alle XML-Elemente mit einem bestimmten Namen in der gerade aktuellen Ebene verarbeiten möchten, verwenden Sie die Methode `findall`:

```
# alle <tagname>-Elemente in der aktuellen Ebene durchlaufen
for sub in e.findall('tagname'):
```

Wenn Sie ohnedies nur das erste passende Subelement verarbeiten wollen, können Sie sich die Schleife mit `findall` sparen und stattdessen `find` einsetzen. Beachten Sie aber, dass die folgende Anweisung einen Fehler auslöst, wenn es kein passendes Element gibt!

```
# auf das erste <tagname>-Element zugreifen
sub = e.find('tagname')
```

Wenn Sie dagegen nach Elementen in allen Ebenen des XML-Dokuments suchen möchten, verwenden Sie die Methode `iter`:

```
# alle <tagname>-Elemente in allen Ebenen durchlaufen
for sub in e.iter('tagname'):
```

Falls die XML-Datei Namespaces verwendet, müssen Sie an `find` bzw. `findall` ein Dictionary mit Kürzel für alle verwendeten Namespaces übergeben. Dieselben Kürzel müssen Sie den Tag-Namen voranstellen. Werfen Sie einen Blick auf das dritte der folgenden Beispiele, das die Vorgehensweise demonstriert.

Persönliches Exemplar für Thomas Bartholomäus

### Beispiel: Dictionary für Ländercodes bilden

Dieser Crashkurs reicht für die meisten Anwendungsfälle bereits aus. Ausgangspunkt für das erste Beispiel ist die Datei `countries.xml`, die wie folgt aufgebaut ist:

```
<countries>
  <country code="AF" iso="4">Afghanistan</country>
  <country code="AL" iso="8">Albania</country>
  <country code="DZ" iso="12">Algeria</country>
  ...
</countries>
```

Das Ziel des folgenden Scripts besteht darin, ein Dictionary zu erstellen, wobei die Ländercodes als Schlüssel dienen:

```
# Beispieldatei read-countries.py
import xml.etree.ElementTree as ET
root = ET.parse('countries.xml').getroot()
countries = {}  # leeres Dictionary
for country in root:
    countries[country.attrib['code']] = country.text

print(countries['CH'])   # Ausgabe: Switzerland
```

### Beispiel: RSS-Feed auswerten

Viele Webseiten bieten über einen *Rich Site Summary Feed* (RSS-Feed) die Möglichkeit, rasch auf die neuesten Artikel zuzugreifen. Den RSS-Feed der BBC-World-News finden Sie hier:

*http://feeds.bbci.co.uk/news/world/rss.xml*

Die resultierende Datei hat (stark gekürzt) diesen Aufbau:

```
<?xml-stylesheet ... ?>
<rss xmlns:dc="http://purl.org/dc/elements/1.1/"
     xmlns:content="http://purl.org/rss/1.0/modules/...">
  <channel>
    <title><![CDATA[BBC News - World]]></title>
    <lastBuildDate>Thu, 23 Feb 2023 11:02:41 GMT</lastBuildDate>
    <copyright>....</copyright>
    <item>
      <title><![CDATA[Gaza-Israel exchange of fire ...]]></title>
      <link>https://www.bbc.co.uk/news/world-...</link>
      <pubDate>Thu, 23 Feb 2023 09:22:38 GMT</pubDate>
    </item>
    <item>
    ...
```


© Rheinwerk Verlag, Bonn 2024

Das Ziel des Scripts besteht darin, die Überschriften und Links der neuesten 10 Artikel anzuzeigen:

```
$ ./read-bbc-news.py

  * Gaza-Israel exchange of fire follows deadly West Bank raid
    https://www.bbc.co.uk/news/world-middle-east-64742259
  * Selfie image shows US pilot flying over Chinese 'spy balloon'
    https://www.bbc.co.uk/news/world-us-canada-64735538
  ...
```

Der erforderliche Code sieht wie folgt aus:

```
# Beispieldatei bbc-world-news.py
import xml.etree.ElementTree as ET
import urllib.request

url = 'http://feeds.bbci.co.uk/news/world/rss.xml'
response = urllib.request.urlopen(url)
binary = response.read()       # binäre Daten
txt = binary.decode('utf-8')   # in UTF-8-Text umwandeln
root = ET.fromstring(txt)      # RSS-Root-Tag
cnt = 0
for item in root.iter('item'): # Schleife über alle Items
    print('*', item.find('title').text)
    print(' ', item.find('link').text)
    cnt += 1
    if cnt >= 10:
        break
```

### Beispiel: MAC-Adressen aus Virtual-Machine-Dateien extrahieren

Wie in Abschnitt 10.4, »XML in der PowerShell«, extrahiert auch das folgende Python-Script MAC-Adressen aus den als Parametern übergebenen `*.vbox`-Dateien:

```
$ ./vbox-mac.py vbox/*.vbox

     kali-2022.vbox: MAC=080027DB966A
  kali-english.vbox: MAC=080027B8A33A
                ...
```

Da die `*.vbox`-Datei einen Namespace verwendet, kann die `iter`-Methode nicht verwendet werden. Stattdessen kommen die Methoden `find` und `findall` zum Einsatz, wobei jeweils ein Dictionary mit dem Namespace-Prefix als zusätzlicher Parameter übergeben wird:

Persönliches Exemplar für Thomas Bartholomäus

```
# Beispieldatei vbox-mac.py
import os, sys
import xml.etree.ElementTree as ET
namesp = { 'ns': 'http://www.virtualbox.org/'}
for xmlfile in sys.argv[1:]:
    basename = os.path.basename(xmlfile)
    root = ET.parse(xmlfile).getroot()
    machine = root.find('ns:Machine', namesp)
    hardware = machine.find('ns:Hardware', namesp)
    network = hardware.find('ns:Network', namesp)
    for adapter in network.findall('ns:Adapter', namesp):
        if 'MACAddress' in adapter.attrib:
            print('%30s: MAC=%s' %
              (basename, adapter.attrib['MACAddress']))
```

## 10.6 XML in der Bash

Die Bash kann mit XML-Dokumenten ebenso wenig umgehen wie mit JSON-Dateien. Der Ausweg sind externe Kommandos, die aber in jedem Fall extra installiert werden müssen. Unter Linux verwenden Sie dazu die Paketverwaltungs-Tools Ihrer Distribution, unter macOS am besten Brew.

### xmllint

Das Kommando `xmllint` hilft dabei, XML-Dokumente mit Einrückungen lesbar darzustellen und die Einhaltung der XML-Syntaxregeln zu verifizieren. Wenn sich eine XML-Datei partout nicht bearbeiten lässt, ist ein kurzer Test mit `xmllint` angebracht. Unter macOS ist das Kommando standardmäßig installiert, unter Linux versteckt es sich je nach Distribution in unterschiedlichen Paketen:

```
$ sudo apt install libxml2-utils    (Debian, Ubuntu)
$ sudo dnf install libxml2          (Fedora, RHEL)
$ yay -yS libxml2                   (Arch Linux)
```

Im einfachsten Fall übergeben Sie an `xmllint` den Dateinamen einer XML-Datei. Das Kommando verifiziert die XML-Syntax und liefert entweder eine Fehlermeldung oder gibt die Datei unverändert wieder aus.

```
$ xmllint countries-malformed.xml

  countries-malformed.xml:6: parser error: Opening and ending tag
    mismatch: country line 4 and countries </countries>
  countries-malformed.xml:7: parser error : Premature end of data
    in tag country line 4
```


© Rheinwerk Verlag, Bonn 2024

Optional können Sie die Validierung einer XML-Datei auch anhand einer *Document Type Definition* (DTD) oder *XML Schema Definition* (XSD) durchführen. Syntaktisch korrekte XML-Dateien können Sie mit `--format` korrekt einrücken und lesbar formatieren:

```
$ xmllint --format countries-unformatted.xml

  <?xml version="1.0"?>
  <countries>
    <country code="AF" iso="4">Afghanistan</country>
    ...
```

Schließlich können Sie mit `xmllint` auch XPath-Ausdrücke verarbeiten:

```
$ xmllint --xpath '/*/MapTZ[57]/@WinID' timezone-mapping.xml

  WinID="Mountain Standard Time"
```

Um `xmllint` als Filter zu verwenden, müssen Sie ein alleinstehendes Minuszeichen als Parameter übergeben, beispielsweise so:

```
$ curl -s <url> | xmllint - --format
```

**Nicht Namespace-kompatibel**

XPath-Ausdrücke in `xmllint` werden nur dann korrekt ausgewertet, wenn das XML-Dokument keine Namespaces verwendet. Zwar gibt es im Internet Anleitungen, wie diese Einschränkung zur Not umgangen werden kann, besser ist es aber, gleich auf ein anderes Tool auszuweichen. Eine Option ist das im folgenden Abschnitt beschriebene `xmlstarlet`.

### XMLStarlet

Unter Linux und macOS stellt das Paket `xmlstarlet` das gleichnamige Kommando oder auch einfach `xml` (z. B. unter Arch Linux) zur Verfügung. Es bietet deutlich mehr Bearbeitungsmöglichkeiten als `xmllint`, insbesondere auch zur Veränderung von XML-Dokumenten. Allerdings ist das Projekt eingeschlafen, die aktuelle Version wurde seit 2014 nicht mehr verändert.

Ich zeige Ihnen an dieser Stelle lediglich das `sel`-Subkommando zur Auswertung von XPath-Ausdrücken. Das folgende Kommando definiert zuerst das Kürzel `ns` für den Default-Namespace in den `*.vbox`-Dateien. `-t -v 'xpath` wendet den angegebenen XPath-Ausdruck auf die XML-Datei an und gibt den resultierenden Wert (`-v` für *value*) aus. `-n` fügt der Ausgabe einen Zeilenumbruch hinzu.

Persönliches Exemplar für Thomas Bartholomäus

```
$ xml|xmlstarlet sel -N ns='http://www.virtualbox.org/' \
        -t -v  './/ns:HardDisk/@location' -n vbox/*.vbox

  Kali-Linux-2022.2-virtualbox-amd64-disk001.vdi
  kali-english.vdi
  ...
```

Die nicht eben intuitive Syntax von `xml sel` können Sie mit `xml sel --help` nachlesen. Weiterführende Dokumentation finden Sie hier:

*https://xmlstar.sourceforge.net/docs.php*

## 10.7 INI-Dateien

INI-Dateien sind Textdateien zur Speicherung von Schlüssel-Wertpaaren. Die Datei kann mit `[section]` in diverse Abschnitte gegliedert werden:

```
; Datei config.ini
[server]
hostname = example.com
port = 8080

[data]
imagePath = /mnt/images
dbName = mydb
```

INI-Dateien werden häufig zur Speicherung von Konfigurationsdaten verwendet. Zur Kennzeichnung von Kommentaren sind eigentlich nur Strichpunkte vorgesehen. Gerade im Linux-Umfeld wird aber zumeist auch # akzeptiert.

### Python

Am unkompliziertesten gelingt das Einlesen von INI-Dateien in Python mit dem Modul `ConfigParser`. Das folgende Listing zeigt die intuitive Anwendung. Beachten Sie, dass der Parser immer Zeichenketten zurückgibt. Wenn Sie die Portnummer 8080 als Zahl nutzen möchten, müssen Sie mit `int()` eine entsprechende Umwandlung durchführen.

```
# Beispieldatei read-config.py
from configparser import ConfigParser
config = ConfigParser()
config.read('config.ini')
print(config['server']['port'])   # Ausgabe: 8080
```


© Rheinwerk Verlag, Bonn 2024

### PowerShell

Obwohl INI-Dateien unter Windows durchaus gebräuchlich sind, fehlt in der PowerShell ein CmdLet zum einfachen Einlesen solcher Dateien. In vielen Fällen können Sie sich wie folgt behelfen:

```
> $config = Get-Content 'config.ini' |
            Select-String '^.*=.*[^=]*$' |
            ConvertFrom-StringData

> $config.port
  8080
```

`Get-Content` liest die Datei und liefert sie zeilenweise weiter. `Select-String` filtert mit einem regulären Ausdruck alle Zeilen heraus, die dem Muster `key = value` entsprechen. `ConvertFrom-StringData` macht aus den Schlüssel-Wert-Paaren eine Hashtable.

Diese simple Vorgehensweise stößt aber an ihre Grenzen, wenn in unterschiedlichen Gruppen dieselben Schlüssel verwendet werden. Wenn Sie komplexe INI-Dateien auswerten wollen, ohne in syntaktische Fallen zu tappen, verwenden Sie besser die beliebte PsIni-Erweiterung:

*https://www.powershellgallery.com/packages/PsIni*

### Bash

Der Bash fehlen wie der PowerShell Funktionen, um INI-Dateien auszuwerten. Bei vielen Linux-Distributionen können Sie dafür `crudini` zu Hilfe nehmen. Das gleichnamige Paket müssen Sie allerdings vorher installieren. Beachten Sie aber, dass es sich dabei um ein Python-Script handelt! `crudini` funktioniert also nur, wenn auch Python installiert ist!

Um einen einzelnen Wert auszulesen, übergeben Sie einfach die INI-Datei, den Namen des Abschnitts und den des Schlüssels. Diverse weitere Anwendungsformen beschreibt `crudini --help`.

```
$ crudini --get config.ini server port
  8080
```

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

# Kapitel 11
# Scripts automatisch ausführen

Viele Scripts haben das Ziel, administrative Aufgaben zu automatisieren: die Durchführung von Backups, den Upload von geänderten/neuen Dateien, das Monitoring und Logging von Servern. Dieses Kapitel beschreibt Möglichkeiten, wie Sie Scripts automatisiert regelmäßig ausführen. Ich konzentriere mich dabei primär auf zwei Verfahren: *Cron* für Linux und den *Windows Task Scheduler*. Außerdem gehe ich kurz auf Mechanismen ein, wie Sie Veränderungen im Dateisystem quasi »live« verfolgen und sofort reagieren können.

### Voraussetzungen für dieses Kapitel

Die Automatisierung von Scripts erfolgt losgelöst von der jeweiligen Script-Sprache. Insofern können Sie dieses Kapitel weitgehend unabhängig vom Rest des Buchs lesen. Der Abschnitt zu Cron ist leichter zu verstehen, wenn Sie über etwas Vorwissen zur Linux-Systemadministration verfügen. Insbesondere müssen Sie in der Lage sein, Systemdateien mit einem Texteditor zu verändern.

Die im Kapitel präsentierten Beispiele greifen auf Arbeitstechniken aus den vorigen Kapiteln zurück, z. B. auf die Anwendung des Kommandos `curl` (siehe Kapitel 8, »Textauswertung mit Filtern und Pipes«) oder auf die Auswertung von XML-Dateien (siehe Kapitel 10, »JSON XML und INI«).

Eine gute Fortsetzung zu diesem Kapitel wäre Kapitel 15, »Backups«: Dort finden Sie weitere Anwendungsbeispiele für die hier vorgestellten Verfahren.

## 11.1 Cron

Cron ist ein Hintergrundprogramm, das auf nahezu jedem Linux-Server läuft. Seine Aufgabe besteht darin, zu festgelegten Zeiten – z. B. jeden Tag um 3:00 Uhr morgens oder jeden Sonntag um 23:00 Uhr – Kommandos oder Scripts zu starten. Die Steuerung von Cron erfolgt durch die Datei `/etc/crontab`, deren ein wenig merkwürdige Syntax ich Ihnen gleich erläutern werde.

Cron ist bei vielen Linux-Distributionen standardmäßig installiert. Sollte die Datei `/etc/crontab` auf Ihrem System nicht existieren, können Sie leicht Abhilfe schaffen:

```
$ sudo apt update && sudo install cron        (Debian, Ubuntu)
$ sudo dnf install cronie && \                (Fedora, RHEL)
  sudo systemctl enable --now crond
```

### /etc/crontab

Die folgenden Zeilen zeigen einen Ausschnitt aus einer `crontab`-Datei.

```
# Datei /etc/crontab
# jeden Tag um 3:00: Backup
0 3 * * * root /etc/myscripts/backup.sh
# am ersten Tag jedes Monats um 0:00:
# statistische Auswertung der Login-Daten
0 0 1 * * root /etc/myscripts/statistics.py
```

`crontab`-Dateien beschreiben die auszuführenden Kommandos zeilenweise, wobei sich jeder Eintrag aus sieben Spalten zusammensetzt:

```
minutes hours day-of-month month weekday user cmd
```

- `minutes`: 0–59
- `hours`: 0–23
- `day-of-month`: 1–31
- `month`: 1–12
- `weekday`: 0–7 (0 = 7 = Sonntag, 1 = Montag usw.)
- `user`: Account, unter dem das Script ausgeführt werden soll (oft `root`)
- `cmd`: das auszuführende Kommando, oft einfach der Pfad zu einem Script

Zeilen, die mit # beginnen, sind Kommentare. Die letzte Zeile der `crontab`-Datei muss mit einem Newline-Zeichen enden. Für die ersten fünf Spalten in der `crontab`-Datei gelten die folgenden Syntaxregeln:

- Aufzählungen: `7,9,12` (ohne Leerzeichen!)
- Bereiche: `3-5`
- immer: `*`
- alle `n`: `*/10` entspricht `0,10,20,30...`

Normalerweise werden die Spalten durch ein logisches Und verknüpft. Ausnahme: Wenn `day-of-month` und `weekday` angegeben werden, gilt für diese beiden Spalten ein logisches Oder. Das Kommando wird sowohl am *n*-ten Tag des Monats als auch am angegebenen Wochentag ausgeführt.

Das folgende Listing gibt einige Syntaxbeispiele. Eigene Cron-Zeitkompositionen können Sie gut unter *https://crontab.guru* verifizieren.


© Rheinwerk Verlag, Bonn 2024

```
# alle 15 Minuten
*/15 * * * * root cmd
# täglich um 0:15, 1:15, 2:15 usw.
15 * * * * peter cmd
# täglich um 1:30
30 1 * * * maria cmd
# jeden Samstag um 0:29
29 0 * * 6 root cmd
# an jedem 1. des Monats um 6:25
25 6 1 * * www-data cmd
# am 1. und 15. des Monats sowie jeden Montag
# jeweils um Mitternacht
0 0 1,15 * 1 root cmd
```

Sofern Cron-Jobs auf einem Rechner ausgeführt werden, auf dem ein Mail-Server konfiguriert ist, werden die Ausgaben oder Fehlermeldungen an `root@localhost` versendet. Diese Adresse können Sie durch die Einstellung `MAILTO=...` innerhalb von `/etc/crontab` ändern.

### Ärger mit PATH und LANG

Für Cron-Jobs gilt eine andere `PATH`-Voreinstellung als im interaktiven Betrieb. Änderungen an `PATH` in `.bashrc` werden nicht berücksichtigt. Deswegen können Kommandos, die außerhalb der üblichen Verzeichnisse wie `/usr/bin` oder `/usr/sbin` installiert sind, nicht ausgeführt werden!

Testen Sie also unbedingt, ob Ihr Script auch beim Start durch Cron korrekt ausgeführt wird. Gegebenenfalls müssen Sie bei manuell installierten Kommandos im Script den vollständigen Pfad angeben (z. B. `/usr/local/bin/aws` zur Ausführung des AWS-Clients). Alternativ können Sie bei einigen Cron-Implementierungen die Variable `PATH` direkt in `/etc/crontab` neu definieren. Dabei geben Sie alle Verzeichnisse an, in denen nach Kommandos gesucht werden soll – beispielsweise so:

```
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
```

Eine weitere mögliche Fehlerquelle kann die Umgebungsvariable `LANG` sein, für die in Cron-Jobs möglicherweise eine andere Einstellung als bei der lokalen Ausführung Ihrer Scripts gilt. Eine Einstellung in `/etc/crontab` wie bei `PATH` ist nicht möglich. Sie können `LANG` aber zu Beginn Ihres Scripts einstellen.

### Persönliche Crontab-Dateien

Die Veränderung von `/etc/crontab` erfordert Systemadministratorrechte. Sollten Ihnen diese nicht zur Verfügung stehen, können Sie Ihre eigene Crontab-Datei verän-

Persönliches Exemplar für Thomas Bartholomäus

dern. Diese befindet sich je nach Linux-Distribution in /var/spool/cron/<user> oder in /var/spool/cron/crontabs/<user>.

Ihre persönliche Crontab-Datei öffnen Sie mit dem Kommando crontab -e. Standardmäßig kommt dabei der schwer zu bedienende Editor vi zum Einsatz. Falls Sie das Programm versehentlich gestartet haben, können Sie es mit [Esc] :q! [↵] verlassen, ohne Änderungen zu speichern. Gegebenenfalls können Sie vor der Ausführung des crontab-Kommandos über die EDITOR-Variable einen einfacheren Editor auswählen:

```
$ export EDITOR=/usr/bin/nano
$ crontab -e
```

Beachten Sie, dass bei der privaten Crontab-Datei die sechste Spalte für den Account entfällt! Das Kommando oder Script wird immer mit den Rechten Ihres Accounts ausgeführt. Das ist auch schon der größte Nachteil dieses Verfahrens: Viele Administrationsaufgaben (z. B. systemweite Backups) erfordern root-Rechte und können nicht in »gewöhnlichen« Accounts ausgeführt werden. Auch sudo hilft hier nicht weiter, weil dieses Kommando ja die interaktive Eingabe Ihres Passworts erfordert.

### hourly-, daily-, weekly- und monthly-Verzeichnisse

Wenn Ihnen der genaue Zeitpunkt egal ist, zu dem Ihre Cron-Aufgaben erledigt werden, können Sie das aufzurufende Script in einem der folgenden Verzeichnisse speichern:

```
/etc/cron.hourly
/etc/cron.daily
/etc/cron.weekly
/etc/cron.monthly
```

Der Vorteil dieses Verfahren besteht darin, dass Sie sich nicht mit der Crontab-Syntax auseinandersetzen müssen. Denken Sie daran, Ihre Script-Datei mit chmod +x als ausführbar zu kennzeichnen! Ihre Scripts werden mit root-Rechten ausgeführt.

### Cron-Alternativen

Unter Linux etabliert sich zunehmend *systemd-timer*. Der größte Vorteil im Vergleich zu Cron besteht darin, dass die gestarteten Jobs besser überwacht werden können. Dem steht aber eine viel größere Komplexität gegenüber. Während das Einrichten eines Cron-Jobs mit etwas Erfahrung in 30 Sekunden gelingt, müssen Sie für Timer-Jobs *zwei* relativ komplizierte Konfigurationsdateien erstellen. Eine gute Beschreibung des Mechanismus finden Sie im Archlinux-Wiki. Die dort zusammengefassten Informationen gelten auch für andere Distributionen.

*https://wiki.archlinux.org/title/Systemd/Timers*


© Rheinwerk Verlag, Bonn 2024

### Jobs unter macOS automatisiert starten

Cron steht auch unter macOS zur Verfügung, gilt dort aber als veraltet. Die Datei `/etc/crontab` existiert standardmäßig nicht. Sie brauchen die Datei aber nur zu erzeugen (z. B. mit `sudo nano /etc/crontab`) und einen korrekten Eintrag einzufügen, dann wird der Job ohne weitere Konfigurationsarbeiten ausgeführt. Insofern ist Cron bis auf Weiteres auch für macOS die einfachste Lösung zur Automatisierung des Script-Aufrufs.

Apple empfiehlt, anstelle von Cron das macOS-spezifische Programm `launchd` zu verwenden. Die Konfiguration erfolgt durch XML-Dateien, deren Syntax hier dokumentiert ist:

*https://bas-man.dev/post/launchd-instead-of-cron*
*https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/ScheduledJobs.html*

## 11.2 Beispiel: Webserver-Monitoring

Die Idee dieses Beispiels besteht darin, mit `curl` die Zeit zu messen, die der Download der Startseite einer Website beansprucht. Das `curl`-Kommando sieht so aus:

```
time=$(curl --connect-timeout 2 -s -S -w '%{time_total}\n' \
        -o /dev/null  2> /tmp/curl.error \
        https://example.com)
```

Die Bestandteile des Kommandos haben die folgende Bedeutung:

- `--connect-timeout 2` bricht den Verbindungsaufbau nach 2 Sekunden ab.
- `-s` unterdrückt Statusausgaben über den Download-Fortschritt.
- `-S` erlaubt abweichend von dieser Option Fehlerausgaben.
- `-w` schreibt die Laufzeit des Kommandos in die Standardausgabe. Diese Information landet in der Variablen `time`.
- `-o` leitet die heruntergeladene HTML-Datei nach `/dev/null` um. In dieser Linux-spezifischen Pseudo-Datei wird nichts gespeichert.
- `2>` leitet Fehlerausgaben in die angegebene Datei um.

Im Script wertet ein ganz ähnliches Kommando die Variablen `TMPFILE` und `HOST` aus. Die Einstellung `LANG=` stellt sicher, dass `curl` und `date` Zahlen und Daten international formatieren.

Wenn das `curl`-Kommando mit einem Fehlercode beendet wird, schreibt `cat` die Fehlermeldung in die Standardausgabe. Bei einer Ausführung des Scripts durch Cron wird diese Ausgabe per Mail an `root@localhost` gesendet (oder an einer andere, in

Persönliches Exemplar für Thomas Bartholomäus

/etc/crontab durch MAILTO angegebene Adresse). Bei einer korrekten Konfiguration werden Sie also per Mail verständigt, wenn die Website nicht mehr korrekt funktioniert.

Wurde curl dagegen ohne Fehler und ohne Überschreitung der Timeout-Zeit ausgeführt, speichert echo die benötigte Zeit und den aktuellen Zeitpunkt in die durch LOGFILE angegebene Datei.

```
# Beispieldatei webserver-ping.sh
LANG=
HOST=example.com
LOGFILE=/var/log/$HOST-ping-time.log
TMPFILE=/tmp/curl.error

time=$(curl --connect-timeout 2 -s -S -w '%{time_total}\n' \
       -o /dev/null  2> $TMPFILE \
       https://$HOST)

if [ $? -ne 0 ]; then
  # output is sent per mail if script is
  # run by cron on a Linux system with mail server
  cat $TMPFILE
else
  now=$(date)
  echo "$time sec @ $now" >> $LOGFILE
fi
```

Um das Script alle 15 Minuten auszuführen, habe ich /etc/crontab um die folgende Zeile ergänzt:

```
# in /etc/crontab
*/15 * * * * kofler /home/kofler/bin/webserver-ping.sh
```

Das Script befindet sich also innerhalb des Heimatverzeichnisses von kofler im Unterverzeichnis bin und soll mit den Rechten des Accounts kofler alle 15 Minuten ausgeführt werden. Die Ergebnisdatei /var/log/example.com-ping-time.log hat den folgenden Aufbau:

```
0.347425 sec @ Mon Feb 27 12:30:01 CET 2023
0.347546 sec @ Mon Feb 27 12:45:02 CET 2023
...
```

### Test und Fehlersuche

Im Unterschied zu vielen anderen Beispielen dieses Buch kann dieses Script nicht ohne Weiteres auf jedem Rechner einfach ausprobiert werden. Vielmehr benötigen Sie ein Linux-System mit Cron, idealerweise samt einem Mail-Server.


© Rheinwerk Verlag, Bonn 2024

Außerdem müssen Sie die Konfigurationsvariablen am Beginn des Scripts Ihren eigenen Wünschen entsprechend anpassen. Falls Sie das Script nicht mit root-Rechten ausführen, müssen Sie sicherstellen, dass die Logging-Datei für das Script schreibbar war. Wenn das Script von `kofler` ausgeführt wird und `HOST` und `LOGFILE` unverändert bleiben, müssen Sie einmalig die beiden folgenden Kommandos ausführen:

```
$ sudo touch /var/log/example.com-ping-time.log
$ sudo chown kofler:root /var/log/example.com-ping-time.log
```

Sobald das Script sich manuell fehlerfrei ausführen lässt, besteht der nächste Schritt darin, `/etc/crontab` einzurichten. Für erste Tests ist es empfehlenswert, die fünf Zeitspalten mit `* * * * *` einzustellen. Damit wird das Script einmal pro Minute ausgeführt.

Syntaxfehler in `/etc/crontab` führen dazu, dass weder Ihr Script noch sonstige Cron-Jobs ausgeführt werden. Um solchen Fehlern auf die Spur zu kommen, führen Sie `systemctl status cron` und `journalctl -u cron` aus. Sie erhalten damit Informationen über den aktuellen Cron-Status sowie die letzten Logging-Ausgaben.

### Echtes Monitoring

Dieses Mini-Script als Monitoring-Lösung zu verkaufen, ist doch ein wenig gewagt. »Echtes« Monitoring überwacht verschiedene Server-Parameter, sendet nur bei jeder Status-Meldung eine Warn-Mail (und nicht bei jedem Fehler, was zu einer Fülle von Monitoring-Mails führt), bietet eine grafische Visualisierung der überwachten Parameter usw.

Natürlich ist es denkbar, das mit eigenen Scripts zu realisieren. Sinnvoller ist es, dezidierte Monitoring-Tools wie *Grafana*, *Prometheus*, *Nagios* etc. einzusetzen. Deren erstmalige Inbetriebnahme ist zwar mit erheblichem Aufwand verbunden, dafür bieten derartige Tools aber eine Fülle von Funktionen kombiniert mit einer zweckmäßigen Benutzeroberfläche.

## 11.3 Windows Task Scheduler

Der *Windows Task Scheduler* (deutscher Programmname: *Aufgabenplanung*) ermöglicht es unter Windows, Programme und Scripts zu vorgegebenen Zeiten auszuführen. Er ist also das Windows-Gegenstück zu Cron. Das Programm ist, wie man es unter Windows erwartet, mit einer Benutzeroberfläche ausgestattet. Diese glänzt allerdings mit dem Charme des vorigen Jahrtausends (siehe Abbildung 11.1). Seine verschachtelten Dialoge sind bestenfalls so intuitiv wie die `crontab`-Syntax. Außerdem ist das Programm verblüffend lahm, manchmal bleibt es ganz hängen. Wenn der Start von Jobs nicht wie beabsichtigt funktioniert, gibt es trotz Event-Logging keine klaren Feh-

Persönliches Exemplar für Thomas Bartholomäus

lermeldungen. Kurz und gut, die Arbeit mit dem Programm ist eine Qual. Warum sich Microsoft nicht aufraffen kann, diese zentrale Windows-Komponente neu zu implementieren, bleibt schleierhaft. (Die aktuelle Version wurde seit 2006 nicht mehr nennenswert verändert.)

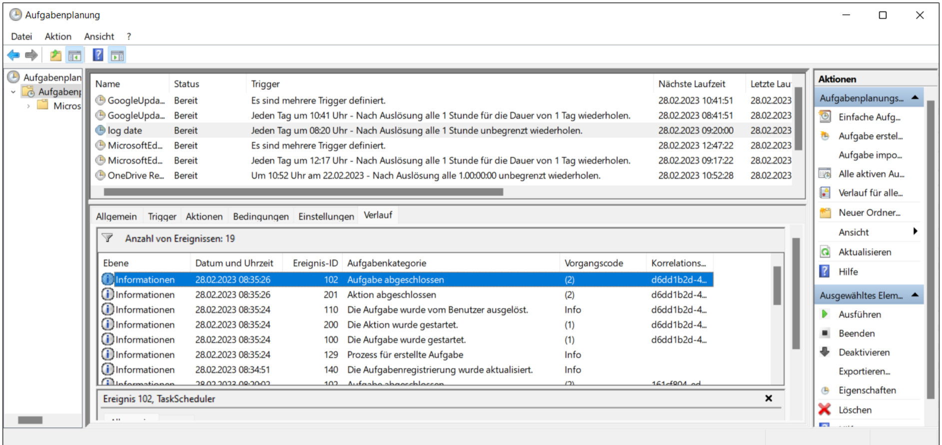



**Abbildung 11.1** Die antiquierte Benutzeroberfläche des Windows Task Schedulers

Um sich mit dem Task Scheduler vertraut zu machen, erstellen Sie ein winziges Script, das Datum und Uhrzeit in eine Logging-Datei protokolliert:

```
# Datei log-date.ps1
Get-Date >> date.log
```

Dieses Script soll nun stündlich ausgeführt werden. Dazu öffnen Sie den Task Scheduler und klicken in der Seitenleiste AKTIONEN auf das Kommando AUFGABE ERSTELLEN. Damit gelangen Sie in einen Dialog mit mehreren Dialogblättern.

- ALLGEMEIN: Hier geben Sie der Aufgabe einen Namen und geben an, in welchem Konto Ihr Script ausgeführt werden soll (üblicherweise Ihrem eigenen) und ob es nur nach einem Login gestartet werden soll oder immer, wenn der Computer läuft.

  Wenn Sie ein Script mit Administratorrechten ausführen wollen, ändern Sie nicht den Benutzer oder die Gruppe, sondern klicken nur auf die Option MIT HÖCHSTEN PRIVILEGIEN AUSFÜHREN.

- TRIGGER: In diesem Dialogblatt fügen Sie einen neuen Trigger ein, um festzulegen, wann und wie oft die Aktion stattfinden soll. Damit das Script stündlich ausgeführt wird, wählen Sie als Startzeitpunkt TÄGLICH mit der gerade aktuellen Uhrzeit sowie WIEDERHOLEN JEDE STUNDE sowie FÜR DIE DAUER VON: 1 TAG. (Die eigentlich plausiblere Einstellung FÜR DIE DAUER VON: UNBEGRENZT hat bei meinen Tests nicht funktioniert.)


© Rheinwerk Verlag, Bonn 2024

Im Unterschied zu Cron unterstützt der Task Scheduler nicht nur zeitliche Trigger, sondern kann Aktionen auch beim Eintreffen diverser Ereignisse ausführen – z. B. beim Login oder wenn sich der Rechner gerade im Leerlaufbetrieb befindet.

- AKTION: Als neue Aktion legen Sie fest, welches Script auszuführen ist. Die Vorgehensweise ist umständlicher als erwartet: Als PROGRAMM/SKRIPT dürfen Sie nämlich nicht einfach den Pfad Ihrer PowerShell-Script-Datei angeben. Stattdessen geben Sie hier `pwsh` für die PowerShell 7.*n* an. Nur wenn Ihr Script von der älteren Windows PowerShell 5.*n* ausgeführt werden soll, lautet der Programmname `powershell`.

  Beim Eingabefeld ARGUMENTE HINZUFÜGEN geben Sie die Option `-File` und den vollständigen Pfad zu Ihrem Script an. Passen Sie bei der Eingabe auf, dass Ihnen keine Tippfehler passieren. Vergessen Sie `-File` nicht!

  Standardmäßig ist bei der Ausführung des Scripts das Verzeichnis `%Windir%\System32\` aktiv. Wenn Sie das nicht wollen (z. B. weil das Script Dateien in einem anderen Verzeichnis lesen oder schreiben soll), müssen Sie bei STARTEN IN das gewünschte Verzeichnis angeben. Damit das Beispiel-Script Datum und Uhrzeit im selben Verzeichnis protokolliert, in dem sich die Script-Datei befindet, müssen Sie dieses Verzeichnis angeben.

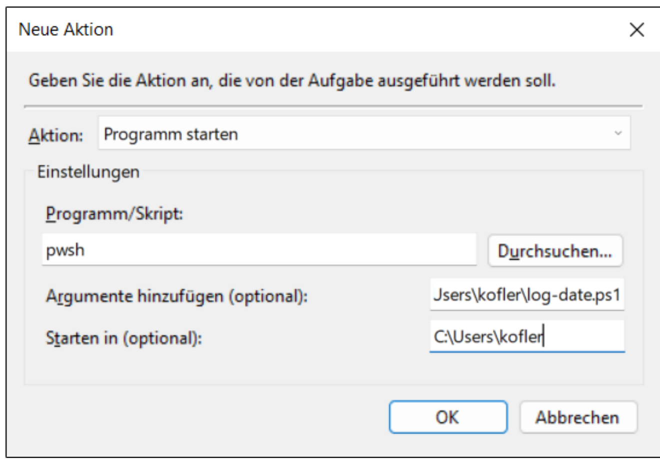



**Abbildung 11.2** Das Script »log-date.ps1« soll durch die PowerShell ausgeführt werden. Als aktives Verzeichnis wird »C:\Users\kofler« verwendet.

- BEDINGUNGEN: Dieses Dialogblatt sieht standardmäßig vor, die Aktion nur dann auszuführen, wenn Ihr Computer im Netzbetrieb läuft. Deaktivieren Sie diese Option, wenn Sie die Aufgabe auch ausführen bzw. testen wollen, während Sie Ihr Notebook im Akku-Betrieb verwenden!

Persönliches Exemplar für Thomas Bartholomäus

- EINSTELLUNGEN: Zuletzt können Sie einige Einstellungen vornehmen, ob Ihre Aufgabe exakt zum vorgesehenen Zeitpunkt erledigt werden soll, ob sie zu einem späteren Zeitpunkt nachgeholt werden darf, ob der Prozess nach einer bestimmten Laufzeit automatisch beendet werden soll (per Default sind hier absurde drei Tage vorgesehen ...) etc. Für erste Tests können Sie alle Optionen in ihren Voreinstellungen belassen.

Mit etwas Glück sollte das Script auf Anhieb laufen. Das können Sie durch einen Blick in die Datei `date.log` im eingestellten Verzeichnis (STARTEN IN) überprüfen. Dieser Datei sollte nun jede Stunde eine weitere Zeile mit dem gerade aktuellen Zeitpunkt hinzugefügt werden.

### Fehlersuche

Erfahrungsgemäß klappt die automatisierte Script-Ausführung nicht immer auf Anhieb. Zur Fehlersuche sollten Sie als ersten Schritt im Task Scheduler das Kommando VERLAUF FÜR ALLE AUFGABEN AKTIVIEREN einschalten. Hinter dieser etwas kryptischen Beschreibung verbirgt sich eine Logging-Funktion. Der Task Scheduler protokolliert nun, wann welche Aktion warum gestartet wird, wann sie endet usw.

Mit dem Kommando AUSFÜHREN können Sie Ihr Script zudem unabhängig vom gewählten Trigger-Zeitpunkt jederzeit manuell starten. Wann der nächste reguläre Startzeitpunkt Ihres Scripts ist, geht in der Aktionsliste aus der Spalte NÄCHSTE LAUFZEIT hervor.

Auf einem meiner Testrechner war der Task Scheduler partout nicht in der Lage, die `pwsh` zu finden, trotz (mutmaßlich) korrekter `PATH`-Einstellungen. Im Event-Logging wurde nur sehr unspezifisch *file not found* protokolliert, ohne klare Angabe, welche Datei nicht gefunden wurde (`pwsh`, das Script, eine andere Datei?). Ich habe letztlich die PowerShell komplett deinstalliert, mit `winget` neu installiert und den Rechner neu gestartet. Danach klappte es.

### Aufgaben per CmdLets einrichten

Erfreulicherweise kann der Task Scheduler auch durch CmdLets gesteuert werden. `Get-ScheduledTask` liefert eine schier endlose Liste aller im System gespeicherten Aktionen. Wenn Sie sich je gefragt haben, warum sich Windows oft so träge anfühlt, wissen Sie nun den Grund. Auf meinem Testrechner lieferte das CmdLet 165 aktive Aufgaben!

Die Oberfläche zum Windows Task Manager zeigt immer nur die Aktionen für einen bestimmten Pfad an, standardmäßig die für den Pfad \, also für das Root-Verzeichnis der AUFGABENBIBLIOTHEK. Alle anderen Aktionen können Sie ansehen, wenn Sie ein Unterverzeichnis auswählen, z. B. `Microsoft\Windows\Bitlocker`.


© Rheinwerk Verlag, Bonn 2024

```
> Get-ScheduledTask | Where-Object { $_.State -ne 'Disabled' }

  TaskPath                 TaskName                     State
  --------                 --------                     -----
  \                        GoogleUpdateTaskMachine...   Ready
  \                        log date                     Ready
  \                        start syncthing              Running
  \Microsoft\Windows\...   .NET Framework NGEN v4.0...  Ready
  ...
```

### Nur für Windows!

Alle hier präsentierten CmdLets stehen nur zur Verfügung, wenn Sie PowerShell-Scripts unter Windows ausführen. Unter Linux und macOS gibt es keinen Task Scheduler, sondern andere Mechanismen (cron, systemd-timer, launchd). Diese können nicht über CmdLets gesteuert werden.

Mit `Get-ScheduledTaskInfo` können Sie ermitteln, wann die Aufgabe zuletzt durchgeführt wurde und wann der nächste Start geplant ist:

```
> Get-ScheduledTaskInfo '\log date'
  LastRunTime        : 28.02.2023 09:24:24
  NextRunTime        : 28.02.2023 11:20:20
  NumberOfMissedRuns : 1
  ...

> Get-ScheduledTask 'log date' | Get-ScheduledTaskInfo
  (gleiches Ergebnis)
```

Durch die Kombination einiger CmdLets können Sie sogar eine neue Aufgabe einrichten. Hier erzeugt `New-ScheduledTaskTrigger` ein Objekt für den Zeitplan und `New-ScheduledTaskAction` ein Objekt für die Aufgabe. `Register-ScheduledTask` speichert den neuen Job.

```
> $trigger = New-ScheduledTaskTrigger -Once -At '6:00am' `
             -RepetitionInterval (New-TimeSpan -Hours 1) `
             -RepetitionDuration (New-TimeSpan -Days 1000)
> $action = New-ScheduledTaskAction  -Execute 'pwsh' `
             -Argument '-File C:\Users\kofler\log-date.ps1' `
             -WorkingDirectory 'C:\Users\kofler'
> Register-ScheduledTask -TaskName 'log-date' `
             -Trigger $trigger -Action $action

  TaskPath    TaskName    State
  --------    --------    -----
  \           log-date    Ready
```

Persönliches Exemplar für Thomas Bartholomäus

## 11.4 Beispiel: Währungskurse speichern

Die europäische Nationalbank veröffentlicht auf der folgenden Seite einmal täglich offizielle Referenzkurse zwischen dem Euro und anderen wichtigen Währungen:

*https://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml*

Die XML-Datei sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<gesmes:Envelope
  xmlns:gesmes="http://www.gesmes.org/xml/2002-08-01"
  xmlns="http://www.ecb.int/vocabulary/2002-08-01/eurofxref">
  <gesmes:subject>Reference rates</gesmes:subject>
  <gesmes:Sender>
    <gesmes:name>European Central Bank</gesmes:name>
  </gesmes:Sender>
  <Cube>
    <Cube time="2023-03-01">
      <Cube currency="USD" rate="1.0684"/>
      <Cube currency="JPY" rate="144.82"/>
      <Cube currency="CHF" rate="0.9997"/>
      ...
```

Das Ziel des folgenden Beispiels besteht darin, den Wechselkurs zwischen Euro und Schweizer Franken (CHF) einmal täglich in einer CSV-Datei zu speichern. Die CSV-Datei soll das folgende Format haben:

```
date;chf_rate
2023-03-01;0.9997
```

Dabei soll nicht das aktuelle Datum protokolliert werden, sondern das in der XML-Datei eingebettete Datum mit dem Tag `<Cube time=...>`. Die Auswertung der Datei gelingt trotz der merkwürdigen `Cube`-Verschachtelung relativ einfach (siehe Abschnitt 10.4, »XML in der PowerShell«). Neu ist an dieser Stelle nur das CmdLet `Invoke-RestMethod` zum Download der XML-Datei, das ich Ihnen in Kapitel 18, »REST-APIs nutzen«, näher vorstellen werde.

```
# Beispieldatei get-exchange-rate.ps1
# XML-Dokument herunterladen und auswerten
$csv = "chf-rates.csv"
$url = "https://www.ecb.europa.eu/stats/eurofxref/
         eurofxref-daily.xml"
$ecb = Invoke-RestMethod $url
$date = $ecb.Envelope.Cube.Cube.time
$rate = ($ecb.Envelope.Cube.Cube.Cube |
           Where-Object {$_.currency -eq 'CHF'}).rate
```


© Rheinwerk Verlag, Bonn 2024

```
# falls notwendig, CSV-Datei erzeugen
if (!(Test-Path $csv)) {
    "date;chf_rate" | Out-File $csv
}
# aktuellen Kurs hinzufügen
Add-Content $csv "$date;$rate"
```

Jetzt müssen Sie das Script nur noch beim Task Scheduler für den täglichen Aufruf anmelden. Da die Kurse üblicherweise gegen 16 Uhr mitteleuropäischer Zeit aktualisiert werden, ist ein zweckmäßiger Zeitpunkt für den täglichen Aufruf zwischen 17:00 und 18:00.

Da die EZB die Währungskurse nur an Werktagen berechnet, wird die CSV-Datei bei täglicher Ausführung Doppelgänger enthalten. Wenn Sie möchten, können Sie das Script dahingehend noch verbessern und vor `Add-Content` überprüfen, ob es bereits einen Eintrag für das in `$date` enthaltene Datum gibt.

## 11.5 Änderungen im Dateisystem verfolgen

Relativ oft kommt es vor, dass Sie Aktionen nicht zu einem vorgegebenen Zeitpunkt ausführen möchten, sondern dann, wenn sich Dateien ändern oder neue Dateien hinzukommen. Das Script, das in meiner täglichen Arbeit am häufigsten läuft, überwacht im lokalen Verzeichnis alle Markdown-Dateien. Sobald sich eine Datei ändert, ruft das Script diverse Kommandos auf, um aus der Markdown-Datei eine aktualisierte PDF mit der Vorschau des Kapitels zu erzeugen, an dem ich gerade arbeite. Dieses Bash-Script hat den folgenden Aufbau:

```
# Beispieldatei run-pandoc.sh
while true; do
    for mdfile in *.md; do
        pdffile=${mdfile%.md}.pdf
        if [ $mdfile -nt $pdffile ]; then
            echo $mdfile
            pandoc ... $mdfile -o $pdffile
        fi
    done
    sleep 1
done
```

Das Script ist leicht zu verstehen: `while true` bildet eine Endlosschleife. Das Script läuft also, bis es mit Strg+C gestoppt wird. `for mdfile` durchläuft alle `*.md`-Dateien und überprüft dann mit dem Operator `-nt` (*newer than*), ob die Markdown-Datei neuer ist als die entsprechende PDF-Datei. In diesem Fall wird der Name der Markdown-Datei ausgegeben und mit `pandoc` eine neue PDF-Datei erzeugt. (`pandoc` ist

Persönliches Exemplar für Thomas Bartholomäus

ein Kommando, um Dokumente zwischen verschiedenen Formaten umzuwandeln – siehe *https://pandoc.org*.)

Eine entscheidende Komponente des Scripts ist `sleep`. Damit legt das Script nach jedem Durchlauf eine Pause von einer Sekunde ein. (Je nachdem, welchen Zweck die Überwachung des Verzeichnisses hat, ist auch eine längere Pause denkbar.) Ohne `sleep` würde das Script eine CPU komplett auslasten. Mit `sleep` ist die CPU-Belastung durch das Script hingegen vernachlässigbar.

Diese Aufgabenstellung lässt sich ebenso elegant auch mit der PowerShell lösen. Der einzige Unterschied ist die `if`-Bedingung, die zusätzlich auch einen Test enthält, ob die PDF-Datei überhaupt existiert. Wenn das nicht der Fall ist, würde der Vergleich von `LastWriteTimeUtc` zu einem Fehler führen. Die Bash ist diesbezüglich etwas entspannter.

```
# Beispieldatei run-pandoc.ps1
while($true) {
    foreach($mdfile in Get-ChildItem -Path *.md) {
        $pdffile = $mdfile.FullName.Replace(".md", ".pdf")
        if(!(Test-Path $pdffile) -or
            ($mdfile.LastWriteTimeUtc -gt
             (Get-Item $pdffile).LastWriteTimeUtc))
        {
            Write-Host $mdfile.Name
            C:\path\to\pandoc.exe $mdfile.FullName -o $pdffile
        }
    }
    Start-Sleep -Seconds 1
}
```

### Inotifiy

Die vorhin präsentierten Scripts sind nur dann zweckmäßig, wenn die Anzahl der zu überwachenden Dateien klein ist. Gibt es denn keine bessere Lösung? Natürlich schon – je nachdem, unter welchem Betriebssystem Sie arbeiten und welche Programmiersprache Sie verwenden.

Ich konzentriere mich an dieser Stelle auf die Nutzung der Inotify-Funktion. Das ist eine Linux-Kernelfunktion zur Überwachung des Dateisystems. Sie können damit ein Verzeichnis anmelden und werden in der Folge immer verständigt, wenn es in diesem Verzeichnis Änderungen gibt.

Um Inotify in Python-Scripts zu nutzen, installieren Sie zuerst das `pyinotify`-Modul:

```
$ pip[3] install pyinotify
```


© Rheinwerk Verlag, Bonn 2024

Zur Anwendung erzeugen Sie ein `WatchManager`-Objekt. Mit `add_watch` geben Sie an, welche Datei bzw. welches Verzeichnis Sie überwachen möchten, welche Ereignisse Sie verarbeiten möchten (hier `IN_CLOSE_WRITE`, also das Schließen einer Datei nach einer Veränderung) und welche Funktion dann aufgerufen wird. Das `WatchManager`-Objekt übergeben Sie nun an den `Notifier`. Mit `loop` starten Sie die Ereignisverarbeitung. Wie in den vorigen Beispielen initiiert auch `loop` eine Endlosschleife. Das Script läuft also, bis Sie es mit Strg+C beenden.

Die Funktion `runPandoc` ist für den Aufruf des `pandoc`-Kommandos zuständig. Dort muss allerdings zuerst getestet werden, ob es sich bei der veränderten Datei überhaupt um eine Markdown-Datei handelt. (Sie können an `add_watch` leider keine Muster wie `*.md` übergeben.) Dazu wird der Parameter `ev` (wie *event*) ausgewertet, der Details über das aufgetretene Ereignis enthält.

```
# Beispieldatei run-pandoc.py
import pyinotify, subprocess

# diese Funktion wird durch pyinotify aufgerufen
def runPandoc(ev):
    if ev.name.endswith('.md'):
        mdfile = ev.name
        pdffile = mdfile.replace('.md', '.pdf')
        cmd = 'pandoc %s -o %s' % (mdfile, pdffile)
        print(cmd)
        subprocess.run(cmd, shell=True)

wm = pyinotify.WatchManager()
wm.add_watch('.', pyinotify.IN_CLOSE_WRITE, runPandoc)
notifier = pyinotify.Notifier(wm)
notifier.loop()
```

In Bash-Scripts können Sie die Inotify-Funktionen mit den Kommandos `inotifywait` und `inotifywatch` nutzen. Dazu müssen Sie vorher das Paket `inotify-tools` installieren. Einige einfache Anwendungsbeispiele sind hier dokumentiert:

*https://github.com/inotify-tools/inotify-tools/wiki#inotifywait*

### Inotify-Alternativen

Unter macOS bietet das *File System Events Application Programming Interface* (kurz FSEvents API) ähnliche Funktionen. In Scripts können Sie FSEvents am einfachsten über das Kommando `fswatch` nutzen. Dieses Kommando kann mit `brew` installiert werden:

*https://github.com/emcrisostomo/fswatch*

Persönliches Exemplar für Thomas Bartholomäus

Eine weitere Überwachungsmöglichkeit bietet das Programm `launchd`. Ein einfaches Anwendungsbeispiel finden Sie in diesem StackOverflow-Beitrag:

*https://stackoverflow.com/a/1516034*

Unter Windows gibt es den `FileSystemWatcher`. Die Nutzung dieser .NET-Funktion ist allerdings ziemlich kompliziert. Dieser Blog-Beitrag zeigt ein Code–Beispiel für die PowerShell:

*https://powershell.one/tricks/filesystem/filesystemwatcher*


© Rheinwerk Verlag, Bonn 2024

# Kapitel 12
# SSH

SSH steht für *Secure Shell* und ist seit Jahrzehnten das bevorzugte Verfahren in Linux und macOS, um im Textmodus auf einem anderen Rechner zu arbeiten und dort Kommandos auszuführen. Erfreulicherweise ist Microsoft vor ein paar Jahren ebenfalls auf den SSH-Zug aufgesprungen, sodass SSH jetzt endlich plattformübergreifend funktioniert. Für das Scripting ist SSH aus zweierlei Gründen wichtig:

- Via SSH können Sie auf anderen Rechnern arbeiten, dort Scripts erstellen und diese schließlich ausführen. SSH ist insbesondere bei der Server-Administration eine administrative Voraussetzung.

  Es muss aber nicht immer ein Server sein! Ich verwende SSH häufig auch, um auf meinem Raspberry Pi zu arbeiten. Der Raspberry Pi braucht dazu weder Bildschirm noch Tastatur – eine Netzwerkverbindung reicht. Analog gilt dies für die zahlreichen virtuellen Maschinen in meinem Arbeitsalltag: Oft brauche ich gar keine grafische Oberfläche und kann die anstehenden Kommandos oder die Scripting-Programmierung in einem Terminal mit einer SSH-Verbindung erledigen.

  Auch der Editor VSCode und SSH bilden ein gutes Team! Mehr dazu erfahren Sie in Kapitel 13, »Visual Studio Code«.

- Für viele Scripts mindestens ebenso wichtig ist die Möglichkeit, Dateien oder ganze Verzeichnisse sicher zwischen verschiedenen Rechnern zu übertragen. Die Kommandos `scp` und `rsync` sowie das Verfahren Secure FTP greifen auf SSH zurück. Backup- und Upload-Scripts setzen daher häufig einen SSH-Zugang voraus. Außerdem müssen Sie wissen, wie die Authentifizierung mit SSH-Keys funktioniert. Der Umgang mit solchen Schlüsseln ist daher ein wichtiges Thema in diesem Kapitel.

### Voraussetzungen für dieses Kapitel

Was das Scripting betrifft, gibt es keine inhaltlichen Voraussetzungen. Sie können dieses Kapitel also losgelöst von den restlichen Kapiteln lesen. Die Beispiel-Scripts setzen Bash- und PowerShell-Grundkenntnisse voraus. Außerdem sollten Sie das Kommando `find` kennen (siehe Abschnitt 6.2, »Dateien suchen«). Soweit Sie unter Linux arbeiten, ist ein administratives Grundwissen natürlich immer von Vorteil.

Persönliches Exemplar für Thomas Bartholomäus

## 12.1 SSH-Client und -Server installieren

Damit Sie eine SSH-Verbindung vom lokalen Rechner *A* zu einem zweiten Rechner *B* herstellen können, muss auf *A* ein SSH-Client zur Verfügung stehen (üblicherweise das Kommando ssh) und auf *B* ein SSH-Server laufen. Dieser Abschnitt klärt die Voraussetzungen, bevor Sie SSH nutzen können.

### Linux

Unter Linux gehört das Kommando ssh gewissermaßen zum Grundwortschatz und ist fast immer installiert. Überzeugen Sie sich mit ssh -V im Terminal! Das Kommando zeigt damit die installierte Version an.

```
$ ssh -V

  OpenSSH_9.2p1, OpenSSL 3.0.8 7 Feb 2023
```

Sollte das Kommando ssh tatsächlich nicht installiert sein, schafft die Installation des gleichnamigen Kommandos Abhilfe:

```
$ sudo apt update                       (Debian, Ubuntu)
$ sudo apt install openssh-client

$ sudo dnf install openssh-client       (Fedora, Red Hat)
```

Nicht ganz so selbstverständlich ist der Betrieb eines SSH-Servers. Klarheit schafft ein kurzer Test:

```
$ systemctl status sshd

  sshd.service - OpenSSH Daemon
     Loaded: loaded (...)
     Active: active (running) since ...
```

Wenn der Dienst sshd nicht bekannt ist oder nicht läuft, gelingt die Installation im Handumdrehen. Ich habe hier wieder die Kommandos für die wichtigsten Distributionen zusammengefasst:

```
$ sudo apt update                       (Debian, Ubuntu)
$ sudo apt install openssh-server

$ sudo dnf install openssh-server       (Fedora, Red Hat)
$ systemctl enable --now sshd
```


© Rheinwerk Verlag, Bonn 2024

### macOS

Unter macOS sind SSH-Client und -Server vorinstalliert. Der SSH-Server muss aber explizit aktiviert werden. Dazu öffnen Sie in den Systemeinstellungen das Modul ALLGEMEIN • TEILEN und aktivieren die Option ENTFERNTE ANMELDUNG. Fertig!

### Windows

Bei aktuellen Windows-Versionen ist der SSH-Client (also das Kommando `ssh`) standardmäßig installiert, der SSH-Server dagegen nicht. Zur Installation fehlender Komponenten öffnen Sie das Programm *Einstellungen*. Sie finden den *OpenSSH-Client* und den *OpenSSH-Server* – gut versteckt – unter APPS • OPTIONALE FEATURES • OPTIONALES FEATURE HINZUFÜGEN. (Warum das Programm nicht einfach über den Microsoft Store installiert werden kann, entzieht sich jeder Logik. Aber ich will hier nicht klagen, es ist großartig, dass Microsoft endlich einen offiziellen SSH-Client und -Server anbietet und so die Installation des unsäglichen PuTTY-Programms unnötig macht.)

Den SSH-Client können Sie nach der Installation als Kommando `ssh` im Terminal ausführen. Beim SSH-Server ist noch ein Schritt notwendig. Das Programm ist zwar installiert, es läuft aber noch nicht als Dienst. Abhilfe schafft das Programm *Dienste*, das Sie im Startmenü finden. Dort suchen Sie den Eintrag *OpenSSH Server* und öffnen per Doppelklick den Einstellungsdialog. Dort klicken Sie auf den Button STARTEN, um den Server-Dienst erstmalig zu starten. Außerdem stellen Sie STARTTYP = AUTOMATISCH ein, um ihn in Zukunft immer beim Hochfahren des Rechners zu aktivieren.

Statt sich durch Einstellungsdialoge zu klicken, können Sie die Installation und den Dienststart auch im Terminal durchführen:

```
> Add-WindowsCapability -Online -Name OpenSSH.Client
> Add-WindowsCapability -Online -Name OpenSSH.Server
> Start-Service sshd
> Set-Service -Name sshd -StartupType 'Automatic'
```

Wenn Sie sich nun von einem externen Rechner via SSH auf Ihrem Windows-Rechner anmelden, landen Sie unvermittelt in der IT-Steinzeit. Unbegreiflicherweise verwendet der SSH-Server `cmd.exe` als Default-Shell. Damit Sie innerhalb der SSH-Session die PowerShell verwenden können, führen Sie `pwsh` aus.

Damit die PowerShell automatisch bei jedem SSH-Login verwendet wird, müssen Sie einen Registry-Eintrag verändern. Das gelingt mit den folgenden drei Kommandos, die in einem Terminal mit Administratorrechten ausgeführt werden müssen:

```
> $PSPATH = (Get-Command -Name pwsh).path
> New-Item "HKLM:\Software\OpenSSH" -Force
> New-ItemProperty "HKLM:\SOFTWARE\OpenSSH"  -Name DefaultShell `
    -Value "$PSPATH" -PropertyType String -Force
```

Persönliches Exemplar für Thomas Bartholomäus

### Plötzlich »Permission denied« beim SSH-Login

Sollte sich bei einem PowerShell-Update oder bei einer PowerShell-Neuinstallation der Ort ändern, wo `pwsh.exe` gespeichert ist, kann der SSH-Server die PowerShell nicht mehr starten. Die Fehlermeldung lautet in solchen Fällen aber nichtssagend *Permission denied*.

Dafür kann es unzählige Gründe geben, vom falschen Passwort bis zur falschen Konfiguration in `opensshd_config`. Zu den möglichen Fehlerursachen zählt aber eben auch ein geänderter Pfad zu `pwsh.exe`!

### Automatisch Administrator

Wenn Sie einen SSH-Login für einen Benutzer durchführen, der Administratorrechte hat, dann erhalten Sie diese Rechte in der SSH-Session unmittelbar!

Das ist ein wichtiger Unterschied zum interaktiven Arbeiten: Normalerweise arbeiten Sie ja mit reduzierten Rechten und führen nur bei Bedarf – etwa durch den expliziten Start eines Terminals mit Administratorrechten – Aktionen mit Administratorrechten aus. Windows verhält sich diesbezüglich auch anders als Linux oder macOS, wo Sie die Administratorrechte für einzelne Kommandos explizit mit `sudo` anfordern müssen.

## Editor in einer SSH-Sitzung verwenden

Unter Linux oder macOS ist die Sache trivial: Um eine Datei direkt in einer SSH-Sitzung zu verändern, starten Sie einfach einen Editor. `nano` und `vim` stehen fast immer zur Verfügung. Unzählige weitere Editoren können bei Bedarf extra installiert werden – beispielsweise `emacs`, `joe` oder `zile`.

Unter Windows ist das Konzept »Editor ohne grafische Benutzeroberfläche« allerdings bis heute unbekannt. Standardmäßig steht kein geeignetes Programm zur Verfügung. Das ändert sich, wenn Sie Git installieren. Bei dieser Gelegenheit wird üblicherweise auch die Git-Bash mit den Editoren `vim` und `nano` installiert. Der Start der beiden Programme gelingt wie folgt:

```
> & "C:\Program Files\Git\usr\bin\nano.exe" myscript.ps1
> & "C:\Program Files\Git\usr\bin\vim.exe" myscript.ps1
```

Es ist übrigens nicht empfehlenswert, `C:\Program Files\Git\usr\bin` in die `Path`-Variablen einzubauen! In diesem Verzeichnis sind unzählige Linux-Tools wie `cd`, `ls`, `cp` usw. installiert. Da diese Namen auch als PowerShell-Aliasse gebräuchlich sind, kann das zu Problemen führen. Besser ist es, in der Profile-Datei (Pfad siehe `$PROFILE`) eine Funktion zum Aufruf Ihres Editors einzubauen:


© Rheinwerk Verlag, Bonn 2024

```
# in der Datei $PROFILE (üblicherweise C:\Users\<name>\
# Documents\PowerShell\Microsoft.PowerShell_profile.ps1)
function nano {
    & "C:\Program Files\Git\usr\bin\nano.exe" $args
}
```

Gegebenenfalls müssen Sie den Installationspfad anpassen. In der Folge können Sie nun `nano` in einer SSH-Sitzung wie folgt ausführen, um eine Script-Datei zu ändern:

```
> nano myscript.ps1
```

Eine denkbare Alternative ist die Verwendung von VSCode über eine SSH-Verbindung (siehe Abschnitt 13.3, »Remote-SSH-Erweiterung«).

### SSH-Server absichern

Der SSH-Server ist ein beliebtes Angriffsziel für Hacker: Diese versuchen, eine Kombination aus Loginname und Passwort zu erraten. (Der Angriff erfolgt natürlich nicht manuell, sondern durch automatisierte Tools.) Das beliebteste Angriffsziel bei Linux-Servern ist der `root`-Account. Es gibt einige elementare Absicherungsmaßnahmen:

- Verwenden Sie für *alle* Accounts gute Passwörter (mindestens 10 Zeichen).
- Blockieren Sie den `root`-Login (`PermitRootLogin no` in `/etc/ssh/sshd_config`).
- Installieren Sie das Programm *Fail2ban*, das IP-Adressen nach mehreren fehlerhaften Login-Versuchen für einige Minuten blockiert.
- Nutzen Sie eine Authentifizierung mit Schlüsseln statt mit Passwörtern. Wie das geht, zeige ich in Abschnitt 12.4.

## 12.2 Mit SSH arbeiten

Sofern Sie noch keine praktische Erfahrung haben, sollten Sie sich zuerst mit dem Kommando `ssh` (dem SSH-Client) vertraut machen. Dazu öffnen Sie ein Terminal und führen das folgende Kommando aus:

```
> ssh user@hostname

  The authenticity of host 'hostname (123.124.125.126)'
  can't be established. ECDSA key fingerprint is
  SHA256:E3IH3027Bc+5DvsvtenJkma1v5nI3owg08ZZqUR2BYk.
  Are you sure you want to continue connecting? yes
  Warning: Permanently added 'hostname,123.124.125.126' (ECDSA)
  to the list of known hosts.

  user@hostname's password: **********
```

Persönliches Exemplar für Thomas Bartholomäus

```
user@hostname$ grep VERSION /etc/os-release

  VERSION="20.04.5 LTS (Focal Fossa)"
  VERSION_ID="20.04"
  VERSION_CODENAME=focal

user@hostname$ exit
```

Dabei ersetzen Sie `user` durch den Account-Namen auf dem betreffenden Rechner und `hostname` durch den Netzwerknamen des Rechners bzw. durch dessen IP-Adresse. Falls die Account-Namen auf dem lokalen Rechner und dem Remote Host übereinstimmen, können Sie ihn auch weglassen (also einfach `ssh hostname` ausführen). Zur Authentifizierung geben Sie das Passwort des Accounts auf dem entfernten Rechner an (nicht Ihr lokales Passwort!).

Um SSH kennenzulernen, können Sie zur Not anstelle des Hostname einfach `localhost` angeben (also `ssh localhost`). Damit stellen Sie eine SSH-Verbindung zum lokalen Rechner her und testen damit sowohl den SSH-Client als auch den lokalen SSH-Server.

Nach dem erfolgreichen Login können Sie auf dem externen Rechner Kommandos ausführen. Im obigen Listing habe ich mit `grep` die Version der installierten Linux-Distribution ermittelt. `exit` bzw. Strg+D beenden die SSH-Sitzung.

## Host-Verifizierung

Beim ersten Verbindungsaufbau zu einem Host fragt der SSH-Client, ob Sie diesem Rechner vertrauen (siehe das obige Listing). Ehrlicherweise haben Sie da nicht viele Optionen – nur wenn Sie zustimmen, können Sie mit dem Login fortsetzen. Falls Sie den externen Rechner selbst eingerichtet haben oder den Administrator bzw. die Administratorin kennen, könnten Sie überprüfen, ob der »Fingerprint« (also die Kurzform) des SSH-Keys des Servers übereinstimmt. Um den Fingerprint eines Servers zu ermitteln, führen Sie dort das folgende Kommando aus, wobei Sie wieder `hostname` durch den tatsächlichen Rechnernamen ersetzen:

```
$ ssh-keyscan hostname | ssh-keygen -lf -

   256 SHA256:Sma6TJ79bONK8PMISjxPIYUi...MK3HQ hostname (ED25519)
  2048 SHA256:i9FlmpRieIFIEHNfHSzFgjp4...Y3ZmO hostname (RSA)
   256 SHA256:E3IH3027Bc+5DvsvtenJkma1...R2BYk hostname (ECDSA)
```

## Mögliche Probleme und ihre Ursache

Bei vielen Linux-Distributionen ist der SSH-Login für `root` aus Sicherheitsgründen blockiert oder nur dann erlaubt, wenn die Authentifizierung mit Schlüsseln erfolgt


© Rheinwerk Verlag, Bonn 2024

(siehe Abschnitt 12.4). Die empfohlene Vorgehensweise zur Erledigung administrativer Arbeiten via SSH besteht darin, den Login mit einem anderen Account durchzuführen und dann in der Sitzung `sudo` zu verwenden. Alternativ können Sie den `root`-Login durch eine Anpassung in der Datei `/etc/ssh/sshd_config` auf dem Rechner mit dem SSH-Server explizit zulassen. Änderungen an dieser Datei werden erst wirksam, wenn Sie den SSH-Server zum Neueinlesen dieser Datei auffordern (`systemctl reload sshd` unter Linux).

SSH-Logins in lokale virtuelle Maschinen scheitern unter Umständen an der Netzwerkanbindung. Insbesondere VirtualBox ist diesbezüglich problematisch. Virtuelle Maschinen sind dort per *Network Address Translation* (NAT) mit dem lokalen Netzwerk verbunden, SSH ist blockiert. Abhilfe schafft eine Portweiterleitung, wie ich dies am Ende des folgenden Blog-Beitrags beschrieben habe:

*https://kofler.info/ubuntu-22-04-in-virtualbox-7-unter-windows*

Der SSH-Client merkt sich bei jedem Login den Hostnamen bzw. die IP-Adresse des Servers und den vom Server verwendeten Schlüssel. Sollte sich der Schlüssel ändern (z. B. im Zuge einer Neuinstallation des Servers), zeigt das `ssh`-Kommando eine drastische Fehlermeldung an und blockiert den Login:

```
WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now
(man-in-the-middle attack)!
```

Wenn Sie wissen, dass der betroffene Server neu aufgesetzt wurde, trifft die Warnung nicht zu. Sie müssen aber die Zeile für den betreffenden Hostnamen oder die IP-Adresse in der lokalen Datei `.ssh/known_hosts` suchen und löschen. Beim nächsten Login erscheint dann die schon bekannte Frage, ob Sie dem Server vertrauen.

### Linux- und macOS-Kommandos ausführen

Anstatt `ssh` interaktiv zu nutzen, können Sie auf dem entfernten Rechner auch direkt ein Kommando ausführen. Das Kommando und seine Parameter werden als Parameter an `ssh` übergeben. `grep` wird hier nicht auf dem lokalen Rechner ausgeführt, sondern auf einem entfernten Linux- oder macOS-Host.

```
> ssh user@linuxhost 'grep VERSION /etc/os-release'

  user@linuxhost's password: **********

  VERSION="20.04.5 LTS (Focal Fossa)"
  VERSION_ID="20.04"
  VERSION_CODENAME=focal
```

Persönliches Exemplar für Thomas Bartholomäus

Aus dieser scheinbar trivialen Funktion ergeben sich weitreichende Möglichkeiten: Sie können nun beispielsweise auf dem entfernten Rechner `tar` starten, das Archiv des Verzeichnisses `data` an die Standardausgabe weiterleiten (geben Sie dazu einen Bindestrich `-` nach der Option `-f` an, also `-f -`). Die Standardausgabe leiten Sie mit `|` als Eingabe für ein zweites `tar`-Kommando weiter, das lokal läuft. Damit können Sie einen ganzen Verzeichnisbaum sicher via SSH von einem Rechner zum nächsten übertragen.

```
$ ssh user@hostname 'tar -czf - data' | tar -xzC ~/copy/ -f -
```

Um in einem Bash-Script mehrere Kommandos via `ssh` ausführen, verwenden Sie am besten die Heredoc-Syntax (siehe Abschnitt 3.10, »Zeichenketten«). Die Option `-T` verhindert, dass `ssh` ein Pseudo-Terminal öffnet. Das ist hier unerwünscht, weil die Kommandoausführung nicht interaktiv erfolgen soll.

```
ssh -T root@host <<ENDSSH
rm -f /etc/file1
cp /root/file2 /userxy/file3
...
ENDSSH
```

Beim ersten SSH-Verbindungsaufbau zu einem neuen Host fragt `ssh`, ob Sie dem Host vertrauen. Normalerweise ist diese Rückfrage sinnvoll. Wenn Sie aber mit `ssh` automatisiert auf mehreren Hosts oder virtuellen Maschinen Arbeiten durchführen möchten, stört die Rückfrage. Abhilfe schafft in solchen Fällen die Option `-o Strict\-HostKeyChecking=no`.

### Windows-Kommandos ausführen

Die Kommandoausführung funktioniert auch für Windows-Hosts, wenngleich die syntaktischen Spielarten hier deutlich kleiner sind. Sofern Sie dort die PowerShell als Standard-Shell eingestellt haben (siehe Abschnitt 12.1, »SSH-Client und -Server installieren«), können Sie via SSH ein Kommando ausführen.

```
$ ssh user@winhost 'Get-ChildItem C:\\'

  user@winhost's password: **********

  Directory: C:\

  Mode            LastWriteTime      Name
  ----      --------------------     ----
  d-r--     17.02.2023    08:20      Program Files
  d-r--     17.02.2023    08:20      Program Files (x86)
  d-r--     13.07.2022    16:02      Users
  d----     20.02.2023    16:13      Windows
```


© Rheinwerk Verlag, Bonn 2024

### SSH Remoting in der PowerShell

Mit den Kommandos `Enter-PSSession` bzw. `New-PSSession` können Sie über das *Windows Remote Management* Kommandos auf externen Windows-Rechnern ausführen. Seit PowerShell 6 gibt es diese Funktion auch für SSH-Verbindungen, dementsprechend ist von *SSH Remoting* die Rede. Vorher sind allerdings serverseitig Vorbereitungsarbeiten erforderlich.

Damit SSH Remoting auf einem externen Windows-Rechner genutzt werden kann, muss in dessen Datei `sshd_config` eine neue `Subsystem`-Zeile eingefügt werden. Die `Subsystem`-Anweisung ist hier nur aus Platzgründen mit \ über zwei Zeilen verteilt. Sie muss in einer Zeile und ohne das Zeichen \ angegeben werden! Laut Dokumentation wird die Option `-nologo` ab der PowerShell-Version 7.4 nicht mehr erforderlich sein.

```
# Windows-Host: Datei $env:ProgramData\ssh\sshd_config
...
Subsystem powershell c:/progra~1/powershell/7/pwsh.exe \
  -sshs -nologo
```

Am einfachsten öffnen Sie die Datei, indem Sie den Editor aus dem Terminal heraus starten (z. B. mit `code $env:ProgramData\ssh\sshd_config`). Dabei wird automatisch `$env:ProgramData` durch den entsprechenden Pfad auf Ihrem Rechner ersetzt (zumeist `C:\ProgramData`).

Die merkwürdige `Subsystem`-Einstellung `c:/progra~1/powershell/7/pwsh.exe` ist der übliche Pfad zur PowerShell-Installation in DOS-Notation (also mit maximal acht plus drei Zeichen pro Datei- oder Verzeichnisname). Diese antike Schreibweise ist notwendig, weil OpenSSH Windows-Dateinamen mit Leerzeichen nicht verarbeiten kann.

Möglicherweise lautet der richtige Pfad auch `c:/progra~2`! Die folgenden Kommandos zeigen, wie Sie die Kurzschreibweise herausfinden können. Zuerst ermitteln Sie mit `Get-Command` den Installationsort der PowerShell. Danach führen Sie `cmd.exe` aus, um die Kurzschreibweise des Programmverzeichnisses zu ermitteln:

```
> (Get-Command pwsh).Path
  C:\Program Files (x86)\PowerShell\7\pwsh.exe
> cmd /c 'for %X in ("C:\Program Files (x86)") do @echo %~sX'
  C:\PROGRA~2
```

Zuletzt müssen Sie den SSH-Server neu starten:

```
> Restart-Service sshd
```

Die Vorbereitungsarbeiten zur Nutzung eines Linux-Rechners für SSH Remoting sehen ganz ähnlich aus. Auch unter Linux müssen Sie eine `Subsystem`-Zeile in `sshd_config` einbauen. Anstelle von `/usr/bin/pwsh` müssen Sie den auf Ihrer Distribution gültigen Pfad zur PowerShell verwenden. Den Pfad ermitteln Sie mit `which pwsh`.

Persönliches Exemplar für Thomas Bartholomäus

```
# Linux-Host: Datei /etc/ssh/sshd_config
...
Subsystem powershell /usr/bin/pwsh -sshs -nologo
```

Das folgende Kommando startet den SSH-Server neu:

```
$ sudo systemctl restart sshd
```

Nach diesen Vorbereitungsarbeiten können Sie mit `New-PSSession` eine SSH-Verbindung herstellen und diese als Parameter an `Invoke-Command` verwenden. Für das folgende Beispiel habe ich lokal auf einem Windows-Rechner gearbeitet, dabei eine Verbindung zum im Netzwerk befindlichen Linux-Rechner hergestellt und dort `Get-ChildItem` ausgeführt. Mit `Remove-PSSession` beenden Sie die SSH-Verbindung.

```
> $session = New-PSSession linuxhostname -Username kofler
> Invoke-Command -Session $session { Get-ChildItem /etc/*.conf }

  Directory: /etc
  Mode          LastWriteTime     Length Name
  ----          -------------     ------ ----
  --r--   02/10/2023    22:08        833 appstream.conf
  --r--   01/27/2023    19:44          0 arptables.conf
  --r--   12/06/2021    13:27       1438 dhcpcd.conf

> Remove-PSSession $session
```

Weitere Konfigurationstipps und Anwendungsbeispiele finden Sie hier:

*https://learn.microsoft.com/en-us/powershell/scripting/learn/remoting/ssh-remoting-in-powershell-core*
*https://github.com/PowerShell/Win32-OpenSSH/issues/1498*

## 12.3 scp und rsync

Mit `scp` können Sie eine Datei zwischen dem lokalen und dem entfernten Rechner hin und her kopieren. Die folgenden Zeilen fassen die wichtigsten `scp`-Syntaxvarianten zusammen. Anders als beim `ssh`-Kommando muss dem Hostnamen immer ein Doppelpunkt folgen! (Zur Erinnerung: `.` ist die Kurzschreibweise für das aktuelle Verzeichnis.)

```
# lokale Datei zu einem externen Host kopieren (Upload)
$ scp filename user@host:
$ scp filename user@host:path/newfilename

# Datei des externen Hosts in das lokale Dateisystem
# kopieren (Download)
```


© Rheinwerk Verlag, Bonn 2024

```
$ scp user@host:filename .
$ scp user@host:path/filename .

# ganze Verzeichnisbäume rekursiv kopieren
$ scp -r localdir/ user@host:
$ scp -r user@host:remotedir .
```

**Lästige Passwortangabe**

Auch wenn dies in den Listings aus Platzgründen nicht enthalten ist, muss jedes scp-Kommando mit der Eingabe des Passworts quittiert werden. Das ist natürlich lästig. Abhilfe schafft die Verwendung eines SSH-Schlüssels (siehe Abschnitt 12.4, »SSH-Authentifizierung mit Schlüsseln«).

Wie bei der interaktiven Verwendung von ssh erfolgt die Kommunikation verschlüsselt. Selbst wenn es einem Angreifer gelingt, die Datenpakete mitzuschneiden, kann dieser die Daten (nach heutigem Wissensstand) ohne Passwort nicht entschlüsseln.

### Copy-Item mit SSH Remoting

scp ist unter Windows ein herkömmliches Kommando (eine *.exe-Datei). Der Aufruf solcher Kommandos erscheint in PowerShell-Scripts immer als Fremdkörper. Eine Alternative besteht darin, SSH Remoting einzurichten (siehe Abschnitt 12.2, »Mit SSH arbeiten«). Unter dieser Voraussetzung können Sie zuerst mit New-PSSession eine SSH-Verbindung herstellen und das Session-Objekt dann an Copy-Item übergeben:

```
> $session = New-PSSession linuxhostname -Username kofler
> Copy-Item localfile /home/kofler/somesubdir -ToSession $session
> Remove-PSSession $session
```

Wichtig ist, dass Sie bei dieser Form der Anwendung von Copy-Item immer einen absoluten Pfad angeben müssen. (Im Gegensatz dazu arbeitet scp automatisch relativ zum Heimatverzeichnis des Accounts der SSH-Verbindung.)

### rsync

Linux- und macOS-Rechner können mit rsync Verzeichnisse synchronisieren. rsync funktioniert sowohl für lokale Verzeichnisse als auch für Verzeichnisse auf entfernten, via SSH erreichbaren Rechnern. In diesem Fall ist die Basissyntax exakt gleich wie bei scp, d. h., Sie müssen lediglich scp durch rsync ersetzen. (Sie können das Detailverhalten von rsync durch eine Menge Zusatzoptionen steuern, auf die ich hier aber nicht eingehe. Werfen Sie mit man rsync ein Blick in das Handbuch!)

Im Vergleich zu scp gibt es drei wesentliche Unterschiede:

Persönliches Exemplar für Thomas Bartholomäus

- Das Kommando/Paket `rsync` muss sowohl auf dem lokalen Rechner als auch auf dem externen Rechner installiert sein. Der bloße SSH-Client bzw. -Server reichen nicht aus.
- `rsync` überprüft, welche Dateien schon übertragen wurden, und kopiert nur neue bzw. veränderte Dateien. Bei großen Verzeichnisbäumen spart das (richtig) viel Zeit.
- `rsync` kann Dateien und Verzeichnisse auch löschen und somit Löschvorgänge synchronisieren. Da dies mit Datenverlusten verbunden sein kann, muss dieses Verhalten explizit mit der Option `--delete` aktiviert werden.

**»rsync« und Windows**

`rsync` steht unter Windows nicht zur Verfügung. Der einfachste Weg zur Nutzung von `rsync` führt über das *Windows-Subsystem für Linux* (WSL). Nach der Installation einer Distribution können Sie `rsync` clientseitig unkompliziert innerhalb von WSL nutzen und dabei auf alle Windows-Dateien zugreifen.

Eine gute (aber nicht SSH-kompatible) Variante zu `rsync` ist das Kommando `robocopy`, das ich Ihnen in Kapitel 15, »Backups«, vorstelle.

## 12.4 SSH-Authentifizierung mit Schlüsseln

Der SSH-Server kennt eine Menge unterschiedlicher Authentifizierungsverfahren: Am gängigsten ist der bisher in diesem Kapitel verwendete Passwort-Login. Alternativen sind die Zweifaktorauthentifizierung (2FA) sowie die Authentifizierung mit Schlüsseln, auf die ich hier eingehe. Diese Art der Authentifizierung erfordert Vorbereitungsarbeiten, die rasch erledigt sind. In der Folge kann `ssh` oder `scp` passwortlos genutzt werden – ideal für die Automatisierung in Scripts.

Als »Schlüssel« dient dabei ein Paar von Dateien. Der öffentliche Teil des Schlüssels wird am SSH-Server hinterlegt, der private Teil bleibt auf dem Rechner, auf dem Sie die Kommandos `ssh`, `scp` oder `rsync` nutzen möchten. Beim SSH-Verbindungsaufbau kann der SSH-Server anhand des öffentlichen Schlüssels überprüfen, ob der lokal gespeicherte private Schlüssel dazu passt. Wenn ein derart geeignetes Schlüsselpaar gefunden wird, gilt dies als ausreichende Authentifizierung; die Passwortabfrage entfällt.

### Schlüsselpaar erzeugen

Um auf dem Client-Rechner ein Schlüsselpaar zu erzeugen, führen Sie einmalig `ssh-keygen` aus. Dieses Kommando ist unter Windows, Linux und macOS verfügbar, sofern ein SSH-Client installiert ist. Wenn das Kommando eine Warnung anzeigt, dass


© Rheinwerk Verlag, Bonn 2024

schon ein Schlüssel existiert, brechen Sie ab oder geben einen neuen Namen an, falls Sie ein zweites Schlüsselpaar benötigen.

```
$ ssh-keygen

  Generating public/private rsa key pair.
  Enter file in which to save the key
     (/home/kofler/.ssh/id_rsa): <Return>
  Enter passphrase (empty for no passphrase): <Return>
  Enter same passphrase again: <Return>
  Your identification has been saved in /home/kofler/.ssh/id_rsa
  Your public key has been saved in /home/kofler/.ssh/id_rsa.pub
```

`ssh-keygen` fragt, ob es den Schlüssel mit einer *Passphrase* (mit einem Passwort) absichern soll. Aus Sicherheitsgründen wäre das empfehlenswert. Wenn Ihr privater Schlüssel in falsche Hände gerät, kann der Dieb mit dem Schlüssel ohne Passwort nichts anfangen. An dieser Stelle geht es aber um den Einsatz von Schlüsseln beim Scripting. Und in diesem Kontext empfehle ich Ihnen, *keine* Passphrase anzugeben, also einfach ↵ zu drücken. Nur so können Sie Ihre Scripts mit `ssh`- oder `scp`-Kommandos ohne Benutzereingriff automatisiert ausführen.

Das Schlüsselpaar wird standardmäßig im Verzeichnis `.ssh` gespeichert und besteht aus zwei Dateien, die je nach dem Verschlüsselungsalgorithmus unterschiedliche Namen haben (siehe Tabelle 12.1). Bei vielen SSH-Installationen kommt noch der Algorithmus von *Rivest*, *Shamir* und *Adleman* (kurz RSA) zum Einsatz. Neuere Versionen verwenden Verfahren wie den *Elliptic Curve Digital Signature Algorithm* (ECDSA). Im Kontext dieses Buchs ist das Verfahren irrelevant; Sie sollten nur wissen, dass die Schlüsseldateien je nach SSH-Version unterschiedliche Namen haben.

| | Privater Schlüssel | Öffentlicher Schlüssel |
|---|---|---|
| **RSA** | `id_rsa` | `id_rsa.pub` |
| **ECDSA** | `id_ecdsa` | `id_ecdsa.pub` |
| **Curve25519** | `id_ed25519` | `id_ed25519.pub` |

**Tabelle 12.1** Namen der Schlüsseldateien je nach Algorithmus

### Geben Sie nie Ihren privaten Schlüssel weiter!

SSH-Schlüssel benötigen Sie bei vielen Hosting-Providern auch zum Einrichten neuer Server oder virtueller Maschinen, bei Git-Portalen zur Authentifizierung von Git-Kommandos usw. Achten Sie darauf, dass Sie immer nur den öffentlichen Teil des Schlüssels weitergeben oder hochladen (Kennung `*.pub`), nie den privaten Teil!



Persönliches Exemplar für Thomas Bartholomäus

Die zweiteiligen Schlüssel entsprechen nicht unserer menschlichen Vorstellung. Besser passt die folgende Analogie:

- Der private SSH-Schlüssel entspricht einem herkömmlichen Schlüssel.
- Der öffentliche SSH-Schlüssel entspricht dagegen einem Schloss.

Sie haben also einen unbegrenzten Vorrat von Schlössern (Kopien des öffentlichen Schlüssels), die Sie an allen erdenklichen Orten platzieren können, insbesondere bei fremden Servern. Den dazu passenden Schlüssel – die private Schlüsseldatei – geben Sie dagegen nie aus der Hand.

### Den öffentlichen Teil des Schlüssels auf dem Server hinterlegen (macOS, Linux)

Wenn Sie clientseitig unter Linux oder macOS arbeiten, ist auch der zweite Schritt ein Kinderspiel: Mit `ssh-copy-id` kopieren Sie den öffentlichen Teil Ihres Schlüssels in das Benutzerverzeichnis des gewünschten Accounts auf dem Server. Dabei ersetzen Sie `name` durch den Account-Namen (Login-Namen) und `hostname` durch den Rechnernamen.

```
$ ssh-copy-id name@hostname

  name@hostnames's password: ********
```

In der Folge können Sie ausprobieren, ob alles geklappt hat. Ein SSH-Login sollte jetzt ohne Passwortabfrage möglich sein:

```
$ ssh name@hostname       (Login ohne Passwort!)
```

### Den öffentlichen Teil des Schlüssel auf dem Server hinterlegen (Windows)

Unter Windows stehen Ihnen die Kommandos `ssh`, `scp` und `ssh-keygen` zur Verfügung, aber `ssh-copy-id` fehlt. Dafür gibt es einfachen Grund: `ssh-copy-id` ist kein kompiliertes Programm, sondern ein Bash-Script. Deswegen wäre eine Portierung für Windows mit mehr Aufwand verbunden. Das ist zum Glück kein Beinbruch: Sie müssen den öffentlichen Teil des Schlüssels nun eben manuell auf den Server kopieren und dort der Datei `.ssh/authorized_keys` im Heimatverzeichnis des gewünschten Accounts hinzufügen.

Das folgende Listing fasst die erforderlichen Kommandos zusammen, wobei ich dem Prompt immer eine Zeichenkette vorangestellt hat, wo das Kommando auszuführen ist: auf dem lokalen Windows-Rechner oder im Rahmen einer SSH-Session auf dem externen Host. Ich gehe hier davon aus, dass auf dem Host Linux oder macOS läuft. Sie müssen `name` durch den Account-Namen, `host` durch den Namen des externen


© Rheinwerk Verlag, Bonn 2024

Rechners und id_rsa.pub durch den Namen für den öffentlichen Teil Ihres Schlüssels ersetzen. (Dieser kann z. B. auch id_ecdsa.pub heißen.)

```
Windows> scp .ssh/id_rsa.pub name@host:
Windows> ssh name@host

host$ mkdir .ssh
host$ touch .ssh/authorized_keys
host$ cat id_rsa.pub >> .ssh/authorized_keys
host$ chmod 700 .ssh
host$ chmod 600 .ssh/authorized_keys
```

Die beiden Kommandos mkdir und touch sind nur notwendig, wenn die Datei .ssh/authorized_keys noch nicht existiert. cat fügt den Schlüssel (dabei handelt es sich einfach um einige Textzeilen mit einem hexadezimalen Code) am Ende dieser Datei hinzu. Die beiden chmod-Kommandos stellen sicher, dass die Zugriffsrechte für das Verzeichnis .ssh und die Datei authorized_keys korrekt sind – andernfalls wird die Datei von SSH-Server ignoriert.

Falls auf dem SSH-Server das Sicherheitssystem SELinux läuft, wie dies unter Fedora, Red Hat & Co. üblich ist, müssen Sie außerdem sicherstellen, dass der SELinux-Kontext korrekt ist:

```
host$ restorecon -R -v .ssh
```

Zuletzt vergewissern Sie sich im Terminal Ihres Windows-Rechners, dass der SSH-Verbindungsaufbau mit ssh user@host jetzt ohne Passwort funktioniert.

## 12.5 Beispiel: Bilder-Upload auf einen Linux-Webserver

Das Ziel des ersten Beispiels besteht darin, durch ein Script alle neuen oder geänderten Bilder aus einem lokalen Verzeichnis mit scp in ein entsprechend vorbereitetes Verzeichnis eines Webservers hochzuladen.

### Vorbereitungsmaßnahmen

Ich gehe in diesem Beispiel davon aus, dass der Webserver unter Ubuntu Linux läuft und dass das Bildverzeichnis in eine bestehende WordPress-Installation eingebettet werden soll. Prinzipiell funktioniert das Beispiel natürlich auch ohne WordPress und für jede andere Distribution; Sie müssen aber gegebenenfalls Pfade anpassen und bei Red-Hat-basierten Distributionen sicherstellen, dass der SELinux-Kontext des Bildverzeichnisses korrekt eingestellt ist.

Persönliches Exemplar für Thomas Bartholomäus

Der Bild-Upload soll mehreren Benutzern/Accounts erlaubt werden. Deswegen wird auf dem Server die Gruppe imageupload eingerichtet. Alle Benutzer, die den Upload durchführen dürfen, werden dieser Gruppe hinzugefügt.

Außerdem wird am Server das Verzeichnis myimages erzeugt. chown ordnet es dem Benutzer www-data zu (das ist bei Debian und Ubuntu der System-Account des Webservers), außerdem der Gruppe imageupload. Das chmod-Kommando bewirkt, dass alle Mitglieder der Gruppe imageupload im Verzeichnis arbeiten, lesen und schreiben dürfen und dass neu hochgeladene Dateien automatisch dieser Gruppe zugeordnet werden.

Sämtliche Kommandos im folgenden Listing sind auf dem Linux-Server auszuführen und erfordern dort root-Rechte (daher das Prompt-Zeichen #).

```
$ sudo -s
# addgroup imageupload
# usermod -a -G imageupload username1
# usermod -a -G imageupload username2
# usermod -a -G imageupload username3
# mkdir /var/www/html/wordpress/myimages
# chown www-data:imageupload /var/www/html/wordpress/myimages
# chmod 2775 /var/www/html/wordpress/myimages
```

Schließlich gehe ich davon aus, dass die Benutzer (username1 usw.) ihren SSH-Key hochgeladen haben, sodass scp ohne interaktiven Login funktioniert. Bevor Sie mit der Entwicklung des Scripts beginnen, sollten Sie das interaktiv testen, z. B. so:

```
$ scp tst.jpg username1@hostname:/var/www/html/wordpress/myimages
```

### Bash-Script

Ich gehe hier zuerst davon aus, dass auf dem lokalen Rechner Linux oder macOS läuft. Für diesen Fall initialisiert das Bash-Script zuerst einige Variablen. Wenn es die Datei last-run nicht gibt, wird sie im aktuellen Verzeichnis mit dem Datum Anfang 2000 erzeugt. Diese Datei dient als Referenz für find, das nur Dateien berücksichtigt, die neuer sind. last-run wird am Ende des Scripts mit dem Zeitpunkt aktualisiert, zu dem das Script gestartet wurde (Datei now). Diese etwas umständliche Vorgehensweise stellt sicher, dass das Script beim nächsten Durchlauf keine Dateien übersieht, die genau während der Ausführung von find dazugekommen sind. Das ist zugegebenermaßen ein unwahrscheinlicher Fall, aber er ist nicht ausgeschlossen.

Das find-Kommando verarbeitet die Kennungen *.png, *.jpg und *.jpeg. Wenn Sie weitere Kennungen – gegebenenfalls auch in Großschreibung – berücksichtigen wollen, müssen Sie weitere Optionen in der Form -o -name ... hinzufügen. -o meint hier ein logisches Oder. -maxdepth 1 bewirkt, dass find keine Unterverzeichnisse durch-


© Rheinwerk Verlag, Bonn 2024

sucht. Sie können diese Option natürlich weglassen. Beachten Sie aber, dass die Verzeichnisstruktur beim Upload verloren geht, d. h., sämtliche Dateien landen direkt in `myimages`. Wenn Sie das nicht wollen, sollten Sie `rsync` anstelle von `scp` verwenden.

Die `-exec`-Option ruft für jede gefundene Datei das `scp`-Kommando auf und übergibt anstelle von `{}` den Dateinamen. Das funktioniert auch für Dateinamen mit Leerzeichen. `\;` gibt an, wo das Kommando für `-exec` endet.

```
# Beispieldatei upload-images.sh
LOCALDIR=$(pwd)
REMOTEDIR=/var/www/html/wordpress/myimages
REMOTEHOST=hostname
REMOTEUSER=username1
LASTRUN=$LOCALDIR/last-run
NOW=$LOCALDIR/now
# Datei last-run erzeugen, wenn sie noch nicht existiert;
# dabei ein altes Datum verwenden (2000-01-01).
if [ ! -f $LASTRUN ]; then
  touch -m -t 200001020000 $LASTRUN
fi
# Datei now mit aktuellem Zeitpunkt erzeugen
touch $NOW

# alle Dateien hochladen, die sich nach last-run geändert haben
find $LOCALDIR -maxdepth 1 \( -name "*.jpg" -o -name "*.jpeg" \
   -o -name "*.png" \) -newer $LASTRUN \
   -exec scp {} $REMOTEUSER@$REMOTEHOST:$REMOTEDIR \;

# last-run aktualisieren
mv $NOW $LASTRUN
```

### PowerShell-Script

Wenn sich Ihre Bilder auf einem Windows-Rechner befinden, ist es naheliegend, das Script mit der PowerShell zu realisieren. Der folgenden Code hat einen ganz ähnlichen Aufbau wie das Bash-Script, weswegen ich es bei einigen Anmerkungen bewenden lasse. Bei `New-Item` verhindert `Out-Null` die ansonsten obligatorische Ausgabe des Dateinamens.

```
# Beispieldatei upload-images.ps1
$localdir = (Get-Location).Path
$remotedir = "/var/www/html/wordpress/myimages"
$remotehost = "hostname"
$remoteuser = "username"
$lastrun = "$localdir/last-run"
$now = "$localdir/now"
```

Persönliches Exemplar für Thomas Bartholomäus

```
# Datei last-run erzeugen, wenn sie noch nicht existiert;
# dabei ein altes Datum verwenden (2000-01-01).
if (-not (Test-Path $lastrun)) {
    (New-Item $lastrun).LastWriteTime = Get-Date "2000-01-02"
}
# Datei now mit aktuellem Zeitpunkt erzeugen
New-Item $now -Force | Out-Null

# alle Dateien hochladen, die sich nach last-run geändert haben
$lastruntime = (Get-Item $lastrun).LastWriteTime
Get-ChildItem -Path $localdir/* `
    -Include "*.jpg", "*.jpeg", "*.png" |
Where-Object { $_.LastWriteTime -gt $lastruntime } |
ForEach-Object {
    scp $_.FullName $remoteuser@${remotehost}:$remotedir
  }

# last-run aktualisieren
Move-Item -Force $now $lastrun
```

Ich habe in diesem Script `scp` direkt aufgerufen, weil auf einem typischen Linux-Server selten die PowerShell installiert ist, geschweige denn die erforderliche Konfiguration von `sshd_config` für das SSH Remoting vorliegt. Aber sollten diese Voraussetzungen erfüllt sein, können Sie natürlich auch mit SSH Remoting arbeiten. Sie finden in den Beispieldateien das vollständige Script. Im Folgenden sind nur die wenigen Zeilen abgedruckt, die sich bei einer derartigen Vorgehensweise ändern:

```
# Beispieldatei upload-images-remoting.ps1
...
$session = New-PSSession $remotehost -Username $remoteuser
Get-ChildItem -Path $localdir/* `
    -Include "*.jpg", "*.jpeg", "*.png" |
Where-Object { $_.LastWriteTime -gt $lastruntime } |
ForEach-Object {
    Copy-Item -ToSession $session $_ $remotedir
}
Remove-PSSession $session
```

## 12.6 Beispiel: Auswertung virtueller Maschinen

Im zweiten Beispiel geht es darum, Informationen aus einer Gruppe gleichartiger virtueller Maschinen auszulesen. Im konkreten Fall waren die virtuellen Maschinen die Basis einer Laborübung auf einer Fachhochschule. Jeder Student bzw. jede Studentin hatte Zugang zu einer virtuellen Maschine. Bereits beim Einrichten der virtuellen


© Rheinwerk Verlag, Bonn 2024

Maschinen habe ich mich darum gekümmert, den öffentlichen Teil meines SSH-Schlüssels dort zu hinterlegen. Die virtuellen Maschinen sind unter den Hostnamen `host<nn>.mylab.com` erreichbar.

Um mir einen raschen Überblick zu verschaffen, wie die Organisation der virtuellen Datenträger und der Dateisysteme aussehen, wollte ich auf jeder Instanz das Kommando `lsblk` ausführen. Das gelingt mit einem Bash-Einzeiler (der hier aber aus Platzgründen und zur besseren Lesbarkeit über vier Zeilen abgedruckt ist). Mit der Option `-o StrictHostKeyChecking=no` verzichtet `ssh` beim ersten Verbindungsaufbau auf die Rückfrage, ob dem Host vertraut werden soll.

```
$ for i in {01..25}; do
    echo "\n\nVM: $i";
    ssh user@host$i.mylab.com -o StrictHostKeyChecking=no lsblk;
  done

  VM: 01
  NAME                  MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
  sr0                    11:0    1 1024M  0 rom
  vda                   252:0    0    5G  0 disk
    vda1                252:1    0    1G  0 part /boot
    vda2                252:2    0    4G  0 part
      almalinux-root 253:0    0  3,5G  0 lvm  /
      almalinux-swap 253:1    0  512M  0 lvm  [SWAP]
  vdb                   252:16   0    1G  0 disk
  ...
```

Wenn für jede virtuelle Maschine mehr Informationen ermittelt und diese in Ergebnisdateien gespeichert werden sollen, lohnt sich dann doch die Programmierung eines kleinen Scripts:

```
# Beispieldatei gather-vm-data.sh
VMNAMES=$(echo host{01..25})
VMHOST=mylab.com
USER=username
CMDS='hostnamectl; echo; ip addr; echo; lsblk'

for vm in $VMNAMES; do
    echo $vm
    ssh $USER@$vm.$VMHOST -o StrictHostKeyChecking=no "$CMDS" \
      > result-$vm.txt
done
```

Das Script durchläuft alle in `VMNAMES` enthaltenen virtuellen Maschinen, führt dort die in `CMDS` aufgelisteten Kommandos aus und speichert die Ergebnisse in Dateien mit Namen wie `result-<vmname>.txt`.

Persönliches Exemplar für Thomas Bartholomäus

Eine stark erweitere Variante dieses Scripts verwende ich zur Auswertung der Prüfungen meiner Lehrveranstaltung »Linux-Systemadministration«. Die Studierenden müssen im Rahmen der Prüfung alle möglichen administrativen Aufgaben in einer frisch aufgesetzten virtuellen Maschine durchführen. Die Kontrolle, wie weit das funktioniert hat, lässt sich zwar nicht vollständig automatisieren, aber das Script ist doch eine starke Arbeitserleichterung meiner Korrekturarbeiten.

Vergleichbare Scripts können Sie natürlich auch abseits der Lehre verwenden, z. B. zur zentralen Überwachung oder Administration großer Gruppen von Servern oder virtuellen Maschinen.

### Andere Tools

Die Verwendung »handgestrickter« Scripts ist nur empfehlenswert, solange die durchzuführenden Arbeiten überschaubar sind. Je komplexer die Aufgabenstellung ist, desto eher sollten Sie sich nach Programmen zur Fernadministration oder für das Monitoring von Server-Pools umsehen. Bewährte Konfigurations-Tools sind *Ansible* oder *Puppet*; ein umfassendes Monitoring können Sie mit *Grafana*, *Prometheus* oder *Nagios* bewerkstelligen.

Wenn es Ihnen nur darum geht, Kommandos per SSH auf vielen Rechnern möglichst effizient auszuführen, sollten Sie sich mit *Cluster SSH* oder *Parallel SSH* anfreunden.


© Rheinwerk Verlag, Bonn 2024

# Kapitel 13
# Visual Studio Code

*Visual Studio Code*, kurz *VSCode* und manchmal noch kürzer einfach *Code*, ist aktuell der beliebteste Universal-Editor für Software-Developer. Zu den Stärken des von Microsoft entwickelten Programms zählen die Unterstützung für alle erdenklichen Programmiersprachen und Plattformen, das riesige Angebot an Erweiterungen, umfassende Konfigurationsmöglichkeiten sowie die großartige Integration mit dem Versionsverwaltungssystem Git.

Obwohl VSCode keineswegs der einzige Editor in meinem alltäglichen Arbeitsumfeld ist, möchte ich seine Stärken nicht mehr missen. Aus der Perspektive dieses Buchs punktet VSCode durch eine einheitliche Bedienung, unabhängig davon, ob Sie gerade an einem Bash-, Python- oder PowerShell-Script arbeiten. Das ist ein Vorteil im Vergleich zu dezidierten Entwicklungssystemen (z. B. *PyCharm* für Python), wo Sie sich unterschiedliche Arbeitstechniken und Tastenkürzel merken müssen.

Eine Einführung in VSCode erscheint mir in diesem Buch überflüssig. Vermutlich kennen Sie den Editor ohnedies schon. Sollte das nicht der Fall sein, finden Sie im Internet haufenweise Tutorials und Videos, die beim Einstieg helfen. Dieses kurze Kapitel konzentriert sich vielmehr auf die Scripting-spezifischen Features von VSCode, also insbesondere auf die Bash-, PowerShell- Python- und Remote-SSH-Erweiterungen.

**Voraussetzungen für dieses Kapitel**

Sie können dieses Kapitel losgelöst vom Rest des Buchs lesen. Die Nutzung der Remote-SSH-Erweiterung setzt allerdings voraus, dass Sie mit SSH vertraut sind (siehe Kapitel 12, »SSH«).

## 13.1 Einführung

Die Installation von VSCode unter Windows und macOS ist ein Kinderspiel. Sie laden einfach das Installationsprogramm bzw. die entsprechende ZIP-Datei von *https://code.visualstudio.com/download* herunter. Unter Windows führen Sie das Pro-

Persönliches Exemplar für Thomas Bartholomäus

gramm aus; unter macOS verschieben Sie den Editor aus dem ZIP-Archiv in den Ordner *Programme*.

Für Linux stellt Microsoft Debian- und RPM-Pakete zur Verfügung. Bei der Installation des Debian-Pakets wird automatisch eine Paketquelle eingerichtet, sodass Sie in Zukunft im Rahmen der System-Updates auch Aktualisierungen für VSCode erhalten. Wenn Sie dagegen unter Fedora, openSUSE oder einer Red-Hat-kompatiblen Distribution arbeiten, müssen Sie sich selbst um die Aktualisierungen kümmern. Dazu besuchen Sie alle paar Monate die Download-Seite, laden sich die neueste Version des RPM-Pakets herunter und wiederholen die Installation.

Falls Sie Arch Linux verwenden, können Sie VSCode als AUR-Paket installieren, wobei Sie die Wahl zwischen einer Arch-Linux-Variante oder dem offiziellen Microsoft-Binary haben. Am besten verwenden Sie dazu einen AUR-Helper, z. B. `yay`. Damit gelingt die VSCode-Installation mit `yay -S code-git` bzw. mit `yay -S visual-studio-code-bin`.

### VSCode versus VSCodium versus Visual Studio

Der Code für VSCode untersteht zwar einer Open-Source-Lizenz und befindet sich in einem öffentlichen GitHub-Repository, die von Microsoft zum Download angebotene Binärversion untersteht aber einer anderen Lizenz und enthält Telemetrie-Funktionen. Wenn Sie das stört, suchen Sie in den Einstellungen nach *Telemetry* und deaktivieren die entsprechenden Optionen.

Wenn Sie mit der Lizenz des VSCode-Downloads von Microsoft unglücklich oder Sie generell misstrauisch sind, können Sie anstelle des Originals die Variante *VSCodium* installieren (siehe auch *https://vscodium.com*). VSCodium verwendet den gleichen GitHub-Code wie VSCode, die Telemetry-Funktionen sind aber deaktiviert, das Programm hat ein anderes Icon und für die Nutzung gilt die sehr liberale MIT-Lizenz.

VSCode bzw. VSCodium haben nichts (oder zumindest nur sehr wenig) mit der Entwicklungsumgebung *Visual Studio* zu tun. Dieses kommerzielle Microsoft-Produkt ist speziell für die Entwicklung von Programmen in C#, C++ und Visual Basic gedacht.

### Denken Sie in Verzeichnissen, nicht in Dateien!

In herkömmlichen Editoren wie Notepad++ öffnen Sie einfach Dateien aus beliebigen Orten und bearbeiten diese. Das geht in VSCode auch, aber das ist nicht die optimale Arbeitsweise. In VSCode öffnen Sie mit FILE • OPEN FOLDER ein Verzeichnis. Die darin enthaltenen Dateien werden nun in der Seitenleiste angezeigt und können per Doppelklick geöffnet und bearbeitet werden.

Warum ist das Verzeichnis-Paradigma von VSCode so wichtig? VSCode stellt für die meisten Programmiersprachen Erweiterungen zur Verfügung. Damit »versteht«


© Rheinwerk Verlag, Bonn 2024

VSCode Ihren Code besser und unterstützt Sie bei der Eingabe. Gleichzeitig können Scripts mit einem RUN-Button direkt aus dem Editor ausgeführt werden. Als aktives Verzeichnis gilt dann das zuletzt in VSCode geöffnete Verzeichnis, nicht das Verzeichnis, in dem sich die Script-Datei befindet!

Nehmen Sie an, Sie haben ein Script entwickelt, das eine im gleichen Verzeichnis befindliche CSV-Datei öffnet und verarbeitet. Beim Ausführen des Scripts tritt aber ein Fehler auf, weil das Script die Datei nicht findet! Wenn das passiert, sind Sie in die VSCode-Verzeichnisfalle getappt. Abhilfe ist einfach: Entweder führen Sie das Script eben manuell aus, am einfachsten im Terminalbereich von VSCode. Oder, noch besser, Sie wechseln mit FILE • OPEN FOLDER in das Verzeichnis, in dem sich Ihr Script befindet.

Die VSCode-Vorliebe für Verzeichnisse zwingt Sie unter Umständen, Arbeitsweisen zu hinterfragen. Die Bearbeitung von Scripts, die über unzählige Verzeichnisse verstreut sind, ist in VSCode mühselig. Andererseits ist es wahrscheinlich sowieso eine gute Idee, inhaltlich zusammengehörende Scripts in einem Verzeichnis abzulegen.

## 13.2 Sprachspezifische VSCode-Erweiterungen

Sobald Sie in VSCode zum ersten Mal eine PowerShell- oder Python-Datei bearbeiten, fragt der Editor, ob er die entsprechende Erweiterung installieren soll. Dieser Empfehlung sollten Sie folgen! VSCode führt nun Syntax-Highlighting durch, vervollständigt Schlüsselwörter, rückt Ihren Code korrekt ein und hilft beim Testen und der Fehlersuche in Ihren Scripts.

### PowerShell-Erweiterung

Zu den interessantesten Funktionen der PowerShell-Erweiterung zählen die folgenden Punkte:

- Run-Button: Die Symbolleiste von VSCode enthält zwei RUN-Buttons. Der eine führt Ihr Script aus, der zweite nur die zuvor markierten Code-Zeilen.
- Navigation: Mit `Strg` und einem Klick navigieren Sie direkt zur Definition der Variable oder Funktion unter dem Mauszeiger.
- PSScriptAnalyzer: Dieses Tool analysiert Ihren Code im Hinblick auf offensichtliche Schwächen bzw. Verstöße gegen *Best Practices*. Das Tool kümmert sich auch um die Code-Einrückung. Nach größeren Umbauten können Sie die korrekten Einrückungen mit `Strg`+`⇧`+`I` wiederherstellen.
- Debugging-Funktionen: Diese Funktion hilft Ihnen bei der Fehlersuche in PowerShell-Scripts. Eine Anleitung finden Sie hier:

  *https://devblogs.microsoft.com/scripting/debugging-powershell-script-in-visual-studio-code-part-1*

Persönliches Exemplar für Thomas Bartholomäus

## Python-Erweiterung

Mit der aktiven Python-Erweiterung (siehe Abbildung 13.1) macht die Eingabe von Python-Code richtig Spaß. VSCode führt automatisch eine Syntaxkontrolle durch, hilft bei der Code-Einrückung, vervollständigt Namen von installierten Modul sowie Funktionen aus importierten Modulen. Sie können Kommentare und Funktionen zusammenklappen und behalten so den Überblick, wenn ein Script doch einmal etwas länger ausfällt.

Sogenannte Refactoring-Funktionen helfen dabei, Code nachträglich zu restrukturieren. Wenn Sie beispielsweise mit F2 eine Variable oder Funktion umbenennen, führt VSCode diese Änderung automatisch an allen Stellen im Code durch.

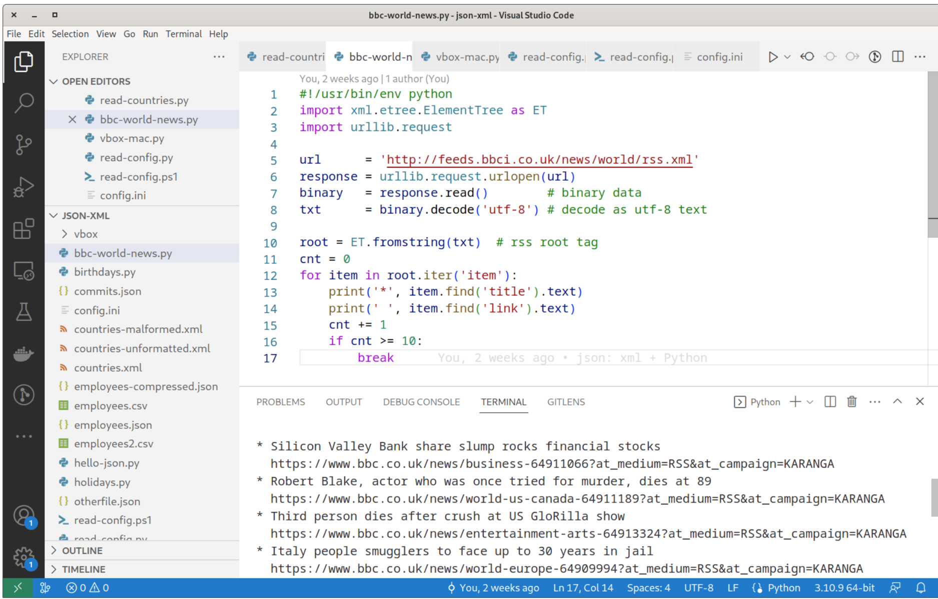



**Abbildung 13.1** Visual Studio Code mit der Python-Erweiterung

VSCode zeigt rechts unten in der Statusleiste die Nummer der am Rechner installierten Python-Version an. Sollte es mehrere Installationen geben, können Sie durch einen Klick auf die Versionsnummer die für die Code-Ausführung gewünschte Version auswählen. (Ich rate Ihnen aber dringend von Parallelinstallationen ab, weil sie oft Ärger mit der Modulverwaltung und `pip` verursachen!)

Unter macOS und Windows ist Ihnen VSCode auch bei der Installation von Python behilflich, insbesondere dann, wenn es keinen Python-Interpreter findet. Das ist sicher gut gemeint, ich halte es aber für eine bessere Idee, die Python-Installation selbst durchzuführen. Damit behalten Sie die Kontrolle über Ihre Python-Installationen und vermeiden Mehrfachinstallationen.


© Rheinwerk Verlag, Bonn 2024

### Bash/Shell-Erweiterungen

Im Gegensatz zu PowerShell und Python gibt es nicht *die* VSCode-Erweiterung für Bash-Scripts. VSCode ergreift auch nicht die Initiative und bietet die Installation einer Erweiterung an, wenn Sie ein Bash- oder Zsh-Script verfassen. Dessen ungeachtet gibt es aber durchaus Erweiterungen, die Ihnen das Leben als Bash-Entwicklerin bzw. -Entwickler leichter machen. Es bleibt Ihnen überlassen, welche davon Sie installieren. (Wenn Sie in der Seitenleiste EXTENSIONS nach *bash* oder *shell* suchen, finden Sie noch mehr Erweiterungen, die aber zum Teil von fragwürdigem Nutzen sind.)

- **Bash Debug** hilft bei der Fehlersuche in Bash-Scripts.
- **shellman** richtet sich speziell an Bash-Einsteiger und hilft bei der Eingabe von Bash-Strukturen wie `if` oder `for`.
- **ShellCheck** analysiert Ihren Code und schlägt syntaktische Verbesserungen und Korrekturen vor. Persönlich finde ich das Tool allerdings zu pingelig und zu wenig konfigurierbar.
- **Code Runner** baut in die Symbolleiste von VSCode einen Button zur direkten Ausführung von Shell-Scripts ein.

## 13.3 Remote-SSH-Erweiterung

Der größte Nachteil von VSCode ist seine grafische Benutzeroberfläche. Diese Aussage mag Ihnen absurd erscheinen: Sie verwenden VSCode ja gerade wegen der intuitiven Bedienung, seiner vielen, elegant in die Benutzeroberfläche integrierten Funktionen. Und solange Sie lokal arbeiten, solange sich Ihre Scripts also auf Ihrem Notebook befinden, kommen die Vorzüge von VSCode tatsächlich voll zur Geltung.

Im Scripting-Alltag verfassen Sie aber oft Scripts, die auf anderen Rechnern oder in virtuellen Maschinen laufen. Häufig besteht zu diesen Rechnern bzw. Servern nur eine SSH-Verbindung. Sie müssen also im Textmodus arbeiten. Linux-Fans setzen dann zumeist auf Editoren wie *Vi* oder *Emacs*, die ihre volle Funktionalität auch im Textmodus entfalten und daher auch in einer SSH-Session anwendbar bleiben. (Persönlich verwende ich in solchen Fällen zumeist `jmacs` oder `zile`. Das sind minimalistische Editoren, die in ihren Grundkommandos Emacs-kompatibel sind.)

Ich mache Ihnen hier aber keinen Vorwurf, wenn Sie keine Lust haben, die unzähligen Tastenkürzel dieser Urgesteine aus der Unix-Vergangenheit zu erlernen. Stattdessen stelle ich Ihnen eine elegante Alternative vor: Sie können VSCode auch über eine SSH-Verbindung verwenden, um Code auf einem anderen Rechner zu bearbeiten! Diese Arbeitsweise ist zwar mit gewissen Einschränkungen verbunden, funktioniert aber prinzipiell ausgezeichnet.

Persönliches Exemplar für Thomas Bartholomäus

### Anwendung der Remote-SSH-Erweiterung

Um VSCode mit SSH zu kombinieren, müssen Sie die von Microsoft entwickelte Erweiterung *Remote SSH* installieren. Außerdem sollten Sie Ihren öffentlichen SSH-Schlüssel zum Remote-Rechner kopieren (`ssh-copy-id`, siehe Abschnitt 12.4, »SSH-Authentifizierung mit Schlüsseln«). Das erspart die wiederholte Eingabe des Passworts für Ihren Account am Remote-Rechner.

Um eine neue Verbindung zu einem externen Rechner einzurichten, führen Sie F1 *Remote-SSH: Add new ssh host* aus und geben dann – wie in einem Terminal – das Kommando `ssh user@hostname` ein. VSCode speichert die Daten in `.ssh/config`, stellt aber vorerst noch keine Verbindung her.

Der Verbindungsaufbau erfolgt mit F1 *Remote-SSH: Connect to host* oder über den grünen Remote-Bereich in der Statusleiste. VSCode greift beim Verbindungsaufbau auf das Kommando `ssh` zurück und wertet entsprechend auch die im Verzeichnis `.ssh` gespeicherten Schlüssel aus.

Nach dem erfolgreichen Verbindungsaufbau öffnet VSCode ein neues Fenster, wobei der grüne Bereich in der Statusleiste klarmacht, dass nicht lokale Daten bearbeitet werden (siehe Abbildung 13.2). Von nun an funktioniert VSCode wie üblich: Sie wählen ein Verzeichnis aus, bearbeiten die darin enthaltenen Dateien, speichern Ihre Änderungen usw. Ausgesprochen praktisch ist im Remote-Betrieb auch der Terminal-Bereich von VSCode. Auch das Terminal läuft *remote*, Sie führen darin also wie in einer SSH-Session Kommandos auf dem Remote-Rechner aus.

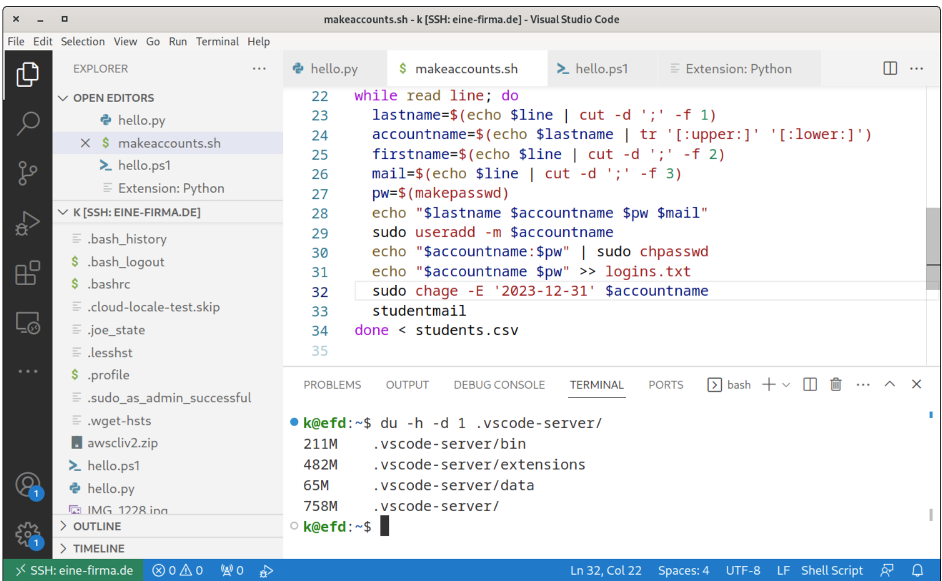



**Abbildung 13.2** VSCode bearbeitet ein Bash-Script auf dem externen Linux-Server »eine-firma.de«


© Rheinwerk Verlag, Bonn 2024

### Einschränkungen

Bevor Sie sich jetzt denken, das ist zu schön, um wahr zu sein, muss ich auf zwei Einschränkungen hinweisen:

- VSCode installiert auf dem Remote-Rechner in das Verzeichnis `.vscode-server` eine Menge Code (typischerweise mehrere Hundert MByte!) und führt diesen Code dort aus. Für die Syntaxkontrolle und das Debugging kommuniziert VSCode auf Ihrem Notebook mit dem Erweiterungscode auf dem Remote-Rechner.

  Damit das flüssig funktioniert, braucht der Remote-Rechner ausreichend Speicherplatz (das betrifft den Datenträger *und* den Arbeitsspeicher), genug Rechenleistung und eine ausgezeichnete Netzwerkanbindung. Andernfalls macht die Arbeit in VSCode wenig Spaß.

- An seine Grenzen stößt VSCode auch, wenn Sie Systemdateien bearbeiten möchten. Es gibt keine mit `sudo` vergleichbare Funktion. Somit erlaubt VSCode nur den Zugriff auf die Dateien, die auch ein SSH-Benutzer lesen darf. Für das Scripting reicht das aus, aber für die Veränderung einer Konfigurationsdatei im Verzeichnis `/etc` brauchen Sie weiterhin einen Editor, den Sie mit `sudo` im Textmodus ausführen können.

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

# Kapitel 14
# Git

Git ist ein Versionsverwaltungssystem. Es wird üblicherweise dann eingesetzt, wenn viele Personen an einem komplexen Projekt arbeiten. Dank Git können dann mehrere Zweige des Programms verwaltet, Änderungen zurückverfolgt und bei Bedarf rückgängig gemacht werden.

Auf den ersten Blick passen Scripting und Git nicht zusammen: Scripts sind kleine Dateien, die oft von nur einer Entwicklerin oder einem Entwickler erstellt werden. Eine Versionsverwaltung mit Git erscheint da überflüssig.

Dieser Eindruck täuscht. Scripting und Git lassen sich wunderbar miteinander kombinieren:

- Git kann auch bei kleinen Script-Sammlungen den Entwicklungsprozess dokumentieren, dient als Backup und hilft dabei, die Scripts unkompliziert über mehrere Zielrechner zu verteilen.
- Manche Entwicklungsvorgänge lassen sich durch den Aufruf der Kommandos `git` und `gh` in Scripts automatisieren.
- Git kann bei bestimmten Vorgängen sogenannte *Hooks* ausführen. Das sind von Ihnen erstellte Scripts, die z. B. vor einem Commit grundlegende Tests vornehmen oder die sich nach einem Commit um das Deployment kümmern.

### Voraussetzungen für dieses Kapitel

Damit Sie optimal von diesem Kapitel profitieren, sollten Sie sich bereits mit SSH-Schlüsseln auseinandergesetzt haben (siehe Kapitel 12, »SSH«). Derartige Schlüssel werden in diesem Kapitel zur Authentifizierung eingesetzt (z. B. bei GitHub oder GitLab). Was die Scripting-Beispiele betrifft, ist dieses Kapitel ziemlich Bash- und Linux-lastig.

Dieses Kapitel beginnt mit einem Git-Crashkurs, der zeigt, wie Sie ein Verzeichnis mit einigen Scripts mit Git versionieren und sichern. Betrachten Sie den Abschnitt aber wirklich nur als Einstieg! (Bernd Öggl und ich haben zu Git ein ganzes Buch verfasst, das ebenfalls im Rheinwerk Verlag erschienen ist. Es behandelt fortgeschrittene Themen, Sonderfälle, Varianten und Best Practices.)

Persönliches Exemplar für Thomas Bartholomäus

Außerdem gehe ich auf die Frage ein, wie Sie sensitive Informationen in Ihren Scripts aus dem Git-Repository fernhalten. Allzu oft ist es schon passiert, dass in Scripts enthaltene Passwörter zu Cloud- oder FTP-Zugängen zu massiven Sicherheitslücken geführt haben. Das gilt es unbedingt zu vermeiden.

## 14.1 Git-Crashkurs

Dieser Crashkurs soll Sie bei Ihren ersten Schritten mit Git unterstützen. Wenn Sie Git schon kennen, blättern Sie einfach weiter zum nächsten Abschnitt!

Ich beschränke mich hier auf die absoluten Basics: Sie lernen weder Zweige noch Merge-Konflikte kennen, geschweige denn Best Practices für den Team-Einsatz. Als einzige Authentifizierungsmethode behandle ich SSH. Kurzum, dieser Crashkurs ist »Git für Minimalisten«. Dessen ungeachtet reicht das vermittelte Wissen aus, um eine Sammlung von Scripts mit Git zu verwalten. Wenn Sie mehr über Git lernen möchten, finden Sie unter *https://git-scm.com/doc* eine umfassende Online-Dokumentation samt Videos. Und habe ich schon erwähnt, dass es auch gute Git-Bücher gibt?

### Vorbereitungsarbeiten

Zuerst müssen Sie Git installieren. Unter Linux verwenden Sie am besten das Git-Paket Ihrer Distribution. Falls Sie Sie unter Debian oder Ubuntu arbeiten, führt `sudo apt install git` zum Ziel. Für macOS und Windows finden Sie unter *https://git-scm.com/downloads* Installationspakete, die unkompliziert anzuwenden sind. Überzeugen Sie sich im Terminal, dass sich das Kommando `git` ausführen lässt. Sollte das nicht klappen, kontrollieren Sie, ob das Git-Installationsverzeichnis in der `PATH`-Variablen enthalten ist!

```
> git --version
  git version 2.39.2
```

**Windows, Linux oder macOS**

Ich verwende in diesem Abschnitt den PowerShell-Prompt >. Die wenigen Screenshots habe ich unter Windows erstellt. An der Vorgehensweise ändert sich aber nichts, egal, ob Sie unter Windows, Linux oder macOS arbeiten bzw. ob Sie `git` in der PowerShell, der Bash oder der Zsh ausführen.

Grundsätzlich können Sie Git *stand-alone* anwenden, den Code Ihres Projekts also ausschließlich lokal auf Ihrem Rechner verwalten. Ich empfehle Ihnen aber dringend, einen kostenlosen Account bei einem Git-Hoster Ihrer Wahl einzurichten. Ich beziehe


© Rheinwerk Verlag, Bonn 2024

mich in den folgenden Beispielen auf GitHub, aber Alternativen wie GitLab funktionieren ebenso gut.

Damit das lokale `git`-Kommando sich beim Git-Host authentifizieren kann, hinterlegen Sie dort Ihren öffentlichen SSH-Schlüssel (also normalerweise die Datei `.ssh/id_xxx.pub`). Bei GitHub finden Sie den entsprechenden Upload-Dialog unter SETTINGS • SSH AND GPG KEYS. Am einfachsten laden Sie die Key-Datei in einen Editor, kopieren den gesamten Text und fügen ihn im Webdialog ein.

Weil diese Information in jedem Commit gespeichert wird, muss Git wissen, wie Sie heißen und welche E-Mail-Adresse Sie haben. (Was ein Commit ist, erkläre ich Ihnen später.) Sie müssen nicht Ihre echte E-Mail-Adresse angeben. Wenn Sie GitHub verwenden und Ihre E-Mail-Adresse anonym bleiben soll, gehen Sie wie folgt vor und ersetzen `aname` durch Ihren GitHub-Accountnamen:

```
> git config --global user.name  "Michael Kofler"
> git config --global user.email "aname@users.noreply.github.com"
```

### Das erste Repository

In der GitHub-Weboberfläche erstellen Sie nun ein neues Repository. Ein *Repository* ist die Sammlung aller Dateien eines Projekts, das Sie mit Git verwalten – und zwar inklusive aller Änderungen, die im Verlauf des Projekts zusammenkommen. Sie müssen dem Repository einen Namen geben. Außerdem sollten Sie die Sichtbarkeit auf PRIVATE stellen. (Ihre ersten Tests sind nicht für die ganze Welt gedacht.) Damit das Repository nicht komplett leer ist, empfiehlt es sich, die Option ADD A README FILE zu aktivieren. Alle anderen Optionen lassen Sie so, wie sie sind.

Anschließend wechseln Sie zurück in Ihr Terminal. Auf meinen Rechnern gibt es zumeist ein eigenes Verzeichnis für Repositories. (`git` ist naheliegend.) Dort führen Sie nun `git clone` aus und wechseln dann in das neue Verzeichnis mit dem Repository:

```
> mkdir git
> cd git
> git clone git@github.com:MichaelKofler/test1234.git
> cd test1234
```

`git clone` lädt das Repository von GitHub herunter und richtet im Verzeichnis `test1234` eine Kopie auf Ihrem Rechner ein. `MichaelKofler/test1234` müssen Sie natürlich durch Ihren eigenen GitHub-Accountnamen und den Namen Ihres Repositorys ersetzen! Die gesamte Repository-Adresse finden Sie in der GitHub-Weboberfläche, wenn Sie auf den grünen Button CODE klicken und dort die SSH-Variante auswählen (siehe Abbildung 14.1).

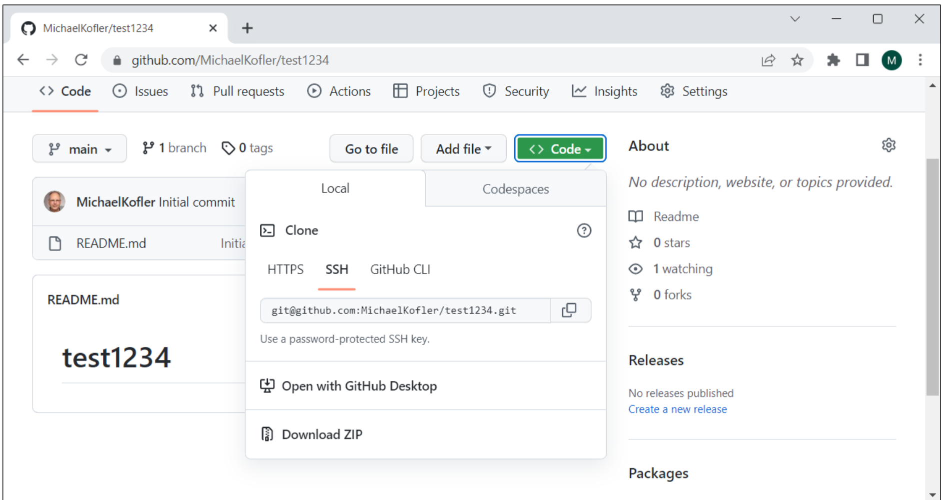



**Abbildung 14.1** Das Test-Repository in der GitHub-Weboberfläche

### Interna

Das Repository-Verzeichnis – hier `git\test1234\` – spiegelt den aktuellen Zustand Ihres Projekts wider. Alte Versionen Ihres Codes sowie weitere interne Daten (eine Art »Git-Datenbank« Ihres Projekts) sind im Unterverzeichnis `.git` versteckt. Dabei handelt es sich um komprimierte Dateien, deren Namen aus hexadezimalen Zeichen zusammengesetzt sind. Verwenden Sie ausschließlich das Kommando `git`, um Ihr Repository zu bearbeiten oder auszulesen. Rühren Sie das `.git`-Unterverzeichnis nicht an!

## Der erste Commit

Die weitere Arbeit findet nun im Verzeichnis des Repositorys statt, in der Logik dieses Beispiels also in `git\test1234\`. Wenn Sie mit VSCode arbeiten, öffnen Sie dieses Verzeichnis. Nehmen wir an, Sie erstellen dort drei neue Scripts `script1.ps1` bis `script3.ps1`.

Am Ende des Tages – oder wenn Sie einen zufriedenstellenden Zwischenstand erreicht haben – wollen Sie diese drei Scripts in das Git-Repository aufnehmen. Um es etwas genauer zu formulieren: Sie wollen eine Kopie der Dateien mit dem gerade aktuellen Zustand in der Git-Datenbank speichern und diese dann mit dem externen Repository auf GitHub synchronisieren. Dazu sind drei Schritte bzw. drei Kommandos notwendig, die ich Ihnen im Anschluss erläutere. Alle Kommandos müssen im Repository-Verzeichnis ausgeführt werden (`git\test1234`, wenn Sie diesem Beispiel folgen):


© Rheinwerk Verlag, Bonn 2024

```
> git add script1.ps1 script2.ps1 script3.ps1

> git commit -m 'script1 to script3: basic functions work'

  [main cd35ac9] script1 to script3: basic functions work
  3 files changed, 3 insertions(+)
  create mode 100644 script1.ps1
  create mode 100644 script2.ps1
  create mode 100644 script3.ps1

> git push

  Writing objects: 100% (5/5), 399 bytes | 199.00 KiB/s, done.
  To github.com:MichaelKofler/test1234.git
     015ed58..cd35ac9  main -> main
```

Beginnen wir mit `git add`: Das Kommando bewirkt zweierlei. Zum einen weiß Git jetzt, dass diese Dateien zum Projekt gehören und unter der Versionsverwaltung stehen. Zum anderen speichert Git den aktuellen Zustand dieser Dateien in der lokalen Git-Datenbank (im Verzeichnis `.git`).

`git commit` erzeugt einen Commit. Damit wird der aktuelle Status des Projekts – alle bis zum Zeitpunkt von `git add` vorgemerkten Änderungen – in einer Art Zwischen-Release gespeichert. Die mit `-m` übergebene *Message* enthält üblicherweise eine Kurzbeschreibung der zuletzt durchgeführten Änderungen. Auch `git commit` wird nur lokal verarbeitet, d. h., das Remote Repository (in unserem Beispiel GitHub) weiß bis jetzt nichts von den lokalen Änderungen!

Erst `git push` überträgt die durchgeführten Änderungen in das Remote Repository, in unserem Fall also auf GitHub. Werfen Sie nach der Durchführung des Kommandos im Terminal noch einmal einen Blick in die GitHub-Weboberfläche. Sie werden sehen, dass dort nun sowohl der Commit als auch die drei Dateien `script1.ps1` bis `script3.ps1` sichtbar sind.

### Git in VSCode

Anstatt die Kommandos `git add`, `git commit` und `git push` in einem Terminal auszuführen, können Sie diese Aktionen auch über die Benutzeroberfläche von VSCode auslösen. VSCode zeichnet sich durch eine exzellente Unterstützung für Git aus. Bei Ihren ersten Versuchen sollten Sie davon aber absehen. Versuchen Sie zuerst, das `git`-Kommando kennen und verstehen zu lernen!

Wenn Sie sich später mit Git in VSCode auseinandersetzen wollen, finden Sie hier einen guten Überblick samt einem Video-Tutorial:

*https://code.visualstudio.com/docs/sourcecontrol/overview*

Persönliches Exemplar für Thomas Bartholomäus

### Weitere Commits

Sie haben sich eine Pause verdient! Am nächsten Tag setzen Sie die Entwicklung fort. Sie beginnen mit der Dokumentation in der Datei `README.md` und führen noch ein paar Verbesserungen in `script2.ps1` durch. Es wird Zeit für den nächsten Commit. Aber anstatt die veränderten Dateien wieder mit `git add` zu markieren, wenden Sie diesmal eine Abkürzung an: Sie führen `git commit` mit der Option `-a` (*all*) aus. Damit werden sämtliche Änderungen von Dateien, die bisher schon unter Git-Kontrolle standen, automatisch in den Commit aufgenommen. Vergessen Sie `git push` nicht, um die Änderungen auch in das Remote Repository hochzuladen!

```
> git commit -a -m 'improved script2.ps1, started documentation'
> git push
```

**Für neue Dateien ist weiter »git add« erforderlich!**

Das Kommando `git commit -a` ist so bequem, dass die Verwendung der Option `-a` rasch zum Automatismus wird. Beachten Sie aber eine wichtige Einschränkung: Das Kommando berücksichtigt nur Dateien, die ohnedies schon unter Git-Kontrolle stehen. Wenn Sie in Ihrem Editor neue Dateien wie `script4.ps1` oder `configuration.ini` erzeugen, dürfen Sie nicht vergessen, diese einmalig mit `git add` dem Repository hinzuzufügen!

### Das Repository auf einem zweiten Rechner einrichten

Ihre Scripts funktionieren soweit zufriedenstellend. Nun sollen diese auf einem zweiten Rechner ausgeführt werden. Dazu installieren Sie dort Git und wiederholen dann in dem Verzeichnis, wo Sie die Scripts nutzen möchten, das bereits bekannte Kommando `git clone`:

```
rechner2> cd some\directory
rechner2> git clone git@github.com:MichaelKofler/test1234.git
rechner2> cd test1234
```

Bei den Tests auf Rechner 2 fallen Ihnen noch zwei Probleme auf. Sie beheben die Fehler und führen nun auch auf Rechner 2 einen Commit durch:

```
rechner2> git commit -a -m 'bugfixes in script1.ps1'
rechner2> git push
```

Die Bugfixes sind damit auf Rechner 2 und im Remote Repository (hier GitHub), aber noch nicht auf Rechner 1. Um dessen Repository mit GitHub zu synchronisieren, führen Sie dort `git pull` aus. Damit übertragen Sie neue Commits, die sich auf dem Remote Repository befinden, auf den Rechner 1:

```
rechner1> git pull
```


© Rheinwerk Verlag, Bonn 2024

**»git pull« vor »git push«**

Sobald Code an mehreren Stellen oder durch mehrere Entwicklerinnen bzw. Entwickler verändert wird, sollten Sie sich angewöhnen, regelmäßig `git pull` auszuführen, um Ihr lokales Repository auf den aktuellen Stand zu bringen. Generell sollten Sie vor `git push` immer `git pull` ausführen. Wenn Sie das vergessen und Git erkennt, dass im lokalen Repository Commits fehlen, führt `git push` zur Fehlermeldung: *failed to push some refs to remote repository xxx*. Abhilfe: Führen Sie zuerst `git pull` aus.

Abbildung 14.2 fasst – zugegebenermaßen stark vereinfacht – die Wirkung grundlegender Git-Kommandos zusammen. Der »Staging-Bereich« ist ein spezieller Bereich der Git-Datenbank, in der mit `git add` hinzugefügte Dateien gespeichert werden, die noch nicht Teil eines Commits sind.

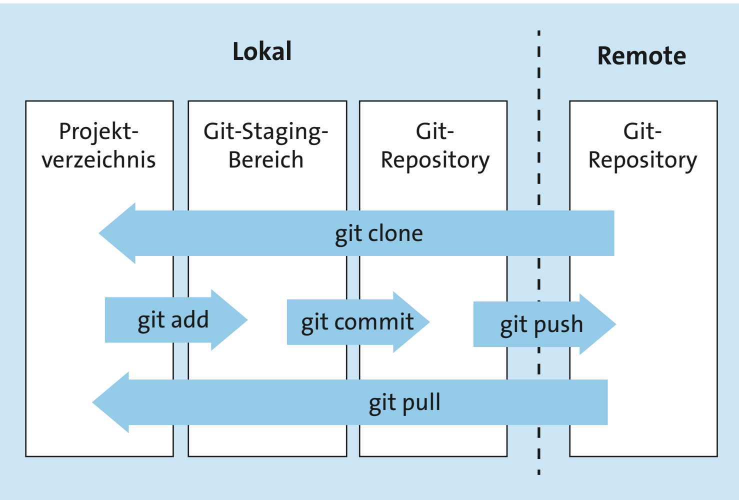



**Abbildung 14.2** Elementare Git-Kommandos

## Git-Status

Je größer Ihr Projekt wird, desto größer wird auch die Gefahr, dass Sie den Überblick in Ihrem Git-Repository verlieren. Abhilfe schafft da `git status`. Das Kommando fasst zusammen, welche Dateien seit dem letzten Commit verändert wurden, welche neu dazugekommen sind usw. (siehe Abbildung 14.3).

Eine weitere Hilfe ist `git log`, das eine Zusammenfassung der letzten Commits ausgibt (der neueste Commit zuerst). Bei `git log` steuern unzählige Optionen, wie viele Details zu jedem Commit angezeigt werden sollen. Die absolute Kurzfassung erhalten Sie mit `--oneline`: Dabei werden nur ein hexadezimaler Identifikationscode (die ersten Stellen des Hash-Codes) und die Commit Message angezeigt:

Persönliches Exemplar für Thomas Bartholomäus

```
> git log

  f2665d9 (HEAD -> main, origin/main, origin/HEAD) more bugfixes
  5d3225d bugfixes in script1.ps1
  679ca1a improved script2.ps1, started documentation
  cd35ac9 script1 to script3.ps1: basic functions work
  015ed58 Initial commit
```

Anstatt sich mit `git log` anzufreunden, können Sie auch die GitHub-Weboberfläche nutzen. Dort können Sie sich durch die Commits klicken, die mit den Commits durchgeführten Änderungen ansehen usw.

```
PowerShell

PS C:\Users\kofler\git\test1234> git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   script1.ps1
        modified:   script3.ps1

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        debugging.log
        script1.ps1.bak
        script4.ps1

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Users\kofler\git\test1234>
```

**Abbildung 14.3** Der Zustand des lokalen Repositorys

## Dateien von Git ausschließen (».gitignore«)

Oft gibt es im Projektverzeichnis Dateien, die Sie nicht in das Repository aufnehmen möchten: vom Editor erstellte Backups Ihrer Scripts, Dateien zum Logging und zur Fehlersuche etc. Damit `git status` diese Dateien nicht auflistet, können Sie die Datei `.gitignore` im lokalen Verzeichnis einrichten. Sie enthält Muster von Dateien, die `git` nicht berücksichtigen soll. Diese Datei kann z. B. so aussehen:

```
*.bak
*.log
*~
tmp/*
```


© Rheinwerk Verlag, Bonn 2024

Diese Datei soll natürlich selbst im Repository gespeichert werden:

```
> git add .gitignore
```

Wenn Sie ausnahmsweise eine Datei in das Repository aufnehmen möchten, für die `.gitignore` ein Sperrmuster enthält, führen Sie `git add` mit der Option `-f` (*force*) aus:

```
> git add debugging.log

  The following paths are ignored by one of your .gitignore
  files: debugging.log.
  Hint: Use -f if you really want to add them.

> git add -f debugging.log
```

### Vorhandenen Code in ein neues Repository übertragen

In diesem Abschnitt bin ich davon ausgegangen, dass Sie zuerst ein Repository auf GitHub einrichten, das leere Repository auf Ihren Rechner klonen und dann dort mit der Entwicklung Ihrer Scripts beginnen. Oft ist es gerade umgekehrt: Sie arbeiten schon ein paar Tage an einem neuen Projekt und entscheiden sich erst dann für den Git-Einsatz. Das ist auch kein Problem!

Zuerst erzeugen Sie das neue Repository in GitHub. Achten Sie darauf, dass Sie die Option ADD A README FILE nicht aktivieren! Fügen Sie dem Repository auch sonst keine Dateien hinzu.

Nun öffnen Sie ein Terminal, wechseln in Ihr Projektverzeichnis, erzeugen mit `git init` ein neues, lokales Repository und fügen diesem die gewünschten Dateien hinzu. `git commit` erstellt den ersten lokalen Commit:

```
> cd my\project
> git init
> git add script*.sh readme.md
> git commit -m 'initial commit'
```

Jetzt muss das lokale Repository mit dem neuen Remote Repository verbunden werden. `git branch` stellt sicher, dass der Hauptzweig im lokalen Repository `main` heißt. Bei aktuellen Git-Versionen ist das standardmäßig der Fall – das Kommando ist dann überflüssig, stört aber nicht. Nur wenn Sie eine recht alte Git-Installation haben, kann es sein, dass diese noch die veraltete Bezeichnung `master` verwendet.

`git remote add origin` legt das Remote Repository fest. Anstelle von `MichaelKofler/test1235.git` müssen Sie wieder Ihren GitHub-Account-Namen und Ihren Repository-Namen angeben!

Persönliches Exemplar für Thomas Bartholomäus

`git push` führt den ersten Upload durch. Die Option `-u` (*set upstream*) stellt sicher, dass die Zuordnung zwischen dem externen Repository für den Zweig `main` dauerhaft gespeichert wird. In Zukunft reicht dann also wieder `git push` ohne weitere Optionen.

```
> git branch -M main
> git remote add origin git@github.com:MichaelKofler/test1235.git
> git push -u origin main

  To github.com:MichaelKofler/test1235.git
   * [new branch]      main -> main
  branch 'main' set up to track 'origin/main'.
```

Damit ist der erste Commit sowohl lokal als auch remote gespeichert. Ab sofort funktionieren die Kommandos `git add`, `git commit`, `git pull` und `git push`, `git status` und `git log` wie vorhin beschrieben.

## 14.2 Der richtige Umgang mit Einstellungen und Passwörtern

Hacker sind immer auf der Suche nach Passwörtern zu Mail-/FTP-/Datenbank-Servern. Fündig werden sie allzu oft auf GitHub, GitLab & Co. Und zwar nicht, weil diese Plattformen selbst so unsicher sind, sondern weil dort gespeicherte und öffentlich zugängliche Code-Dateien die Passwörter ganz einfach im Klartext enthalten. Das kann passieren, wenn Ihr Code so ähnlich wie das folgende Backup-Script aussieht:

```
# Achtung, Negativ-Beispiel!
... code
mysqldump -u root -ptopSecret wordpress > db.sql
lftp -u ftpuser,topSecret2 backupserver.example.com << EOF
cd remote/dir/
put db.sql
bye
EOF
... more code
```

Hier erzeugt zuerst `mysqldump` ein Backup der Datenbank. Das dazu erforderliche Passwort wird direkt mit der Option `-p` übergeben. Danach stellt `lftp` eine Verbindung zu einem FTP-Server her, um das Backup dort zu speichern. Das Passwort wird mit der Option `-u` übergeben. Wenn Sie dieses Backup-Script in einem Git-Repository speichern, kennt jeder, der Zugriff auf das Repository hat, die Passwörter zum MySQL- und FTP-Server.

Im Übrigen sollten Sie FTP möglichst ganz vermeiden. Das Protokoll ist inhärent unsicher. Wenn überhaupt, speichern Sie auf einem FTP-Server nur Dateien, die Sie vorher verschlüsselt haben!


© Rheinwerk Verlag, Bonn 2024

### Wie geht's besser?

Es gibt mehrere *Best Practices*, um derartige Sicherheitslücken zu verhindern:

- **Passwörter extern speichern:** Nahezu alle Programme, die ein Passwort zur Authentifizierung benötigen, können dieses aus einer Datei lesen. Bei `mysqldump` kommt `.my.cnf` zum Einsatz, bei `lftp` die Datei `.netrc` usw. Bei einigen Tools brauchen Sie gar keine Passwörter und können stattdessen Schlüsseldateien verwenden (z. B. SSH, siehe Abschnitt 12.4, »SSH-Authentifizierung mit Schlüsseln«).

  Selbstverständlich dürfen Passwort- oder Schlüsseldateien auf keinen Fall in das Git-Repository aufgenommen werden. Sie können aber in der Dokumentation darauf hinweisen, welche Passwortdateien Ihr Script voraussetzt.

- **Sonstige Einstellungen in einer getrennten Datei speichern:** Generell ist es eine gute Idee, sämtliche Parameter, die bei der Verwendung des Scripts auf einem anderen Rechner oder in einem anderen Kontext zu ändern sind, getrennt vom restlichen Code anzugeben. In einfachen Fällen ohne Sicherheitsrisiken kann dies in den ersten Zeilen des Scripts erfolgen. Ich benenne derartige Parameter oft in Großbuchstaben, um eine optische Differenzierung von den restlichen Variablen herzustellen.

  In aller Regel ist es noch besser, sämtliche Parameter in einer eigenen Datei zu speichern, die am Beginn des Scripts importiert wird. Gut geeignet sind JSON- oder INI-Dateien (siehe insbesondere Abschnitt 10.7, »INI-Dateien«).

- **Nur ein Muster der Einstellungsdatei in das Git-Repository aufnehmen:** Selbst wenn die Konfigurationsdatei keine Passwörter enthält, sollen die restlichen Parameter selten der ganzen Welt bekannt gegeben werden. Kritische Informationen sind auch Hostnamen, Port-Nummern sowie die Namen von AWS-Buckets oder von virtuellen Maschinen.

  Konfigurationsdateien mit solchen Daten sollten *nicht* in das Git-Repository aufgenommen werden! Ohne die Konfigurationsdatei wird die Inbetriebnahme Ihres Scripts für andere Personen aber zum Ratespiel. Deswegen ist es zweckmäßig, eine Musterkonfigurationsdatei in das Repository zu inkludieren. Wenn die echte Konfigurationsdatei `mybackup.conf` heißt, können Sie die Musterdatei z. B. `mybackup.conf.sample` nennen. Der Zweck dieser Datei ist es, die Syntax der Konfigurationsdatei zu erläutern. Sie enthält aber keine echten Daten, sondern frei erfundene Musterdaten.

Die obigen Regeln gelten für *alle* Git-Repositories, auch für private Repositories bzw. für Repositories auf eigenen, nicht öffentlichen GitLab-Instanzen! Wenn Sie ein Repository einrichten bzw. nutzen, können Sie nie vorhersehen, wer später noch alles Zugriff darauf erhält. Passwörter und andere kritische Informationen haben *nie* etwas in Repositories verloren.

Persönliches Exemplar für Thomas Bartholomäus

### Was tun, wenn ein Passwort doch im Git-Repository gelandet ist?

Das Konzept von Git-Repositories besteht darin, dass jede Datei auch in einem alten Zustand wiederhergestellt werden kann. Daher nützt es nichts, die betreffende Datei im nächsten Commit zu entfernen oder das Passwort in der Datei zu überschreiben. Die einzig sichere Vorgehensweise besteht darin, das Passwort beim betreffenden Programm oder Server zu ändern. Sollte das unmöglich sein, weil das externe System nicht von Ihnen kontrolliert wird, bleibt noch die Möglichkeit, das Repository als Ganzes zu löschen. Eine unerfreuliche Perspektive, ich weiß.

### Beispiel

Wie könnte das fragmentarische Backup-Script aus der Einleitung dieses Abschnitts besser und sicherer organisiert werden? Das Script beginnt mit der Importanweisung `. filename` (eine Kurzschreibweise für `source filename`). Damit wird die angegebene Datei in das Script eingelesen und von der Bash verarbeitet.

```
# Beispieldatei mybackup.sh
# Konfigurationseinstellungen laden
. mybackup.conf

# mysqldump liest das Passwort aus .my.cnf
mysqldump -u $MYSQLUSER $DBNAM > db.sql

# lftp liest das Passwort aus .netrc
lftp $FTPHOST << EOF
cd remote/dir/
put db.sql
bye
EOF
```

Die dazugehörige Konfigurationsdatei sieht so aus:

```
# Konfigurationsdatei mybackup.conf
MYSQLUSER=root
DBNAME=wordpress
FTPHOST=backupserver.example.com
```

Um sicherzustellen, dass diese Datei nicht im Git-Repository landet, wird `.gitignore` um die folgende Zeile erweitert:

```
# Datei .gitignore
mybackup.conf
```

Als Hilfestellung für Anwenderinnen und Anwender des Scripts wird dagegen die folgende Musterdatei in das Git-Repository aufgenommen:



© Rheinwerk Verlag, Bonn 2024

```
# Konfigurationsdatei mybackup.conf.sample
# Kopieren Sie diese Datei nach mybackup.conf und
# ersetzen Sie die Mustertexte durch zutreffende Namen!
# Beachten Sie, dass myscript.sh das MySQL-Passwort in .my.cnf
# und das FTP-Passwort in .netrc erwartet.
MYSQLUSER=mysqlaccountname
DBNAME=databasename
FTPHOST=ftphostname
```

**Powershell und Python**

Der Source-Operator `.` funktioniert auch in der PowerShell. Bei Python-Scripts speichern Sie die Einstellungen am besten in einer INI-Datei (siehe Abschnitt 10.7).

## 14.3 Git-Automatisierung

Viele `git`-Kommandoabfolgen wiederholen sich immer wieder, beispielsweise `git add`, `git commit`, `git pull` und `git push`. Es liegt nahe, den Aufruf dieser Kommandos in Scripts zu automatisieren.

Um Ihnen eine Vorstellung zu geben, wie derartige Scripts aussehen können, überprüfen die Kommandos im folgenden Listing, ob das aktuelle Verzeichnis überhaupt unter Git-Kontrolle steht und ob ein Remote Repository eingerichtet ist. Wenn diese Voraussetzungen erfüllt sind, zeigt `git add . --dry-run`, welche Dateien in den nächsten Commit aufgenommen würden. `git add .` setzt voraus, dass Sie zuvor eine `.gitignore`-Datei eingerichtet haben, die sämtliche nicht benötigten Dateien von Git-Operationen ausschließt.

Jetzt haben Sie die Wahl: Wenn Sie eine Commit-Message angeben, wird `git add` ohne die Option `--dry-run` ausgeführt und ein Commit durchgeführt; die Änderungen werden mit `git pull` und `git push` mit dem Remote Repository synchronisiert. Andernfalls wird der Vorgang abgebrochen. Aufgrund der Option `-e` im Hashbang endet das Script, sobald ein Kommando einen Fehler verursacht.

```
#!/bin/bash -e
# Beispieldatei git-ace.sh = git add/commit/push
# testen, ob aktuelles Verzeichnis unter Git-Kontrolle
if ! git rev-parse --is-inside-work-tree >& /dev/null; then
    echo "not in git repo, exit"
    exit 1
fi
```

Persönliches Exemplar für Thomas Bartholomäus

```
# testen, ob Remote Repository für aktuellen Zweig existiert
if ! git ls-remote >& /dev/null; then
    echo "no git remote, exit"
    exit 1
fi

# Dateien für den Commit auflisten
echo "Dry run:"
git add -A --dry-run

# Abbruch oder Eingabe der Commit Message
echo "Do you want to add and commit all files listed above?"
echo "Enter commit message or press return to exit."
echo -n "> "
read msg

# git add, commit, pull + push ausführen
if [[ $msg ]]; then
    git add -A
    git commit -m "$msg"
    git pull
    git push
else
    echo "exit"
fi
```

Die folgenden Zeilen zeigen einen Beispiellauf des Scripts. Ich habe das Script im Arbeitsverzeichnis dieses Buchs ausgeführt. (Natürlich verwalte ich alle Text-, Bild-, Layout- und Code-Dateien dieses Buchs mit Git.)

```
$ ./code/git/git-acp.sh

  Dry run:
    add '.gitignore'
    add 'bash.md'
    add 'code/git/git-acp.sh'
  Do you want to add and commit all files listed above?
  Enter commit message or press return to exit.
  > git scripting sample, work in progress
  ...
  To github.com:MichaelKofler/scripting-buch.git
     a0ff68f..a60f1d8  main -> main
```

Falls Sie unter Windows arbeiten, können Sie das Script in der Git-Bash ausführen oder müssen eine PowerShell-Variante des Codes entwickeln.


© Rheinwerk Verlag, Bonn 2024

### Eine Prise Skepsis

Sie werden im Internet eine Menge Bash- und PowerShell-Scripts finden, die so ähnlich wie das obige Beispiel aufgebaut sind. Ich bin grundsätzlich ein Fan davon, Abläufe zu automatisieren – sonst hätte ich dieses Buch nicht geschrieben. Das obige Beispiel hat aber nur Demonstrations-Charakter, ich benutze es selbst normalerweise nicht.

Ich arbeite täglich mit Git. Die wenigen Kommandos, die ich ständig brauche, kenne ich auswendig. Der Tippaufwand ist minimal. Da lohnt sich keine Automatisierung, bzw. allenfalls reichen ein paar Aliase, wie sie z. B. in der Zsh-Erweiterung »Oh My Zsh« enthalten sind.

Außerdem sind meine Git-Repositories von ganz unterschiedlichem Charakter: Manche Repositories haben nur einen einzigen Zweig, und ich bin der einzige Anwender. Bei anderen Repositories gibt es viele Benutzer, mehrere Zweige und diverse Sonderfälle. Bevor da etwas schiefgeht, führe ich `git` lieber manuell aus und verlasse mich nicht auf Automatismen!

### GitHub-Fernsteuerung mit »gh«

Das Kommando `git` funktioniert für jedes Git-Repository. Es kann allerdings keine Operationen ausführen, die GitHub-spezifisch sind. Dazu zählen etwa das Einrichten eines neuen Repositorys auf GitHub, die Durchführung eines Pull Requests etc. Für solche Zwecke bietet GitHub das Kommando `gh` an. Downloads und eine Installationsanleitung finden Sie hier:

*https://github.com/cli/cli/releases*

Nach der Installation führen Sie mit `gh auth login` eine einmalige Authentifizierung durch, die in einem Webbrowser durchzuführen ist. (`gh` kümmert sich um den Start des Webbrowsers und die Navigation zur richtigen Seite.)

Jetzt können Sie mit `gh repo list` Ihre Repositories auf GitLab auflisten, mit `gh repo create` ein neues Repository einrichten, mit `gh pr create` einen Pull-Request hinzufügen, mit `gh pr status` den Zustand eines Pull-Requests abfragen usw. Sie können `gh` wie `git` in Ihren Scripts aufrufen. Eine Referenz aller `gh`-Kommandos finden Sie hier:

*https://cli.github.com/manual/gh*

### GitLab-Fernsteuerung

Wie der Name schon andeutet, eignet sich `gh` nur für GitHub. Für GitLab gibt es ein analoges Projekt (Kommando `glab`, siehe *https://gitlab.com/gitlab-org/cli*).

Persönliches Exemplar für Thomas Bartholomäus

## 14.4 Git Hooks

*Git Hooks* sind Scripts, die automatisch vor oder nach bestimmten Git-Aktionen ausgeführt werden. In jedem neuen Git-Repository finden Sie im Verzeichnis `.git/hooks` eine Sammlung von Script-Mustern für verschiedene Aktionen:

```
$ ls .git/hooks

  applypatch-msg.sample       prepare-commit-msg.sample
  commit-msg.sample           pre-push.sample
  fsmonitor-watchman.sample   pre-rebase.sample
  post-update.sample          pre-receive.sample
  pre-applypatch.sample       push-to-checkout.sample
  pre-commit.sample           update.sample
  pre-merge-commit.sample
```

Bei den Sample-Dateien handelt es sich um gut dokumentierte Shell-Scripts. Ein Blick in die Dateien verrät nicht nur den Zweck des jeweiligen Hooks, sondern zeigt auch gleich funktionierenden Muster-Code.

Um einen Hook zu aktivieren, müssen Sie die Kennung `.sample` entfernen. (Anders als bei den Beispieldateien in diesem Buch ist auch die Kennung `.sh` tabu. Die Scripts werden nur ausgeführt, wenn sie gar keine Kennung haben!)

Unter Linux und macOS müssen zudem Sie sicherstellen, das das Execute-Bit gesetzt ist, also `chmod +x script.sh` ausführen. Unter Windows wird das Script von der Git Bash ausgeführt. Weitere Grundlageninformationen zu den Git Hooks finden Sie im Git-Handbuch:

*https://git-scm.com/book/de/v2/Git-einrichten-Git-Hooks*

### GitHub Actions, GitLab Pipelines

Als Alternative zu den Git-Hooks stellen die großen Git-Plattformen alternative Mechanismen zur Auswahl, um bei der Ausführung von Commits automatisch Aktionen auszuführen, z. B. zum automatisierten Testen oder zum Deployment auf andere Systeme.

Die Konfigurationsdateien dieser Mechanismen enthalten Script-Abschnitte, in denen Sie wiederum Code angeben können. Scripting-Grundkenntnisse helfen Ihnen somit auch dann, wenn Sie nicht mit Hooks arbeiten, sondern fortgeschrittene GitHub- oder GitLab-Features nutzen möchten.

### Beispiel: Nicht versionierte Dateien vor dem Commit erkennen

Das Script `.git/hooks/pre-commit` wird vor jedem Commit ausgeführt. Es kann dazu verwendet werden, die Einhaltung bestimmter Spielregeln für einen Commit zu


© Rheinwerk Verlag, Bonn 2024

garantieren. Wenn das Script einen Fehlercode zurückgibt (also einen Exit-Code ungleich 0), wird `git commit` abgebrochen, und der Benutzer muss den erkannten Mangel beheben.

Das folgende Beispiel habe ich aus dem Git-Buch entnommen, das ich zusammen mit Bernd Öggl verfasst habe. `pre-commit` testet, ob es geänderte oder neue Dateien gibt, die nicht im Commit enthalten und auch nicht durch `.gitignore` ausgeschlossen sind. Wenn das der Fall ist, werden diese Dateien durch `git status` aufgelistet.

`exit 1` verhindert, dass Sie bei einem Commit eine Datei vergessen. Das Script zwingt Sie aber gleichzeitig, alle Dateien, die nicht berücksichtigt werden sollen, explizit in `.gitignore` aufzuzählen. Das kann ziemlich lästig sein.

```
#!/bin/sh
# Beispieldatei pre-commit, muss in das Verzeichnis
# .git/hooks kopiert werden
untracked=$(git ls-files --others --exclude-standard | wc -l)
if [ $untracked -gt 0 ]; then
    git status
    echo
    echo "Pre commit fail! There are untracked files. Either run"
    echo "'git add' or add an entry to .gitignore."
    exit 1
fi
```

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

TEIL III

# Anwendungen und Beispiele

Persönliches Exemplar für Thomas Bartholomäus

© Rheinwerk Verlag, Bonn 2024

Kapitel 15

# Backups

Das automatisierte Erstellen von Backups ist *die* klassische Scripting-Anwendung. In diesem Kapitel zeige ich Ihnen einige konkrete Beispiele:

- Synchronisierung von Verzeichnissen eines Desktop-Rechners
- Backup eines WordPress-Systems auf einem Linux-Rechner
- Backup von SQL-Server-Datenbanken unter Windows

**Voraussetzungen für dieses Kapitel**

Die Beispiele dieses Kapitels sind in den Sprachen Bash und PowerShell formuliert. Außer den Sprachgrundlagen setze ich voraus, dass Sie mit dem Windows Task Scheduler und mit Cron umgehen können (siehe Kapitel 11, »Scripts automatisch ausführen«) und sich mit SSH auskennen (siehe Kapitel 12, »SSH«). Des Weiteren sollten Sie mit Kommandos wie `rsync`, `scp` und `tar` bzw. CmdLets wie `Compress-Archive` vertraut sein.

Anstelle eigener Scripts können Sie natürlich fertige Backup-Lösungen verwenden. Das Internet ist voll von Backup-Tools, die das ganze Spektrum von einfachen, kostenlosen Scripts bis hin zu teuren, kommerziellen Anwendungen abbilden. Wenn die Aufgabenstellung wie in den Beispielen dieses Kapitels relativ simpel ist, reichen eigene Scripts definitiv aus. Aber das gilt natürlich nicht, wenn Sie firmenweite, redundante Backups brauchen, die alle erdenklichen Speicherorte berücksichtigen sollen – also z. B. Active Directories, Cloud-Speicher oder außerhalb des Firmennetzwerks befindliche Mitarbeiter-Notebooks.

## 15.1 Verzeichnisse auf externen Datenträger synchronisieren

Vermutlich verwenden Sie in irgendeiner Form die Cloud als Backup-Speicher: Als Entwickler bzw. Entwicklerin synchronisieren Sie Ihren Code mit Git in ein Remote Repository und kombinieren so die Vorteile der Versionskontrolle mit denen eines externen Backups. Als »gewöhnliche« Anwenderin bzw. »gewöhnlicher« Anwender speichern Sie wichtige Dateien einfach in einem OneDrive-, NextCloud-, DropBox-

Persönliches Exemplar für Thomas Bartholomäus

oder iCloud-Drive-Verzeichnis. All diese Verfahren sind bei einer korrekten Anwendung ebenso praktisch wie sicher.

Der Ausgangspunkt für dieses Beispiel ist der Wunsch, wichtige Verzeichnisse des Notebooks *zusätzlich* auf einen externen Datenträger zu synchronisieren. Der Vorteil dieses altmodischen Backup-Verfahrens besteht darin, dass die gespeicherten Dateien auch dann zur Verfügung stehen, wenn – aus welchem Grund immer – gerade kein Cloud-Zugang möglich ist. Lokale Backups bieten sich auch dann an, wenn die Datenmengen für Cloud-Speicher zu groß sind – etwa bei virtuellen Maschinen, Video-Projekten usw.

Ich möchte auch gleich auf den wichtigsten Nachteil hinweisen: »Synchronisieren« heißt, dass auf Ihrem Notebook gelöschte Dateien auch auf dem externen Datenträger gelöscht werden. Dieses Backup-Verfahren bietet also keine Möglichkeit, gelöschte oder überschriebene Dateien wiederherzustellen. Insofern sollten die hier vorgestellten Bash- bzw. PowerShell-Scripts nicht die alleinige Form Ihres Backups sein, sondern andere Formen ergänzen.

#### Vorsicht bei Backups von geöffneten Dateien

Wenn Sie mit dem Script Dateien synchronisieren, die gerade geöffnet sind und sich während des Kopierens verändern, ist das Backup zumeist wertlos. Das betrifft z. B. Image-Dateien von virtuellen Maschinen oder Datenbanken.

Für dieses Problem gibt es leider keine allgemeingültige Lösung. Idealerweise führen Sie das Script zu einem Zeitpunkt aus, zu dem möglichst wenige Dateien aktiv in Verwendung sind. Die meisten Datenbank-Server bieten Möglichkeiten, konsistente Backups auch im laufenden Betrieb durchzuführen – aber zumeist nicht auf Dateiebene. (Zwei Beispiele dazu folgen in den weiteren Abschnitten dieses Kapitels.) Virtuelle Maschinen könnten Sie in Ihrem Script für den Zeitpunkt des Backups herunterfahren und danach wieder neu starten. Solche Lösungsansätze sind aber ganz spezifisch von der Software abhängig, die auf Ihrem Rechner läuft.

Je nach Betriebssystem stellen Dateisystem-Snapshots einen weiteren Lösungsweg dar: Dabei frieren Sie vorübergehend eine Kopie des Dateisystems ein und verwenden diese statische Kopie als Basis für das Backup. Unter Linux bieten der *Linux Volume Manager* (LVM) oder das btrfs-Dateisystem solche Möglichkeiten.

### PowerShell-Script mit robocopy

`robocopy` steht für *Robust File Copy* und ist ein Windows-Kommando, das seit 2008 mit allen Windows-Versionen ausgeliefert wird. Auch wenn das Kommando leider nicht in Form von CmdLets aufgerufen werden kann, lässt es sich gut in PowerShell-


© Rheinwerk Verlag, Bonn 2024

Scripts integrieren. Die unzähligen Optionen des Kommandos sind hier dokumentiert:

*https://en.wikipedia.org/wiki/Robocopy*
*https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/robocopy*

Das folgende Script beginnt mit der Initialisierung von drei Parametern, die angeben, welche Verzeichnisse wohin synchronisiert werden sollen.

`Get-Volume` ermittelt eine Liste aller Dateisysteme. Wenn darin der Zieldatenträger nicht erkannt wird, endet das Script. Beachten Sie bei `Where-Object`, dass `Get-Volume` den Dateisystemnamen zwar in der Spalte `FriendlyName` anzeigt, die Eigenschaft aber tatsächlich `FileSystemLabel` heißt. Was sich die Entwicklerinnen und Entwickler von `Get-Volume` dabei gedacht haben, darüber kann man nur den Kopf schütteln!

Wenn das Zieldateisystem erkannt wurde, wird der dazugehörige Laufwerkbuchstabe ermittelt. Dieser muss nicht immer gleich bleiben, je nachdem, welche Datenträger gerade in Verwendung sind!

Das Script protokolliert alle `robocopy`-Ausgaben in eine Datei der Form `robocopy-2023-12-31--17-30.log`. Diese Datei und ein Logging-Verzeichnis werden automatisch eingerichtet.

Zuletzt durchläuft das Script eine Schleife mit allen in `$syncdirs` aufgezählten Verzeichnissen und führt `robocopy` aus. Die dabei eingesetzten Optionen haben die folgende Bedeutung:

- `/e`: Verzeichnisse rekursiv durchlaufen
- `/purge`: lokal gelöschte Dateien und Verzeichnisse auch im Backup löschen (Vorsicht!)
- `/xo`: nur Dateien kopieren, die sich geändert haben; diese letzte Option beschleunigt den Synchronisationsvorgang ab dem zweiten Durchlauf enorm
- `/log+:filename`: fügt der Datei Logging-Ausgaben hinzu

```
#!/usr/bin/env pwsh
# Beispieldatei sync-folders.ps1
# $destvolume: Name des Backup-Datenträgers (z. B. ein USB-Stick)
# $destdir:   Name des Zielverzeichnisses auf diesem Datenträger
# $logdir:    Name des Logging-Verzeichnisses auf dem Datenträger
# $syncdirs:  Liste der zu synchronisierenden Verzeichnisse
#             (relativ zu den persönlichen Dateien)
$destvolume = "mybackupdisk"
$destdir    = "backup-kofler"
$logdir     = "$destdir\sync-logging"
$syncdirs   = "Documents", "Pictures", "myscripts"
```

Persönliches Exemplar für Thomas Bartholomäus

```
# Laufwerkbuchstabe des Ziel-Dateisystems ermitteln
$disk = Get-Volume |
        Where-Object { $_.FileSystemLabel -eq $destvolume }
if (! $disk) {
    Write-Output "Backup disk $destvolume not found. Exit."
    Exit 1
}
$drive = ($disk | Select-Object -First 1).DriveLetter
Write-Output "Syncing with disk ${drive}:"

# Zielverzeichnis erzeugen, wenn es noch nicht existiert;
# | Out-Null verhindert die Anzeige des Verzeichnisnamens
New-Item -ItemType Directory -Force "${drive}:\${destdir}" |
  Out-Null

# Logging-Verzeichnis und Logging-Datei erzeugen
New-Item -ItemType Directory -Force "${drive}:\${logdir}" |
  Out-Null
$logfile = `
  "${drive}:\${logdir}\robocopy-{0:yyyy-MM-dd--HH-mm}.log" `
  -f (Get-Date)
New-Item $logfile | Out-Null

# Schleife über die Sync-Verzeichnisse
foreach ($dir in $syncdirs) {
    $from = "${HOME}\$dir"
    $to = "${drive}:\${destdir}\$dir"
    Write-Output "sync from $from to $to"
    robocopy /e /purge /xo /log+:$logfile "$from" "$to"
}
```

Bei meinen Tests hat das Script immer wieder den Fehler *access denied* (Error-Code 5) beim Schreiben der Dateien auf den USB-Datenträger ausgelöst. Im Internet gibt es unzählige Berichte über diesen Fehler und fast ebenso viele Lösungsvorschläge. Zuverlässig wirkt anscheinend nur die Ausführung des Scripts mit Administratorrechten. Erste Tests führen Sie einfach in einem Terminal-Fenster mit Administratorrechten durch. In der Folge richten Sie das Script im *Windows Task Scheduler* so ein, dass es einmal täglich mit Administratorrechten gestartet wird (Option MIT HÖCHSTEN PRIVILEGIEN AUSFÜHREN im Dialogblatt ALLGEMEIN).

## Verbesserungsideen

Bevor Sie dem Script Ihre Daten anvertrauen, möchte ich Sie auf ein paar Einschränkungen hinweisen. Wenn Sie möchten, können Sie das Script ja dahingehend optimieren.


© Rheinwerk Verlag, Bonn 2024

- **Warnung bei Fehler bzw. Nichtausführung:** Das Script bricht einfach ab, wenn der externe Datenträger gerade nicht zur Verfügung steht. Besser wäre es, nach mehreren erfolglosen Versuchen in irgendeiner Form eine Warnung darzustellen oder zu versenden.
- **Ausschluss-Regeln:** Das Script kann Verzeichnisse entweder ganz oder gar nicht synchronisieren. In der Praxis wären Ausschlusskriterien für Dateien oder Verzeichnisse hilfreich, die nicht berücksichtigt werden.

### Bash-Script mit rsync

Wenn Sie unter Linux oder macOS arbeiten, realisieren Sie vergleichbare Synchronisations-Aufgaben am besten als Bash-Script mit `rsync`. Das folgende Script hat denselben Aufbau wie das vorhin erläuterte PowerShell-Script. Es testet, ob der Backup-Datenträger an einem bestimmten `mount`-Punkt zur Verfügung steht und führt dann für eine Liste lokaler Verzeichnisse das Kommando `rsync` aus.

Wie Sie aus Abschnitt 12.3, »scp und rsync«, wissen, funktioniert `rsync` auch im Zusammenspiel mit SSH. Sie können das Script also relativ einfach so anpassen, dass Ihre Verzeichnisse nicht mit einem externen Datenträger, sondern mit einem anderen Rechner synchronisiert werden.

```
# Beispieldatei sync-folders.sh
# was soll wohin synchronisiert werden
DESTVOLUME="/run/media/kofler/mybackupdisk"
DESTDIR="backup-kofler"
LOGDIR="$DESTDIR/sync-logging"
SYNCDIRS=("Documents" "Pictures" "myscripts")

# ist das Backup-Dateisystem verfügbar?
if ! mount | grep $DESTVOLUME --quiet; then
    echo "Backup disk $backupdisk not found. Exit."
    exit 1
fi

# Ziel- und Logging-Verzeichnis erzeugen
mkdir -p "$DESTVOLUME/$DESTDIR"
mkdir -p "$DESTVOLUME/$LOGDIR"

# Dateiname für Logging zusammensetzen
logfname=$(date "+rsycn-%Y-%m-%d--%H-%M.log")
log="$DESTVOLUME/$LOGDIR/$logfname"

# Schleife über alle Verzeichnisse
for dir in "${SYNCDIRS[@]}"; do
    from=$HOME/$dir
    to=$DESTVOLUME/$DESTDIR/$dir
```

Persönliches Exemplar für Thomas Bartholomäus

```
    echo "sync from $from to $to"
    rsync -a --delete -v "$from" "$to" >> $log
done
```

Die `rsync`-Optionen haben die folgende Wirkung:

- `-a` (*archive*): Verzeichnisse rekursiv verarbeiten, Dateiinformationen (Besitzer, Zugriffsrechte) erhalten
- `--delete`: lokal gelöschte Verzeichnisse und Dateien auch im Backup löschen (Vorsicht!)
- `-v` (*verbose*): detailliert ausgeben, was gerade vor sich geht

Damit das Script automatisch einmal täglich um 12:30 Uhr aufgerufen wird, habe ich den folgenden Eintrag im `/etc/crontab` eingebaut. Passen Sie einfach die gewünschte Zeit, den Account-Namen und den Pfad zum Backup-Script an Ihre Gegebenheiten an:

```
# in der Datei /etc/crontab
30 12 * * * kofler /home/kofler/myscripts/sync-folders.sh
```

**macOS**

Ich habe dieses Script nur unter Linux getestet. Für macOS sind kleine Adaptierungen erforderlich, insbesondere was den Test betrifft, ob der Backup-Datenträger gerade mit dem Rechner verbunden ist.

## 15.2 WordPress-Backup

Ausgangspunkt für das zweite Beispiel ist ein Linux-Rechner, auf dem ein Web- und ein Datenbank-Server laufen, um einen Webauftritt mit WordPress zu realisieren. Gesichert werden sollen einmal täglich der Inhalt der Datenbank `wp`, das Verzeichnis `/var/www/html/wordpress` sowie die ganze Server-Konfiguration, also das Verzeichnis `/etc`. Es sollen immer komprimierte Backups der letzten sieben Tage aufbewahrt werden, und zwar sowohl lokal als auch auf einem zweiten Server, der via SSH erreichbar ist.

Die Implementierung erfolgt durch ein Bash-Script, das mit root-Rechten läuft. Für das Backup der MySQL-Datenbank wird das Kommando `mysqldump` verwendet. Es stellt die Verbindung zum MySQL-Server mit dem Account `wpbackupuser` her. Dieser Account, der lediglich Rechte zur Durchführung von Backups umfasst, kann in MySQL so eingerichtet werden:


© Rheinwerk Verlag, Bonn 2024

```
CREATE USER wpbackupuser@localhost IDENTIFIED BY 'TopSecret123';
GRANT Select, Lock Tables, Show View
   ON wp.* TO wpbackupuser@localhost;
GRANT Process, Reload ON *.* TO wpbackupuser@localhost;
```

Um zu vermeiden, dass das Script das Passwort im Klartext enthält, wird dieses in der Datei `/root/.my.cnf` gespeichert. `mysqldump` sucht automatisch nach dieser Datei und wertet die darin enthaltenen Daten aus. Stellen Sie mit `chmod 600 .my.cnf` sicher, dass niemand außer `root` diese Datei lesen darf!

```
# in der Datei /root/.my.cnf
[mysqldump]
user = wpbackupuser
password = TopSecret123
default-character-set = utf8
```

Nachdem alle Voraussetzungen geschaffen sind, können Sie mit der Programmierung starten. In den ersten Zeilen des Scripts sind diverse Konfigurationsparameter zusammengefasst. Zur besseren Lesbarkeit habe für diese Variablen Großbuchstaben verwendet.

`date +%u` erzeugt eine durchlaufende Nummer für den Wochentag (1 für Montag bis 7 für Sonntag). Diese Nummer wird in die Namen der Backup-Dateien integriert.

`mysqlopt` enthält Optionen zur Durchführung des Backups. Das Verhalten von `mysqldump` kann durch unzählige Optionen gesteuert werden. Ich habe mich hier auf `--single-transaction` beschränkt. Diese Option stellt sicher, dass sich die Datenbank während des Backups nicht ändern kann. Die Ausgaben von `mysqldump` werden an `gzip` weitergeleitet, komprimiert und schließlich in der Datei `$dbfile` gespeichert. Falls die Datei schon existiert, wird sie überschrieben. (Das passiert zum ersten Mal, wenn das Script an acht aufeinanderfolgenden Tagen läuft. Sie haben also immer Backup-Versionen der letzten sieben Tage.)

Zur Durchführung des Backups der WordPress-Installation reicht ein kurzes `tar`-Kommando (siehe auch Abschnitt 6.3, »Dateien komprimieren und archivieren«). Das WordPress-Verzeichnis enthält auch alle installierten Plugins, alle hochgeladenen Bilder etc. In der Regel ist die mit `tar` erzeugte Datei viel größer als das MySQL-Backup.

In der letzten Zeile des Scripts werden die vorerst nur lokal gespeicherten Backups auf einen zweiten Rechner kopiert. Damit befinden sich alle Backup-Dateien redundant an zwei unterschiedlichen Orten.

```
# Beispieldatei lamp-backup.sh
BACKUPDIR=/localbackup
DB=wp
DBUSER=wpbackupuser
WPDIR=/var/www/html/wordpress
SSH=user@otherhost:wp-backup/
```

Persönliches Exemplar für Thomas Bartholomäus

```
# MySQL-Backup
weekday=$(date +%u)
dbfile=$BACKUPDIR/wp-db-$weekday.sql.gz
mysqlopt='--single-transaction'
mysqldump -u $DBUSER $mysqlopt $DB | gzip -c > $dbfile

# Backup der WordPress-Installation samt Plugins und Uploads
htmlfile=$BACKUPDIR/wp-html-$weekday.tar.gz
tar czf $htmlfile -C $WPDIR .

# Backup des /etc-Verzeichnisses
etcfile=$BACKUPDIR/etc-$weekday.tar.gz
tar czf $etcfile -C /etc .

# Backup-Dateien auf einen zweiten Rechner kopieren
scp $dbfile $htmlfile $etcfile $SSH
```

Nachdem Sie das Script manuell ausgeführt haben, müssen Sie sich noch um den automatisierten Aufruf kümmern. Die erforderliche Zeile in `/etc/crontab` können Sie entsprechend dem folgenden Muster gestalten:

```
# in /etc/crontab
# Script jede Nacht um 3:00 mit root-Rechten ausführen
00 3 * * * root /path/to/lamp-backup.sh
```

Nachdem Sie das Backup-System eingerichtet haben, sollten Sie es ein paar Tage beobachten. Funktioniert der automatische Cron-Aufruf? Eine häufige Fehlerursache besteht darin, dass für Cron-Jobs oft eine andere `PATH`-Einstellung gilt als für den `root`-Account, in dem Sie die Tests durchgeführt haben. Das gilt insbesondere, wenn Sie im Script Kommandos verwenden, die manuell abseits der üblichen Pfade installiert wurden. Bei solchen Kommandos müssen Sie gegebenenfalls den vollständigen Pfad angeben.

Testen Sie unbedingt auch, ob Sie auf einem anderen Rechner oder in einer virtuellen Maschine in der Lage sind, Ihre WordPress-Installation aus den Backup-Dateien wieder herzustellen! Was nützt das schönste Script, wenn Sie erst im Katastrophenfall bemerken, dass das Backup aus irgendeinem Grund unvollständig ist!

**Erweiterung: Verschlüsselte Backups auf AWS S3 hochladen**

Die Aufbewahrung von Backup-Dateien auf externen Rechnern ist – was die Zugriffsmöglichkeiten durch Angreifer betrifft – oft weniger sicher als auf dem lokalen Server. Deswegen ist es eine gute Idee, Backup-Dateien vor dem Upload auf andere Hosts zu verschlüsseln.


© Rheinwerk Verlag, Bonn 2024

Wie Sie dazu vorgehen und wie Sie den Cloud-Speicher Amazon S3 als Speicherort verwenden können, zeige ich Ihnen in Kapitel 20, »Scripting in der Cloud«. Dort greife ich dieses Beispiel noch einmal auf und erweitere es diesbezüglich.

## 15.3 SQL-Server-Backup

Das Ziel des folgenden Scripts ist es, alle Datenbanken eines SQL Servers in komprimierten Dateien in einem vorgegebenen Verzeichnis zu sichern. Dabei gehe ich davon aus, dass Sie unter Windows arbeiten. (Grundsätzlich lässt sich der SQL Server auch unter Linux installieren, aber das habe ich für dieses Beispiel nicht getestet.)

Zur Administration des SQL Servers durch PowerShell-Scripts gibt es in der PowerShell-Gallery das von Microsoft entwickelte Modul `SqlServer`:

```
> Install-Module SqlServer
```

Eine Referenz aller CmdLets dieses Moduls finden Sie hier:

*https://learn.microsoft.com/en-us/powershell/module/sqlserver*

Mit `Backup-SqlDatabase` ist es ganz einfach, ein Backup einer Datenbank zu erstellen:

```
> Backup-SqlDatabase -ServerInstance "localhost\SQLEXPRESS01" `
                     -Database "dbname"
```

Im obigen Kommando geben Sie mit `-ServerInstance` an, unter welchem Namen der SQL Server erreichbar ist. Das Kommando kann natürlich nur ausgeführt werden, wenn Ihr Windows-Account Zugriffsrechte auf den Datenbank-Server hat. Das Backup landet standardmäßig im Backup-Verzeichnis des Servers, üblicherweise also hier:

```
C:\Program Files\Microsoft SQL Server\MSSQLn.n\MSSQL\Backup
```

Alternativ geben Sie den gewünschten Backup-Dateinamen mit der Option `-BackupFile` vor. (Es gibt eine Menge weiterer Optionen, mit denen Sie die Funktionsweise von `Backup-SqlDatabase` steuern können. Werfen Sie einen Blick in die Dokumentation!) Die durch `Backup-SqlDatabase` erzeugten Dateien liegen in einem binären Format vor, das sich gut komprimieren lässt.

### dbatools

Eine attraktive Alternative zum `SqlServer`-Modul sind die `dbatools`. Das Modul ist wesentlich umfangreicher und umfasst wesentlich mehr CmdLets. Wenn Sie sich auf die Administration von SQL-Server-Instanzen spezialisieren, sollten Sie sich unbedingt mit den `dbatools` anfreunden!

*https://dbatools.io/commands*

Persönliches Exemplar für Thomas Bartholomäus

### Alle Datenbanken per Script sichern und komprimieren

Mit diesen einleitenden Informationen sollte es kein Problem sein, das folgende Script zu verstehen. Es beginnt mit der Deklaration einiger Parameter, die Sie entsprechend Ihrem eigenen Setup anpassen müssen. `$exclude` enthält eine Liste von Datenbanken, die *nicht* gesichert werden sollen. Diese Liste muss unbedingt `tempdb` enthalten. Der Versuch, `Backup-SqlDatabase` auf diese SQL-Server-interne Datenbank anzuwenden, führt zu einem Fehler.

`Get-SqlDatabase` ermittelt alle am SQL Server verfügbaren Datenbanken. Eine Schleife verarbeitet nun alle Datenbanken. Jede Backup-Datei wird sofort in eine eigene Archiv-Datei verpackt. (Es wäre natürlich denkbar, alle Datenbanken gemeinsam in einem Archiv zu sichern. Aber da Datenbanken riesig werden können, ist das keine gute Idee.) Mit der Option `-Force` werden schon existierende ZIP-Dateien ohne Fehlermeldung überschrieben. Die ursprüngliche Backup-Datei wird in der Folge gelöscht.

```
# Beispieldatei backup-all-dbs.ps1
$instance = "localhost\SQLEXPRESS01"
$backupdir = "C:\sqlserver-backups"
$exclude = "tempdb", "other", "tmp"

# Backup-Verzeichnis erzeugen, wenn es noch nicht existiert
New-Item -ItemType Directory -Force $backupdir | Out-Null

# Schleife über alle Datenbanken
$dbs = Get-SqlDatabase -ServerInstance $instance
foreach($db in $dbs) {
    # Datenbanken aus $exclude überspringen
    if ($db.name -in $exclude) { continue }

    # Name der Backup-Dateien zusammensetzen
    $backupname = "$backupdir\$($db.name).bak"
    $archivename = "$backupdir\$($db.name).zip"
    Write-Output "Backup database $($db.name) to $archivename"

    # backup, create archive (-Force to overwrite existing),
    # delete original backup file
    Backup-SqlDatabase -ServerInstance $instance `
      -Database $db.name -BackupFile $backupname
    Compress-Archive -Force $backupname $archivename
    Remove-Item $backupname
}
```


© Rheinwerk Verlag, Bonn 2024

# Kapitel 16
# **Bildverarbeitung**

In meinem Arbeitsalltag helfen Scripts häufig dabei, die digitale Bilderflut unter Kontrolle zu halten. Die in diesem Kapitel vorgestellten Scripts erledigen die folgenden Aufgaben:

- Bilder von einem Format in ein anderes umwandeln, dabei gegebenenfalls auch verkleinern und mit Wasserzeichen ausstatten
- Fotos anhand ihres Aufnahmedatums (EXIF-Daten) in Verzeichnisse sortieren
- Metadaten von Fotos in eine Datenbank eintragen

Zum Einsatz kommen dabei die Open-Source-Tools *ImageMagick* und *ExifTool*.

> **Voraussetzungen für dieses Kapitel**
>
> Zur Lektüre dieses Kapitels benötigen Sie lediglich Grundkenntnissen in den drei Sprachen Bash, PowerShell und Python.

## 16.1 Bilddateien manipulieren

Ganz egal, ob Sie PNG-Bilder in das JPG-Format umwandeln möchten oder umgekehrt, dabei Wasserzeichen einbauen oder Transparenzeffekte berücksichtigen wollen, die maximale Auflösung begrenzen oder ein Teilbild ausschneiden möchten – die kostenlosen Tools der ImageMagick-Bibliothek helfen Ihnen dabei. Das einzige Hindernis ist die nicht ganz triviale Steuerung der Kommandos durch unzählige Optionen.

### ImageMagick installieren

Die meisten Linux-Distributionen stellen ImageMagick als Paket zur Verfügung. Unter Ubuntu führen Sie zur Installation `apt install imagemagick` aus. Unter macOS gelingt die Installation am einfachsten mit Brew (also `brew install imagemagick`).

Für Windows finden Sie unter *https://imagemagick.org/script/download.php* ein Installationsprogramm. Falls Sie eine bereits installierte Version aktualisieren möch-

Persönliches Exemplar für Thomas Bartholomäus

ten, sollten Sie diese vorher deinstallieren – andernfalls werden die alte und die neue Version parallel installiert, was selten zweckmäßig ist. Das Setup-Programm schlägt vor, das Installationsverzeichnis zum PATH hinzuzufügen – diese Option sollten Sie unbedingt aktiviert lassen. Generell ist es zumeist zweckmäßig, die vorausgewählten Optionen zu belassen (siehe Abbildung 16.1).

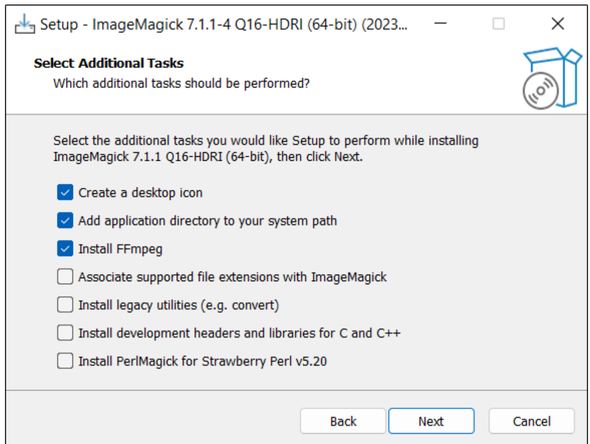



**Abbildung 16.1** ImageMagick-Installation unter Windows

Überzeugen Sie sich mit `magick --version` in einem Terminal, dass die Installation erfolgreich war und das Kommando auch gefunden wird:

```
$ magick --version
  Version: ImageMagick 7.1.1-4 Q16-HDRI x86_64 ...
```

Wenn Sie mit ImageMagick auch PostScript-, PDF- oder EPS-Dateien lesen möchten, müssen Sie außerdem den PostScript-Interpreter GhostScript installieren. Unter Linux und macOS können Sie entsprechende Pakete mit `apt`, `dnf` oder `brew` installieren. Windows-Downloads finden Sie hier:

*https://ghostscript.com/releases/gsdnld.html*

## ImageMagick ausprobieren

ImageMagick stellt das Kommando `magick` zur Verfügung. Es ist sehr vielseitig anwendbar. Allerdings gibt es mehrere Hundert Optionen, um das Verhalten des Kommandos zu steuern:

*https://imagemagick.org/script/magick.php*

Lassen Sie sich davon nicht abschrecken! In vielen Fällen lässt sich `magick` mühelos anwenden. Um eine vorhandene Datei in ein anderes Format umzuwandeln, geben Sie einfach den Dateinamen der Quelldatei und den einer neuen Ergebnisdatei an. `magick` erkennt den gewünschten Datentyp anhand der Kennung:


© Rheinwerk Verlag, Bonn 2024

```
$ magick in.bmp out.png
```

Das folgende Kommando verkleinert ein PNG-Bild auf eine Größe von maximal 1024 × 1024 Pixel. Die Bildproportionen bleiben dabei erhalten. Ein Bild mit ursprünglich 6000 × 4000 Pixel wird auf 1024 × 683 Pixel skaliert. Das >-Zeichen hinter `1024x1024` bewirkt, dass kleine Bilder *nicht* hochskaliert werden. 1024 × 1024 Pixel sind also die Obergrenze. Das Ergebnis wird im JPG-Format gespeichert.

Transparente Bereiche des PNG-Bilds werden hellrosa dargestellt. Der Farbcode entspricht dabei der vertrauten CSS-Syntax. `-flatten` fügt alle Bildebenen zusammen. `-quality 75` reduziert die Qualität der JPEG-Kompression auf 75 Prozent. Das verringert die Qualität, aber auch die Größe der Bilddatei. (Standardmäßig gilt eine Qualitätsstufe von 92 Prozent.)

```
$ magick in.png -resize '1024x1024>' -background '#ffdddd' \
    -flatten -quality 75 out.jpg
```

Im dritten Beispiel baut `magick` in der Mitte des Bilds einen diagonalen Watermark-Text mit einem Copyright-Hinweis ein:

```
$ magick in.png -gravity center \
    -pointsize 40 -font Arial -fill black \
    -draw "rotate 45 text 0,0 '(c) Michael Kofler 2023'" out.png
```

Im Gegensatz zu vielen anderen Kommandos ist bei `magick` die Reihenfolge der Optionen nicht egal! Sie werden vielmehr in der angegebenen Reihenfolge ausgeführt bzw. beziehen sich auf das gerade aktuelle Bild (falls mehrere Bilddateien zusammengefügt werden).

Wenn Sie mehrere Bilder auf einmal verarbeiten möchten, empfiehlt sich zumeist die Programmierung eines Scripts. Das hat den Vorteil, dass die oft mühsam zusammengesuchten Optionen beim nächsten Mal wieder zur Verfügung stehen. Sie können aber selbstverständlich einfache Aufgaben auch direkt im Terminal erledigen.

Das abschließende Beispiel in Bash-Syntax durchläuft alle `*.jpeg`-Dateien im aktuellen Verzeichnis und macht daraus sogenannte *Thumbnails*, also stark verkleinerte Bilder. Passen Sie aber auf! Weil das folgende Kommando den gleichen Quell- und Zieldateinamen verwendet, werden die ursprünglichen Bilder überschrieben! Wenn Sie das nicht möchten, müssen Sie einen anderen Zieldateinamen verwenden (z. B. `"thumb-$file"`):

```
$ for file in *.jpeg; do magick "$file" -resize '320x320' \
    -quality 80 "$file"; done
```

Soweit das Zieldateiformat EXIF-Daten unterstützt, bleiben diese Daten erhalten. EXIF-Daten sind Metadaten, die unter anderem Auskunft über Ort und Zeit der Aufnahme geben. EXIF-Daten stehen im Mittelpunkt der weiteren Abschnitte dieses

Persönliches Exemplar für Thomas Bartholomäus

Kapitels. Wenn Sie EXIF-Daten entfernen möchten, können Sie `magick` mit der Option `-strip` aufrufen. Sicherer ist es, das im nächsten Abschnitt vorgestellte Kommando `exiftool -all:all= file.jpg` einzusetzen.

**Sonderzeichen quotieren**

Beachten Sie, dass die `magick`-Syntax diverse Sonderzeichen vorsieht, die in der Bash und der PowerShell eine besondere Bedeutung haben. Entweder stellen Sie die betreffenden Zeichenketten wie in den vorangegangenen Beispielen in Anführungszeichen, oder Sie verwenden das Quotierungszeichen \ (Bash) bzw. ` (PowerShell).

**KI-Hilfe**

Auch KI-Tools wie ChatGPT kennen ImageMagick. Wenn Sie eine spezifische Aufgabenstellung haben, ist es zumindest einen Versuch wert, die erforderliche Kombination von Optionen mit KI-Unterstützung zu ermitteln. Bei meinen Versuchen lag die Trefferquote etwa bei 50 Prozent. Nicht immer entsprach das Ergebnis meinen Vorstellungen, aber oft führte mich ChatGPT immerhin auf die richtige Fährte. Beachten Sie, dass sowohl ChatGPT als auch viele Lösungsvorschläge im Internet statt `magick` das veraltete Kommando `convert` verwenden.

### Beispiel: convert2eps (Bash-Variante)

Um aus den Textdateien meiner Bücher in den Formaten Markdown bzw. LaTeX drucktaugliche PostScript-Dateien zu machen, brauche ich zu jedem Screenshot im PNG-Format eine äquivalente EPS-Datei (*Encapsulated PostScript*). Genau darum kümmert sich das folgende Script.

Es durchläuft in einer Schleife alle PNG-Dateien im aktuellen Verzeichnis. Wenn es zu `<name>.png` keine Datei `<name>.eps` gibt bzw. wenn die PNG-Datei neuer ist als die EPS-Datei (Vergleichsoperator `-nt`, also *newer than*), erzeugt `magick` eine neue EPS-Datei.

Die `magick`-Optionen bzw. -Parameter haben die folgende Bedeutung: `-quiet` unterdrückt die Anzeige von Warnungen und Statusmeldungen. `-background white` verwendet einen weißen Hintergrund für die transparenten Bereiche des Bilds. `-flatten` fügt die Bildebenen zusammen.

`eps2:$epsfile` gibt an, dass beim Schreiben der EPS-Datei die Spezifikationen des PostScript Levels 2 gelten sollen. Das hat einen großen Einfluss auf die Bildgröße, weil die Datei viel besser komprimiert wird (ohne Qualitätsverlust) als im Default-Level 1. Noch kompaktere Dateien verspricht Level 3 (also `eps3:$epsfile`). Dieser Level stellt aber höhere Voraussetzungen an den Drucker bzw. die Belichtungs-Software und kann Kompatibilitätsprobleme verursachen.


© Rheinwerk Verlag, Bonn 2024

```
# Beispieldatei convert2eps.sh
for pngfile in *.png; do
    # Kennung .png durch .eps ersetzen
    epsfile=${pngfile%.png}.eps
    if [ $pngfile -nt $epsfile ]; then
        echo "$pngfile -> $epsfile"
        magick "$pngfile" -quiet -background white \
           -flatten "eps2:$epsfile"
    fi
done
```

#### Beispiel: convert2eps (PowerShell-Variante)

Sie können das obige Script als Muster für diverse Anwendungsformen von `magick` verwenden – ganz egal, ob Sie Hunderte Bilder verkleinern, mit Wasserzeichen markieren oder in ein anderes Format umwandeln möchten. Falls Sie unter Windows arbeiten, erfüllt das Script im nächsten Listing denselben Zweck:

```
# Beispieldatei convert2eps.ps1
$pngfiles = Get-ChildItem *.png
foreach ($pngfile in $pngfiles) {
    $pngname = $pngfile.Name
    $epsname = $pngname -replace ".png", ".eps"
    # überspringen, wenn es schon eine aktuelle EPS-Datei gibt
    if ( (Test-Path $epsname) -and
         (Get-ChildItem $epsname).LastWriteTime
                        -gt $pngfile.LastWriteTime )
    {
        continue
    }
    Write-Host "$pngname -> $epsname"
    magick $pngname -quiet -background white -flatten `
      eps2:$epsname
}
```

#### Bildbearbeitung unter Windows ohne ImageMagick

Natürlich gibt es im .NET-Framework diverse Klassen mit elementaren Funktionen zur Bildbearbeitung, die Sie in PowerShell-Scripts anwenden können. Keine dieser Klassen kann auch nur annähernd mit den Möglichkeiten von ImageMagick mithalten. Dem steht der Vorteil gegenüber, dass keine externen Tools installiert werden müssen. Ein gutes Beispiel für die Anwendung der Klasse `System.Drawing.Bitmap` finden Sie hier:

*https://stackoverflow.com/questions/47602716*

Persönliches Exemplar für Thomas Bartholomäus

## 16.2 Fotos nach Aufnahmedatum sortieren

Ausgangspunkt dieses Beispiels ist eine große Sammlung von Fotos (JPEG-Format). Diese sollen in Verzeichnisse verschoben werden, die den Monat des Aufnahmedatums angeben. Ein im Juli 2023 aufgenommenes Foto soll also in das Verzeichnis `2023-07` verschoben werden. Wenn es dieses Verzeichnis noch nicht gibt, wird es erzeugt.

Die Lösung dieser Aufgabe gelingt durch die Auswertung der EXIF-Daten. Dabei handelt es sich um Metadaten, die von der Kamera in die JPEG-Datei eingebaut werden. Je nach Smartphone- bzw. Kamera-Modell enthalten die EXIF-Daten das Aufnahmedatum, den Aufnahmeort (GPS-Koordinaten), diverse Kamera-Parameter (Blende, Belichtungszeit), die Orientierung (Bilddrehung, z. B. *Portrait* oder *Landscape*) usw.

### ExifTool installieren und ausprobieren

Das beste Werkzeug zur Auswertung von EXIF-Daten ist *ExifTool*. Dabei handelt es sich um ein Paket, das aus einer Perl-Bibliothek und dem Kommando `exiftool` besteht. Unter Linux können Sie `exiftool` aus den Paketquellen installieren, wobei der Paketname je nach Distribution variiert. `exiftool -ver` ermittelt anschließend die Versionsnummer.

```
$ sudo apt install exiftool                (Debian, Ubuntu)
$ sudo dnf install perl-Image-ExifTool     (Fedora)
$ sudo pacman -S perl-exiftool             (ArchLinux)

$ exiftool -ver
  12.50
```

Etwas mühsamer ist die Installation unter Windows: Unter *https://exiftool.org* finden Sie eine ZIP-Datei zum Download. Sie enthält allerdings *kein* Setup-Programm. Stattdessen müssen Sie die im Archiv enthaltene Datei `exiftool(-k).exe` manuell in ein Verzeichnis kopieren, das in der `PATH`-Variablen aufgelistet wird. Am einfachsten verwenden Sie `C:\Windows`. (Das erfordert allerdings Administratorrechte.) Anschließend benennen Sie die Datei in `exiftool.exe` um.

Der Microsoft Defender blockiert den ersten Versuch, `exiftool` unter Windows auszuführen, weil der Herausgeber unbekannt ist. Klicken Sie auf Weitere Informationen und Trotzdem ausführen. In der Folge gelingt der Start ohne Probleme, auch aus dem Terminal heraus.

Auch unter macOS müssen Sie kleine Hürden überwinden: Das DMG-Image enthält eine `*.pkg`-Datei mit einem Installer. Die Ausführung wird aber von macOS blockiert, weil das Installationsprogramm nicht signiert ist. Sie müssen in den Systemein-


© Rheinwerk Verlag, Bonn 2024

stellungen das Modul DATENSCHUTZ öffnen. Dort klicken Sie auf EXIFTOOL N.N.PGK DENNOCH ÖFFNEN.

Ist die Installation einmal geschafft, ist die Anwendung des `exiftool`-Kommandos denkbar einfach: Sie übergeben einen Dateinamen und erhalten als Ergebnis eine schier endlose Liste von EXIF-Werten:

```
$ exiftool cimg2647.jpg

  ExifTool Version Number       : 12.50
  File Name                     : cimg2647.jpg
  ...
  Date/Time Original            : 2008:08:28 12:53:22
  ...
  Shutter Speed                 : 1/250
  Circle Of Confusion           : 0.005 mm
  Field Of View                 : 50.7 deg
```

Zum Sortieren der Bilder nach dem Aufnahmedatum ist nur der Eintrag *Date/Time Original* relevant. Die gleichnamige Option in Kombination mit dem gewünschten Format für das Datum (`-d`) sowie (`-s3` für *shortest*) reduziert die Ausgabe auf die gewünschte Information:

```
$ exiftool -d '%Y-%m' -s3 -DateTimeOriginal cimg2647.jpg

  2008-08
```

EXIF-Daten können eine Menge Informationen über den Ursprung der Bilder verraten. `exiftool` bietet daher die Möglichkeit, die EXIF-Daten aus einer Datei zu entfernen. Das empfohlene Kommando lautet `exiftool -All= image.jpg`. Es entfernt fast alle Metadaten, belässt aber den APP14-Adobe-Block mit Farbinformationen. Dieser Datenblock ist notwendig, damit die Farben des Bilds korrekt dargestellt werden können. Die Daten haben aber keine Privacy-Relevanz. Lesen Sie zum Löschen von EXIF-Daten auch den folgenden Beitrag im ExifTool-Forum!

*https://exiftool.org/forum/index.php?topic=13034.0*

`exiftool` kennt unzählige weitere Optionen, auf die ich hier aber nicht eingehe. Werfen Sie gegebenenfalls einen Blick in das Handbuch!

*https://exiftool.org/exiftool_pod.html*

### Bash-Beispiel

Die folgenden Zeilen zeigen eine Implementierung des Sortier-Scripts in der Bash. Der Code sollte ohne weitere Erläuterungen verständlich sein.

Persönliches Exemplar für Thomas Bartholomäus

```
# Beispieldatei sort-photos.sh
# Schleife über alle als Parameter übergebenen Dateinamen
for file in "$@"; do
    yearmonth=$(exiftool -s3 -d '%Y-%m' -DateTimeOriginal $file)
    if [ $yearmonth ]; then
        echo "$file -> $yearmonth/"
        mkdir -p $yearmonth
        mv "$file" $yearmonth/
    fi
done
```

Damit Sie das Script unkompliziert ausprobieren können, finden Sie in den Beispieldateien ein Archiv mit einigen kleinen Fotos.

```
$ ./sort-photos.sh *.jpg

  cimg2372.jpg -> 2007-12/
  cimg2374.jpg -> 2007-12/
```

Das Script ignoriert alle Dateien ohne EXIF-Daten bzw. mit fehlender Aufnahmezeit und belässt diese am bisherigen Speicherort. Um das zu testen, enthält `photos.zip` einige Sonderfälle: `z-empty.jpg` ist einfach eine leere Datei (0 Byte). `z-almost-no-metadata.jpeg` wurde mit einem iPhone aufgenommen, dann verkleinert und schließlich mit `exiftool -All=` nahezu aller EXIF-Daten beraubt. `z-gimp-metadata.jpeg` wurde als Screenshot in das Bildverarbeitungsprogramm Gimp eingefügt und dann gespeichert. Es enthält daher nur die von Gimp für neu erstellte Bilder vorgesehen Metadaten.

### PowerShell-Beispiel

Der Code für die PowerShell-Implementierung sieht nahezu gleich aus wie beim Bash-Script. Der größte Unterschied ist die verschachtelte Schleife. Sie ist erforderlich, weil die PowerShell kein *Globbing* durchführt, also Muster wie `*.jpg` nicht in eine Liste von Dateinamen umwandelt. Das müssen Sie selbst tun!

Die erste Schleife durchläuft alle Parameter. Die zweite Schleife führt für jeden Parameter `Get-Item` aus. Bei einem Dateinamen wie `myimg.jpg` liefert `Get-Item` einfach die Datei zurück. Wurde dagegen ein Muster übergeben, dann liefert `Get-Item` die Liste der passenden Dateien. Die beiden Schleifen stellen sicher, dass auch ein Aufruf von `.\sort-images.ps1 *.jpg *.jpeg` korrekt verarbeitet wird.

```
# Beispieldatei sort-images.ps1
# Schleife über alle Parameter
foreach ($arg in $args) {
    # Schleife über die Get-Item-Ergebnisse
    foreach ($file in Get-Item $arg) {
```


© Rheinwerk Verlag, Bonn 2024

```
        $yearmonth = exiftool -s3 -d '%Y-%m' `
                              -DateTimeOriginal $file
        if ($yearmonth) {
            Write-Output "$($file.Name) -> $yearmonth/"
            $targetdir = New-Item -ItemType Directory `
                                  -Path $yearmonth -Force
            Move-Item $file $targetdir
        }
    }
}
```

**Geschwindigkeitsprobleme**

Wenn es darum geht, »nur« ein paar Hundert Bilder zu sortieren, ist der hier präsentierte Lösungsansatz gut genug. Wenn es sich aber um Tausende von Bildern handelt, ist es effizienter, die in `exiftool` integrierten Scripting-Funktionen zu nutzen. Das erfordert ein genaues Handbuch-Studium; dafür muss `exiftool` nur einmal und nicht Tausende Male gestartet werden. Das ist deswegen entscheidend, weil `exiftool` auf die Programmiersprache Perl zurückgreift. Mit jedem Aufruf wird nicht ein kleines kompiliertes Tool, sondern eine komplexe Perl-Umgebung gestartet. Ein Code-Beispiel finden Sie hier:

*https://exiftool.org/mistakes.html#M3*

## 16.3 EXIF-Metadaten in SQL-Kommandos umwandeln

Das Ziel des folgenden Beispiels ist es, einen Pool von Fotos zu verarbeiten und die wichtigsten Metadaten jedes Bilds in eine Datenbank einzutragen. Gespeichert werden sollen unter anderem der Name, die Dateigröße, das Änderungsdatum und die GPS-Koordinaten. Die etwas höhere Komplexität der Aufgabenstellung legt hier den Einsatz von Python nahe.

### PyExifTool

Es existieren diverse Exif-Module für Python. Das hier vorgestellte `PyExifTool` greift auf das `exiftool`-Kommando zurück. Sie müssen also zuerst das ExifTool installieren (siehe Abschnitt 16.2, »Fotos nach Aufnahmedatum sortieren«) und dann das folgende `pip`-Kommando ausführen:

```
$ pip  install pyexiftool    (Windows, neue Linux-Distributionen)
$ pip3 install pyexiftool    (macOS, alte Linux-Distributionen)
```

Persönliches Exemplar für Thomas Bartholomäus

Das folgende Listing zeigt die Anwendung des Moduls. `ExifToolHelper` stellt die Verbindung zum `exiftool`-Kommando her. Aus Effizienzgründen bleibt diese Verbindung aufrecht, bis sie nicht mehr benötigt wird. `exiftool` muss also nicht für jedes Bild neu aufgerufen werden. Die `with`-Konstruktion kümmert sich darum, dass die Verbindung auf jeden Fall geschlossen wird, entweder beim Ende des Code-Blocks oder beim Auftreten eines Fehlers.

Nun können Sie an die Methode `get_metadata` eine Liste von Dateinamen übergeben. Die Methode antwortet mit einer Liste von Dictionaries, wobei jedes Dictionary alle für das jeweilige Bild verfügbaren Metadaten enthält. Im Beispiel-Script werden nur wenige ausgewählte EXIF-Parameter ausgelesen.

```
# Beispieldatei hello-pyexiftool.py
# Aufruf: ./hello-pyexiftool.py file1.jpg file2.jpg
import exiftool, sys
filenames = sys.argv[1:]
keys = ['SourceFile', 'EXIF:Model', 'EXIF:ExposureTime',
        'EXIF:ISO', 'EXIF:DateTimeOriginal']

with exiftool.ExifToolHelper() as exifhelper:
    # gibt Liste mit Dictionaries zurück, ein Dictionary für
    # jeden Dateinamen
    metadatalst = exifhelper.get_metadata(filenames)
    # zu jedem Foto ausgewählte EXIF-Elemente anzeigen
    for metadata in metadatalst:
        print()
        for key in keys:
            # gibt es den Key überhaupt?
            if key in metadata:
                print('%25s : %-30s' % (key, metadata[key]))
```

Die folgenden Zeilen zeigen einen Testdurchlauf für zwei Dateien:

```
$ ./exif-to-sql.py IMG_3798.jpg IMG_3799.jpg

               SourceFile : IMG_3798.jpg
               EXIF:Model : Canon IXUS 210
        EXIF:ExposureTime : 0.0008
                 EXIF:ISO : 100
    EXIF:DateTimeOriginal : 2012:11:03 08:44:32

               SourceFile : IMG_3799.jpg
               EXIF:Model : Canon IXUS 210
        EXIF:ExposureTime : 0.0008
                 EXIF:ISO : 125
    EXIF:DateTimeOriginal : 2012:11:03 08:44:39
```


© Rheinwerk Verlag, Bonn 2024

Beachten Sie, dass je nach Smartphone- bzw. Kamera-Modell unterschiedliche EXIF-Daten zur Verfügung stehen. Deswegen sollten Sie den Zugriff auf einzelne Einträge immer absichern (im obigen Code mit `if key in metadata`).

Die Anweisung `exifhelper.get_metadata(filenames)` im obigen Beispiel hat leider ihre Tücken:

- Wenn auch nur eine einzige Datei falsche EXIF-Metadaten enthält oder diese ganz fehlen, dann tritt ein `ExifToolExceptionError` auf. Der Fehler macht es unmöglich, die Ergebnisse der restlichen Dateien auszuwerten.
- Ein zweites Problem besteht darin, dass `get_metadata` alle Metadaten liest, auch solche, an den Sie gar nicht interessiert sind. Je nach Betriebssystem (Windows!) kommt es dabei unter Umständen zum Fehler *charmap: codec can't decode byte xxx in position nnn*. Offenbar verschluckt sich das `pyexiftool` an UTF8-Zeichenketten in den Metadaten. Bei meinen Tests trat der Fehler nur unter Windows auf und nur in `pyexiftool`, nicht aber beim direkten Aufruf von `exiftool`.

Insofern ist es besser, jede Bilddatei einzeln zu verarbeiten und die Auswertung mit `try-except` abzusichern. Außerdem verwenden Sie anstelle von `get_metadata` die Methode `get_tags`, die nur ausgewählte EXIF-Keys verarbeitet. Diese Vorgehensweise zeige ich Ihnen im nächsten Script `exif-to-sql.py`.

### EXIF2SQL

Die erste Teilaufgabe des Scripts `exif-to-sql.py` ist das *Globbing*, also die Auswertung von Mustern wie `*.jpg`. Unter macOS und Linux ist das überflüssig – da erzeugt die Shell aus `*.jpg` die entsprechende Liste von Dateien und gibt diese an das Script weiter.

Unter Windows landet der Parameter `*.jpg` dagegen unverändert in der Parameterliste. Wenn das Script durch die Auswertung von `platform.system()` feststellt, dass es unter Windows läuft, muss es selbst die passenden Dateinamen ermitteln. Diese Aufgabe übernimmt die Funktion `glob` aus dem gleichnamigen Modul. Die Schleife über `argv[:1]` stellt sicher, dass das Globbing auch dann korrekt funktioniert, wenn mehrere Dateimuster übergeben werden (also beispielsweise `.\exif-to-sql.py *.jpg *.jpeg`).

Die Liste `keys` wird mit den Namen aller EXIF-Parameter initialisiert, die das Script auswerten soll. Beachten Sie, dass Sie dabei die zweiteilige Nomenklatur von `pyexiftool` anwenden müssen. (`exiftool` verwendet hingegen kürzere Namen ohne vorangestellte Gruppennamen.)

Die Variable `sql` enthält das auszuführende SQL-Kommando. `inspect.cleandoc` entfernt dabei die Einrückungen der mehrzeiligen Zeichenkette. Das `INSERT`-Kommando setzt voraus, dass die Datenbank eine entsprechende Tabelle enthält. Wie Sie die

Persönliches Exemplar für Thomas Bartholomäus

Tabelle bei Bedarf einrichten können, zeige ich Ihnen in Abschnitt 19.3, »EXIF-Metadaten in einer Datenbank speichern«.

```
# Beispieldatei exif-to-sql.py
import exiftool, inspect, platform, sys
from glob import glob

# Globbing (nur unter Windows notwendig)
if platform.system() == 'Windows':
    filenames = []
    for arg in sys.argv[1:]:
        filenames.extend(glob(arg))
else:
    filenames = sys.argv[1:]

# nur diese EXIF-Parameter auswerten
keys = ['File:FileName', 'File:FileSize', 'EXIF:Orientation',
        'EXIF:DateTimeOriginal', 'EXIF:GPSLatitude',
        'EXIF:GPSLongitude', 'EXIF:GPSAltitude']

# SQL-Kommando um einen Datensatz einzufügen;
# cleandoc entfernt die Einrückung
sql = inspect.cleandoc("""
  INSERT INTO photos (name, size, orientation, datetimeoriginal,
                      latitude, longitude, altitude)
  VALUES (%s, %s, %s, %s, %s, %s, %s);""")
```

Nach diesen Vorbereitungsarbeiten stellt `ExifToolHelper` die Verbindung zum lokal installierten `exiftool` her.

Die `for`-Schleife verarbeitet alle Bilddateien. `get_tags` liest nur die durch `keys` vorgegebenen EXIF-Tags aus. Diese Schlüssel werden jetzt in der Ergebnisliste `results` gesammelt, wobei fehlende Tags durch `NULL` ersetzt werden. Beim `DateTimeOriginal`-Tag muss außerdem die Schreibweise des Datums von `2023:12:13` auf die ISO-konforme Syntax `2023-12-13` umgestellt werden. Dazu ersetzt `replace` die ersten beiden Doppelpunkte durch Strichpunkte.

Zuletzt werden die so gesammelten Ergebnisse in ein Tupel umgewandelt und mit dem Formatierungsoperator `%` in die SQL-Zeichenkette eingebaut.

```
# ... Fortsetzung, Verbindung zu exiftool herstellen
with exiftool.ExifToolHelper() as exifhelper:
    # Schleife über alle Bilddateien
    for file in filenames:
        # EXIF-Daten sammeln
        results = []
```


© Rheinwerk Verlag, Bonn 2024

```
        try:
            # get_tags liest nur ausgewählte Keys
            metadata = exifhelper.get_tags(file, keys)[0]
            for key in keys:
                if key in metadata:
                    if key == 'EXIF:DateTimeOriginal':
                        # 2023:12:13 -> 2023-12-13
                        date = str(metadata[key]).\
                                 replace(':', '-', 2)
                        results += [ "'%s'" % (date) ]
                    else:
                        results += [ "'%s'" % (metadata[key]) ]
                else:
                    results += ['NULL']
            print(sql % tuple(results))
        except Exception as e:
            print("-- skipped %s" % (file))
```

Wenn Sie das Script aufrufen und einige JPEG-Dateien übergeben, sieht die Ausgabe so aus:

```
$ ./exif-to-sql *.jpg *.jpeg

  INSERT INTO photos (name, size, orientation, datetimeoriginal,
                      latitude, longitude, altitude)
  VALUES ('IMG_3422.jpeg', '23521', '1', '2022-01-07 15:54:01',
          '47.1209027777778', '13.6318833333333', '2033.231752');
  ...
```

Sie können die Ausgabe natürlich auch in eine SQL-Datei umleiten:

```
$ ./exif-to-sql *.jpg *.jpeg > photos.sql
```

**Fehlende GPS-Daten**

Fotos, die Sie mit einem Smartphone aufnehmen, enthalten immer geografische Daten – es sei denn, Sie haben diese Funktion explizit deaktiviert. Beachten Sie aber, dass manche Fotoprogramme die Ortinformationen beim Exportieren aus Datenschutzgründen entfernen. Wenn Sie beispielsweise mit Apple Fotos arbeiten, achten Sie beim Export darauf, dass die Option INKLUSIVE • ORTSINFORMATIONEN gesetzt ist!

### Erweiterungsmöglichkeiten

Naturgemäß können Sie das Script nach Ihren eigenen Vorstellungen adaptieren, je nachdem, welche EXIF-Daten Sie in der Datenbank speichern möchten.

Persönliches Exemplar für Thomas Bartholomäus

Anstatt die SQL-Kommandos am Bildschirm auszugeben bzw. in eine Datei umzuleiten, könnte das Script auch gleich eine Verbindung zu einem Datenbank-Server herstellen und die Kommandos dort ausführen. Eine entsprechende Erweiterung des Scripts präsentiere ich Ihnen in Kapitel 19, »Datenbanken«.

Eine Datenbankverbindung einmal vorausgesetzt, könnte das Script auch noch eine Doppelgängerkontrolle durchführen, also Einträge ersetzen statt hinzufügen, wenn nur der Dateiname oder die Kombination aus Dateinamen und die Dateigröße übereinstimmt.


© Rheinwerk Verlag, Bonn 2024

# Kapitel 17
# Web Scraping

*Web Scraping* bezeichnet das gezielte Extrahieren von Daten aus einer Webseite. Die Grundidee besteht darin, den HTML-Code einer Seite zu laden und dort nach den gewünschten Daten zu suchen. Mit Web Scraping können Sie automatisiert auf Shopping-Websites den Preis eines Produkts ermitteln oder auf einer Wetter-Website die aktuelle Temperatur in Hawaii, oder auf einer News-Website die neuesten Schlagzeilen, oder auf einer Security-Site die neuesten Sicherheitslücken für eine Software, die Sie einsetzen.

In diesem Kapitel zeige ich Ihnen einige Techniken rund um das Web Scraping:

- Ich beginne mit dem Kommando `wget`: Damit können Sie ganze Websites inklusive aller verlinkten Bilder, CSS-Dateien usw. herunterladen. Technisch gesehen hat das nur am Rande mit Web Scraping zu tun und geht eher in die Richtung *Web Crawling*. (Den Begriff erkläre ich etwas weiter unten.) Der Vorteil von `wget` besteht darin, dass Sie komplexe Aufgabenstellungen mit einem einzigen Kommando lösen können, ohne überhaupt ein Script entwickeln zu müssen.
- Mit regulären Mustern können Sie den HTML-Code von Dokumenten auswerten und auf diese Weise die gewünschten Daten ermitteln. Das funktioniert aber nur in einfachen Fällen zufriedenstellend.
- Besser ist die Verwendung einer Bibliothek, die ein HTML-Dokument als Objektbaum darstellt. Der Fachausdruck lautet *Document Object Model* (DOM). Ich stelle Ihnen in diesem Kapitel die Python-Module *Beautiful Soup* und *Requests-HTML* sowie das PowerShell-Modul *PowerHTML* vor. Damit können Sie unkompliziert durch komplexe HTML-Dokumente navigieren. Der resultierende Code ist übersichtlicher und zuverlässiger als bei der Verwendung regulärer Muster.

### Voraussetzungen für dieses Kapitel

Die meisten Scripts in diesem Kapitel verwenden die Programmiersprache Python. Es wäre gut, wenn Sie mit regulären Ausdrücken umgehen können. Außerdem benötigen Sie HTML-Grundkenntnisse, die ich in diesem Buch aber nicht vermitteln kann.

Persönliches Exemplar für Thomas Bartholomäus

### Einschränkungen

Web Scraping ist ein populäres Scripting-Konzept, aber es ist mit vielen Einschränkungen verbunden:

- Der Aufbau von Webseiten ändert sich häufig. Das Web-Scraping-Script muss dann angepasst werden.
- Moderne Webseiten greifen clientseitig auf JavaScript zurück. Ein simples Auslesen des HTML-Codes versagt dann unter Umständen. (Glück haben Sie, wenn JavaScript nur für die Einblendung der allgegenwärtigen Werbung zuständig ist.)
- Es ist grundsätzlich nicht erlaubt, per Web Scraping erfasste Daten kommerziell zu nutzen. Um es an einem einfachen Beispiel zu illustrieren: Sie können nicht Wetterdaten von einer bekannten Wetter-Website sammeln, in irgendeiner Form an Ihre Kunden weitergeben und dabei womöglich noch Geld durch Werbung verdienen.

  Natürlich ist nicht *alles* verboten: Die Grundidee des Internets ist es ja, dass Informationen öffentlich geteilt werden. Eine populäre erlaubte Anwendung des Web Scrapings sind Preisvergleichsseiten wie *https://geizhals.de*. Aber auch für solche Dienste gibt es Spielregeln, z. B. was die Verwendung von Produktfotos betrifft.

  Juristisch ist das ein schwieriges Terrain. Ich belasse es hier bei dieser Warnung und konzentriere mich in diesem Kapitel auf technische Details. Sie müssen aber unbedingt die rechtlichen Voraussetzungen abklären, bevor Sie Web Scraping über den rein privaten Bereich hinaus anwenden!

Die Lösung dieser Einschränkungen sind APIs (*Application Programming Interfaces*), die einen geordneten Datenzugang geben. Während Webseiten im Hinblick auf schönes Layout optimiert sind, erlauben APIs einen effizienten Datenaustausch ohne Overhead, wobei manche APIs sogar Datenänderungen erlauben. Außerdem schaffen API-Nutzungsregeln Klarheit, welche Daten wie verwendet werden dürfen bzw. welche Kosten dabei anfallen.

Langer Rede kurzer Sinn: Wenn die Website, deren Daten Sie nutzen möchten, eine API anbietet, sollten Sie gleich weiterblättern zu Kapitel 18, »REST-APIs nutzen«.

### Web Scraping, Web Crawling und Data Mining

Zusammen mit *Web Scraping* werden häufig zwei weitere Begriffe gebraucht, die eine ähnliche Bedeutung haben. Während beim *Web Scraping* Daten einer Website ausgelesen werden, verfolgt *Web Crawling* die Links einer Website. Mit Web Crawling können Sie ein Verzeichnis aller Seiten einer Website erstellen, und beim Verfolgen von Links auf andere Seiten sogar ein Verzeichnis von großen Teilen des Internets. Suchmaschinen wie Google oder Bing basieren auf Web Crawling.


© Rheinwerk Verlag, Bonn 2024

Auch *Data Mining* geht einen Schritt weiter als Web Scraping: Jetzt geht es darum, Informationen aus verschiedenen Datenquellen und über längere Zeiträume zu kombinieren und statistisch auszuwerten. Dafür kann es viele Gründe geben, z. B. Marketing-Untersuchungen oder wissenschaftliche Aufgabenstellungen.

## 17.1 Websites mit wget herunterladen

In seiner einfachsten Anwendung können Sie mit dem Kommando `wget` eine Datei per HTTP(S) oder FTP herunterladen. Das folgende Kommando lädt ein Bild mit dem Cover dieses Buchs von meiner Website herunter und speichert es im lokalen Verzeichnis:

```
$ wget https://kofler.info/wp-content/uploads/scripting.jpg
```

`wget` ist unter Linux zumeist standardmäßig installiert; sollte das nicht der Fall sein, führt `apt/dnf install wget` zum Ziel. Unter macOS installieren Sie das Kommando am besten mit `brew install wget` (siehe *https://brew.sh*). Eine für Windows kompilierte Variante können Sie von der folgenden Webseite herunterladen. Sie müssen sich allerdings selbst darum kümmern, das Programm in einem der in `PATH` aufgezählten Verzeichnisse zu speichern.

*https://eternallybored.org/misc/wget*

`wget` wird durch Dutzende von Optionen gesteuert, die in der `man`-Seite bzw. hier dokumentiert sind:

*https://linux.die.net/man/1/wget*

### Beispiel 1: Unmittelbar verlinkte Bilder herunterladen

Ich verzichte hier auf eine enzyklopädische Beschreibung, sondern zeige Ihnen stattdessen drei typische `wget`-Anwendungen.

Das erste Kommando, das ich einmal mit den besser lesbaren Langoptionen und ein zweites Mal kompakter mit gleichwertigen Kurzoptionen angegeben habe, lädt alle Bilddateien mit den Endungen `*.jpg`, `*.jpeg`, `*.png` und `*.gif` herunter, die unmittelbar auf der Startseite von *https://example.com* verwendet werden. Das Kommando berücksichtigt aber keine Bilder, die in anderen Seiten von *https://example.com* verwendet werden.

```
$ wget --no-clobber --span-hosts --page-requisites \
       --accept jpg,jpeg,png,gif --execute robots=off example.com

$ wget -nd -H -p -A jpg,jpeg,png,gif -e robots=off example.com
```

Persönliches Exemplar für Thomas Bartholomäus

Kurz zur Bedeutung der Optionen:

- `--no-directories` erzeugt auf Ihrem Rechner keine Bildverzeichnisse, die den Ort der Bilder auf der Website repräsentieren. Vielmehr werden alle Bilder im gerade aktuellen Verzeichnis gespeichert.
- `--span-hosts` berücksichtigt auch Links auf Bilder, die sich auf anderen Websites als der Startseite (hier *https://example.com*) befinden.
- `--page-requisites` weist `wget` an, nicht nur die HTML-Startseite herunterzuladen, sondern auch alle dort verlinkten zusätzlichen Dateien (Bilder, aber auch CSS-Dateien usw.)
- `--accept` schränkt die vorige Option wieder ein: Wir interessieren uns nur für Dateien mit den angegebenen Kennungen.
- `--execute` gibt an, dass eventuell auf der Website in der Datei `robots.txt` formulierte Regeln für Suchmaschinen ignoriert werden sollen. Das ist nicht die feine Art!

Das Kommando versagt bei Bildern, deren URL nicht mit der Kennung endet, etwa wie im folgenden Beispiel:

```
https://example.com/images/img_1234.jpg?s=2b63769b
```

### Beispiel 2: Alle auf der Website verlinkten PDF-Dateien herunterladen

Das zweite Kommando ist eine Variante des ersten. Diesmal werden PDF-Dokumente anstelle von Bildern heruntergeladen. Allerdings soll nicht nur die Startseite von *https://example.com* berücksichtigt werden, sondern auch alle weiteren Seiten dieser Website, die `wget` durch die Auswertung von Links ausgehend von der Startseite innerhalb von vier Sprüngen erreichen kann.

```
$ wget --recursive -no-directories --level=4 --accept pdf \
       --execute robots=off example.com

$ wget -r -nd -l 4 -A pdf -e robots=off example.com
```

Das Kommando verwendet zwei neue Optionen:

- `--recursive` weist `wget` an, die auf der Startseite befindlichen Links weiter zu verfolgen und auch die Zielseiten zu durchsuchen. Standardmäßig berücksichtigt das Kommando dabei aber nur Seiten auf der Ausgangs-Domain. (Auf `--span-hosts` habe ich bei diesem Beispiel verzichtet. Lesen Sie dazu die nachfolgende Erläuterung.)
- `--level` gibt an, wie viele Link-Ebenen berücksichtigt werden sollen. Verwenden Sie `--level inf`, um eine Website wirklich vollständig zu durchsuchen.


© Rheinwerk Verlag, Bonn 2024

Bei der Ausführung des Kommandos brauchen Sie Geduld. Beachten Sie, dass das Kommando nur PDF-Dateien berücksichtigt, die direkt auf der angegebenen Website gespeichert sind, nicht aber PDF-Dokumente, die sich woanders befinden (z. B. in einem Amazon-Cloud-Verzeichnis).

Der Einsatz der Option `--span-hosts` ist im Gegensatz zum ersten Beispiel nicht empfehlenswert. Zwar lädt `wget` dann auch PDF-Dateien herunter, die sich auf anderen Servern befinden, aber gleichzeitig weitet das Kommando die Suche nun auch auf die Seiten anderen Server aus. Tipps, wie Sie dieses Problem beispielsweise durch eine Auflistung ausgewählter externer Hosts einschränken können, gibt StackOverflow:

*https://stackoverflow.com/questions/16780601*

> **Viele Seitenzugriffe und hohe Datenmengen**
>
> Die Option `--recursive` führt dazu, dass Sie in kurzer Zeit unzählige Dateien von der angegebenen Website herunterladen. Das verursacht je nach Netzwerkgeschwindigkeit eine relativ hohe Last für den Webserver. Manche Administratorinnen und Administratoren limitieren deswegen die Maximalanzahl der von einer IP-Adresse ausgehenden Requests. Im ungünstigsten Fall kann der Crawling-Versuch unter Missachtung der Robot-Regeln sogar als Angriff interpretiert werden.

### Beispiel 3: Lokale Kopie der Website erstellen

Das dritte Kommando kopiert alle Dateien einer Website in ein lokales Verzeichnis. Dabei werden auch Bilder und CSS-Dateien berücksichtigt. Die Verzeichnisstruktur der Website wird lokal nachgebildet. Zum Schluss (erst nachdem der Download *aller* Dateien abgeschlossen ist!) werden alle Links durch Links auf lokale Dateien ersetzt.

Im Idealfall können Sie nun die lokale Kopie der Website in einem Browser ohne Internetverbindung ansehen. In der Praxis funktioniert das nicht immer so gut. Eine wesentliche Fehlerquelle ist JavaScript-Code, der nicht wie erwartet funktioniert.

```
$ wget --mirror --convert-links --adjust-extension --no-parent \
  --page-requisites --max-redirect=3  https://example.com

$ wget -m -k -E -np -p --max-redirect=3  https://example.com
```

Die Optionen haben die folgende Bedeutung:

- `--mirror` aktiviert die Mirroring-Funktionen. Dazu zählen die rekursive Verarbeitung von Links und die unbegrenzte Verfolgung von Links.
- `--convert-links` wandelt Links innerhalb der Website in lokale Links um. Diese Aktion wird erst nach dem Ende aller Downloads durchgeführt.

Persönliches Exemplar für Thomas Bartholomäus

- `--adjust-extension` fügt heruntergeladenen HTML-Dateien ohne Endung die Kennung `.html` hinzu.
- `--no-parent` verfolgt Links nur »nach unten« (in Subverzeichnisse), aber nicht »nach oben« (außerhalb des Startpunkts).
- `--max-redirect` limitiert die Anzahl von Redirection-Anweisungen des Webservers.

Für die Ausführung des Kommandos gelten die gleichen Warnungen wie für das zweite Beispiel.

## 17.2 Web Scraping mit regulären Mustern

In Kapitel 9, »Reguläre Muster«, habe ich bereits davor gewarnt, reguläre Muster zur Verarbeitung von HTML-Dokumenten zu verwenden. Dennoch werden reguläre Muster in der Praxis oft genau dazu verwendet – ganz einfach, weil ihre Anwendung naheliegend und mit wenig Aufwand verbunden ist. Das folgende Python-Script ist dafür ein gutes Beispiel. Das Ziel des Scripts besteht darin, alle auf einer HTML-Seite mit `<img src=...>` verlinkten Bilder herunterzuladen.

Der Code ist grundsätzlich leicht zu verstehen: Das Script erzeugt zuerst das lokale Verzeichnis `tmp`, falls dieses noch nicht existiert, und lädt dann das HTML-Dokument mit der in `url` angegebenen Adresse herunter. `findall` sucht darin nach Mustern der Art `<img src="..."` und liefert als Ergebnis eine Liste der Regex-Gruppe (also jeweils die Adresse zwischen den Anführungszeichen) zurück.

Die Schleife verarbeitet alle Links und stellt relativen Adressen die Startadresse voran. Danach extrahiert es aus den oft langen URLs den eigentlichen Bildnamen. Dabei kommen `urllib.urlsplit` und `os.path.basename` zum Einsatz. Die folgenden Zeilen veranschaulichen die Anwendung der beiden Funktionen:

```
import os
from urllib import parse
url = 'https://example.com/dir1/dir2/img.jpg?p1=123&p2=534'
path = str(parse.urlsplit(url).path) # '/dir1/dir2/img.jpg'
base = os.path.basename(path)        # 'img.jpg'
```

Zuletzt versucht das Script, alle Bilder herunterzuladen und im temporären Verzeichnis zu speichern.

```
# Beispielprogramm load-images.py
from urllib import parse, request
import os, re

# temporäres Verzeichnis erzeugen
os.makedirs("tmp/", exist_ok=True)
```



© Rheinwerk Verlag, Bonn 2024

```
# HTML-Code laden
url = "https://rheinwerk-verlag.de"
with request.urlopen(url) as response:
    html = response.read().decode('utf8')
# mit regulärem Ausdruck nach <img src="..." suchen
imageUrls = re.findall(r'<img.*?src=\"(.+?)\"', html)
for imgUrl in imageUrls:
    # relative Links in absolute umwandeln
    if not imgUrl.startswith('http'):
        imgUrl =  parse.urljoin(url, imgUrl)
    # Zieldateiname extrahieren
    path = str(parse.urlsplit(imgUrl).path)
    basename = os.path.basename(path)
    destination = 'tmp/' + basename
    try:
        # Bild herunterladen und speichern
        with request.urlopen(imgUrl) as response, \
             open(destination, 'wb') as f:
            f.write(response.read())
        print(basename)
    except:
        print("skipped", basename[:30])
```

Bei einfachen HTML-Seiten funktioniert das Script wunderbar, und man könnte den Eindruck gewinnen, dass hier eigentlich nicht allzu viel schiefgehen kann. Aber weit gefehlt!

- Das Script erkennt das Muster `<img src="..."` auch in auskommentiertem HTML-Code, in Listings (`<pre>`) sowie in JavaScript-Code, wo Teile des Dateinamens dynamisch generiert werden. Die Folge sind Download-Fehler.
- Das Script versagt, wenn sich zwischen `<img` und `src="..."` ein Zeilenumbruch befindet. `findall(..., re.MULTILINE)` hilft in solchen Fällen auch nicht ohne Weiteres, weil dann in das reguläre Muster explizit `\n` eingebaut werden muss.
- Das Script kommt nicht mit Bildern zurecht, deren Inhalt direkt in den HTML-Code eingebettet ist (z. B. in der Form `<img src="data:image/png;base64 ...`).

Es bleibt dabei: Auch wenn die Versuchung groß ist, sollten Sie den Einsatz regulärer Muster beim Web Scraping vermeiden!

## 17.3 Web Scraping mit Beautiful Soup

Der »richtige« Weg zum Web Scraping ist die Verwendung eines HTML-Parsers, der aus dem HTML-Code ein *Document Object Model* bildet. Für Python stehen gleich mehrere HTML-Parser zur Auswahl, die im Hinblick auf unterschiedliche Eigenschaf-

Persönliches Exemplar für Thomas Bartholomäus

ten (Geschwindigkeit, Fehlertoleranz, einfache Anwendung) optimiert sind. Die beiden folgenden Seiten geben einen ersten Überblick:

- *https://elitedatascience.com/python-web-scraping-libraries*
- *https://html5-parser.readthedocs.io/en/latest/*

## Beautiful Soup und Requests

Die folgenden Beispiele setzen auf das Python-Modul »Beautiful Soup« in der Version 4, das in Kombination mit dem Default-HTML-Parser von Python eingesetzt wird. Sie müssen dieses Modul vorweg mit `pip` installieren:

```
$ pip  install beautifulsoup4  (Windows, Linux)
$ pip3 install beautifulsoup4  (macOS, alte Linux-Distributionen)
```

Anstelle des in den bisherigen Beispielen verwendeten Python-Standardmoduls `urllib` mit der `request`-Funktion führe ich Downloads in diesem Beispiel mit dem Modul `requests` durch. Es muss ebenfalls mit `pip` installiert werden:

```
$ pip  install requests        (Windows, Linux)
$ pip3 install requests        (macOS, alte Linux-Distributionen)
```

`requests` werde ich in Kapitel 18, »REST-APIs nutzen«, genauer vorstellen. Vorerst reicht es, wenn Sie wissen, wie Sie ein HTML-Dokument herunterladen und an Beautiful Soup weitergeben:

```
import requests
from bs4 import BeautifulSoup
response = requests.get('https://example.com')
dom = BeautifulSoup(response.content, 'html.parser')
```

## Hello, Beautiful Soup!

Beautiful Soup stellt unzählige Funktionen zur Verfügung, um innerhalb des Document Object Models an einen bestimmten Punkt zu navigieren, dort Attribute auszuwerten oder Schleifen über Subelemente auszuführen. Ich beschränke mich hier auf die Vorstellung der allerwichtigsten Grundfunktionen. Dabei enthält die Variable `x` entweder das DOM-Root-Objekt oder ein zuvor ermitteltes anderes HTML-Objekt.

- `x.attrs('attributename')` oder einfach `x['attributename']` liefert das gewünschte Attribut als String.
- `x.contents` liefert den Inhalt als String-Array (als Array deswegen, weil es mehrere Teile geben kann).
- `x.find('elementname')` ermittelt ausgehend von der aktuellen Position im DOM das erste HTML-Element vom angegebenen Typ, etwa `x.find('<table>')`.


© Rheinwerk Verlag, Bonn 2024

Beachten Sie, dass find nur *innerhalb* des aktuellen Teilbaums sucht. Wenn x ein <table>-Element ist, sucht x.find('th') nur innerhalb dieser einen Tabelle, nicht aber in anderen Tabellen des Dokuments.

- x.find('name', aname='value') sucht nach dem ersten HTML-Element, bei dem ein Attribut den gewünschten Wert hat – beispielsweise x.find('div', id='1234'). Für CSS-Klassen müssen Sie den Attributnamen class_ verwenden, also x.find('p', class_='intro').
- x.find_all(...) funktioniert wie find, liefert aber alle passenden Elemente ausgehend vom Startpunkt. Das Ergebnis kann dann in einer Schleife durchlaufen werden.
- x.select('css selector') funktioniert ähnlich wie find_all, verarbeitet als Parameter aber einen CSS-Selektor.

Unzählige weitere Funktionen sind mit vielen Anwendungsbeispielen hier dokumentiert:

*https://www.crummy.com/software/BeautifulSoup/bs4/doc/*

Das folgende Einführungsbeispiel durchsucht eine HTML-Seite nach allen img-Elementen und liefert die dazugehörigen src-Attribute:

```
# Beispieldatei hello-beautifulsoup
import requests
from bs4 import BeautifulSoup
siteurl = "https://kofler.info/"
response = requests.get(siteurl)
dom = BeautifulSoup(response.content, 'html.parser')
images = dom.find_all('img')
for img in images:
    print(img['src'])
```

### Beispiel: Liste der Top-Titel des Rheinwerk Verlags ermitteln

Als Ausgangspunkt des ersten Beispiels dient die Website des Rheinwerk Verlags (*https://rheinwerk-verlag.de*). Dort werden jeden Monat acht Bücher – Neuerscheinungen oder Bestseller – zentral präsentiert. Ziel des Scripts ist es, diese acht Titel sowie die Links zu den Detailseiten zu extrahieren.

Um sich innerhalb des HTML-Codes zu orientieren, öffnen Sie die Website in einem Webbrowser und aktivieren dort die Entwicklungs-Tools. Mit wenigen Mausklicks navigieren Sie nun durch die verschachtelte Struktur der HTML-Elemente, bis Sie den für Sie relevanten Teilbereich gefunden haben (siehe Abbildung 17.1).

Persönliches Exemplar für Thomas Bartholomäus

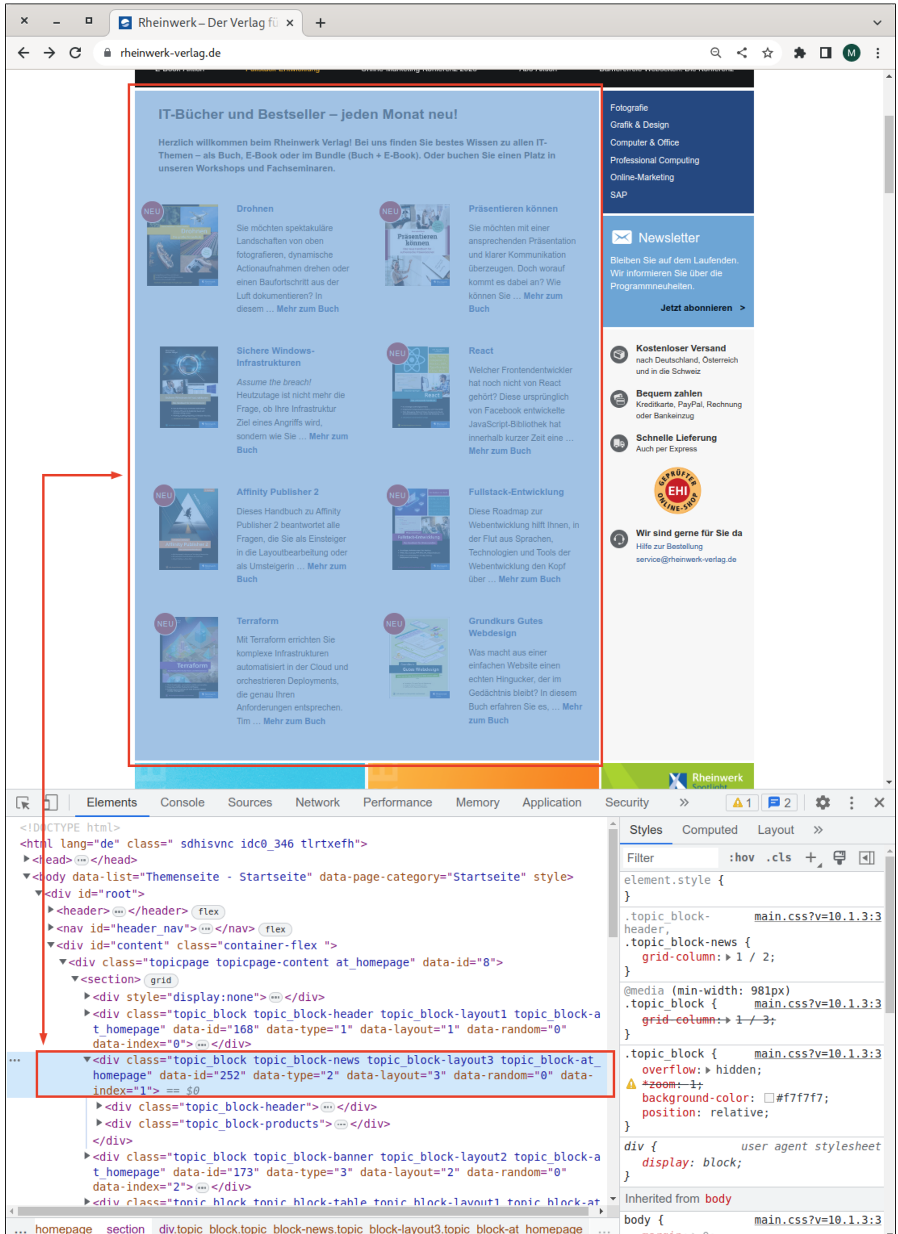



**Abbildung 17.1** Die Website des Rheinwerk Verlags in einem Webbrowser im Entwicklungsmodus

Jetzt geht es nur noch darum, die so analysierte Struktur im Python-Code nachzubilden. Das ist in diesem Fall recht einfach: Zuerst sucht `dom.find` nach dem `<div>`-Element mit der Klasse `topic_block-products`. Innerhalb dieses Elements durch-



© Rheinwerk Verlag, Bonn 2024

läuft eine Schleife über `find_all`-Ergebnisse alle `<article>`-Objekte. Bei jedem Artikel verrät das Attribut `data-title` den Buchtitel. Der erste Link (`<a>`) verweist auf die Detailseite des Buchs. Allerdings enthält das `href`-Attribut relative Links. Diese müssen mit `urljoin` vervollständigt werden.

```
# Beispieldatei rheinwerk-block-products.py
import requests
import urllib.parse
from bs4 import BeautifulSoup

# parse HTML content using BeautifulSoup
siteurl = "https://www.rheinwerk-verlag.de/"
response = requests.get(siteurl)
dom = BeautifulSoup(response.content, 'html.parser')

# find the section that contains the eight new titles
new_titles_section = dom.find('div', class_ = 'topic_block-
    products')

# extract the eight new titles and their detail page links
articles = new_titles_section.find_all('article')
for article in articles:
    print('*', article['data-title'])
    link = article.find('a')['href']
    bookurl = urllib.parse.urljoin(siteurl, link)
    print(' ', bookurl)
```

Im März 2023 produzierte ein Testlauf die folgenden Ergebnisse:

```
* Drohnen
  https://www.rheinwerk-verlag.de/drohnen-die-grosse-fotoschule/
  ...
* Grundkurs Gutes Webdesign
  https://www.rheinwerk-verlag.de/grundkurs-gutes-webdesign-...
```

**Fehler bei Relaunch**

Web Scraping verlässt sich auf den Aufbau einer Website. Sobald das Layout einer Website sich ändert, muss das Script entsprechend angepasst werden. Das gilt auch für dieses Beispiel. Es hat wunderbar funktioniert, als ich den Code im März 2023 entwickelte. Aber sollte sich der Verlag zu einem Relaunch der Seite oder auch nur zu Layout-Änderungen entscheiden, wird das Script anstelle der Titelliste voraussichtlich nur noch Fehlermeldungen liefern.

Persönliches Exemplar für Thomas Bartholomäus

## 17.4 Web Scraping mit Requests-HTML

Bei manchen Webseiten liegt die Tücke im Detail. Wenn Sie die Seite im Webbrowser betrachten, sehen Sie den fertigen HTML-Code und erwarten keine Probleme. Aber beim Versuch, die gleiche Seite als *Document Object Model* zu verarbeiten, fehlen plötzlich wichtige Bestandteile. (Sie können auch versuchen, die Seite einfach mit `curl` herunterzuladen. Dann vergleichen Sie den heruntergeladenen Code mit dem HTML-Code, den Sie im Browser sehen.)

Die Ursache für dieses merkwürdige Verhalten hat mit JavaScript zu tun: Viele moderne Websites liefern quasi ein halb fertiges HTML-Dokument aus. Teile davon werden erst im Webbrowser durch die lokale Ausführung von eingebetteten JavaScript-Codes fertiggestellt.

Bei derartigen Webseiten reicht es nicht aus, das HTML-Dokument einfach herunterzuladen; der enthalten JavaScript-Code muss durch Python auch ausgeführt werden. Damit das gelingt, müssen Sie anstelle des `requests`-Moduls das wesentlich leistungsfähigere Modul `requests-html` einsetzen. Es wird wie üblich vorweg mit `pip` installiert:

```
$ pip  install requests-html   (Windows, Linux)
$ pip3 install requests-html   (macOS, alte Linux-Distributionen)
```

Die folgenden Zeilen zeigen die Anwendung des Moduls:

```
from requests_html import HTMLSession
url = "https://www.githubstatus.com/"
session = HTMLSession()
# HTML-Code samt JavaScript herunterladen
response = session.get("https://example.com")
# eingebetteten JavaScript-Code ausführen
response.html.render()
```

Beim ersten Ausführen von `render` wird eine lokale Kopie von Chromium heruntergeladen. (Der Download ist über 100 MByte groß, dauert also eine Weile.) Chromium kümmert sich dann um die Verarbeitung des JavaScript-Codes. Erfreulicherweise funktioniert das auch unter Windows problemlos. Allerdings muss Chromium mit jedem Script-Start neu ausgeführt werden, was den `render`-Aufruf spürbar verzögert. Verwenden Sie `render` also nur, wenn dies wirklich notwendig ist!

### Beispiel: GitHub-Status auswerten

Auf der Seite *https://www.githubstatus.com* fasst die Firma GitHub den aktuellen Status ihrer Dienste zusammen. Das Ziel dieses Beispiels besteht darin, diese Statusinformationen in ein Dictionary einzulesen. Auf den ersten Blick (siehe Abbildung 17.2) sieht die Aufgabe unkompliziert aus.

 © Rheinwerk Verlag, Bonn 2024

Allerdings wird der fertige HTML-Code erst clientseitig durch JavaScript erzeugt. Das Web Scraping funktioniert erst, nachdem Sie die Seite mit dem Requests-HTML-Modul laden und mit `render` ausführen.

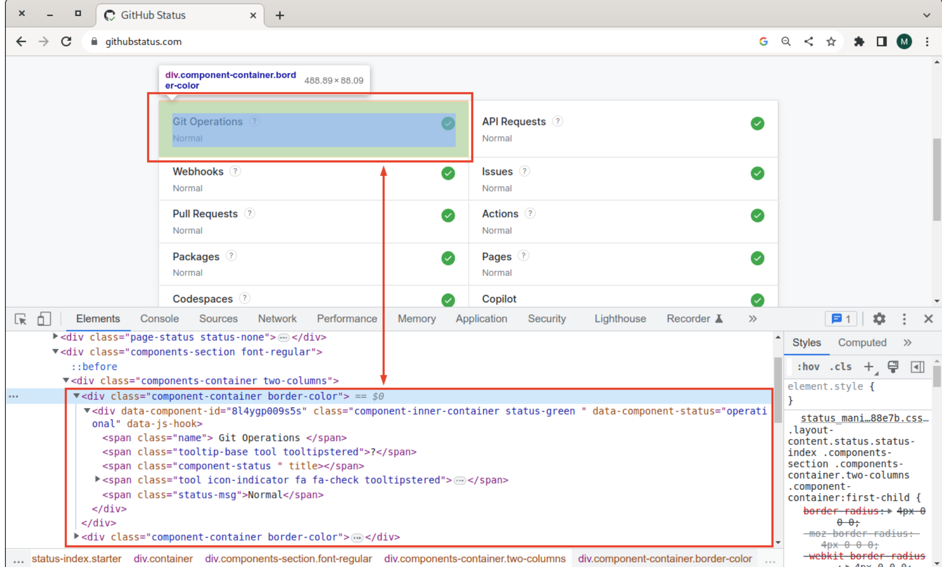



**Abbildung 17.2** GitHub-Status

Danach haben Sie die Wahl: `response.html` liefert ähnlich wie Beautiful Soup Zugriff auf ein Document Object Model. Die Methoden haben ein wenig andere Namen als bei Beautiful Soup, sind aber nach einem kurzen Blick in die Dokumentation (siehe *https://requests.readthedocs.io/projects/requests-html*) intuitiv anzuwenden. Die folgenden Zeilen zeigen die erste Variante des GitHub-Beispiels, die ausschließlich das Requests-HTML-Modul verwendet.

```
# Beispielprogramm github-status1.py
from requests_html import HTMLSession
url = "https://www.githubstatus.com/"
session = HTMLSession()
response = session.get(url)  # response object
response.html.render()

# zum Speichern der Ergebnisse
dict = {}

# Schleife über alle component-container
containers = response.html.find('div.component-container')
```

```
for container in containers:
    # unsichtbare Container überspringen
    if 'style' in container.attrs and
      container.attrs['style'] == 'display: none;':
        continue
    # die restlichen Container auswerten, Ergebnisse speichern
    name = container.find('span.name', first=True).text.strip()
    status = container.find('span.status-msg',
                            first=True).text.strip()
    dict[name] = status

# Ergebnis anzeigen
print(dict)
```

Sofern die GitHub-Dienste gerade alle fehlerfrei laufen, sollte das Script-Ergebnis so aussehen:

```
{'Git Operations': 'Normal', 'API Requests': 'Normal',
 'Webhooks': 'Normal', 'Issues': 'Normal',
 'Pull Requests': 'Normal', 'Actions': 'Normal',
 'Packages': 'Normal', 'Pages': 'Normal',
 'Codespaces': 'Normal', 'Copilot': 'Normal'}
```

Wenn Sie keine Lust haben, sich mit den DOM-Funktionen von `requests-html` auseinanderzusetzen, können Sie den HTML-Code auch mit Beautiful Soup weiterverarbeiten. Die Verdoppelung `.html.html` ist hier kein Fehler. `response.html` verweist auf das DOM, `response.html.html` liefert dagegen den dazugehörigen HTML-Code.

```
# Beispielprogramm github-status2.py
# die ersten Zeilen wie bei github-status1.py ...
# parse HTML content using BeautifulSoup
dom = BeautifulSoup(response.html.html, 'html.parser')
dict = {}

# loop through components
containers = dom.find_all('div', class_ = 'component-container')
for container in containers:
    # skip invisible containers
    if container.has_attr('style') and container['style'] == '
    display: none;':
        continue
    name = container.find('span', class_='name').contents[0].
    strip()
    status = container.find('span', class_='status-msg').contents
    [0].strip()
    dict[name] = status
```


© Rheinwerk Verlag, Bonn 2024

## 17.5 Web Scraping mit der PowerShell

Von den drei in diesem Buch behandelten Programmiersprachen ist Python für das Web Scraping am besten geeignet. Einfache Aufgabenstellungen lassen sich aber auch mit der PowerShell lösen. Das CmdLet `Invoke-WebRequest` liefert ein `BasicHtml-WebResponseObject` zurück. Es gibt Zugriff auf ausgewählte Eigenschaften der Antwort auf den Request. Damit haben Sie zwar kein vollwertiges *Document Object Model*, aber für manche Aufgabenstellungen reichen die Eigenschaften aus.

| Eigenschaft | Bedeutung |
|---|---|
| `Content` | HTML-Code (Zeichenkette) |
| `Headers` | die zurückgelieferten Header (Dictionary mit Zeichenketten) |
| `Status` | der HTML-Status-Code der Antwort (Integer) |
| `Images` | Liste aller im HTML-Code enthaltenen Bilder |
| `InputFields` | Liste aller Eingabefelder in den Formularen der HTML-Seite |
| `Links` | Liste aller enthaltenen Links |

**Tabelle 17.1** Die wichtigsten Eigenschaften des »BasicHtmlWebResponseObject«

Das folgende Script lädt alle auf einer Webseite verlinkten Bilder in ein temporäres Verzeichnis herunter. Der Download erfolgt in einer Schleife mit `Invoke-WebRequest` und der Option `-OutFile`. Um die von diesem CmdLet produzierten Statusausgaben zu unterdrücken, wird die Variable `ProgressPerference` vorübergehend auf `'SilentlyContinue'` gestellt.

```
# Beispieldatei load-images.ps1
New-Item -ItemType Directory -Force tmp | Out-Null

$url = 'https://rheinwerk-verlag.de/'
$response = Invoke-WebRequest $url

# Schleife über alle Bilder auf der Startseite der Website
foreach($image in $response.Images) {
    $src = $image.src
    if ($src.StartsWith('//')) {
        # bei protokoll-relativen Links 'https:' voranstellen
        $src = "https:" + $src
    } elseif (! $src.StartsWith('http')) {
        # bei relativen Links die Basis-URL voranstellen
        $src = $url + $src
    }
```

Persönliches Exemplar für Thomas Bartholomäus

```
    $filename = 'tmp/' + (Split-Path $image.src -Leaf)
    Write-Output ($src + " -> " +  $filename)
    $ProgressPreference = 'SilentlyContinue'  # keine Status-
    Invoke-WebRequest $src -OutFile $filename # ausgaben
    $ProgressPreference = 'Continue'
}
```

### PowerHTML-Modul

Um eine HTML-Seite in ein richtiges *Document Object Model* umzuwandeln, setzen Sie am besten das CmdLet `ConvertFrom-HTML` aus dem Modul *PowerHTML* ein. Das Modul müssen Sie vorweg installieren:

```
> Install-Module PowerHTML
```

Das Modul wurde seit 2019 nicht mehr verändert. Es hat keine mit *Requests-HTML* vergleichbaren JavaScript-Funktionen. *PowerHTML* ist zudem schlecht dokumentiert. Intern greift das Modul auf das *Html Agility Pack* (HAP) zurück:

*https://html-agility-pack.net*

Das folgende Script liest die Wikipedia-Seite zur PowerShell und extrahiert daraus alle `<h2>`-Überschriften. Da jede Überschrift aus mehreren Komponenten aufgebaut ist, wird innerhalb der Überschrift das `<span>`-Element der Klasse `mw-headline` ermittelt.

```
# Beispieldatei wikipedia.ps1
$url = 'https://en.wikipedia.org/wiki/PowerShell'
$response = Invoke-WebRequest $url
$dom = ConvertFrom-Html $response
# <div id='bodyContent'> suchen
$content = $dom.SelectSingleNode("//div[@id='bodyContent']")
# darin alle <h2>-Überschriften suchen
$headers = $content.SelectNodes("//h2")
foreach ($header in $headers) {
    $headline =
      $header.SelectSingleNode("span[@class='mw-headline']")
    Write-Output $headline.InnerText
}
```

#### PowerHTML-Alternativen

Das Modul PowerHTML funktioniert plattformübergreifend, also auch in PowerShell-Scripts unter Linux. Eine Alternative besteht darin, die `HTMLFile`-Klasse zu nutzen, die aber nur unter Windows zur Verfügung steht und die sich selbst dort in PowerShell-Scripts nur mühsam anwenden lässt. Ein Beispiel finden Sie auf StackOverflow:

*https://stackoverflow.com/a/71855426*


© Rheinwerk Verlag, Bonn 2024

# Kapitel 18
# REST-APIs nutzen

Schon in der Einleitung von Kapitel 17 habe ich klargemacht, dass Web Scraping nur eine Notlösung ist. Der Informationsaustausch zwischen Websites bzw. mit externen Diensten klappt wesentlich zuverlässiger, wenn dazu ein *Application Programming Interface* genutzt werden kann. Durchgesetzt haben sich in den letzten Jahrzehnten sogenannte REST-APIs. Der *Representational State Transfer* beruht auf einem Datenaustausch über etablierte HTTP-Requests wie Get, Put oder Post. Die Daten werden zumeist im JSON-Format oder, zum Glück viel seltener, als XML übertragen. Manche APIs erfordern zudem eine Authentifizierung, wobei diverse Verfahren zur Auswahl stehen (Basic, Bearer, Digest, OAuth etc.).

In diesem Kapitel demonstriere ich Ihnen die Nutzung von REST-APIs und zeige Ihnen einige elementare Programmiertechniken.

### Voraussetzungen für dieses Kapitel

Die Beispiel-Scripts verwenden alle drei in diesem Buch behandelten Programmiersprachen. Eine wichtige Voraussetzung für die Lektüre ist zudem Kapitel 10, »JSON XML und INI«: JSON ist als REST-Datenformat allgegenwärtig.

Sie werden von diesem Kapitel nur profitieren, wenn Sie über ein Basiswissen bezüglich HTTP und REST verfügen. Wenn Sie typische Request-Typen wie Get, Put oder Post nicht kennen oder noch nie von der *Bearer Authentication* gehört haben, sollten Sie sich zuerst ein wenig einlesen.

## Werkzeuge

Wie ich Ihnen gleich zeigen werde, reichen die Kommandos `curl` oder `wget` schon aus, um REST-APIs zu testen. Wirklich Spaß macht das aber nicht. Sie sollten sich zur Analyse fremder APIs sowie zur Fehlersuche unbedingt eine grafische Benutzeroberfläche zur Ausführung von HTTP-Requests installieren. Meine Empfehlung lautet *Postman* (*https://postman.com*), aber auch *Swagger UI*, der *Insomnia REST Client* oder *RapidAPI* (ehemals *Paw*, nur für macOS) sind gute Alternativen. Es existieren auch diverse Browser-Erweiterungen, um REST-APIs im Webbrowser auszuprobieren.

Persönliches Exemplar für Thomas Bartholomäus

### Beispiel-APIs zum Ausprobieren

Um den Umgang mit REST-APIs zu üben, brauchen Sie eine Spielwiese. Auf den folgenden Seiten finden Sie APIs, die Sie (teilweise nur in beschränkten Ausmaß) kostenlos ausprobieren können:

- *https://httpbin.org*
- *https://restcountries.com*
- *https://www.postman.com/explore*
- *https://apipheny.io/free-api*
- *https://github.com/public-apis/public-apis*

Wenn Sie unkompliziert eine reale API simulieren möchten, können Sie dies mit *WireMock* tun (*https://wiremock.org*). WireMock ist an sich ein Open-Source-Projekt. Die Nutzung des zugehörigen Cloud-Dienstes ist aber nur mit großen Einschränkungen kostenlos und richtet sich eher an große Firmen, die abseits des Produktivbetriebs APIs testen möchten.

### REST-APIs selbst implementieren

Mit der Bash oder der PowerShell ist die Realisierung eigener REST-APIs nahezu unmöglich. Anders sieht die Sache für Python aus: Mit *Flask* und *Django* gibt es zwei weit verbreitete Frameworks zur Server-Programmierung. Mögliche Alternativen sind *FastAPI*, *Pyramid* oder *Bottle*. Allerdings liegen sowohl der API-Entwurf als auch deren Implementierung weit außerhalb der Reichweite dieses Buchs mit einem Fokus auf typischerweise kurze Scripts.

## 18.1 curl und wget

Vor allem Linux-Nutzerinnen und -Nutzern sind die Kommandos `curl` und `wget` zumeist vertraut. Sie werden meist zum Download von Dateien im Terminal eingesetzt. `curl` zeigt standardmäßig den Inhalt der heruntergeladenen Datei an; die Ausgabe kann aber mit `-o` (»oh«, nicht null) in eine Datei gespeichert werden. `wget` speichert die Datei standardmäßig im lokalen Verzeichnis. Wenn der Dateiname nicht aus der URL hervorgeht, wird der von der Website vorgegebene Dateiname der Startseite verwendet (oft `index.html`). Wenn Sie die Datei in der Standardausgabe anzeigen möchten, verwenden Sie `-O -` (»Oh«, gefolgt vom gewünschten Dateinamen oder - für die Standardausgabe).

```
$ curl https://example.com/some/file.txt -o file.txt
$ wget https://example.com/some/file.txt
```


© Rheinwerk Verlag, Bonn 2024

### Installation

Unter Linux und macOS steht `curl` standardmäßig zur Verfügung. `wget` kann mit den üblichen Paketverwaltungswerkzeugen installiert werden, also `apt/dnf/brew install wget`.

`curl` für Windows finden Sie unter *https://curl.se/windows* zum Download, `wget` unter *https://eternallybored.org/misc/wget*.

## curl

`curl` und `wget` führen standardmäßig einen Get-Request aus und verzichten auf eine Authentifizierung. Beide Kommandos können aber auch lokale Daten hochladen (Put/Post), unterstützen diverse Authentifizierungsarten etc. Ich konzentriere mich im Folgenden auf `curl` und fasse hier kurz die wichtigsten Optionen im REST-Kontext zusammen.

- `-X` oder `--request GET/POST/PUT...` legt den gewünschten Request-Typ fest (standardmäßig Get).
- `-H` oder `--header 'txt'` übergibt den gewünschten HTTP-Header – z. B. `-H "Content-Type: application/json"`. Die Option kann wiederholt werden, wenn mehrere Header erforderlich sind. `-H @file.txt` liest die Header aus einer Datei.
- `-d` oder `--data 'data'` sendet die angegebenen Daten mit dem Request. Zu `--data` gibt es die Alternativen `--data-binary`, `--data-raw` und `--data-urlencode`. In jedem Fall können die Daten auch in der Form `@file` aus der angegebenen Datei gelesen werden.
- `-u` oder `--user name:pw` übergibt einen Benutzernamen und das dazugehörende Passwort.
- `-n` bzw. `--netrc` weist `curl` an, die Login-Daten aus der Datei `.netrc` zu lesen. Mit `--netrc-file file.txt` können Sie alternativ einen anderen Ort für diese Datei angeben. Beide Varianten sind sicherer als die Passwort-Übergabe durch `-u`.
- `--oauth2-bearer 'token'` sendet mit dem Request das angegebene Bearer-Token mit.
- `-L` oder `--location` erlaubt `curl`, einen HTTP-Redirect auszuführen. Wenn der Server also angibt, dass die Seite jetzt an einem anderen Ort ist, wird diese Umleitung befolgt.
- `-k` oder `--insecure` verzichtet auf die Überprüfung der SSH- oder HTTPS-Zertifikate. Das ist unsicher!
- `-v` oder `--verbose` zeigt während der Ausführung des Request umfassende Debugging-Informationen an.

Persönliches Exemplar für Thomas Bartholomäus

- `-I` oder `--head` liefert als Ergebnis nur den Header der Antwort. Das ist z. B. praktisch, wenn Sie nur am HTTP-Status-Code interessiert sind, nicht an den sonstigen Daten der Antwort.

Die folgenden Beispiele beziehen sich auf *https://httpbin.org*. Diese Website hat den Zweck, REST-Werkzeuge oder -Scripts zu testen. Beispielsweise liefert ein Get-Request der Adresse `/headers` ein JSON-Dokument mit allen übertragenen Headern zurück:

```
$ curl https://httpbin.org/headers -H 'myheader1: lorem' \
                                   -H 'myheader2: ipsum'

  {
    "headers": {
      "Accept": "*/*",
      "Host": "httpbin.org",
      "Myheader1": "lorem",
      "Myheader2": "ipsum",
      "User-Agent": "curl/8.0.1",
      "X-Amzn-Trace-Id": "Root=1-64228236-74fe..."
    }
  }
```

Im zweiten Beispiel erzeugen Sie zuerst die winzige, lokale JSON-Datei `mydata.json`. Diese Datei übertragen Sie mit Put an die Adresse `/put`. Die Website antwortet mit den gesendeten Daten, sodass Sie eine Kontrolle haben, ob alles funktioniert:

```
$ cat > mydata.json << EOF
{
    "firstName": "John",
    "lastName": "Doe"
}
EOF

$ curl -X PUT https://httpbin.org/put?para=123 -d @mydata.json  \
       -H "Content-Type: application/json"

  {
    "args": {
      "para": "123"
    },
    "data": "{ \"firstName\": \"John\",  \"lastName\": \"Doe\"}",
    "json": {
      "firstName": "John",
      "lastName": "Doe"
    }, ...
  }
```


© Rheinwerk Verlag, Bonn 2024

Zur Fehlersuche ist es oft hilfreich, Requests mit einer grafischen Benutzeroberfläche zusammenzustellen (siehe Abbildung 18.1).

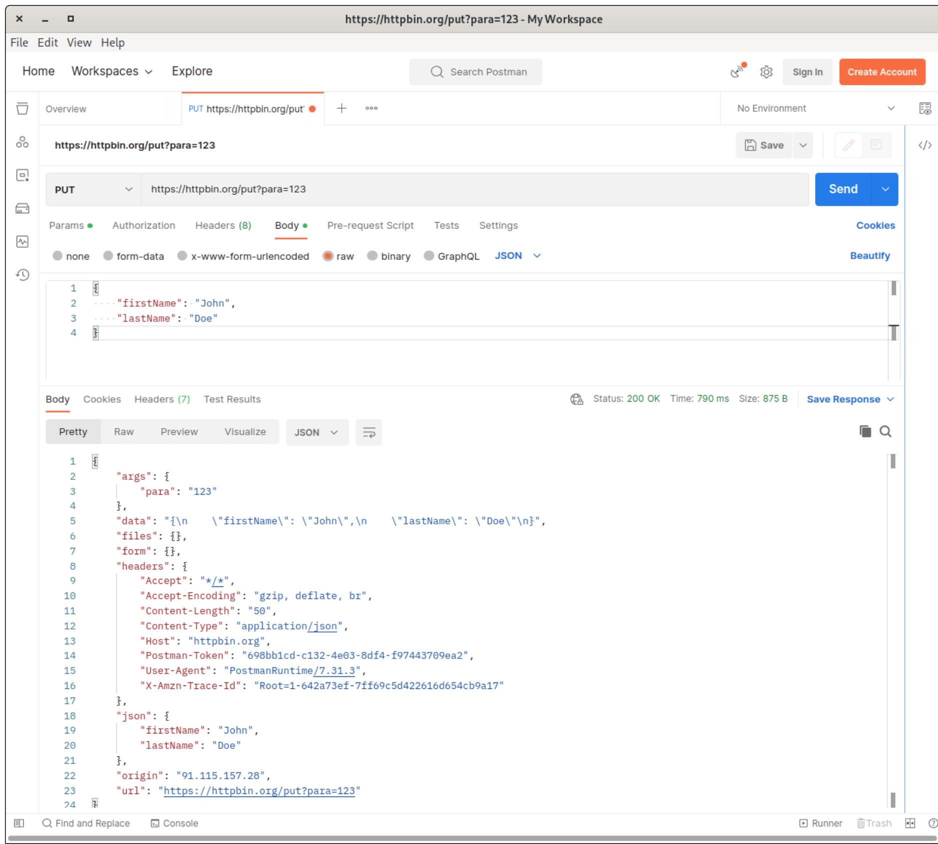



**Abbildung 18.1** Mit »Postman« ausgeführter Put-Request

Im letzten Beispiel testen Sie die *Basic Authentication*. Dabei geben Sie mit `/basic-auth/name/pw` die Login-Daten vor. Die Website überprüft, ob diese Daten mit denen von `-u 'name:pw'` übereinstimmen. Mit `-I` wird der Status-Code der Antwort angezeigt, im ersten Fall 200 (*successful authentication*), im zweiten Fall 401 (*unsuccessful authentication*).

```
$ curl -X GET "https://httpbin.org/basic-auth/maria/topsecret" \
       -u 'maria:topsecret' -I

  HTTP/2 200

$ curl -X GET "https://httpbin.org/basic-auth/maria/topsecret" \
       -u 'maria:wrong' -I

  HTTP/2 401
```

Persönliches Exemplar für Thomas Bartholomäus

Eine Referenz aller *httpbin.org*-Funktionen sowie weitere `curl`-Beispiele finden Sie hier:

*https://httpbin.org*
*https://www.naleid.com/2017/11/26/using-http-apis-on-the-command-line-1-curl.html*

## wget

`wget` unterstützt grundsätzlich die gleichen Funktionen wie `curl`, allerdings haben die Optionen andere Namen. Es ist eine Frage der persönlichen Präferenz, ob Sie lieber mit `wget` oder `curl` arbeiten. Ich verzichte an dieser Stelle auf eine Beschreibung der Optionen (lesen Sie bei Bedarf `man wget`!) und belasse es bei zwei Beispielen. Das erste Kommando ist ein simpler Get-Request. *httpbin.org* antwortet mit einer UUID, die in ein JSON-Dokument verpackt ist. Die Rückgabe wird wegen `-O -` direkt im Terminal angezeigt. Dafür verhindert `-q` (*quiet*) die Ausgabe von Statusinformationen.

```
$ wget https://httpbin.org/uuid -O - -q

  {
    "uuid": "9672f6c6-a27e-4059-8d87-35f0dc171bc7"
  }
```

Das zweite Beispiel testet die Übertragung von JSON-Daten mit einem Post-Request:

```
$ wget --header "Content-Type: application/json" \
       --post-data '{"firstName": "John", "lastName": "Doe"}' \
       https://httpbin.org/post -O -

  { ...
    "json": {
      "firstName": "John",
      "lastName": "Doe"
    }, ...
  }
```

## REST-APIs in Bash-Scripts nutzen

Die Bash ist nicht die optimale Scripting-Sprache für REST-Anwendungen. Mit Python oder der PowerShell können Sie Ihre Scripts besser, effizienter und zuverlässiger formulieren. Für einfache Aufgabenstellungen reichen die Kombination aus `curl`/`wget` plus `jq` (siehe Kapitel 10, »JSON XML und INI«) aber aus.

Das folgende Beispiel greift auf die Website *https://ipinfo.io* zurück. Diese antwortet auf Get-Requests mit einem JSON-Dokument, das außer der ursprünglichen


© Rheinwerk Verlag, Bonn 2024

IP-Adresse auch eine ungefähre geografische Zuordnung enthält. (Verlassen Sie sich nicht auf die Genauigkeit der Antwort!)

```
$ curl https://ipinfo.io

  {
    "ip": "91.115.157.28",
    "hostname": "91-115-157-28.adsl.highway.telekom.at",
    "city": "Graz",
    "region": "Styria",
    "country": "AT",
    "loc": "47.0667,15.4500",
    "postal": "8041",
    "timezone": "Europe/Vienna", ...
  }
```

Das Script `get-location.sh` extrahiert mit `jq` aus der Antwort die Stadt, die Region und das Land:

```
# Beispieldatei get-location.sh
json=$(curl https://ipinfo.io -s)
for key in city region country; do
    echo -n "$key: "
    echo $json | jq .$key
done
```

Bei mir zu Hause liefert das Script dann die folgende Ausgabe:

```
city: "Graz"
region: "Styria"
country: "AT"
```

## 18.2 REST-APIs in der PowerShell nutzen

Die PowerShell stellt zwei CmdLets zur Auswahl, um HTTP-Requests durchzuführen:

- `Invoke-WebRequest` haben Sie schon im vorigen Kapitel kennengelernt (siehe Abschnitt 17.5, »Web Scraping mit der PowerShell«). Das CmdLet kann auch für REST-Methoden verwendet werden.
- `Invoke-RestMethod` hat Vor- und Nachteile im Vergleich zu `Invoke-WebRequest`: Praktisch ist, dass das CmdLet Antworten in einem flachen (nicht verschachtelten) JSON-Format direkt in ein `PSCustomObject` umwandelt. Wenn das Web-Service Antworten im XML-Format liefert, gibt `Invoke-WebRequest` direkt ein Objekt vom Typ `XmlDocument` zurück. Dieser relativ intelligente Umgang mit typischen REST-Datentypen vereinfacht die weitere Auswertung.

Persönliches Exemplar für Thomas Bartholomäus

Ärgerlich ist dagegen, dass das CmdLet den HTTP-Status-Code nicht zurückgibt. Wenn der Request erfolgreich (Code 2xx) ist, erhalten Sie Ergebnisdaten, andernfalls tritt ein Fehler auf. In manchen REST-Anwendungen ist dieser Ansatz zu minimalistisch.

Aus meiner Sicht überwiegen bei `Invoke-RestMethod` die Nachteile. In den meisten Anwendungen ziehe ich `Invoke-WebRequest` vor und wandle die `Content`-Zeichenkette dann in ein JSON-Objekt um:

```
> $json = (Invoke-WebRequest `
     'https://httpbin.org/headers').Content | ConvertFrom-Json

> $json.headers.host
  httpbin.org
```

Bei meinen Tests lieferten manche REST-Services HTML- anstelle von JSON-Code auf Requests, die ich mit `Invoke-RestMethode` oder `Invoke-WebRequest` durchgeführt habe. Abhilfe schafft dann die Angabe des User-Agents `curl` im Header, damit das Service nicht irrtümlich annimmt, dass der User-Agent ein Webbrowser ist:

```
> Invoke-RestMethod 'https://ipinfo.io'

  <!DOCTYPE html><html lang="en"> ... (seitenlanger HTML-Code)

> Invoke-RestMethod 'https://ipinfo.io' `
      -Headers @{'User-Agent' = 'curl'}

  ip       : 91.115.157.28
  hostname : 91-115-157-28.adsl.highway.telekom.at
  city     : Graz
  region   : Styria
  country  : AT
  ...
```

## Optionen

Die beiden `Invoke`-CmdLets liefern nicht nur unterschiedliche Ergebnisdatentypen, auch diverse Spezialoptionen weichen voneinander ab. Glücklicherweise stimmen zumindest die Optionen für die Grundeinstellungen überein:

- `-Uri` übergibt die Adresse. Diese Option muss nicht angegeben werden, d. h., die Adresse kann auch direkt an das CmdLet übergeben werden.
- `-Method` legt den gewünschten Request fest. Zulässige Einstellungen sind `Get`, `Put`, `Post` usw.
- `-Authentication` wählt das gewünschte Authentifizierungsverfahren (`None`, `Basic`, `Bearer` oder `OAuth`).


© Rheinwerk Verlag, Bonn 2024

- `-Credential` übergibt Daten für Basic-Authentifizierung an den Request. Das Paar aus Loginname und Passwort muss als `PSCredential`-Objekt übergeben werden. Ein derartiges Objekt können Sie interaktiv mit `Get-Credential` initialisieren. Alternativ können Sie ein derartiges Objekt auch per Code initialisieren:

```
$password = ConvertTo-SecureString 'topSecret' -AsPlainText `
                                               -Force
$credential = New-Object `
   System.Management.Automation.PSCredential('loginname',
                                             $password)
```

- `-Token` übergibt eine Zeichenkette für die `Bearer`- und `OAuth`-Authentifizierung an das CmdLet.
- `-Body` übergibt die an den Webserver zu übertragenden Daten (Upload). Mit `-ContentType` können Sie – soweit notwendig – das Format und den Zeichensatz dieser Daten festlegen.
- `-Header` erwartet eine Hashtable mit den Header-Einstellungen.

Die folgenden Zeilen zeigen, wie Sie mit `Invoke-WebRequest` eigene Daten an einen Server übertragen.

```
$data = @{ firstName = 'John'; lastName = 'Doe'}
$jsondata = $data | ConvertTo-Json
$url = 'https://httpbin.org/put'
$response = Invoke-WebRequest -Method Put $url -Body $jsondata
$response.Content

  {
    "headers": {
      "Content-Length": "46",
      "Host": "httpbin.org",
      "User-Agent": "Mozilla/5.0 (Linux; ...) PowerShell/7.3.2",
      "X-Amzn-Trace-Id": "Root=1-6426e776-337f635051c3c6753..."
    },
    "json": {
      "firstName": "John",
      "lastName": "Doe"
    }, ...
  }
```

Da bei einem Request-Aufruf typischerweise viele Optionen und Parameter erforderlich sind, bietet es sich an, die Daten zuerst in einer Hashtable zu sammeln und per *Splatting* (siehe Abschnitt 4.3, »Aufruf von CmdLets und Funktionen«) an das CmdLet zu übergeben:

```
$data = @{ firstName = 'John'; lastName = 'Doe'}
$para = @{ Uri = 'https://httpbin.org/put';
           Method = 'Put';
           Body = $data | ConvertTo-Json}
$response = Invoke-WebRequest @para
```

## 18.3 Beispiel: Aktuelles Wetter ermitteln

Das folgende Script kombiniert zwei REST-APIs:

- Zuerst wird das schon bekannte Geolocation-Service *https://ipinfo.io* verwendet, um den mutmaßlichen Ort für die aktuelle IP-Adresse zu ermitteln. Das funktioniert nicht immer ganz exakt und kann komplett schiefgehen, wenn Sie eine VPN-Verbindung verwenden. Aber in den meisten Fällen stimmen die zurückgegebenen Koordinaten einigermaßen gut.
- Im zweiten Schritt wird für den ermittelten Ort das aktuelle Wetter ermittelt. Die API unter *https://api.weatherapi.com* steht mit gewissen Einschränkungen kostenlos zur Verfügung. Sie müssen sich allerdings einen API-Key besorgen. (Die Angabe Ihrer E-Mail-Adresse reicht aus, Sie müssen weder eine Kreditkartennummer noch sonstige persönliche Daten angeben.)

Für den ersten Schritt habe ich `Invoke-RestMethod` verwendet, weil sich damit die von *ipinfo.io* übermittelten Daten am bequemsten auswerten lassen. Die deutlich komplexere JSON-Struktur von *weatherapi.com* habe ich dagegen explizit mit `ConvertFrom-Json` in PowerShell-Datenstrukturen umgewandelt.

```
# Beispieldatei get-weather.ps1
# Verwenden Sie einen eigenen Key!
$key = "7901..."

# ermittelt aus der aktuellen IP-Adresse Längen- und Breitengrad
$location = (Invoke-RestMethod "https://ipinfo.io" `
          -Headers @{'User-Agent' = 'curl'}).loc

# ermittelt das Wetter für diesen Ort
$base = "https://api.weatherapi.com/v1/current.json"
$url = "${base}?key=$key&q=$location&aqi=no"
$response = Invoke-WebRequest $url
$content = $response.Content | ConvertFrom-Json
$city = $content.location.name
$temp = $content.current.temp_c
$text = $content.current.condition.text
Write-Output "Current weather in ${city}: $text at $temp °C"
```


© Rheinwerk Verlag, Bonn 2024

Wenn Sie das Script ausführen, sieht das Ergebnis so ähnlich wie die folgende Zeile aus:

```
Current weather in Graz: Partly cloudy at 5 °C
```

Ein weiteres Beispiel für die Anwendung von `Invoke-RestMethod` finden Sie in Abschnitt 11.4, »Beispiel: Währungskurse speichern«. Bei diesem Beispiel liefert das CmdLet direkt ein XML-Objekt, das in der Folge ausgewertet wird.

## 18.4 REST-APIs in Python nutzen

Dreh- und Angelpunkt von REST-Scripts in Python ist das Modul `requests`, das ich Ihnen in Kapitel 17, »Web Scraping«, schon kurz vorgestellt habe. Das Modul hat den Nachteil, dass es im Gegensatz zum `urllib`-Modul mit `pip` extra installiert werden muss:

```
$ pip  install requests        (Windows, Linux)
$ pip3 install requests        (macOS, alte Linux-Distributionen)
```

Dafür können mit der gleichnamigen Funktion ohne syntaktische Verrenkungen alle erdenklichen Requests ausgeführt werden. Die folgenden Zeilen zeigen einen einfachen Get-Request. Das binäre Ergebnis (`response.content`) wird mit `decode` in eine UTF-8-Zeichenkette umgewandelt und ausgegeben:

```
import requests
# Get-Request
response = requests.get('https://httpbin.org/get?q=123')
print(response.content.decode('utf-8'))
# {
#   "args": {
#     "q": "123"
#   }, ...
```

Wenn die REST-API ein JSON-Dokument zurückgibt, machen Sie daraus mit der integrierten Methode `json` einen Python-Objektbaum. (Sie brauchen also nicht auf das `json`-Modul zurückgreifen.) Den HTTP-Status der Webserver-Antwort ermitteln Sie mit der Eigenschaft `status_code`.

```
data = response.json()
print(data)
# {'args': {'q': '123'}, 'headers': {'Accept': '*/*', ...
print(response.status_code)
# 200
```

Persönliches Exemplar für Thomas Bartholomäus

Je nachdem, welche Art von Request Sie ausführen möchten, verwenden Sie anstelle von `requests.get` entsprechend `put`, `patch`, `delete` usw. An sämtliche Methoden können Sie diverse optionale Parameter übergeben:

- `header` erwartet ein Dictionary mit den Header-Daten.
- An `data` können Sie wahlweise ein Dictionary mit Parametern (z. B. für einen Post-Request) oder eine Zeichenkette mit sonstigen Daten übergeben.
- Alternativ können Sie mit dem Parameter `json` ein Dictionary übergeben. Sein Inhalt wird im JSON-Format übertragen.
- Mit `files` können Sie lokale Dateien zum Server hochladen.

Die folgenden Zeilen zeigen einen Put-Request, bei dem Daten im JSON-Format zum Server übertragen werden:

```
data = {'firstName': 'John', 'lastName': 'Doe'}
response = requests.put('https://httpbin.org/put', json=data)
```

Wenn Sie bei einem Request eine Basic-Authentifizierung durchführen wollen, übergeben Sie den Benutzernamen und das Passwort an die `auth`-Option:

```
url = 'https://httpbin.org/basic-auth/maria/topsecret'
response = requests.get(url, auth=('maria', 'topsecret'))
print("Status Code", response.status_code)
```

Wenn Sie alternativ eine einfache Bearer-Authentifizierung wünschen, übergeben Sie das Token am besten als Header:

```
token = "234f1523werf"
headers = {"Authorization": "Bearer %s" % (token)}
url = 'https://httpbin.org/bearer'
response = requests.get(url, headers=headers)
print("Status Code", response.status_code)
```

Unzählige weitere Features des `requests`-Moduls sind hier dokumentiert:

*https://requests.readthedocs.io*

## 18.5 Beispiel: Strompreise ermitteln und grafisch darstellen

Der Ausbau erneuerbarer Energien führt dazu, dass die Strompreise heute stärker schwanken als früher. Je mehr Wind bläst bzw. desto stärker die Sonne scheint, desto mehr Strom steht zur Verfügung – manchmal mehr, als gerade gebraucht wird. Zu solchen Zeitpunkten ist der Strom auf den Strombörsen sehr billig.

Unter diesem Hintergrund etablieren sich momentan neue Stromanbieter mit stündlich wechselnden Preisen (Awattar, Tibber etc.). Derart dynamische Tarife motivieren


© Rheinwerk Verlag, Bonn 2024

dazu, energieintensive Aktionen (E-Auto aufladen, Wäschetrockner starten) möglichst dann durchzuführen, wenn viel Strom zur Verfügung steht. Davon profitieren sowohl die Kundinnen und Kunden als auch die Energieversorger.

Das Beispiel dieses Abschnitts bezieht sich auf die Firma *Awattar*. Diese Firma bietet eine API an, die über die Strompreise für die kommenden Stunden informiert (maximal für 24 Stunden). Als Datengrundlage dient die europäische Strombörse EPEX SPOT, wo diese Preise aufgrund von Angebot und Nachfrage errechnet werden. Die Preise für den nächsten Tag stehen ab ca. 14:00 Uhr zur Verfügung.

Das im Folgenden präsentierte Script macht aus diesen Daten ein Diagramm (siehe Abbildung 18.2). Beachten Sie, dass zu den im Diagramm dargestellten Preisen für die reinen Energiekosten noch eine monatliche Grundgebühr, Leitungskosten, Stromzählerpauschalen etc. hinzukommen.

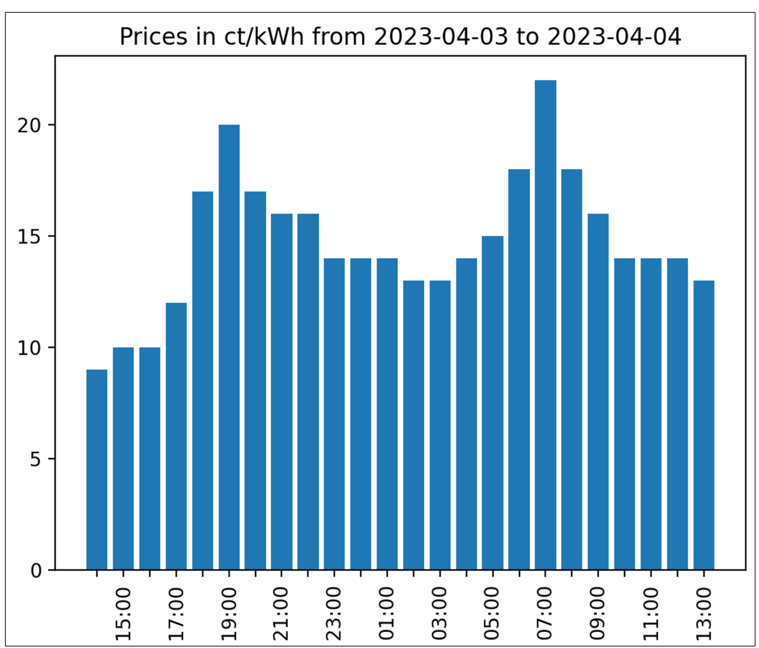



**Abbildung 18.2** Das Diagramm zeigt die Strompreise für die kommenden 24 Stunden (reine Energiepreise inkl. MWSt für Österreich Anfang April 2023).

## Awattar-API

Die API ist aktuell kostenlos zugänglich, bei allerdings nur maximal 100 Zugriffen pro Tag:

*https://api.awattar.at/v1/marketdata* (Preise für Österreich)
*https://api.awattar.de/v1/marketdata* (Preise für Deutschland)

Das JSON-Ergebnis ist für Menschen leider nicht besonders gut lesbar. Die Zeitspannen, während derer ein bestimmter Preis gilt, werden in Epoch-Millisekunden angegeben. Diese in Unix übliche Zeitangabe rechnet die Sekunden bzw. Millisekunden seit dem 1. Januar 1970 UTC. Preise werden in Euro pro Megawattstunde angegeben, allerdings ohne Mehrwertsteuer und ohne den von Awattar verrechneten Aufpreis von 3 Prozent.


```
{
   "object":"list",
   "data":[
      {
         "start_timestamp":1680508800000,
         "end_timestamp":1680512400000,
         "marketprice":107.11,
         "unit":"Eur/MWh"
      },
      {
         "start_timestamp":1680512400000,
         "end_timestamp":1680516000000,
         "marketprice":93.08,
         "unit":"Eur/MWh"
      },
      ...
```


### Auswertung der Daten

Nach der Initialisierung diverser Variablen führt das Script einen API-Aufruf durch und wertet die gesammelten Daten dann in einer Schleife aus. Die Umrechnung der Epoch-Zeitangaben in »gewöhnliche« DateTime-Objekte gelingt mit `fromtimestamp` mühelos. Zu den Nettopreisen werden die Mehrwertsteuer und der Awattar-Aufschlag hinzugerechnet.

```
# Beispieldatei electricity-prices.py
import locale, requests
from datetime import datetime
import matplotlib.pyplot as plt
# Sprache laut Systemeinstellung verwenden
locale.setlocale(locale.LC_ALL, '')
# Grundeinstellungen
surcharge = 0.03   # Awattar-Aufpreis auf EXPO SPOT
vat = 0.20         # 20% MWSt (Österreich)
url = 'https://api.awattar.at/v1/marketdata'
# alternativ für Deutschland
# vat = 0.19
# url = 'https://api.awattar.de/v1/marketdata'
```


© Rheinwerk Verlag, Bonn 2024

```
# API-Aufruf
response = requests.get(url)
jsondata = response.json()

# Daten sammeln
hours = []          # Liste für Uhrzeiten
prices = []         # Liste für Preise
dateStart = None    # Startdatum der Zeitspanne

# Daten auswerten und in Textform darstellen
for price in jsondata['data']:
    startDt = \
      datetime.fromtimestamp(price['start_timestamp'] / 1000)
    hour = startDt.strftime('%H:%M')
    day = startDt.strftime('%a')
    if not dateStart:         # einmalig initialisieren
      dateStart = startDt.strftime('%Y-%m-%d')
    dateEnd = startDt.strftime('%Y-%m-%d')  # überschreiben
    priceCentKw = round(price['marketprice'] / 10 * \
      (1 + surcharge) * (1 + vat))
    priceBar = '*' * int(priceCentKw)       # ASCII-Art-Balken
    print('%s %s %3d ct/kWh %s' % \
      (day, hour, priceCentKw, priceBar))
    hours += [hour]
    prices += [priceCentKw]
```

Das Script gibt die verarbeiteten Daten zuerst in Textform aus. Das sieht z. B. so aus:

```
Mo 19:00  20 ct/kWh ********************
Mo 20:00  17 ct/kWh *****************
Mo 21:00  16 ct/kWh ****************
...
```

### Matplotlib

Python ist auch im (natur)wissenschaftlichen Bereich sehr populär. Dementsprechend gibt es diverse Module, die beim Zeichnen von technischen Diagrammen helfen. Am populärsten ist die Matplotlib. Das folgende Listing zeigt, dass wenige Zeilen Code zur Gestaltung eines einfachen Diagramms ausreichen:

```
# Fortsetzung von electricity-prices.py
fig, ax = plt.subplots()
ax.bar(hours, prices)
plt.xticks(rotation=90)
plt.title('Prices in ct/kWh from %s to %s' % \
  (dateStart, dateEnd))
```

Persönliches Exemplar für Thomas Bartholomäus

```
# jeden zweiten Beschriftungspunkt ausblenden
for label in ax.xaxis.get_ticklabels()[::2]:
    label.set_visible(False)
fig.savefig('prices.png', dpi=200)
```

`subplots` liefert *zwei* Objekte zurück, die für unterschiedliche Aspekte des Diagramms verantwortlich sind. `ax.bar` erstellt ein einfaches Balkendiagramm aus den in zwei Parameter übergebenen X- und Y-Werten. `plt.xticks(rotation=90)` bewirkt eine platzsparende Beschriftung der X-Achse. `plt.title` kümmert sich um die Beschriftung des Diagramms.

Die folgende Schleife blendet jeden zweiten Beschriftungspunkt auf der X-Achse aus. `savefig` speichert das Diagramm schließlich in einer PNG-Datei. Der `dpi`-Parameter (*Dots per Inch*) bestimmt die dabei gewünschte Auflösung. (Der Defaultwert von nur 100 DPI führt zu recht pixeligen Diagrammen.)

Der Platz in diesem Buch reicht für eine intensive Auseinandersetzung mit der Matplotlib leider nicht aus. Dafür finden Sie auf der Projektwebseite unzählige Beispiele sowie sehr hilfreiche Cheatsheets:

*https://matplotlib.org*

Weitere Informationen zu Diagrammen mit Matplotlib sowie zu anderen Python-Funktionen im naturwissenschaftlichen Bereich finden Sie im Rheinwerk-Buch »Der Python-Kurs für Ingenieure und Naturwissenschaftler«.

### Energieverbrauch steuern

Noch spannender als die grafische Darstellung der Strompreise ist der automatisierte Betrieb großer Verbraucher bevorzugt in den Zeiten, wo die Energiekosten am niedrigsten sind. Ein Raspberry Pi, der die Preisdaten regelmäßig ausliest, kann z. B. einen Elektroboiler über einen steuerbaren Schalter zum preistechnisch idealen Zeitpunkt ein- und ausschalten. Bei Wärmepumpen bietet die *Smart-Grid*-Schnittstelle eine Möglichkeit, je nach Stromverfügbarkeit bzw. -preis zwischen Normal-, Spar- oder Extrabetrieb zu wechseln.

Mangels eigener Erfahrungen kann ich Ihnen diesbezüglich leider keine Beispiel-Scripts anbieten. Im Internet finden Sie Anleitungen und Erfahrungsberichte, z. B. hier:

- *https://www.awattar.at/blog/kdm_mai18*
- *https://www.awattar.at/blog/boilerpionier*
- *https://www.viessmann-community.com/t5/Waermepumpe-Hybridsysteme/Dynamischer-Strompreis-und-Waermepumpensteuerung/td-p/280599*


© Rheinwerk Verlag, Bonn 2024

# Kapitel 19
# Datenbanken

»Scripting und Datenbanken« ist ein fast unerschöpfliches Thema. Sie können mit Scripts Daten in eine Datenbank einfügen, Informationen von dort auslesen und verarbeiten, die Struktur von Datenbanken ändern usw.

Im Detail hängt die Vorgehensweise davon ab, welches Datenbankmanagementsystem (DBMS), welche Programmiersprache, welche Bibliothek bzw. welches Modul Sie verwenden. Insofern ist eine systematische Behandlung aller Aspekte hier von vornherein unmöglich. Stattdessen konzentriere ich mich in diesem Kapitel auf einige konkrete Beispiele, die sich auf den SQL Server bzw. auf MySQL beziehen:

- gleichartige Datenbanken ändern, warten und auslesen (Bash, PowerShell)
- Datenbank-Setup für neue Kunden (Bash)
- EXIF-Daten speichern (Python)
- JSON-Dateien importieren (PowerShell)

### Voraussetzungen für dieses Kapitel

Für die Lektüre dieses Kapitels benötigen Sie wie üblich Bash-, Python- und PowerShell-Grundkenntnisse. Außerdem sollten Sie mit JSON-Dateien umgehen können (siehe Kapitel 10). Schließlich benötigen Sie ein Grundverständnis über die Funktionsweise von Datenbank-Servern.

Wenn Sie mit Ihren Scripts Backups von Datenbanken erstellen möchten, empfehle ich Ihnen einen Blick in das Kapitel 15, »Backups«.

Um die Beispiele übersichtlich zu halten, habe ich auf eine Fehlerabsicherung sowie auf die Auslagerung der Login-Daten und anderer Einstellungen verzichtet. Bei »echten« Datenbank-Scripts sollten beide Punkte selbstverständlich sein. Tipps zur Fehlerabsicherung finden Sie in den Grundlagenkapiteln zu Bash, PowerShell und Python. Wie Sie sicher mit Einstellungen und Passwörtern umgehen, habe ich in Abschnitt 14.2 beschrieben.

Die Beispiel-Scripts aus diesem Kapitel können Sie nicht ohne Weiteres ausprobieren! Sie müssen zuerst den passenden Datenbank-Server installieren, eine passende Datenbank mit Tabellen einrichten, sich um die Authentifizierung beim Datenbank-Server kümmern etc.

Persönliches Exemplar für Thomas Bartholomäus

## 19.1 Datenbanken aktualisieren und warten

In meiner beruflichen Praxis betreue ich einen MySQL-Server, auf dem sich viele gleichartige Datenbanken befinden. Jeder Kunde hat seine eigene Datenbank. Zwar variiert der Inhalt dieser Datenbanken je nach Kunde, der Aufbau ist aber immer gleich.

Hin und wieder kommt es nun vor, dass ich – beispielsweise zur Fehlersuche – auf alle Datenbanken dieselben SELECT-Kommandos anwenden möchte. Das erforderliche Script ist – dem Motto dieses Buchs entsprechend – nicht einmal 10 Zeilen lang:

```
# Beispieldatei apply-select.sh
# diese Datei enthält die Namen aller Datenbanken
DBLIST=dbs.txt
# diese Datei enthält die auszuführenden Kommandos;
# die Kommandos müssen durch ; getrennt sein
SQLFILE=select.sql

for db in $(cat $DBLIST | sort) ; do
  echo "Database: $db"
  mysql $db < $SQLFILE
  echo "---"
done
```

Eine zentrale Bedeutung in dem Script nimmt das Kommando mysql ein. Es stellt eine Verbindung zu der durch die Variable db angegebenen Datenbank her und führt alle in der durch Eingabeumleitung ausgelesenen Datei enthaltenen Kommandos aus. Der Verbindungsaufbau zum Datenbank-Server setzt voraus, dass der aktive Benutzer ausreichende MySQL-Zugriffsrechte hat und die Authentifizierung entweder auf Betriebssystemebene erfolgt (auth_socket-Verfahren bei MySQL bzw. unix_socket bei MariaDB) oder dass die Datei .my.cnf das erforderliche Passwort enthält.

Die Ergebnisse der SELECT-Kommandos werden direkt am Bildschirm angezeigt. Zumeist führe ich das Script in der Form ./apply-select.sh | less durch, damit ich in Ruhe durch die Ausgaben scrollen bzw. darin suchen kann.

Grundsätzlich könnte select.sql auch Kommandos enthalten, um alle Datenbanken zu ändern, also z. B. um in jeder Datenbank bei einer Tabelle eine Spalte hinzuzufügen. Weil derartige Änderungen aber ungleich gefährlicher sind als reine Abfragen, speichere ich derartige Kommandos in einer eigenen Datei updates.sql, die von einem zweiten Script apply-updates.sh verarbeitet wird. Das Script ist bis auf die Initialisierung der Variable SQLFILE ident zu apply-select.sh.


© Rheinwerk Verlag, Bonn 2024

### PowerShell und sqlcmd

Die gleiche Idee lässt sich natürlich auch mit einem PowerShell-Script realisieren. Das Script ruft das Kommando `sqlcmd` auf, also das SQL-Server-Gegenstück zu `mysql`. Der Code sieht dann so aus:

```
# Beispieldatei apply-select.ps1
$dblist = "dbs.txt"         # Liste mit Datenbanknamen
$sqlfile = "select.sql"     # SQL-Kommandos
$server = ".\sqlexpress01" # SQL-Server-Instanz

# Schleife über alle Datenbanken
foreach($db in Get-Content $dblist) {
    Write-Output "Database: $db"
    sqlcmd -S $server -d $db -i $sqlfile
}
```

Wenn der SQL Server nicht auf dem lokalen Rechner läuft, ersetzen Sie `.` durch den Hostnamen. Wiederum setze ich hier voraus, dass das Script in einem Account läuft, der Zugriff auf alle betreffenden Datenbanken hat. Beachten Sie, dass die in `select.sql` enthaltenen SQL-Kommandos durch Strichpunkte getrennt sein müssen. Das Script setzt voraus, dass sich `sqlcmd.exe` in einem Verzeichnis befindet, das in der `PATH`-Variable aufgelistet ist.

## 19.2 Neuen Kunden-Account einrichten

Der Ausgangspunkt für dieses Beispiel ist ein Webserver für Kunden-Accounts. Bei einer Bestellung erhält der Kunde eine eigene Datenbank und kann auf die Web-Applikation über eine eigene Adresse zugreifen:

*https://example.com/name*

Das folgende Bash-Script kümmert sich um die Initialisierungsarbeiten, die beim Einrichten eines neuen Accounts notwendig sind:

- neue MySQL-Datenbank erzeugen und initialisieren
- neuen MySQL-User erzeugen und diesem Zugriff auf die Datenbank geben
- das Verzeichnis `/var/www/html/name` einrichten
- dort eine Konfigurationsdatei erzeugen
- eine Mail an den Kunden mit den Login-Daten senden

Auf einem Server, den ich betreue, gibt es tatsächlich ein Script, das so ähnlich funktioniert. Aus didaktischen Gründen habe ich den Code hier stark vereinfacht. Ich will Ihnen hier also die prinzipielle Vorgehensweise erläutern.

Persönliches Exemplar für Thomas Bartholomäus

### Keine Testmöglichkeit

Der Code für das Script befindet sich natürlich in den Beispieldateien zum Buch. Sie können das Script aber nicht ohne Weiteres ausprobieren. Dazu würden Sie einen Linux-Server brauchen (inklusive Web-, Datenbank- und Mail-Server), außerdem den Code für eine Web-Applikation, eine Muster-Datenbank etc.

## Account-Daten

`accountdata` enthält die Daten des neuen Kunden. Die Variable `URL` gibt den kundenspezifischen Teil der Webadresse an (also z. B. *https://example.com/sd-architects*). `DB` wird als Name für die neue MySQL-Datenbank und für den dazugehörenden MySQL-Benutzer verwendet. `DB` ist maximal 16 Zeichen lang und darf nur Buchstaben, Ziffern und Unterstriche enthalten.

```
# Beispieldatei accountdata
FIRSTNAME='Maria'
LASTNAME='Smith'
COMPANY='Smith & Davis Architects'
EMAIL='maria.smith@example.com'
URL='sd-architects'
DB='sdarchitects'
```

## Script-Aufbau

Das Script `make-new-account.sh` muss mit `root`-Rechten ausgeführt werden. Die im Script enthaltenen `mysql`- und `mysqldump`-Kommandos setzen voraus, dass `root` uneingeschränkte Zugriffsrechte auf den MySQL-Server hat bzw. dass das Passwort für den MySQL-root-User in der Datei `/root/.my.cnf` enthalten ist.

Der Code beginnt mit dem Einlesen der Account-Daten des neuen Kunden. Das Script testet zuerst, ob schon eine gleichnamige Datenbank existiert. In diesem Fall wird die Ausführung beendet:

```
# Beispieldatei make-new-account.sh
# Account-Daten importieren
. accountdata
# Abbruch, wenn die Datenbank schon existiert
sql="SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA
     WHERE SCHEMA_NAME='$DB'"
result=$(mysql -s -N -e "$sql")
echo "result = $result"
if [ "$result" ]; then
  echo "Database $DB already exists."
  exit
fi
```


© Rheinwerk Verlag, Bonn 2024

Das Script verwendet den Passwortgenerator mkpasswd, um zwei zufällige Passwörter zu erzeugen:

- Die Variable dbpw enthält das Passwort für den Zugriff auf die Kundendatenbank. Dieses Passwort wird nur intern verwendet und in einer Konfigurationsdatei auf dem Server gespeichert.
- loginpw enthält das Kundenpasswort für den Web-Login. Ein Hash-Code dieses Passworts wird in der Datenbank gespeichert. Das Passwort wird per Mail an den Kunden gesendet.

In den nächsten Zeilen wird eine neue Datenbank erzeugt und mit einer Musterdatenbank initialisiert. Außerdem wird ein MySQL-Account eingerichtet, der Zugriff auf diese Datenbank hat.

```
# Datenbank-Passwort (intern)
dbpw=$(makepasswd)
# Kunden-Passwort für den Web-Login
loginpw=$(makepasswd)

# Datenbank erzeugen und mit einer Kopie der 'template'-Datenbank
# initialisieren
mysqladmin create $DB
mysql $DB | mysqldump template

# Account für den Datenbankzugriff einrichten
sql="CREATE USER $DB@localhost IDENTIFIED BY '$dbpw';
     GRANT ALL ON $DB.* TO $DB@localhost"
mysql -e "$sql"
```

In der Kundendatenbank wird ein Hash-Code des Login-Passworts gespeichert. Der Hashcode wird mit htpasswd generiert. Anschließend werden die Kundendaten in der vorhin erzeugten Kundendatenbank gespeichert:

```
# Hash-Code des Login-Passworts erzeugen
hash=$(htpasswd -bnBC 10 "" $loginpw | tr -d ':\n')
# Kundendaten in der Kundendatenbank speichern
sql="INSERT INTO accounts (firstname, lastname, company,
                           email, hashcode)
     VALUES ('$FIRSTNAME', '$LASTNAME', '$COMPANY',
             '$EMAIL', '$hash')"
mysql $DB -e "$sql"
```

Im nächsten Schritt richtet das Script in /var/www/html ein neues Kundenverzeichnis ein und speichert dort in dbconfig.php die Login-Daten für den Datenbankzugriff. chown und chmod stellen sicher, dass die Zugriffsrechte auf die Dateien stimmen:

Persönliches Exemplar für Thomas Bartholomäus

```
# Webverzeichnis für den Kunden einrichten
cd /var/www/html/myapplication
mkdir $URL
cat > $URL/dbconfig.php << EOF
<?php
LocalConfig::set('dbname', '$DB');
LocalConfig::set('dbhost', 'localhost');
LocalConfig::set('dbuser', '$DB');
LocalConfig::set('dbpass', '$dbpw');
EOF

chown -R www-data:www-data $URL
chmod -R o-rwx $URL
chown root:www-data $URL/dbconfig.php
chmod 640 $URL/dbconfig.php
```

Zuletzt sendet das Script eine Mail an den neuen Kunden. Das Kommando `mail` setzt einen funktionierenden Mail-Server voraus.

```
# E-Mail an den neuen Kunden senden
BODY="Dear customer,
\n
\nthis is your login data:
\n
\nLogin:        https://example.com/$URL
\nAccount:      $EMAIL
\nPasswort:     $loginpw
\n
\nPlease change your password after the first login!
"
echo -e $BODY | mail -s "your new account"  \
    -a "From: support@example.com" \
    -a "Content-Type: text/plain; charset=UTF-8" \
    $EMAIL
```

## 19.3 EXIF-Metadaten in einer Datenbank speichern

In Abschnitt 16.3, »EXIF-Metadaten in SQL-Kommandos umwandeln«, habe ich Ihnen ein Python-Script präsentiert, das die EXIF-Metadaten aus Fotos im aktuellen Verzeichnis extrahiert und SQL-Kommandos mit entsprechenden `INSERT`-Kommandos erzeugt. Die SQL-Kommandos konnten per Ausgabeumleitung in einer Datei gespeichert und später auf eine Datenbank angewendet werden (z. B. mit dem Kommando `mysql`).


© Rheinwerk Verlag, Bonn 2024

Wesentlich eleganter ist es, wenn das Script eine Datenbankverbindung herstellt und die Metadaten direkt in der Datenbank speichert. Das erspart den Umweg über eine SQL-Datei. Ich gehe für das folgende Beispiel davon aus, dass die Datenbank von einem MySQL-Server verwaltet wird und dass dort die Tabelle `photos` existiert. Das Kommando `CREATE TABLE` dokumentiert den Aufbau der Tabelle:

```
CREATE TABLE photos(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  size INT,
  orientation INT,
  datetimeoriginal DATETIME,
  latitude DOUBLE,
  longitude DOUBLE,
  altitude DOUBLE,
  -- ts enthält den Zeitpunkt der letzten Änderung
  ts TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP()
     ON UPDATE CURRENT_TIMESTAMP()
);
```

### PyMySQL

Es gibt mehrere Python-Module für den Zugriff auf MySQL- oder MariaDB-Datenbanken. Ich konzentriere mich hier auf PyMySQL. Das Modul ist unkompliziert zu installieren, einfach anzuwenden und hat sich in meiner beruflichen Praxis bewährt. Informationen zu alternativen MySQL- bzw. Datenbank-Modulen finden Sie hier:

*https://stackoverflow.com/questions/372885*
*https://wiki.openstack.org/wiki/PyMySQL_evaluation*

PyMySQL muss wie üblich mit `pip` installiert werden:

```
$ pip  install requests        (Windows, Linux)
$ pip3 install requests        (macOS, alte Linux-Distributionen)
```

Beim Verbindungsaufbau mit `connect` geben Sie den Hostnamen des Servers an (`localhost`, wenn der Datenbank-Server auf demselben Rechner wie Ihr Script läuft), den Benutzernamen, das Passwort und den Namen der Datenbank, auf die Sie zugreifen möchten.

MySQL und MariaDB unterstützen diverse UTF-8-Varianten. Für das Zusammenspiel mit Python ist `utf8mb4` am besten geeignet. Der Parameter `cursorclass` steuert, wie PyMySQL `SELECT`-Ergebnisse zurückgibt. Der Cursortyp `DictCursor` verpackt jeden Datensatz in ein Dictionary, wobei die Spaltennamen als Schlüssel dienen. Das erleichtert die Weiterverarbeitung der Ergebnisse.

Persönliches Exemplar für Thomas Bartholomäus

Jedes Mal, wenn Sie ein SQL-Kommando ausführen möchten, benötigen Sie ein Cursor-Objekt. Es ist empfehlenswert, derartige Cursors mit `with` zu nutzen – dann ist sichergestellt, dass Sie die Objekte schnellstmöglich wieder freigeben und keine unnötigen Ressourcen blockieren.

An die `execute`-Methode übergeben Sie die Zeichenkette mit dem SQL-Kommando. Dieses kann beliebig viele `%s`-Codes enthalten. `execute` ersetzt sie durch Zeichenketten aus dem Tupel, der Liste oder dem Dictionary, das im zweiten Parameter übergeben wird. Beachten Sie, dass hier keine anderen Codes zulässig sind, also z. B. `%d` für Zahlen! Wenn Sie `NULL` speichern möchten, verwenden Sie das Python-Schlüsselwort `None`.

```
# Beispieldatei hello-pymysql.py
conn = pymysql.connect(host='localhost',
                       user='username',
                       password='topsecret',
                       db='dbname',
                       port=3306,
                       charset='utf8mb4',
                       cursorclass=pymysql.cursors.DictCursor)

# Einträge aus der 'photos'-Tabelle auslesen
with conn.cursor() as cur:
    sql = 'SELECT * FROM photos WHERE id < %s'
    cur.execute(sql, (1000))
    while row := cur.fetchone():
        print(row)
```

Mit `execute` können Sie auch `INSERT`-, `UPDATE`- oder `DELETE`-Kommandos ausführen. `commit` schließt die Transaktion ab. Die ID-Nummer des neuen Datensatzes können Sie anschließend der Eigenschaft `lastrowid` entnehmen:

```
sql = '''INSERT INTO photos (name, size, orientation)
         VALUES (%s, %s, %s)'''
with conn.cursor() as cur:
    cur.execute(sql, ('img_1234.jpg', 3231283, 0))
    conn.commit()
    print('ID of new record:', cur.lastrowid)
```

Wenn Ihr Script nach Abschluss der Datenbankoperationen weiterläuft, sollten Sie die Datenbankverbindung schließen:

```
conn.close()
```

Mehr Details zum Umgang mit PyMySQL finden Sie in der Projektdokumentation:

*https://pymysql.readthedocs.io*


© Rheinwerk Verlag, Bonn 2024

### EXIF-Metadaten speichern

Wie Sie die in den Bilddateien von Fotos eingebetteten EXIF-Daten in Python auswerten, habe ich Ihnen bereits in Abschnitt 16.3, »EXIF-Metadaten in SQL-Kommandos umwandeln«, anhand eines Scripts erläutert. An dieser Stelle geht es nur noch darum, die Metadaten direkt in einer Datenbank zu speichern. Dazu muss am Beginn des Scripts eine Datenbankverbindung hergestellt werden. Danach werden in einer Schleife alle Bilddateien durchlaufen.

```
# Beispieldatei exif-to-mysql.py (gekürzt)
import ...
# Verbindungsaufbau
conn = pymysql.connect(host='localhost', ...)

# nur die folgenden EXIF-Keys berücksichtigen
keys = ['File:FileName', 'File:FileSize', ...]

# Parameter auswerten (sys.argv)
filenames = ...

# INSERT-Kommando
sql = '''INSERT INTO photos (name, size, orientation,
             datetimeoriginal, latitude, longitude, altitude)
         VALUES (%s, %s, %s, %s, %s, %s, %s)'''

# Verbindung zum exiftool-Kommando herstellen, DB-Cursor
with exiftool.ExifToolHelper() as exifhelper, \
     conn.cursor() as cur:

    # Schleife über alle Dateien
    for file in filenames:
        # EXIF-Daten in Liste sammeln
        results = []
        try:
            metadata = exifhelper.get_tags(file, keys)[0]
            for key in keys:
                if key in metadata:
                    if key == 'EXIF:DateTimeOriginal':
                        # Datum an ISO-Syntax anpassen
                        date = str(metadata[key])
                        results += [ date.replace(':', '-', 2) ]
                    else:
                        results += [str(metadata[key])]
                else:
                    results += [ None ]   # entspricht NULL
```

Persönliches Exemplar für Thomas Bartholomäus

```
            # INSERT ausführen
            cur.execute(sql, results)
        except Exception as e:
            print("-- skipped %s" % (file))

# alle Änderungen speichern (Commit)
conn.commit()
conn.close()
```

Eine Sammlung mit Fotos zum Ausprobieren des Scripts finden Sie bei den Beispieldateien zu Kapitel 16, »Bildverarbeitung«.

## 19.4 JSON-Daten in eine Tabelle importieren

Als Ausgangspunkt für diesen Abschnitt dient die JSON-Datei `employees.json`. Sie ist wie folgt aufgebaut:

```
[
  {
    "FirstName": "Ruthanne",
    "LastName": "Ferguson",
    "DateOfBirth": "1977-06-04",
    "Street": "4 Dewy Turnpike",
    "Zip": "27698",
    "City": "Clifton Hill",
    "State": "NJ",
    "Gender": "F",
    "Email": "ruthanne_ferguson5693@fastmail.cn",
    "Job": "Junior Engineer",
    "Salary": "5201.45"
  }, ...
```

Ein PowerShell-Script soll diese Datei verarbeiten und die Datensätze in eine Tabelle einer SQL-Server-Datenbank einfügen. Ich gehe dabei davon aus, dass die JSON-Keys mit den Spaltennamen der Tabelle übereinstimmen. Eine geeignete Tabelle können Sie mit dem folgenden Kommando erstellen:

```
CREATE TABLE employees(
  id INT IDENTITY(1, 1) PRIMARY KEY,
  FirstName TEXT NOT NULL,
  LastName TEXT NOT NULL,
  DateOfBirth DATE,
  Street TEXT,
  City TEXT,
  State CHAR(2),
```


© Rheinwerk Verlag, Bonn 2024

```
  Gender CHAR(1),
  Email TEXT,
  Job TEXT,
  Salary FLOAT);
```

Das Script verwendet das Modul `SQLServer`, das Sie aus Abschnitt 15.3, »SQL-Server-Backup«, schon kennen. Falls Sie das noch nicht erledigt haben, müssen Sie das Modul mit `Install-Module SQLServer` installieren.

Der Code beginnt mit der Initialisierung einiger Variablen. `Get-Content` liest die JSON-Datei. `ConvertFrom-JSON` macht daraus ein Array mit `PSCustomObjects`. Die äußere `foreach`-Schleife durchläuft alle Array-Elemente. Die innere Schleife durchläuft alle Spalten und erzeugt den `VALUES`-Teil des `INSERT`-Kommandos. Die resultierenden Kommandos sehen dann aus wie das folgende Beispiel:

```
INSERT INTO employees (FirstName, LastName, DateOfBirth, ...)
VALUES('Sebastian', 'James', '1953-05-14', ...);
```

Das Script verlässt sich darauf, dass bei jedem Datensatz für alle Spalten Daten zur Verfügung stehen. Wenn das nicht der Fall ist, müssen Sie entsprechende Tests einbauen und gegebenenfalls `NULL` in das SQL-Kommando einbauen.

Zuletzt werden sämtliche gesammelten Kommandos mit `Invoke-Sqlcmd` an den SQL Server übergeben.

```
# Beispieldatei json-to-sql-server.ps1
Import-Module SQLServer

# Verbindung zum SQL Server
$connectionString = "Server=.\sqlexpress01;Database=mydb;" +
                    "Trusted_Connection=true;Encrypt=false"

# Name der Tabelle und der Spalten
$tableName = "employees"
$columns = "FirstName", "LastName", "DateOfBirth", "Street",
           "City", "State", "Gender", "Email", "Job", "Salary"

$sql = "INSERT INTO $tablename (" +  ($columns -Join ", ") + ") "
$sqlcmds = ""

# JSON-Datei lesen und in PowerShell-Objekte umwandeln
$jsonFilePath = "employees.json"
$json = Get-Content $jsonFilePath | ConvertFrom-Json

# Schleife über alle Array-Elemente (Datensätze)
foreach ($record in $json) {
    $values = ""
```

Persönliches Exemplar für Thomas Bartholomäus

```
    # Schleife über alle Spalten
    foreach ($column in $columns) {
        if ($values) {
            $values += ", "
        }
        $values += "'" + $record.$column + "'"
    }
    $sqlcmds += $sql + "`nVALUES(" + $values + ");`n"
}

# SQL-Kommandos ausführen
Invoke-Sqlcmd -ConnectionString $connectionString -Query $sqlcmds
```


© Rheinwerk Verlag, Bonn 2024

# Kapitel 20
# Scripting in der Cloud

Die Cloud ist groß und vielfältig – insofern ist der Titel dieses Kapitels vermessen. Man könnte ein ganzes Buch zu diesem Thema schreiben und dabei auf unterschiedliche Cloud-Spielarten, -Anbieter und -Programmiervarianten eingehen.

An dieser Stelle beschränke ich mich auf den *Simple Storage Service* (S3) der *Amazon Webservices* (AWS). Dieser Dienst hat sich als eine kostengünstige Möglichkeit etabliert, Dateien zu speichern. Es gibt zwei zentrale Anwendungsmöglichkeiten: Sie können dort Dateien ablegen, die öffentlich zugänglich sind (etwa zur Entlastung des eigenen Webservers), oder Sie verwenden S3 als Ort für Backups oder selten benötigten Dateien. Dabei müssen Sie die Kosten im Auge behalten: Je nach Angebot kann der Transfer oder die eigentliche Aufbewahrung der entscheidende Faktor sein.

Zur Steuerung der AWS-S3-Funktionen gibt es zwei großartige Bibliotheken: die AWS-CLI für Bash-Scripts und das AWS-Modul für PowerShell-Scripts. In diesem Kapitel gehe ich auf beide Varianten ein.

**Voraussetzungen für dieses Kapitel**

Damit Sie von diesem Kapitel profitieren, müssen Sie die Bash bzw. die PowerShell einigermaßen beherrschen. Außerdem benötigen Sie Grundkenntnisse zu AWS-S3: Insbesondere müssen Sie wissen, was Buckets sind und wie die Benutzerverwaltung (*Identity and Access Management*, kurz IAM) funktioniert.

Die beiden in diesem Kapitel präsentierten Beispiele bauen auf Techniken auf, die ich in Kapitel 15, »Backups«, bzw. Kapitel 17, »Web Scraping«, vorstelle. Insofern ist es empfehlenswert, zuerst einen Blick in diese beiden Kapitel zu werfen.

## 20.1 AWS CLI

Das AWS CLI (*Command Line Interface*) bietet die Möglichkeit, AWS-Funktionen auf Kommandoebene zu steuern. Das CLI kann grundsätzlich auf allen gängigen Plattformen installiert werden. Ich setze im Folgenden aber voraus, dass Sie unter Linux oder macOS arbeiten und Ihre Scripts in der Bash entwickeln möchten.

Persönliches Exemplar für Thomas Bartholomäus

Keine Angst, PowerShell-Freunde kommen nicht zu kurz. Blättern Sie weiter zu Abschnitt 20.3, »AWS-PowerShell-Modul«!

### Installation unter Linux und macOS

AWS-CLI-Downloads und Installationsanleitungen für alle Plattformen finden Sie hier:

*https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html*

Unter Linux laden Sie die ZIP-Datei herunter, packen das Archiv aus und führen das Installations-Script aus. Das `aws`-Kommando landet im Verzeichnis `/usr/local/bin` und sollte dort ohne `PATH`-Änderungen ausführbar sein. Falls Sie eine schon vorhandene Installation aktualisieren möchten, übergeben Sie an das `install`-Kommando die Option `--update`.

```
$ curl https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip \
     -o "awscliv2.zip"
$ unzip awscliv2.zip
$ sudo ./aws/install
$ aws --version
  aws-cli/2.11.11 Python/3.11.2 ...
```

> **Ärger mit Cron**
>
> Bei einigen Linux-Distributionen ist `/usr/local/bin` nicht in der für Cron-Jobs relevanten `PATH`-Variablen enthalten. Der Aufruf von `aws` in Cron-Jobs scheitert dann. Abhilfe: Erstellen Sie einen Link von `/usr/bin/aws` aus `/usr/local/bin/aws`, oder geben Sie im Script immer den vollständigen Pfad des `aws`-Kommandos an.

Für macOS stellt Amazon einen PKG-Installer zur Verfügung. Die Installation gelingt mit wenigen Klicks. Auch in diesem Fall wird das `aws`-Kommando in das Verzeichnis `/usr/local/bin` installiert.

### Konfiguration

Für die folgenden Beispiele benötigen Sie einen AWS-Nutzer, der Zugriffsrechte auf ein oder mehrere Buckets hat. Nachdem Sie den Nutzer in der AWS-Weboberfläche unter *Identity and Access Management* eingerichtet und mit einem Access Key ausgestattet haben, führen Sie `aws configure` aus. Das Kommando fragt nach den Zugangsdaten und speichert diese in den lokalen Konfigurationsdateien `.aws/config` und `.aws/credentials`. Führen Sie `aws configure` in dem Account aus, unter dem später auch Ihre Scripts laufen sollen. Wenn es sich um Backup-Scripts handelt, die `root`-Rechte benötigen, muss auch `aws configure` im `root`-Account ausgeführt werden!


© Rheinwerk Verlag, Bonn 2024

```
$ aws configure

  AWS Access Key ID [None]:      AKxxxxxxx
  AWS Secret Access Key [None]: xxxxxxxxxxxxxxxx
  Default region name [None]:   eu-central-1
  Default output format [None]: <Return>
```

Die Default-Region können Sie anfangs leer lassen. Später ermitteln Sie mit `aws s3api get-bucket-location` die Region Ihrer Buckets und wiederholen dann `aws configure`:

```
$ aws s3api get-bucket-location --bucket my.bucket.name
  "LocationConstraint": "eu-central-1"
```

### Erste Schritte

`aws`-Kommandos bestehen normalerweise aus drei Teilen: dem Hauptkommando `aws`, dem Namen eines AWS-Dienstes (z. B. `s3` oder `ec2`) und schließlich einem Subkommando für diesen Dienst. Um festzustellen, auf welche Buckets Sie in Ihrer Konfiguration zugreifen können, führen Sie das folgende Kommando aus:

```
$ aws s3 ls

  2020-06-23 00:40:38 my.bucket.name
  2022-12-12 02:51:07 my.bucket.othername
```

Wenn Sie an `ls` als weiteren Parameter `s3://<bucketname>` übergeben, zeigt `aws` die Dateien in diesem Bucket an:

```
$ aws s3 ls s3://my.bucket.name

  2023-12-15 11:11:53      61431 test.txt
  2023-03-22 15:23:01      61431 test2.txt
```

Mit `aws s3 cp` kopieren Sie eine lokale Datei in ein Bucket bzw. von dort zurück auf den lokalen Rechner:

```
$ aws s3 cp local-file.txt s3://my.bucket.name
$ aws s3 cp s3://my.bucket.name/local-file.txt copy.txt
```

Neben `cp` gibt es einige weitere Subkommandos, die Ihnen aus Ihrer Arbeit in der Bash vertraut sind, z. B. `mv` oder `rm`. Viele Kommandos kennen die Optionen `--recursive`, `--include 'pattern'` und `--exclude 'pattern'`, wobei das Muster unter anderem die Zeichen `?` und `*` unterstützt. Die Zeichen haben die Bedeutung wie in der Bash (siehe Tabelle 3.3), nicht wie in regulären Mustern.

Beachten Sie, dass S3-Buckets das Konzept von Verzeichnissen nur mit Einschränkungen kennen. Zwar darf der Name einer im Bucket gespeicherten Datei aus mehreren

Persönliches Exemplar für Thomas Bartholomäus

Teilen zusammengesetzt sein (`dir1/dir2/file`), es gibt aber keine Möglichkeit, mit `mkdir` ein Verzeichnis zu erstellen oder mit `rmdir` ein Verzeichnis zu löschen.

Zu den wichtigsten AWS-S3-Kommandos zählt `aws s3 sync`. Damit können Sie ein lokales Verzeichnis und ein Verzeichnis in einem Bucket synchronisieren. Gerade für Backups ist das enorm praktisch: Wenn es ein komplettes Backup Ihrer Daten in einem lokalen Verzeichnis gibt, schaffen Sie mit einem regelmäßig ausgeführten Synchronisationskommando Redundanz. Sollte das lokale Backup verloren gehen, haben Sie immer noch eine Kopie in der Cloud. (Beachten Sie, das Sie bei AWS nicht nur für die Menge der gespeicherten Daten zahlen, sondern auch für jeden Transport in die eine oder andere Richtung. Aus Kostengründen ist es zweckmäßig, das Backup inkrementell zu organisieren, sodass nur die von einem zum nächsten Tag gespeicherten Änderungen übertragen werden.)

```
$ aws s3 sync my-local-backupdir s3://my.bucket.name
```

Eine Referenz aller AWS-S3-Kommandos finden Sie hier:

*https://awscli.amazonaws.com/v2/documentation/api/latest/reference/s3/index.html*

### Dateien verschlüsseln

Auch wenn Amazon damit wirbt, dass Ihre Dateien verschlüsselt gespeichert werden, verbessert das die Datensicherheit nur minimal, solange Amazon den Schlüssel hat. Wenn Sie möchten, dass keine fremden Unternehmen oder Geheimdienste die Backups Ihrer Organisation oder Firma lesen, müssen Sie alle Dateien verschlüsseln, bevor Sie sie in die Cloud übertragen. Diese Grundregel ist nicht Amazon-spezifisch, sondern gilt für *jede* Speicherung von Daten auf externen Servern oder bei Cloud-Diensten!

Im Folgenden stelle ich Ihnen das Kommando `gpg` vor, mit dem Sie Dateien unkompliziert symmetrisch verschlüsseln und später wieder entschlüsseln können.

#### Symmetrisch versus asymmetrisch

»Symmetrische Verschlüsselung« bedeutet, dass zum Verschlüsseln und Entschlüsseln jeweils der gleiche Schlüssel verwendet wird. Dem stehen asymmetrische Verfahren mit einem Schlüsselpaar gegenüber (so wie bei SSH): Der öffentliche Schlüssel wird zum Verschlüsseln verwendet, der private zum Entschlüsseln. Das ist vor allem dann vorteilhaft, wenn die Verschlüsselung an verschiedenen Orten (Rechnern) stattfinden soll. Der dazu erforderliche öffentliche Schlüssel kann bedenkenlos verteilt werden.


© Rheinwerk Verlag, Bonn 2024

> Leider sind asymmetrische Verfahren für große Dateien ineffizient. Um dennoch die Vorzüge asymmetrischer Verfahren zu nutzen, ist es üblich, die Dateien weiterhin symmetrisch zu verschlüsseln. Zusätzlich werden nun aber auch die Schlüssel verschlüsselt – und das asymmetrisch! Man spricht nun von einem »hybriden Verschlüsselungssystem«:
>
> *https://en.wikipedia.org/wiki/Hybrid_cryptosystem*
>
> Für unsere Zwecke, also zur sicheren Verwahrung von Backup-Dateien in einer Cloud, ist ein symmetrisches Verfahren absolut ausreichend. Sie müssen nur sicherstellen, dass der von Ihnen verwendete Schlüssel nicht in falsche Hände kommt.

Zuerst brauchen Sie einen Schlüssel, also einfach eine binäre Datei mit Zufallsdaten. Ein guter Weg zur Erzeugung eines neuen Schlüssels ist das Kommando `openssl`, das unter Linux Bestandteil des gleichnamigen Pakets ist. Das folgende Kommando erzeugt einen Schlüssel mit 32 Bytes Länge. (32 Bytes erscheinen wenig, aber das sind 256 Bits. Für symmetrische Verfahren gelten aktuell 128 Bits als ausreichend sicher.)

```
$ openssl rand 32 > mykey
```

Falls Ihnen `openssl` nicht zur Verfügung steht, funktioniert unter Linux auch das folgende Kommando:

```
$ dd if=/dev/random of=mykey bs=16 count=1
```

Ich habe schon klargestellt: Die Key-Datei ist – kleines Wortspiel – der Schlüssel zur Sicherheit. Auf meinen Servern bewahre ich derartige Dateien in einem nur von `root` lesbaren Verzeichnis auf und stelle mit `chown root:root mykey` und `chmod 600` sicher, das wirklich niemand außer `root` die Datei lesen darf. (Denken Sie auch daran, an einem sicheren Ort ein Backup Ihres Schlüssels aufzubewahren! Sollte Ihr Server samt der Schlüsseldatei verloren gehen – etwa wegen eines Hardware-Defekts – können Sie die verschlüsselten Backup-Dateien in der Cloud nie wieder entschlüsseln.)

Zum Ver- bzw. Entschlüsseln setzen Sie das Kommando `gpg` ein. Da jeweils eine Menge Optionen übergeben werden müssen, verpacken Sie den Aufruf am besten in zwei winzige Scripts:

```
# Beispieldatei mycrypt.sh
# Verwendung: mycrypt.sh < plain > crypted
gpg -c -q --batch --cipher-algo AES256 --compress-algo none \
  --passphrase-file /path/to/mykey

# Beispieldatei myuncrypt.sh
# Verwendung: myuncrypt.sh < crypted > plain
gpg -d --batch --no-tty -q --cipher-algo AES256 \
  --compress-algo none --passphrase-file /path/to/mykey
```

Persönliches Exemplar für Thomas Bartholomäus

Kurz eine Erläuterung der Optionen:

- `-c` (*symmetric crypt*) verschlüsselt die Standardeingabe, schreibt in die Standardausgabe.
- `-d` (*symmetric decrypt*) entschlüsselt die Standardeingabe, schreibt in die Standardausgabe.
- `-q` (*quiet*) unterdrückt Statusmeldungen.
- `--batch` aktiviert den Batch-Modus (keine interaktiven Rückfragen).
- `--cipher-algo` legt den Verschlüsselungsalgorithmus fest.
- `--compress-algo` legt den Komprimieralgorithmus fest.
- `--passphrase-file` gibt an, aus welcher Datei `gpg` den Schlüssel lesen soll.

Sie können die beiden Scripts wie folgt ausprobieren:

```
$ ./mycrypt < readme.txt > readme.crypt
$ ./myuncrypt < readme.crypt > readme.copy
$ diff readme.txt readme.copy
```

Das erste Kommando verschlüsselt `readme.txt`. Das zweite Kommando entschlüsselt die Datei und speichert das Ergebnis als `readme.copy`. Das dritte Kommando vergleicht die beiden Dateien und listet alle Unterschiede auf. Wenn `diff` nichts ausgibt, sind die Dateien identisch.

**Zuerst komprimieren, dann verschlüsseln**

Wenn Sie eine Datei komprimieren *und* verschlüsseln möchten, sollten Sie immer zuerst komprimieren und dann verschlüsseln. Umgekehrt scheitert das Komprimieren, weil schon verschlüsselte Dateien für das Komprimierprogramm wie eine Folge von Zufallsdaten aussehen. Eine Komprimierung ist damit unmöglich.

## 20.2 Beispiel: Verschlüsselte Backup-Dateien in die Cloud hochladen

In Abschnitt 15.2, »WordPress-Backup«, habe ich Ihnen ein Script vorgestellt, das Backups einer MySQL-Server-Datenbank und eines Webserver-Verzeichnisses in einem lokalen Verzeichnis speichert. Mit nur wenig Mühe können Sie dieses Script dahingehend verbessern, dass es die lokalen Dateien zuerst komprimiert und dann in ein AWS-Bucket hochlädt. Weil das Script recht kurz ist, habe ich hier nochmals den ganzen Code abgedruckt. Auf einige Besonderheiten möchte ich kurz hinweisen:

- Die Verschlüsselungskommandos sind in Funktionen verpackt.
- Die Funktionen werden in einer Pipe aufgerufen: `mysqldump` erzeugt das Backup, `gzip` komprimiert es und `mycrypt` verschlüsselt es. Erst das Ergebnis wird in einer


© Rheinwerk Verlag, Bonn 2024

Datei gespeichert. Diese Vorgehensweise vermeidet die zeitaufwendige Erzeugung von Zwischendateien.

- Analog erzeugt `tar` ein komprimiertes Archiv. `-f -` leitet es an die Standardausgabe weiter. `mycrypt` verschlüsselt diese und speichert wiederum das Ergebnis in einer Datei.
- Beim Aufruf von `aws` habe ich jeweils den gesamten Pfad des Kommandos angegeben, sodass das Script auch bei einer Automatisierung durch Cron fehlerfrei funktioniert.
- Im Gegensatz zum gewöhnlichen `cp`-Kommando, bei dem Sie mehrere Dateien in ein Zielverzeichnis kopieren können (also `cp file1 file2 file3 dir`), akzeptiert `aws s3 cp` nur eine Quelldatei. Daher muss ich das Kommando für jede Datei extra aufrufen.

```
# Beispieldatei lamp-backup-to-aws.sh
BACKUPDIR=/localbackup
DB=wp
DBUSER=wpbackupuser
WPDIR=/var/www/html/wordpress
BUCKET=s3://your.bucket.name

function mycrypt {
  gpg -c -q --batch --cipher-algo AES256 --compress-algo none \
      --passphrase-file /etc/mykey
}
function myuncrypt {
  gpg -d --batch --no-tty -q --cipher-algo AES256 \
      --compress-algo none --passphrase-file /etc/mykey
}

# MySQL-Backup
weekday=$(date +%u)
dbfile=$BACKUPDIR/wp-db-$weekday.sql.gz.crypt
mysqlopt='--single-transaction'
mysqldump -u $DBUSER $mysqlopt $DB | gzip -c | mycrypt > $dbfile

# Backup der WordPress-Dateien
htmlfile=$BACKUPDIR/wp-html-$weekday.tar.gz.crypt
tar czf - -C $WPDIR . | mycrypt > $htmlfile

# Upload in ein AWS-Bucket
/usr/local/bin/aws s3 cp $dbfile   $BUCKET
/usr/local/bin/aws s3 cp $htmlfile $BUCKET
```

**Wiederherstellung überprüfen**

Wenn Ihr Backup-Script fertig ist, sollten Sie unbedingt testen, ob Sie Ihre Daten aus den Backups wiederherstellen können!

## 20.3 AWS-PowerShell-Modul

Grundsätzlich spricht nichts dagegen, das vorhin behandelte AWS-CLI (also das Kommando `aws`) in PowerShell-Scripts aufzurufen. Es gibt aber einen noch eleganteren Weg: Amazon stellt mit den *AWS Tools for PowerShell* mehrere Module für verschiedene AWS-Services zur Verfügung. Die Module werden ausgezeichnet gewartet und üblicherweise ein- bis zweimal pro Woche aktualisiert. Weil die AWS Tools echte PowerShell-Objekte zurückgeben, können Sie Ihre Scripts oft klarer formulieren als mit dem AWS-CLI.

Die Installation gelingt mühelos mit `Install-Module`, übrigens nicht nur unter Windows, sondern auch unter Linux und macOS:

```
> Install-Module AWS.Tools.Common
> Install-Module AWS.Tools.S3
```

### Erste Schritte

Wie beim AWS-CLI setze ich im Folgenden voraus, dass es einen AWS-Nutzer mit ausreichenden Zugriffsrechten auf ein oder mehrere Buckets gibt. Dessen *Access-* und *SecretKey* müssen Sie nun mit `Set-AWSCredentials` angeben. Außerdem können Sie einen Profilnamen angeben. Damit können Sie für unterschiedliche Scripts jeweils eigene Zugriffsdaten verwenden. Mit `-StoreAs default` können Sie ein Default-Profil anlegen.

```
> Set-AWSCredential -AccessKey AKxxx -SecretKey xxxx `
   -StoreAs MyProfile
```

Unter Windows werden die Zugriffsdaten in verschlüsselter Form in der folgenden Datei gespeichert:

```
C:\Users\<name>\AppData\Local\AWSToolkit\RegisteredAccounts.json
```

Unter Linux und macOS ist der Speicherort wie beim AWS-CLI `.aws/credentials`; die Keys werden im Klartext gespeichert.

Um zu testen, ob die Konfiguration funktioniert hat, führen Sie `Get-S3Bucket` aus. Das CmdLet listet Ihre Buckets auf.


© Rheinwerk Verlag, Bonn 2024

```
> Get-S3Bucket -ProfileName MyProfile
> Get-S3Bucket                              # für das default-Profil
```

In den weiteren Beispielen gehe ich davon aus, dass Sie ein Profil mit dem Namen `default` eingerichtet haben. Sollte das nicht der Fall sein, müssen Sie sämtlichen CmdLets die Option `-ProfileName` mit Ihrem Profilnamen hinzufügen. Alternativ können Sie zu Beginn einer Session oder eines Scripts mit `Set-AWSCredential` das gewünschte Profil voreinstellen:

```
> Set-AWSCredential -ProfileName MyProfile
```

Wenn Sie wissen möchten, in welcher Region sich Ihre Buckets befinden, führen Sie `Get-S3BucketLocation` aus. Beachten Sie, dass das CmdLet ein leeres Ergebnis liefert, wenn sich das Bucket in der Region US-East (North Virginia) befindet (`us-east-1`).

```
> Get-S3Bucket | ForEach-Object {
    $name = $_.BucketName
    $region = Get-S3BucketLocation -BucketName $name
    Write-Output "$name : $region"
  }

  my.first.bucket : eu-central-1
  my.other.bucket : eu-west-2
  ...
```

Den Inhalt eines Buckets zeigt `Get-S3Object` an – und zwar etwas ausführlicher, als Ihnen vielleicht recht ist:

```
> Get-S3Object -BucketName my.first.bucket -Region eu-central-1

  ChecksumAlgorithm : {}
  ETag              : "b81e..."
  BucketName        : my.first.bucket
  Key               : duplicati1.png
  LastModified      : 13.05.2019 21:36:35
  Owner             : Amazon.S3.Model.Owner
  Size              : 31729
  StorageClass      : STANDARD

  ChecksumAlgorithm : {}
  ETag              : "ab98"
  ...
```

Vermutlich sind Sie nur an den Dateinamen interessiert (Eigenschaft `Key`). Außerdem werden Sie nur in den seltensten Fällen wirklich *alle* Dateien aus einem Bucket benötigen. (Tatsächlich liefert das CmdLet maximal 1000 Treffer.) Das folgende Kommando liefert nur die Namen aller Dateien, die mit `dir1/` beginnen:

Persönliches Exemplar für Thomas Bartholomäus

```
> Get-S3Object -BucketName my.first.bucket -KeyPrefix 'dir1/' |
  Select-Object Key
```

Leider bietet `Get-S3Object` keine Möglichkeit, Dateien nach anderen Kriterien zu filtern. Soweit Sie die 1000-Dateien-Grenze nicht überschreiten, müssen Sie *alle* Dateinamen ermitteln und dann `Filter-Object` anwenden. Das folgende Kommando zeigt die Namen aller Dateien an, die mit `.txt` enden:

```
> Get-S3Object -BucketName my.first.bucket |
  Select-Object Key |
  Where-Object { $_.Key -like '*.txt' }
```

Relativ oft wird es Ihnen passieren, dass `Get-S3Object` bzw. diverse andere CmdLets die folgende Fehlermeldung liefern: *The bucket you are attempting to access must be addressed using the specified endpoint.*

Der Fehler deutet darauf hin, dass Sie die Option `-Region` vergessen haben und AWS nicht weiß, wo sich Ihr Bucket befindet. Da die ständige Angabe der Option `-Region` lästig ist, können Sie die Default-Region für die aktuelle Session bzw. für Ihr Script mit `Set-DefaultAWSRegion` einstellen.

```
> Set-DefaultAWSRegion -Region eu-central-1
```

### Dateien kopieren

Die wohl häufigste Aufgabe bei der Nutzung von AWS S3 besteht darin, Dateien in die Cloud hoch- bzw. von dort wieder herunterzuladen. Dazu dienen `Write-S3Object` und `Read-S3Object`:

- Das erste Kommando lädt eine lokale Datei in ein Bucket hoch. Der Key-Name im Bucket stimmt mit dem der lokalen Datei überein.
- Das zweite Kommando lädt eine weitere Datei hoch, gibt der Datei im Bucket aber einen anderen Namen (Option `-Key`).
- Das dritte Kommando lädt die Datei `readme.txt` aus dem Bucket herunter und speichert die Datei lokal unter dem Namen `local-file.txt`.
- Das vierte Kommando lädt alle Dateien, deren Key mit `dir1/` beginnt, in das aktuelle Verzeichnis herunter.

```
> Write-S3Object -BucketName my.first.bucket -File local-file.txt
> Write-S3Object -BucketName my.first.bucket -File local.txt `
                 -Key dir1/tst.txt
> Read-S3Object  -BucketName my.first.bucket -Key  readme.txt `
                 -File local-file.txt
> Read-S3Object  -BucketName my.first.bucket -KeyPrefix dir1/ `
                 -Folder .
```


© Rheinwerk Verlag, Bonn 2024

Mit `Remove-S3Object` löschen Sie Dateien im Bucket:

```
> Remove-S3Object -BucketName my.first.bucket -Key readme.txt
```

Neben den bisher aufgezählten CmdLets stellen die AWS Tools unzählige weitere Funktionen zur Auswahl. Eine Referenz aller S3-CmdLets finden Sie hier:

*https://docs.aws.amazon.com/powershell/latest/reference/items/S3_cmdlets.html*

Angesichts der Fülle von CmdLets irritiert das Fehlen einer ganz wichtigen Funktion: Es gibt in den AWS Tools kein CmdLet, das dem CLI-Kommando `aws s3 sync` entspricht. Wenn Sie ein lokales Verzeichnis mit einem Bucket synchronisieren möchten, empfehle ich Ihnen, auch in PowerShell-Scripts auf das CLI zurückzugreifen.

## 20.4 Beispiel: Große Dateien einer Webseite in die Cloud auslagern

Ausgangspunkt für das folgende Beispiel ist eine eigene Website. Das Ziel besteht darin, die dort verlinkten sehr großen PDF-Dateien in die Cloud auszulagern. Die Website soll also weiter mit einem eigenen Server betrieben werden. Wenn die Besucherinnen und Besucher auf einen PDF-Download-Link klicken, soll diese Datei aber von AWS S3 heruntergeladen werden. Damit bleibt die Website unter Ihrer eigenen Verwaltung, gleichzeitig können Sie die Last Ihres Webservers durch große Downloads minimieren.

Das Script `migrate-pdf-to-aws.ps1` funktioniert wie folgt:

- Es lädt den HTML-Code einer Webseite herunter.
- Es durchsucht alle Links nach solchen, die mit `.pdf` enden.
- Es lädt die PDF-Dokumente zuerst in ein temporäres Verzeichnis herunter und dann in ein öffentliches AWS-S3-Bucket hoch.
- Im HTML-Code werden die Links angepasst. Eine neue Version der HTML-Seite wird lokal gespeichert.

### Vorbereitungen

Für dieses Beispiel benötigen Sie ein Bucket, das öffentlich im Web erreichbar ist. Dazu müssen Sie in der AWS-Console mehrere Einstellungen vornehmen:

- Aus Sicherheitsgründen ist standardmäßig jeder öffentliche Zugriff auf neue Buckets blockiert. Die entsprechende Option BLOCK ALL PUBLIC ACCESS muss deaktiviert werden.
- Sie müssen eine *Bucket Policy* festlegen, die den Zugriff auf alle Objekte erlaubt. In der Konsole formulieren Sie dazu ein JSON-Dokument nach dem folgenden Muster:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::my.public.bucket/*"
        }
    ]
}
```

In diesem Code müssen Sie `my.public.bucket` durch den Namen Ihres Buckets ersetzen. Das `Version`-Datum darf hingegen nicht geändert werden.

- Schließlich müssen Sie die Option STATIC WEBSITE HOSTING aktivieren. An dieser Stelle erfahren Sie auch, unter welcher HTTP-Adresse Ihre Bucket-Objekte zu finden sind. Sie müssen ein Startdokument für das Webhosting angeben, üblicherweise `index.html`.

  Eigentlich ist diese Seite als Startpunkt für die statische Website gedacht. Für dieses Beispiel ist eine derartige Startseite aber gar nicht erforderlich. Sie können dennoch eine minimale HTML-Datei hochladen, die den Zweck der Website erläutert (z. B.: *This site hosts PDF documents*).

### Script

Das Script beginnt mit der Initialisierung einiger Variablen. `Set-DefaultAWSRegion` und `Set-AWSCredential` stellt die Region und das Profil für alle weiteren AWS-CmdLets ein. Anschließend erzeugt es ein temporäres Verzeichnis, in dem die PDF-Dokumente zwischengespeichert werden können, und lädt mit `Invoke-WebRequest` den zu verarbeitenden HTML-Code herunter.

```
# Beispieldatei migrate-pdf-to-aws.ps1
$bucket = "my.pdf.bucket"
$awsurl = `
  "http://my.pdf.bucket.s3-website.eu-central-1.amazonaws.com/"
$region = "eu-central-1"
$awsprofile = "MyProfile"      # profile for AWS credentials
$htmlsource = "https://example.com/page-with-pdf-links.html"
$htmldest = "updated.html"  # file name for new HTML code

# Verzeichnis .\tmp erzeugen
New-Item -ItemType Directory -Force tmp | Out-Null
```



© Rheinwerk Verlag, Bonn 2024

```
# AWS-Default-Region und -Profil einstellen
Set-DefaultAWSRegion $region
Set-AWSCredential -ProfileName $awsprofile

# HTML-Code herunterladen
$response = Invoke-WebRequest $htmlsource
$html = $response.Content
```

Eine Schleife durchläuft nun alle Links im HTML-Dokument. Wenn der Link mit `.pdf` endet, wird die Datei zuerst in das lokale Verzeichnis heruntergeladen und dann in die Cloud hochgeladen. `Replace` ersetzt dann die ursprüngliche Link-Adresse durch die neue. (Beachten Sie, dass `Replace` hier zweckmäßiger ist als der Operator `-replace`, der reguläre Muster verarbeitet. In einer URL kommen häufig Punkte vor, die in regulären Ausdrücken eine besondere Bedeutung haben.) Zuletzt speichert das Script den geänderten HTML-Code in einer lokalen Datei.

```
# (Fortsetzung ...)
# Schleife über alle Links
foreach ($link in $response.links) {
    $href = $link.href
    if ($href -match '.*pdf$') {
        Write-Output $href
        # Dateiname aus URL extrahieren
        $filename = $href.Substring($href.LastIndexOf("/") + 1)
        # PDF herunterladen
        Invoke-WebRequest $href -OutFile tmp\$filename
        # PDF in Cloud hochladen
        Write-S3Object -BucketName $bucket -File tmp\$filename `
                       -Key $filename
        # AWS-URL für PDF zusammenstellen
        $pdfAtAws = "$awsurl$filename"
        # Link in HTML-Dokument aktualisieren
        $html = $html.Replace("href=`"$href`"",
                              "href=`"$pdfAtAws`"")
    }
}
# geänderten HTML-Code speichern
$html | Out-File $htmldest
```

## Einschränkungen

Bevor Sie auf die Idee kommen, dieses Script auf eine reale Website anzuwenden, sollten Sie sich mit den nicht unerheblichen Einschränkungen auseinandersetzen:

- Die meisten modernen Websites sind mit einem *Content Management System* realisiert. Der HTML-Code einer Seite setzt sich aus diversen Komponenten zusam-

Persönliches Exemplar für Thomas Bartholomäus

men. Das Script sieht diesen HTML-Code in seiner Gesamtheit. Tatsächlich wollen Sie aber in der Regel nur Teile davon verändern. Dazu müsste das Script Zugriff auf die Einzelseite innerhalb des CMS haben. Dabei kommt oft nicht HTML, sondern Markdown oder eine CMS-spezifische Sprache zum Einsatz.

- Die Direkt-Links auf die PDF-Dateien im AWS-S3-Bucket verwenden nur HTTP, nicht das zeitgemäße Protokoll HTTPS. Das lässt sich ändern, ist aber mit erheblichem Konfigurationsaufwand verbunden. Die Details sind hier dokumentiert:

  *https://docs.aws.amazon.com/AmazonS3/latest/userguide/WebsiteHosting.html*

- Auch das Script selbst könnte noch optimiert werden. Es sollte nur PDF-Links berücksichtigen, die auf Dateien der eigenen Website zeigen. Fremde PDFs bzw. schon ausgelagerte PDFs sollen nicht noch einmal verarbeitet werden.

  Empfehlenswert ist auch eine Doppelgängerkontrolle, die sicherstellt, dass ein mehrfach verlinktes Dokument nur einmal in die Cloud hochgeladen wird.

- Das Script setzt absolute Links voraus (also `href="https://hostname/..."`). HTML erlaubt aber auch relative Links (`href="mydocument.pdf"`). Gegebenenfalls müssen Sie Ihr Script dahingehend ergänzen. Entsprechenden Beispielcode – dort allerdings in Python formuliert – finden Sie in Abschnitt 17.2, »Web Scraping mit regulären Mustern«.

Kurzum: Der Realisierung der simplen Idee dieses Beispiels stehen in der Praxis doch einige Hürden entgegen.


© Rheinwerk Verlag, Bonn 2024

# Kapitel 21
# Virtuelle Maschinen

Solange Sie virtuelle Maschinen nur vereinzelt anwenden, lohnt sich eine Automatisierung nicht. Sie können eventuell in Erwägung ziehen, ein Tool zum rascheren Setup neuer virtueller Maschinen zu verwenden (etwa Vagrant).

Ganz anders ist die Ausgangslage, wenn Sie automatisiert viele virtuelle Maschinen erzeugen, konfigurieren, warten und auswerten wollen – etwa für ein Labor (Unterricht), für den Cluster-Betrieb (Wissenschaft) oder für ein skalierbares Deployment (Container- oder Server-Betrieb). Natürlich gibt es für solche Anwendungsfälle alle möglichen Spezial-Tools, von OpenStack bis Kubernetes. Allerdings ist die Anwendung solcher Programme kompliziert und erfordert eine intensive Einarbeitung. Für einfache Aufgabenstellungen reichen oft ein paar kleine Scripts aus.

Die Scripts in diesem Kapitel beziehen sich auf die Virtualisierungssysteme KVM (Linux) und Hyper-V (Windows).

**Voraussetzungen für dieses Kapitel**

Neben Bash- bzw. PowerShell-Basiswissen brauchen Sie für dieses Kapitel natürlich ein Grundverständnis für das jeweilige Virtualisierungssystem und für die zugrunde liegenden Netzwerktechniken.

Eines der Beispiele setzt die Kommandos `cut`, `grep` und `sed` ein und wendet reguläre Muster an, um Netzwerkkonfigurationsdateien zu ändern. Die dazugehörenden Grundlagen habe ich in Kapitel 8, »Textauswertung mit Filtern und Pipes«, sowie in Kapitel 9, »Reguläre Muster«, behandelt.

Auch die SSH-Authentifizierung mit Schlüssel spielt in den Beispielen eine Rolle. Werfen Sie gegebenenfalls noch einmal einen Blick in Kapitel 12, »SSH«!

## 21.1 Virtuelle Maschinen einrichten und ausführen (KVM)

Der Ausgangspunkt für die folgenden Scripts ist ein Ubuntu-Server. Dort sind das Virtualisierungssystem KVM, das Kommando `virsh` aus dem Paket `libvirt-clients` sowie das Kommando `virt-clone` aus dem gleichnamigen Paket installiert. Das Ziel

Persönliches Exemplar für Thomas Bartholomäus

besteht darin, mehrere gleichartige virtuelle Maschinen auszuführen, die von einer vorhandenen virtuellen Maschine mit dem Namen `vm-base` geklont werden. Das Ausgangssystem `vm-base` verfügt über drei Netzwerkschnittstellen und vier virtuelle Datenträger.

### Virtuelle Maschinen klonen

Das Script `make-vms.sh` erwartet zwei numerische Parameter. Es durchläuft dann eine Schleife vom Startwert bis zum Endwert und erzeugt die virtuellen Maschinen `vm-<nn>`. Das Kommando `make-vms.sh 10 29` erzeugt also 20 virtuelle Maschinen mit den Namen `vm-10` bis `vm-29`.

Das Script testet zuerst, ob zwei Parameter übergeben wurden. Bevor es mit dem Klonen beginnt, stellt das Script sicher, dass die originale virtuelle Maschine (das Klonbasissystem, Variable `orig`) heruntergefahren ist. Es wertet dazu die mit `virsh list` erzeugte Liste aller laufenden virtuellen Maschinen aus.

`virt-clone` erzeugt automatisch die erforderlichen virtuellen Datenträger und verwendet dabei die mit `--file` angegebenen Dateinamen und die mit `--mac` spezifizierten MAC-Adressen (*Media Access Control*, zur Identifizierung von Netzwerkgeräten). Allerdings verwenden diese Datenträger immer das RAW-Imageformat. Die nachfolgenden `qemu-img`-Kommandos wandeln die Image-Dateien in das effizientere QCOW2-Format um.

```
# Beispieldatei make-vms.sh
if [ $# -ne 2 ]; then
    echo "usage: make-vms.sh <start> <end>"
    exit 1
fi
vmstart=$1
vmend=$2
orig='vm-base'     # base VM to clone

# Klonbasissystem herunterfahren
result=$(virsh list | grep $orig)
if [ ! -z "$result" ]; then
  echo "shutting down $orig"
  virsh shutdown $orig
  sleep 10
fi

# VMs erzeugen
for (( nr=$vmstart; nr<=$vmend; nr++ )); do
  echo "create vm-$nr"
  disk1=/var/lib/libvirt/images/vm-$nr-disk1.qcow2
```


© Rheinwerk Verlag, Bonn 2024

```
  disk2=/var/lib/libvirt/images/vm-$nr-disk2.qcow2
  disk3=/var/lib/libvirt/images/vm-$nr-disk3.qcow2
  disk4=/var/lib/libvirt/images/vm-$nr-disk4.qcow2
  tmpdisk=/var/lib/libvirt/images/tmpdisk.qcow2
  virt-clone --name "vm-$nr" --original $orig \
    --mac 52:54:00:01:00:$nr --mac 52:54:00:02:00:$nr \
    --mac 52:54:00:03:00:$nr \
    --file $disk1 --file $disk2 --file $disk3 --file $disk4

  # RAW-Disks in QCOW2-Disks konvertieren
  qemu-img convert $disk1 -O qcow2 $tmpdisk
  mv $tmpdisk $disk1
  qemu-img convert $disk2 -O qcow2 $tmpdisk
  mv $tmpdisk $disk2
  qemu-img convert $disk3 -O qcow2 $tmpdisk
  mv $tmpdisk $disk3
  qemu-img convert $disk4 -O qcow2 $tmpdisk
  mv $tmpdisk $disk4
done
```

### Virtuelle Maschinen starten und herunterfahren

`make-vms.sh` erzeugt die virtuellen Maschinen `vm-<nn>` nur, startet diese aber nicht. Diese Aufgabe übernimmt ein weiteres Script `start-vms.sh`, das wiederum zwei Zahlen als Parameter erwartet. Es führt `virsh start <name>` aus, um die betreffende virtuelle Maschine zu starten.

```
# Beispieldatei start-vms.sh
vmstart=$1
vmend=$2
for (( nr=$vmstart; nr<=$vmend; nr++ )); do
  echo "start vm-$nr"
  virsh start "vm-$nr"
done
```

Daneben gibt es zwei analoge Scripts, um die virtuellen Maschinen herunterzufahren (`virsh shutdown`) bzw. um diese zu löschen und dabei alle virtuellen Datenträger zu löschen (`virsh undefine --remove-all-storage`).

### Scripts auf mehreren virtuellen Maschinen ausführen

Nachdem Sie 20 virtuelle Maschinen zum Laufen gebracht haben, fällt Ihnen auf, dass Sie ein Konfigurationsdetail vergessen haben. Sie könnten sich nun auf jeder der virtuellen Maschinen mit SSH anmelden und die Konfiguration vervollständigen. Aber natürlich gibt es eine elegantere Lösung: Mit `run-script-on-vms.sh` führen Sie die

Persönliches Exemplar für Thomas Bartholomäus

in myscript.sh gespeicherten Kommandos per SSH auf allen gewünschten virtuellen Maschinen aus.

run-script-on-vms.sh setzt voraus, dass es auf Ihrem lokalen Server ein SSH-Schlüsselpaar gibt und dass der öffentliche Schlüssel im root-Account der virtuellen Maschinen bekannt ist. Dazu müssen Sie – natürlich vor dem Klonen! – auf vm-base SSH-Logins für root erlauben und den lokalen Schlüssel mit ssh-copy-id root@basevm dorthin kopieren. Anstelle von basevm geben Sie den Hostnamen oder die IP-Adresse der virtuellen Maschine an.

run-script-on-vms.sh ist verblüffend kurz. Im Prinzip wird für jede virtuelle Maschine in einer Schleife das Kommando ssh root@host < myscripts.sh > result.txt ausgeführt. Anstelle von host müssen im Script der Host-Name (hier vm-<nn>.example.com) oder die IP-Adresse der betreffenden virtuellen Maschine angegeben werden. Die Option -o StrictHostKeyChecking=no bewirkt, dass SSH auf die Rückfrage verzichtet, ob einem Host vertraut werden soll, zu dem erstmalig eine Verbindung hergestellt wird.

```
# Beispieldatei run-script-on-vms.sh
vmstart=$1
vmend=$2
for (( nr=$vmstart; nr<=$vmend; nr++ )); do
    ssh -o StrictHostKeyChecking=no \
      root@vm-$nr.example.com 'bash -s' \
      < myscript.sh > results-$nr.txt
done
```

## 21.2 Netzwerkkonfiguration automatisieren (KVM)

Eine virtuelle Maschine zu »klonen«, bedeutet, dass sämtliche Eigenschaften des Ausgangssystems erhalten bleiben. Geklont werden daher auch alle Konfigurationsdateien. In den meisten Fällen ist genau das wünschenswert, aber es gibt Ausnahmen. Eine betrifft die statische Netzwerkkonfiguration. Soweit die Netzwerkadapter Ihre Adressen nicht automatisiert via DHCP beziehen, müssen die Netzwerkkonfigurationsdateien jeder virtuellen Maschine angepasst werden – sonst gibt es Netzwerkkonflikte.

Diese Aufgabe übernimmt ein weiteres Script, das sich allerdings nicht auf dem Virtualisierungs-Host befindet, sondern *in* der virtuellen Maschine. Es muss also im Klon-Basissystem (laut den Namen des vorigen Beispiels in vm-base) ein Script geben, das beim Hochfahren der virtuellen Maschine bzw. seiner Klone ausgeführt wird.

In meinem Setup für den Linux-Unterricht sind die virtuellen Maschinen kompatibel zu Red Hat Enterprise Linux 9. (Ich verwende AlmaLinux, aber RHEL 9,


© Rheinwerk Verlag, Bonn 2024

Oracle Linux 9 oder Rocky Linux 9 funktionieren diesbezüglich exakt gleich.) In der virtuellen Maschine `vm-base` gibt es das Script `/etc/myscripts/setup-vm-network`. Es wird bei jedem Boot-Prozess ausgeführt. Verantwortlich dafür ist die Datei `/etc/rc.d/rc.local`, die wie folgt aussieht:

```
#!/bin/bash
touch /var/lock/subsys/local
. /etc/myscripts/setup-vm-network
```

Sie müssen diese Datei mit `chmod +x /etc/rc.d/rc.local` ausführbar machen, damit sie berücksichtigt wird.

### Ausgangspunkt

Das Bash-Script `setup-network` setzt voraus, dass es für zwei Netzwerkadapter bestehende Konfigurationsdateien gibt:

```
/etc/NetworkManager/system-connections/enp1s0
/etc/NetworkManager/system-connections/enp7s0
```

Die Dateien verwenden die Syntax des Linux-NetworkManagers. Die Dateien enthalten unter anderem die folgenden Zeilen:

```
# statische IPv4-Konfiguration mit 192.168.122.1 als Gateway
address1=192.168.122.27/24,192.168.122.1
# statische IPv6-Konfiguration mit 2a01:abce:abce::2 als Gateway
address1=2a01:abcd:abcd::27/64,2a01:abce:abce::2
```

Diese Dateien sollen so angepasst werden, dass jede virtuelle Maschine eine eindeutige IPv4- und IPv6-Adresse hat. Dazu werden die letzten zwei Stellen der MAC-Adresse des ersten Netzwerkadapters auswertet. Lautet die MAC-Adresse z. B. `52:54:00:01:00:27`, dann soll die virtuelle Maschine die folgenden IP-Adressen verwenden:

- IPv4: `192.168.122.27`
- IPv6: `2a01:abcd:abce::27`

In den ersten Zeilen des Scripts werden einige Variablen initialisiert. Danach wertet das Script die Systemdatei `/sys/class/net/enp1s0/address` aus, die die MAC-Adresse des ersten Adapters enthält. `cut` extrahiert daraus die sechste hexadezimale Gruppe. Die `if`-Anweisung eliminiert eine führende 0, macht also beispielsweise aus `07` einfach `7`.

Die Variablen `ipv4old` bzw. `ipv4new` sowie `ipv6old` bzw. `ipv6new` enthalten ein Muster für die bisherige IP-Adresse sowie die erforderliche richtige IP-Adresse. Wenn das Script mit `grep` feststellt, dass die aktuelle Netzwerkkonfiguration nicht mit der

Persönliches Exemplar für Thomas Bartholomäus

Wunschadresse übereinstimmt, werden beide Konfigurationsdateien mittels sed verändert. Vereinfacht dargestellt haben die sed-Kommandos die folgende Wirkung:

- conffile1: 192.168.122.*/24 wird ersetzt durch 192.168.122.<nn>/24
- conffile2: 2a01:abcd:abcd::.*/64 wird ersetzt durch 2a01:abcd:abcd::<nn>/64

Im Anschluss daran löscht das Script die Datei /etc/machine-id und richtet sie dann mit einer zufälligen ID neu ein. Auch diese Datei hat mit der Netzwerkkonfiguration zu tun. Sie wird vom NetworkManager (einer Linux-Systemkomponente) ausgewertet und dazu verwendet, IPv6-Link-Local-Unicast-Adressen zu erzeugen (fe80-xxx). Wenn alle virtuellen Maschinen die gleiche interne ID-Nummer haben, dann stimmen auch die Unicast-Adressen überein und es kommt – trotz ansonsten korrekter IPv6-Konfiguration – zu Adresskonflikten.

Bei Bedarf können Sie das Script natürlich um weitere Funktionen ergänzen, z. B. zum Erzeugen neuer Keys für den OpenSSH-Server oder zur Einstellung des Hostnamens.

```
# Beispieldatei setup-vm-network.sh
NMPATH=/etc/NetworkManager/system-connections
IF1=enp1s0
IF2=enp7s0

# Ort der Netzwerkkonfigurationsdateien
conffile1=$NMPATH/$IF1.nmconnection
conffile2=$NMPATH/$IF2.nmconnection

# extrahiert die letzten 2 MAC-Stellen, eliminiert führende 0
mac=$(cut -d ':' -f 6 /sys/class/net/$IF1/address)
if [ ${mac:0:1} == 0 ]; then mac=${mac:1:2}; fi

# IPv4- und IPv6-Adressen: old = bisher, new = gewünscht
ip4old="192\.168\.122\..*/24"
ip4new="192\.168\.122\.$mac/24"
ip6old="2a01:abcd:abcd::.*/64"
ip6new="2a01:abcd:abcd::$mac/64"

# falls die Konfigurationsdatei eine von ip4new abweichende
# Adresse verwendet: Konfigurationsdateien mit sed korrigieren
if ! grep -q "address1=$ip4new" $conffile1; then
  sed -E -i.old  "s,$ip4old,$ip4new," $conffile1
  sed -E -i.old  "s,$ip6old,$ip6new," $conffile2
  # /etc/machine-id neu einrichten
  rm /etc/machine-id
  systemd-machine-id-setup
```


© Rheinwerk Verlag, Bonn 2024

```
  # virtuelle Maschine neu starten
  echo "reboot"
  reboot
else
  echo "no network changes"
fi
```

Das Script endet mit einer `reboot`-Anweisung. Wenn Sie ein Script wie in diesem Beispiel selbst entwickeln, müssen Sie mit `reboot` äußerst vorsichtig umgehen! Wenn Ihr Script nicht richtig funktioniert, wird die virtuelle Maschine ununterbrochen neu gestartet. Testen Sie Ihr Script also ausführlich, bevor Sie `reboot` einbauen!

Da das Script die letzten beiden Stellen der MAC-Adresse dezimal (nicht hexadezimal) verarbeitet, ist es zur Administration von 100 virtuellen Maschinen geeignet. 192.168.122.0 ist reserviert. 192.168.122.1 sowie 2a01:abce:abce::2 werden als Gateway-Adressen verwendet. Damit verbleibt für IPv4 der Adressbereich 192.168.122.3 bis .99. Sie können also maximal 97 virtuelle Maschinen einrichten. Dieses Limit können Sie bei Bedarf umgehen, indem Sie die letzten zwei MAC-Ziffern hexadezimal auswerten oder mehrere MAC-Ziffern berücksichtigen.

**Alternativen**

Die Ausführung eines Scripts im Rahmen des Init-Systems ist nicht der einzige Weg, um virtuelle Maschinen zu konfigurieren. Große Virtualisierungs-Frameworks wie OpenStack setzen auf Cloud-Init (siehe *https://cloud-init.io*).

Zur Konfiguration laufender virtueller (oder realer) Maschinen können Sie auch Konfigurations-Tools wie Puppet oder Ansible verwenden. Diese Programme setzen aber voraus, dass alle Maschinen im Netzwerk erreichbar sind, dass die Netzwerkkonfiguration also bereits abgeschlossen ist.

## 21.3 Hyper-V steuern

Hyper-V ist für Windows, was KVM für Linux ist. Deswegen verwundert es nicht, dass Microsoft dem hauseigenen Virtualisierungssystem ein umfassendes PowerShell-Modul spendiert hat. Sofern Sie über Windows Pro verfügen und Hyper-V noch nicht aktiviert haben, gelingt dies am schnellsten mit dem folgenden Kommando:

```
> Add-WindowsFeature Hyper-V  -IncludeManagementTools
```

Über das PowerShell-Modul `Hyper-V` können nun unter Windows (nicht aber unter Linux oder macOS) annähernd 250 Aliase und CmdLets ausgeführt werden:

Persönliches Exemplar für Thomas Bartholomäus

```
> Get-Command -Module Hyper-V

  CommandType  Name                     Version   Source
  -----------  ----                     -------   ------
  Alias        Export-VMCheckpoint      2.0.0.0   Hyper-V
  Alias        Get-VMCheckpoint         2.0.0.0   Hyper-V
  Alias        Remove-VMCheckpoint      2.0.0.0   Hyper-V
  Alias        Rename-VMCheckpoint      2.0.0.0   Hyper-V
  Alias        Restore-VMCheckpoint     2.0.0.0   Hyper-V
  Cmdlet       Add-VMAssignableDevice   2.0.0.0   Hyper-V
  Cmdlet       Add-VMDvdDrive           2.0.0.0   Hyper-V
  ...
```

### Admin-Rechte erforderlich

Standardmäßig setzt die Ausführung von Hyper-V-CmdLets administrative Rechte voraus. Sie müssen also ein PowerShell-Terminal mit Administratorrechten öffnen.

Alternativ können Sie einzelne Benutzer oder Gruppen zu den Hyper-V-Administratoren hinzufügen. Derartige Einstellungen nehmen Sie im Gruppenrichtlinienverwaltungs-Editor vor.

`Get-VM` listet alle installierten virtuellen Maschinen auf und verrät Details zu deren aktuellem Zustand:

```
> Get-VM

  Name    State     CPUUsage(%)   MemoryAssigned(M)   ...
  ----    -----     -----------   -----------------   -----
  alma9   Running   24            2024
  kali    Running   0             5976
  ...
```

`Get-VM` liefert `VirtualMachine`-Objekte zurück. `Get-Member` zeigt, dass die zugrunde liegende Klasse unzählige Eigenschaften ausweist:

```
> Get-VM | Select-Object -First 1 | Get-Member

  TypeName: Microsoft.HyperV.PowerShell.VirtualMachine

  Name                          MemberType    Definition
  ----                          ----------    ----------
  CheckpointFileLocation        AliasProperty ...
  VMId                          AliasProperty ...
  VMName                        AliasProperty ...
  Equals                        Method        ...
```



© Rheinwerk Verlag, Bonn 2024

```
GetHashCode                        Method        ...
AutomaticCheckpointsEnabled        Property      ...
AutomaticCriticalErrorAction       Property      ...
AutomaticStartAction               Property      ...
AutomaticStartDelay                Property      ...
...
```

Das Starten oder Herunterfahren aller virtuellen Maschinen gelingt mit zwei Einzeilern, die die CmdLets `Start-VM` bzw. `Stop-VM` aufrufen:

```
> Get-VM | Where-Object {$_.State -eq 'Off'} | Start-VM

> Get-VM | Where-Object {$_.State -eq 'Running'} | Stop-VM
```

Mit dem ausgesprochen praktischen CmdLet `Set-VM` können Sie diverse Eigenschaften von virtuellen Maschinen verändern:

```
> $vm = Get-VM "alma9-clone1"
> Set-VM -VM $vm -MemoryStartupBytes 2GB -ProcessorCount 2
```

Die Veränderung des Arbeitsspeichers ist unmöglich, wenn die virtuelle Maschine einen dynamisch zugewiesenen Speicherbereich verwendet. In diesem Fall können Sie die Ober- und die Untergrenze sowie die anfängliche Speichergröße einstellen, wobei diese nicht größer als die Obergrenze sein darf. Mit der Option `-DynamicMemory` können Sie von der statischen auf eine dynamische Speicherverwaltung umstellen.

```
> Set-VM -VM $vm -DynamicMemory -MemoryStartupBytes 512MB `
    -MemoryMinimumBytes 512MB -MemoryMaximumBytes 1GB
```

Eine Referenz aller Hyper-V-CmdLets mit ihren unzähligen Optionen finden Sie wie üblich online:

*https://learn.microsoft.com/en-us/powershell/module/hyper-v*

### Virtuelle Maschine klonen

Das `Hyper-V`-Modul stellt kein eigenes CmdLet zum Klonen virtueller Maschinen zur Verfügung. Diese Aufgabe kann aber über einen Umweg erledigt werden: Dazu exportieren Sie die virtuelle Maschine zuerst (`Export-VM`) und erstellen die neue virtuelle Maschine dann über einen Import (`Import-VM`).

In der Praxis ist der Vorgang allerdings komplexer, als diese kurze Zusammenfassung vermuten lässt. Das folgende Script erstellt `noOfClones` Kopien einer vorhandenen virtuellen Maschine. Das Script beginnt damit, die Basis-VM herunterzufahren und alle Snapshots zu löschen. (Entfernen Sie `Remove-VMSnapshot`, wenn Sie die Snapshots erhalten möchten! Beachten Sie aber, dass `Export-VM` auch alle Snapshots umfasst und dass es keine Option gibt, dies zu verhindern.)

Persönliches Exemplar für Thomas Bartholomäus

```
# Beispieldatei clone-vm.ps1
$basename = "alma9"
$noOfClones = 3
$tmp = $env:TEMP  # Vorsicht: $env:TEMP funktioniert nur unter
                  # Windows, nicht unter Linux/macOS

# die Funktion testet, ob ein Verzeichnis existiert, und
# löscht es dann; Vorsicht!
function delete-dir($path) {
    if (Test-Path "$path") {
        Write-Output "delete $path"
        Remove-Item "$path" -Recurse -Force
    }
}

# wenn die VM läuft: herunterfahren
$basevm = Get-VM $basename
if ($basevm.State -eq 'Running' ) {
    Write-Output "shutdown $basename"
    Stop-VM -VM $basevm
}
# alle Snapshots der VM löschen (Vorsicht!)
Get-VMSnapshot -VM $basevm | Remove-VMSnapshot
```

Die nächsten Zeilen ermitteln mit `Get-VMHardDiskDrive` den Ort, in dem sich die erste Disk der Basis-VM befindet. Die Disks der geklonten VMs werden später relativ dazu in Unterverzeichnissen angelegt.

Falls das temporäre Verzeichnis, in das der Export durchgeführt werden soll, bereits existiert (z. B. als Überbleibsel von einem vorherigen Script-Aufruf), dann wird es gelöscht.

`Export-VM` exportiert schließlich die Basis-VM. Dabei entsteht im temporären Verzeichnis ein Unterverzeichnis mit dem Namen der virtuellen Maschine. Die eigentliche Beschreibung der virtuellen Maschine befindet sich in einer `*.vmcx`-Datei. `Get-ChildItem` ermittelt den Dateinamen.

```
# (Fortsetzung ...)
# ermittelt das Verzeichnis, in dem das erste Disk Image
# gespeichert ist
$pathFirstDisk =  (Get-VMHardDiskDrive -VM $basevm |
                   Select-Object -First 1).Path
$dirFirstDisk = Split-Path $pathFirstDisk -Parent
Write-Output "Virtual disk directory: $dirFirstDisk"

# temporäres Export-Verzeichnis löschen (falls es existiert)
delete-dir "$tmp\$basename"
```


© Rheinwerk Verlag, Bonn 2024

```
# Export durchführen
Export-VM -VM $basevm -Path $tmp
$vmcxfile = Get-ChildItem `
    "$tmp\$basename\Virtual Machines\*.vmcx"
Write-Output "VMCX file: $vmcxfile"
```

In einer Schleife werden nun mehrere Klone der virtuellen Maschine erzeugt. Die Option `-Copy` bedeutet, dass dabei wirklich eine neue VM mit eigenen Dateien erzeugt wird. `-GenerateNewId` gibt der VM eine Hyper-V-eigene Identifikationsnummer. Die beiden Optionen `-XxxPath` geben an, wo die Dateien der virtuellen Maschine gespeichert werden sollen.

`Rename-VM` gibt der VM einen neuen Namen. Hyper-V hat zwar kein Problem damit, dass mehrere VMs den gleichen Namen haben, allerdings stiften gleichnamige VMs nur Verwirrung.

Neuen VMs weist Hyper-V automatisch 2 CPUs sowie 2 GByte Speicher zu. `Set-VM` reduziert diese Werte. Außerdem legt `Set-VMNetworkAdapter` für jede VM eine statische MAC-Adresse fest. Standardmäßig verwendet Hyper-V dynamische MACs, die erst beim Start einer VM generiert werden. Die ersten sechs Stellen der MAC sollten immer `00155d` lauten. Dieser Teil der MAC wurde von Microsoft für Hyper-V reserviert.

```
# (Fortsetzung ...)
# $noOfClones Kopien der exportierten VM erzeugen
for ($i = 1; $i -le $noOfClones; $i++) {
    $cloneName = "${basename}-clone${i}"
    $mac="00155d1234{0:d2}" -f $i
    Write-Output "setup $cloneName with MAC $mac"

    $clone = Import-VM -Path $vmcxfile -Copy -GenerateNewId `
        -VhdDestinationPath "$dirFirstDisk\$cloneName" `
        -VirtualMachinePath "$dirFirstDisk\$cloneName"

    # den Klon umbenennen
    Rename-VM -VM $clone -NewName $cloneName

    # Eigenschaften der VM einstellen
    Set-VM -VM $clone -ProcessorCount 1 `
      -MemoryStartupBytes 1GB

    # statische MAC-Adresse einstellen
    $mac="00155d1234{0:d2}" -f $i
    Set-VMNetworkAdapter -VMName $cloneName -StaticMacAddress
    $mac
}
```

Persönliches Exemplar für Thomas Bartholomäus

```
# temporäres Export-Verzeichnis löschen
delete-dir "$tmp\$basename"
```

Beachten Sie, dass das Script mit administrativen Rechten ausgeführt werden muss. Das Script liefert einen Fehler, wenn die VM-Dateien eines Klons bereits existieren – also z. B., wenn das Script ein zweites Mal ausgeführt wird. Wenn Sie möchten, können Sie das Script dahingehend erweitern, dass Sie bereits vorhandene Klone im Script erkennen und löschen.


© Rheinwerk Verlag, Bonn 2024

# Kapitel 22
# Docker und Scripting

Docker ist ein System zur Erzeugung und Verwaltung von Containern. Das sind (teilweise) vom Betriebssystem isolierte Software-Umgebungen. Docker hat sich in den letzten Jahren vor allem auf Entwicklerrechnern etabliert. Sie wollen schnell ein Node.js-Programm ausprobieren? `docker run node` stellt die Testumgebung zur Verfügung! (Ich gebe zu, *ganz* so einfach ist es in der Praxis nicht, aber nahezu.)

Mit Docker können Sie mit viel geringerem Overhead als bei virtuellen Maschinen ein reproduzierbares Setup schaffen. Diese aus diversen Komponenten bestehende Umgebung können Sie dann bei Bedarf rasch auf andere Rechner übersiedeln, auch über Plattform-Grenzen hinweg. Gleichzeitig sind Docker bzw. vergleichbare Tools wie Podman oder Kubernetes inzwischen eine Schlüsselkomponente für stark skalierbare bzw. häufig veränderbare Server-Infrastrukturen.

In diesem Kapitel zeige ich Ihnen anhand von Beispielen, wie stark Scripting und Docker miteinander zusammenhängen:

- Das `Dockerfile`, das die Zusammensetzung eines eigenen Image beschreibt, kann Bash-Anweisungen enthalten, z. B. zur Installation von Software-Komponenten. Der Script-Code ist damit quasi Teil des Dockerfiles.
- Docker macht eine Software-Lösung, die aus diversen vorinstallierten Komponenten plus ein paar eigenen Scripts besteht, »transportabel«. Sie können also quasi eine Komplettlösung anbieten, die unkompliziert auf jedem Betriebssystem läuft (Docker vorausgesetzt).

### Voraussetzungen

Die wichtigste Voraussetzung für dieses stark beispielorientierte Kapitel besteht darin, dass Sie Docker gut kennen und auf Ihrem Rechner installiert haben. Da die Basis der eigenen Beispiel-Images jeweils Linux ist, ist etwas Linux-Grundwissen ebenfalls sehr hilfreich. Als einzige Scripting-Sprache kommt die Bash zum Einsatz.

Persönliches Exemplar für Thomas Bartholomäus

## 22.1 Beispiel: EXIF-Sorter als Docker-Image

In Kapitel 16, »Bildverarbeitung«, habe ich Ihnen ein zehnzeiliges Bash-Script vorgestellt, das Fotos anhand ihres Datums sortiert. Unter Linux ist dessen Anwendung recht einfach, weil sich das zugrunde liegende `exiftool` unkompliziert aus den Paketquellen installieren lässt. Unter Windows und macOS ist die Installation aber umständlicher.

Wenn Sie sowieso Docker verwenden, können Sie das Script in ein Docker-Image verpacken und dieses im Docker-Hub speichern. In der Theorie können nun Sie und andere Anwenderinnen und Anwender das gesamte Setup mit ein, zwei Docker-Kommandos herunterladen und ausführen. Wie Sie gleich sehen, stimmt das prinzipiell, auch wenn in der Praxis ein paar Hürden zu überwinden sind.

### Das Script adaptieren

Das Hauptproblem bei diesem Beispiel besteht darin, dass in Docker ausgeführter Code nicht ohne Weiteres auf das Dateisystem des Host-Computers zugreifen kann. Das EXIF-Script muss aber lokale Dateien lesen, neue Verzeichnisse erstellen und die Bilder dann dorthin verschieben. Die Lösung für dieses Problem sind Volumes, also lokale Verzeichnisse, die mit einem Docker-Container verbunden werden.

Eine zweite Hürde ist der Umgang mit Dateimustern: Während die Shell in Linux und macOS aus `*.jpg` eine Liste mit den entsprechenden Dateinamen macht, funktioniert das in der PowerShell nicht. Aus diesem Grund habe ich das ursprüngliche Script ein wenig adaptiert. Es erwartet jetzt keine Parameter mehr, sondern verarbeitet einfach alle JPEG-Dateien im aktuellen Verzeichnis.

```
# Beispieldatei sort-photos.sh
echo "exif sorter 1.0"

# *.JPG löst keinen Fehler aus, wenn es keine JPG-Datei gibt
shopt -s nullglob

# Schleife über alle JPEG-Dateien
for file in *.jpeg *.jpg *.JPEG *.JPG; do
    echo "$file"
    yearmonth=$(exiftool -s3 -d '%Y-%m' \
                         -DateTimeOriginal "$file")
    if [ $yearmonth ]; then
        echo "$file -> $yearmonth/"
        mkdir -p $yearmonth
        mv "$file" $yearmonth/
    fi
done
```


© Rheinwerk Verlag, Bonn 2024

`shopt -s nullglob` bewirkt, dass nicht zutreffende Globbing-Muster eliminiert werden. Ohne diese Option würde z. B. `'*.JPG'` als Parameter an `exiftool` übergeben. `exiftool` würde dann einen Fehler liefern, dass es die Datei `'*.JPG'` gar nicht gibt.

### Das Docker-Image zusammenstellen

Nun geht es darum, dieses Script in ein Docker-Image zu integrieren. Ich habe mich dazu entschieden, Ubuntu 22.04 als Basis zu verwenden. Die Datei `Dockerfile` beschreibt, welche Änderungen das eigene Image im Vergleich zum Ausgangs-Image aufweisen soll:

- Mit `RUN` können Sie nun diverse Linux-Kommandos übergeben, die beim Erstellen des Images ausgeführt werden. Die Trennung durch `&&` bewirkt, dass jedes Kommando nur dann ausgeführt wird, wenn das vorige Kommando fehlerfrei war. Wenn Sie so wollen, ist die an `RUN` übergebene Zeichenkette selbst ein Mini-Script.

  `apt-get` installiert das Paket `exiftool`. Damit das resultierende Image nicht unnötig groß wird, werden danach alle temporären Dateien gelöscht (`apt-get clean` und `rm`).
- `COPY` kopiert das zuvor abgedruckte Script aus dem lokalen Verzeichnis (also aus demselben Verzeichnis, in dem auch das `Dockerfile` gespeichert ist) in das neue Image.
- `VOLUME` legt fest, dass das lokale Verzeichnis `/data` als *Volume* mit einem Verzeichnis des Docker-Host-Systems verbunden werden muss.
- `WORKDIR` macht dieses Verzeichnis für die Ausführung des Docker-Containers zum Default-Verzeichnis.
- `CMD` legt fest, dass beim Start des Containers sofort das Script `sort-photos.sh` ausgeführt werden soll. Sobald das Script fertig ist, endet das aktive »Leben« des Containers.

```
# Datei Dockerfile
# Ubuntu 22.04 als Basis
FROM ubuntu:22.04

# lokales Verzeichnis /data erstellen, exiftool installieren
RUN mkdir /data && \
    apt-get update && \
    apt-get install -y exiftool && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# das Script ./sort-photos.sh aus dem lokalen Verzeichnis in das
# Image kopieren
COPY sort-photos.sh /usr/local/bin
```

Persönliches Exemplar für Thomas Bartholomäus

```
# /data als 'Volume' und als Default-Verzeichnis verwenden
VOLUME ["/data"]
WORKDIR /data

# beim Start des Containers sort-photos.sh ausführen
CMD ["/usr/local/bin/sort-photos.sh"]
```

### Docker-Image erstellen und in den Docker Hub hochladen

Der nächste Schritt besteht darin, aus dem `Dockerfile` das entsprechende Image zu erzeugen und zu testen:

```
$ docker buildx build -t koflerinfo/exif-sorter .
```

Wenn das Script bei lokalen Tests (siehe dazu auch den folgenden Abschnitt) zufriedenstellend funktioniert, können Sie es in den Docker Hub hochladen. Dazu brauchen Sie ein Konto bei Docker. Mit `docker login` melden Sie sich dort an.

Damit Ihr Image nicht nur auf Rechnern mit der gleichen CPU-Architektur wie auf Ihrem eigenen Rechner funktioniert, ist es empfehlenswert, gleich einen Multiarch-Build durchzuführen. Wenn Sie z. B. wie ich normalerweise auf einem Linux-Notebook mit Intel- oder AMD-CPU arbeiten, kann das Image später auch auf dem Raspberry Pi oder auf einem Apple-Rechner mit einer ARM-CPU (»Apple Silicon«) nativ ausgeführt werden. Die erforderlichen Kommandos habe ich im Folgenden zusammengefasst. Naturgemäß müssen Sie `koflerinfo/exif-sorter` durch den Namen Ihres Docker-Accounts sowie den Namen Ihres Images ersetzen.

```
$ docker login

$ docker buildx create --name multiarch \
    --driver docker-container --use

$ docker buildx build --push \
    --platform linux/arm/v7,linux/arm64/v8,linux/amd64 \
    -t koflerinfo/exif-tool .
```

### Image testen bzw. anwenden

Die Anwendung des Images setzt voraus, dass auf dem jeweiligen Rechner Docker installiert ist. Nun wechseln Sie in das Verzeichnis, in dem sich Ihre Fotos befinden, und führen `docker run` aus. Die genauen Optionen hängen davon ab, unter welchem Betriebssystem Sie arbeiten.

Ich beginne hier mit macOS. Die Option `--rm` bewirkt, dass der Container nach der Ausführung sofort gelöscht wird. (Das einmal heruntergeladene Image bleibt auf Ihrem Rechner. Mit jedem `docker run` wird automatisch ein neuer Container erzeugt.)



© Rheinwerk Verlag, Bonn 2024

Die Option `-v` verbindet das aktuelle Verzeichnis (das Ergebnis von `pwd`) mit dem Verzeichnis `/data` im Container. Der letzte Parameter gibt schließlich den Namen des Images an.

```
$ cd path/to/images
$ docker run --rm -v $(pwd):/data koflerinfo/exif-sorter
```

Wenn Sie unter Windows in einem Terminal mit der PowerShell arbeiten, sieht das Kommando ganz ähnlich aus. Der einzige Unterschied besteht darin, dass der Zugriff auf das aktuelle Verzeichnis nun in der Form `${PWD}` erfolgt.

```
> docker run --rm -v ${PWD}:/data koflerinfo/exif-sorter
```

Am umständlichsten ist der Aufruf leider unter Linux. Im Vergleich zu macOS kommt die Option `-u` hinzu, die Docker anweist, neue Dateien und Verzeichnisse mit den Rechten des aktuellen Benutzers zu erzeugen. (Standardmäßig verwendet Docker unter Linux `root`-Rechte. Das macht aber die weitere Nutzung der Dateien und Verzeichnisse mühsam.)

```
$ docker run --rm -v $(pwd):/data -u $UID:$GID \
    koflerinfo/exif-sorter
```

In allen drei Fällen führt Docker nun innerhalb des Containers das Script `sort-photos.sh` aus, verarbeitet alle Bilder im gerade aktuellen Verzeichnis, erzeugt neue Unterverzeichnisse wie `2023-12` und verschiebt Fotos, die in diesem Monat entstanden sind, dorthin.

### Fazit

Docker einmal vorausgesetzt, funktioniert die Weitergabe des Scripts samt allen Voraussetzungen wunderbar. Der große Wermutstropfen ist der umständliche Aufruf des Containers. Selbst Docker-Profis werden sich die erforderlichen Optionen kaum auswendig merken.

## 22.2 Beispiel: Markdown/Pandoc-Umgebung

Den Text dieses Buchs habe ich in Markdown-Dateien verfasst, je eine Datei für jedes Kapitel. Das Programm Pandoc macht daraus zwei Versionen: Eine HTML-Version, aus der der Rheinwerk Verlag dann E-Book-Dateien produziert (EPUB, Mobi), und eine LaTeX-Version, aus der ich selbst die Druckdatei erzeuge (PostScript bzw. PDF).

Ganz so einfach, wie es hier klingt, ist die Angelegenheit aber leider nicht: So fehlen in Markdown (auch in der erweiterten Pandoc-Variante) Steuerungselemente, um alle Layoutwünsche des Rheinwerk Verlags nachzubilden. Da, wo Markdown/Pandoc zu kurz greift, habe ich den Build-Prozess durch eigene Scripts ersetzt. Beispielsweise

Persönliches Exemplar für Thomas Bartholomäus

habe ich Markdown um eine eigene Syntax ergänzt, um Tastensymbole oder Menükommandos im Text abzubilden. [Strg]+[X] oder DATEI • ÖFFNEN formuliere ich in meinem Markdown-Code so:

```
`[Strg]`+`[X]` oder *°Datei°Öffnen°*
```

Das ist der Punkt, wo Scripting ins Spiel kommt:

- Eine ganze Reihe von Scripts manipulieren Markdown-, HTML- und LaTeX-Dateien vor und nach dem Aufruf von `pandoc` und `latex`.
- Mehrere Scripts kümmern sich darum, dass der mehrteilige Prozess zum Erstellen neuer HTML- und PDF-Varianten meines Texts automatisch startet, sobald ich eine Markdown-Datei speichere.
- Ein weiteres Script überwacht das `images`-Verzeichnis: Sobald ich dort einen neuen Screenshot speichere, erzeugt das Script sofort die dazugehörende EPS-Variante. Die ist erforderlich, damit LaTeX eine PostScript-Version des Buchs erzeugen kann.

Was hat nun Docker damit zu tun? Das erforderliche Setup, damit ich mit einem Kommando aus allen Markdown-Dateien dieses Buchs die fertige Druckdatei erzeugen kann, ist erheblich komplexer, als es hier den Anschein hat. Insgesamt sind rund 30 Scripts im Einsatz! Diese verlassen sich teilweise darauf, dass eine ganz bestimmte Version von Pandoc zum Einsatz kommt. Außerdem brauche ich eine komplette LaTeX-Umgebung, diverse Schriften, Zusatztools wie `imagemagick` usw.

Also habe ich mit Unterstützung meines Freunds und Docker-Experten Bernd Öggl ein eigenes Docker-Image zusammengesetzt. Es enthält die gesamte erforderliche Software. Alle weiteren Dateien des Buchprojekts befinden sich in einem Verzeichnis. Anstatt nun ein Script zu starten, führe ich `docker compose up` aus. Es erzeugt aus dem Image einen neuen Container und startet parallel drei Auto-Build-Scripts, die wiederum andere Scripts aufrufen. Erst durch den Einsatz von Docker lässt sich mein Setup relativ unkompliziert für verschiedene Bücher adaptieren, von einem Rechner zum nächsten transportieren usw.

### Beispiel ausprobieren

Für das hier präsentierte Beispiel habe ich mein Markdown-Setup so gut wie möglich abgespeckt. Einige Teile musste ich aus Copyright-Gründen entfernen. So verwende ich diverse Schriften, die ich nicht im Rahmen der Beispieldateien weitergeben darf.

Damit Sie das Beispiel ausprobieren können, wechseln Sie in das Verzeichnis mit den Beispieldateien und erzeugen das Docker-Image:

```
$ cd sample-files/markdown-setup/docker-file
$ docker build . -t my-pandoc-image
```


© Rheinwerk Verlag, Bonn 2024

Der Build-Prozess inklusiver aller Downloads dauert eine Viertelstunde. Das resultierende Image wird lokal auf Ihrem Rechner gespeichert (nicht im Docker Hub). Es ist rund 3 GByte groß.

Falls Sie einen Mac mit M1/M2/M3-CPU verwenden, müssen Sie vor dem Start von `docker build` ein `RUN`-Kommando im `Dockerfile` ändern. Damit wird eine für diese Architektur kompilierte Version von `pandoc` installiert. (Die erforderliche Anweisung ist schon enthalten, Sie müssen lediglich die Kommentarzeichen entfernen.)

Sobald das Image `my-pandoc-image` zur Verfügung steht, können Sie den Typesetting-Prozess starten:

```
$ cd sample-files/markdown-setup
$ docker compose up
```

Damit wird aus dem Image ein Container erzeugt. Dort werden drei schon erwähnte Scripts ausgeführt, um Markdown zu LaTeX umzuwandeln, LaTeX zu PostScript/PDF sowie PNG-Bilder zu EPS-Bildern. Als Ergebnis erhalten Sie nach einigen Sekunden die Datei `typesetting/latex/preview.pdf` (siehe Abbildung 22.1).

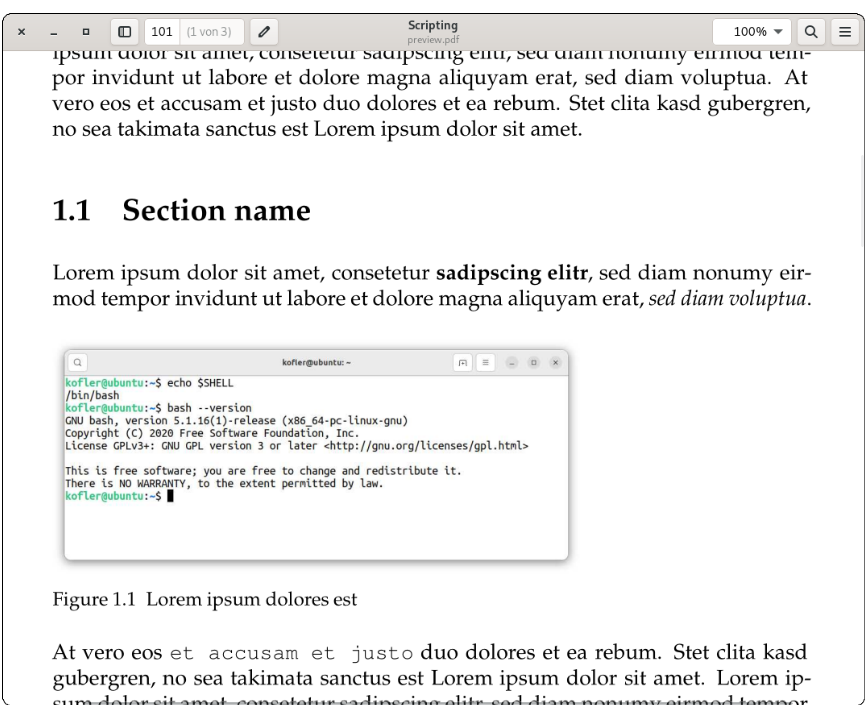



**Abbildung 22.1** Die aus »sample.md« erzeugte PDF-Datei

### Zugriffsrechte auf die Ergebnisdateien

Falls Sie docker compose unter Linux ausführen, wird allen von Docker erzeugten Dateien die Benutzer-ID 1000 zugewiesen. Wenn Sie eine andere ID-Nummer wünschen, können Sie dies in compose.yaml einstellen (Schlüsselwort user). Ihre eigene ID-Nummer erfahren Sie, wenn Sie in einem Terminal das Kommando id ausführen.

Sie können nun sample.md in einen Editor laden und dort eine Änderung vornehmen. Sobald Sie die Datei speichern, beginnt der Build-Prozess erneut. Wenige Sekunden später steht eine aktualisierte Version von preview.pdf zur Verfügung. Wenn Sie mit dem Test fertig sind, beenden Sie docker compose mit [Strg]+[C].

Sofern Sie unter Linux oder macOS arbeiten, können Sie mit ./clean.sh die Aufräumarbeiten durchführen. Das Script löscht alle temporär erzeugten Dateien.

### Der Nachteil »echter« Beispiele

Die meisten Beispiele in diesem Buch habe ich in der einen oder anderen Form tatsächlich irgendwann benötigt. Ich habe die Aufgabenstellung oder den Lösungsweg aber oft vereinfacht, um die Beispiele einfacher nachvollziehbar zu machen. Viele Beispiele dieses Buch sind also – wie ich hoffe – kurz und didaktisch prägnant.

Dieses Beispiel ist eine Ausnahme. Obwohl ich im Vergleich zu meinem tatsächlichen Setup viel vereinfacht habe, bleibt das Gesamtkonzept doch ziemlich komplex. Das Beispiel zeigt, was in der Realität passiert, wenn über den Zeitraum von zehn Jahren immer wieder Code geändert oder hinzugefügt wird.

## Dockerfile

Das Dockerfile sieht im Prinzip ganz ähnlich aus wie im ersten Beispiel dieses Kapitels. Die folgenden Zeilen zeigen nur einige Ausschnitte:

```
# Beispieldatei docker-file/Dockerfile
# Basis Ubuntu 20.04; diverse Pakete sowie Pandoc installieren
FROM ubuntu:20.04
RUN ln -fs /usr/share/zoneinfo/Europe/Berlin /etc/localtime && \
    apt-get update -y && \
    DEBIAN_FRONTEND=noninteractive apt-get install -y \
        --no-install-recommends \
        texlive-latex-recommended ...
RUN mkdir /data && \
    curl -L https://github.com/.../pandoc-2.7.3-1-amd64.deb \
      -o /data/pandoc.deb && \
```


© Rheinwerk Verlag, Bonn 2024

```
    dpkg -i /data/pandoc.deb && \
    rm /data/pandoc.deb
...
# Grundeinstellungen
VOLUME  ["/data"]
WORKDIR /data
CMD ["/bin/bash"]
```

Das auf dieser Basis erzeugte Image enthält nur die Software (also LaTeX, Pandoc usw.), aber keine eigenen Scripts.

### Docker-Compose-Datei

Eine interaktive Ausführung eines Containers mit dem durch `Dockerfile` beschriebenen Image ist zwar möglich, aber bei diesem Beispiel nicht vorgesehen. Stattdessen beschreibt die Datei `compose.yaml`, wie der Container ausgeführt werden soll:

```
# Beispieldatei compose.yaml
services:
  pics:
    build:
      context: pandoc
    image: my-pandoc-image
    volumes:
      - ./:/data
    command: ./typesetting/autoconvert ./images
    user: "1000"
  markdown:
    build:
      context: pandoc
    image: my-pandoc-image
    volumes:
      - ./:/data
    command: ./typesetting/autobuild
    user: "1000"
  latex:
    build:
      context: pandoc
    image: my-pandoc-image
    volumes:
      - ./:/data
    working_dir: /data/typesetting/latex
    command: ./autobuild
    user: "1000"
```

Persönliches Exemplar für Thomas Bartholomäus

Die Datei `compose.yaml` besagt, dass nach dem Start des Containers drei Scripts ausgeführt werden sollen:

- `typesetting/autoconvert` für die PNG-zu-EPS-Umwandlung
- `typesetting/autobuild` für die Verarbeitung der Markdown-Dateien in die Formate HTML und LaTeX
- `typesetting/latex/autobuild` für die Verarbeitung der durch das Docker-Service `markdown` erzeugten Datei `typesetting/latex/alltext.tex` zu einer PDF-Datei

### Scripts

Nach dieser langen Einleitung sind wir wieder beim eigentlichen Thema dieses Buchs angelangt, bei den Scripts.

- `typesetting/autoconvert`: Dieses Bash-Script überwacht das Verzeichnis `images`. Wenn dort eine PNG-Datei entdeckt wird, die neuer als die entsprechende EPS-Datei ist, wird `convert2eps` im gleichen Verzeichnis aufgerufen. Dieses Script verwendet wiederum `convert` aus dem Imagemagick-Paket (siehe auch Abschnitt 16.1, »Bilddateien manipulieren«).
- `typesetting/autobuild`: Dieses Script überwacht, ob sich eine der in `chapters` aufgezählten Markdown-Dateien oder diverse im Verzeichnis `typesetting` gespeicherte Scripts seit dem letzten Durchlauf verändert haben. Wenn das der Fall ist, kümmert sich das Script `build-html-latex` um den Aufruf von `pandoc`.
- `typesetting/build-html-latex`: An dieses Script werden alle zu verarbeitenden Markdown-Dateien als Parameter übergeben. Diese Dateien werden gleich zweimal mit `pandoc` verarbeitet: einmal, um `preview.html` mit der HTML-Version der Markdown-Dateien zu erzeugen, ein zweites Mal, um `latex/alltext.tex` zu generieren. An dieser Stelle zeige ich Ihnen lediglich die Zeilen, die für den LaTeX-Code verantwortlich sind. Die Variable `chaps` enthält dabei die Namen der Markdown-Dateien ohne die Kennung `.md`:

```
# Beispieldatei typesetting/build-html-latex (gekürzt)
...
lchaps=""
for fn in $chaps; do
   ./sed-before-pandoc-for-latex $fn.md  ./tmp/$fn.pandoc
   lchaps="$lchaps ./tmp/$fn.pandoc"
done
pandoc --from markdown+smart -t latex --listings \
       --top-level-division=chapter  \
       -o ./latex/tmptex $lchaps
./sed-after-pandoc-for-latex \
    ./latex/tmptex ./latex/alltext.tex
```


© Rheinwerk Verlag, Bonn 2024

Die for-Schleife ist für das Pre-Processing zuständig. Auf jede Markdown-Datei werden in sed-befor-pandoc-for-latex diverse Suchen-und-Ersetzen-Operationen angewendet. Die resultierenden Dateien werden im Unterverzeichnis tmp mit der Kennung .pandoc gespeichert. Die Dateinamen werden in der Variablen lchaps gespeichert.

Das nachfolgende pandoc-Kommando verarbeitet diese Dateien und erzeugt die Datei latex/tmptex. Ein weiteres Script mit sed-Kommandos kümmert sich nun um das Post-Processing und erzeugt die Datei latex/alltext.tex.

- typesetting/sed-xxx: Insgesamt vier Scripts sind für diverse Suchen- und Ersetzenoperationen zuständig. Ich zeige Ihnen hier einige Zeilen aus einer dieser vier Dateien. Die Anweisungen ändern Querverweise auf Bilddateien von images/name.png zu ../images/name.eps und fügen bei Abkürzungen wie »z. B.« die typografisch erforderlichen kleinen Abstände ein (LaTeX-Code \,).

```
# Beispieldatei typesetting/sed-after-pandoc-for-latex
sed -E -e 's,images/(.*)\.png,../images/\1.eps,' \
       -e 's/z\.B\./z.\\,B./' \
       -e 's/d\.h\./d.\\,h./' \
       -e 's/u\.a\./u.\\,a./' \
       ...
```

- typesetting/python-xxx: Der von pandoc erzeugte LaTeX-Code entspricht nicht ganz den Layoutwünschen des Verlags. Diverse Python-Scripts kümmern sich daher um eine Nachverarbeitung, z. B. für die Gestaltung von Tabellen oder für die Einstellung der gewünschten Größe von Abbildungen. Der Code ist weder besonders effizient noch besonders elegant, weswegen ich hier auf einen Abdruck verzichte.

  Die meisten Scripts werden am Ende von sed-before-pandoc-for-latex aufgerufen. Aus Debugging-Gründen sind die Scripts nicht durch Pipes verbunden, sondern erzeugen jeweils temporäre Dateien.

- typesetting/latex/autobuild: Dieses Script überwacht die *.tex-Dateien im Verzeichnis latex. Sobald sich eine der Dateien ändert, wird build-pdf aufgerufen:

```
# Beispieldatei typesetting/latex/build-pdf
latex book.tex &&  \
  dvips book -o book.ps && \
  ps2pdf book.ps && \
  mv book.pdf preview.pdf
```

  build-pdf verarbeitet die Datei book.tex. Diese liest zuerst diverse Header-Dateien, die für die Grundeinstellungen verantwortlich sind (z. B. für die Seitengröße oder die Gestaltung der Kopf- und Fußzeilen). Dann folgt die schon erwähnte Datei alltext.tex:

Persönliches Exemplar für Thomas Bartholomäus

Möglicherweise fragen Sie sich, warum sich die vielen Scripts außerhalb des Docker-Images befinden und nicht innerhalb. Das hat ganz pragmatische Gründe: Das Neuerstellen des Images ist ein relativ zeitaufwendiger Vorgang. In der Praxis kommt es immer wieder vor, dass sich kleinere Layoutänderungen nur durch einen Umbau der Scripts bewerkstelligen lassen. Solche Arbeiten lassen sich außerhalb des Images viel einfacher durchführen und testen.


© Rheinwerk Verlag, Bonn 2024

# Index


.bashrc-Datei ... 42
.zshrc-Datei ... 46
/var/log-Verzeichnis ... 217
&& (Bash) ... 54
? (Where-Object) ... 227

## A

Add-Content ... 226
Add-WindowsCapability ... 333
Add-WindowsFeature ... 467
add_watch ... 328
Administratorrechte ... 213
alias ... 220
Alias ... 101
- *CmdLets-Referenz* ... 247

AllSigned (ExecutionPolicy) ... 109
Amazon Web Services ... 447
append ... 167
apt ... 219
Archivdateien ... 211
args-Variable (PowerShell) ... 123
argv (Python) ... 195
Arithmetische Substitution ... 72
Arrays ... 65, 125
Assignment Expression ... 178
Assoziative Arrays ... 66
attrs ... 410
Aufgabenplanung ... 321
Ausgabe duplizieren ... 131
Ausgabe formatieren ... 131
Ausgabeumleitung ... 129, 213
authorized_keys ... 344
Automatische Variablen (PowerShell) ... 123
Awattar (REST-API) ... 430
AWS-S3 ... 447
- *AWS Tools for PowerShell* ... 454
- *CLI* ... 447
- *PowerShell* ... 454

## B

Backslash ... 76
Backtick ... 107
Backup-SqlDatabase ... 387
basename ... 309
Bash ... 21
- *INI-Datei* ... 313
- *JSON* ... 299
- *Konfiguration* ... 42
- *Reguläre Muster* ... 278
- *VSCode* ... 355
- *Windows-Installation* ... 39

Basic-Authentifizierung ... 423, 426, 430
BasicHtmlWebResponseObject ... 417
Bearer-Authentifizierung ... 421, 426, 430
Beautiful Soup ... 31, 409
Bedingungen ... 78
Betriebssystem feststellen ... 399
bg ... 55
Bilddateien manipulieren ... 389
Bilder sortieren ... 28
Bilder-Upload ... 345
Bourne Again Shell ... 37
Bourne Shell ... 21
Brace Extension ... 61
break ... 86, 180
brew ... 219
BSD ... 207

## C

Call-Operator (PowerShell) ... 103
CalledProcessError ... 197
capture_out ... 196
case ... 80
cat ... 221
cd ... 208
chgrp ... 214
chmod ... 49, 112, 214
chown ... 214
Chromium ... 414
chsh ... 46
cleandoc ... 159
Cloud ... 447
CmdLet ... 100
CmdletBinding ... 145
Code (Editor) ... 351
Command line arguments ... 81
Commit (Git) ... 362
Common Information Module (CIM) ... 135
Compress-Archive ... 133, 231, 388
ConfigParser ... 312
connect ... 441
contents ... 410
continue ... 86, 180
ConvertFrom-Json ... 291, 426, 428


Persönliches Exemplar für Thomas Bartholomäus

ConvertFrom-JSON ........ 445
ConvertFrom-StringData ........ 313
ConvertTo-Json ........ 291
ConvertTo-Xml ........ 306
copy ........ 176
Copy-Item ........ 224
Count ........ 106, 242
cp ........ 209
Cron ........ 315
*Beispiel* ........ 386
*crontab-Datei* ........ 316
crudini ........ 313
CSS-Farbcodes validieren ........ 285
CSV-Dateien ........ 264
*csv-Modul (Python)* ........ 264
*csvkit* ........ 264
*Export (PowerShell)* ........ 243
*Import-CSV (PowerShell)* ........ 266
curl ........ 258, 319, 420
cut ........ 252, 465


## D


Data Mining ........ 404
date ........ 221
Dateien
*überwachen* ........ 327
*archivieren* ........ 211
*komprimieren* ........ 211, 231
*kopieren* ........ 209
*löschen* ........ 209
*suchen* ........ 209, 227
*verschlüsseln* ........ 450
Dateinamen mit Leerzeichen ........ 85
Datentypen ........ 115
datetime-Modul ........ 432
dbatools ........ 387
Debug Stream ........ 128
DebugPreference-Variable ........ 129
declare ........ 64
decode ........ 196, 298
deepcopy ........ 176
def (Python) ........ 186
del ........ 167
df ........ 218
DictCursor ........ 441
Dictionary (Python) ........ 170, 183
dnf ........ 219
do ........ 83
do-until ........ 134
do-while ........ 134
Docker ........ 473
*Dockerfile* ........ 475
Document Object Model (DOM) ........ 409
done ........ 83
Doppelgänger eliminieren ........ 254
duf ........ 218
dump (Python/JSON) ........ 295


## E


E-Mail-Adressen erkennen ........ 278
echo ........ 69
Eingabeumleitung ........ 132
ElementTree-Modul ........ 306
elif ........ 77, 178
else ........ 77, 135, 178, 180
elseif ........ 135
Enable-PSRemoting ........ 236
Encapsulated PostScript ........ 392
Endlosschleife ........ 327
Enter-PSSession ........ 235
env-Kommando ........ 154
env-Variable ........ 125
EPEX SPOT (REST-API) ........ 431
Epoch-Zeit ........ 431
EPS-Dateien erzeugen ........ 392
Error Stream ........ 128
esac ........ 80
etree-Modul ........ 306
execute ........ 442
Execution Policy ........ 109
EXIF-Daten ........ 394
*in Datenbank speichern* ........ 397, 440
*exiftool* ........ 394
exit ........ 89, 195
Exit ........ 148
Exit-PSSession ........ 235
Expand-Archive ........ 231
Expansionsmechanismen ........ 72
export ........ 67
Export-Clixml ........ 306
Export-Csv ........ 243, 267
Export-ModuleMember ........ 144
Export-VM ........ 469
Extended Regular Expressions (ERE) ........ 270


## F


false ........ 123
Farben (bash) ........ 70
Fehlerabsicherung ........ 88, 145
fg ........ 55
fi ........ 77
file (Python) ........ 190
File System Events ........ 329
filter ........ 168
Filter ........ 142, 249




© Rheinwerk Verlag, Bonn 2024

find ... 209, 346, 410
findall ... 287, 307, 408, 410
for ... 83, 134, 180
foreach ... 132
ForEach-Item ... 241
ForEach-Object ... 126, 133
format ... 164
Format-Custom ... 131
Format-List ... 105, 131, 242
Format-Operator (PowerShell) ... 135
Format-Table ... 131, 242
Format-Wide ... 242
Fotos sortieren ... 28, 394
free ... 217
fromtimestamp ... 432
FSEvents API ... 329
fswatch ... 329
fullmatch ... 287
function ... 87
Funktionen ... 87, 103, 137, 186
fx ... 301

## G

Geolocation (REST-API) ... 424
Get-Alias ... 247
Get-ChildItem ... 99, 224, 227
Get-CimInstance ... 135
Get-Clipboard ... 244
Get-Command ... 104, 143, 244, 245, 333
Get-ComputerInfo ... 238
Get-Content ... 226, 445
    *Raw-Option* ... 304
Get-Credential ... 426
Get-Date ... 99
Get-EventLog ... 239
Get-ExecutionPolicy ... 109
Get-Help ... 104
Get-History ... 244
Get-InstalledModule ... 245
Get-ItemProperty ... 237
Get-ItemPropertyValue ... 237
Get-Location ... 99, 223
Get-Member ... 99, 240
Get-Module ... 245
Get-Package-Provider ... 246
Get-Process ... 106, 232
Get-Random ... 244
Get-S3Bucket ... 454
Get-S3Object ... 455
Get-ScheduledTask ... 324
Get-ScheduledTaskInfo ... 325
Get-Service ... 234
Get-SqlDatabase ... 388
Get-VMHardDiskDrive ... 470
Get-Volume ... 381
getopts ... 82
gh ... 373
Gierige reguläre Muster ... 274
Git ... 359
    *Git Bash* ... 41
    *GitHub-Status (Web Scraping)* ... 414
    *gitignore-Datei* ... 366
    *Hooks* ... 374
Globale Variable ... 187
Globbing ... 59, 184
    *PowerShell* ... 124
    *Python* ... 399
Glue Code ... 19
gpg ... 451
Greedy (Regex) ... 274
grep ... 210, 250, 465
    *Reguläre Muster* ... 279
Group-Object ... 240
gunzip ... 211
gzip ... 211

## H

Hash-Bang ... 47
Hash-Code generieren ... 439
Hashtables ... 123, 127
head ... 253
Hello World ... 108, 153
Heredoc ... 75
Herestring ... 76, 118
Hintergrundprozesse ... 55
history ... 221
Hooks ... 374
HTML ... 403
    *HTMLSession* ... 414
    *Parser* ... 409
htpasswd ... 439
Hyper-V ... 33, 467

## I

if ... 77, 135, 178
IFS-Variable ... 66, 86
ImageMagick ... 389
Immutable Types ... 175
import ... 197
Import-Clixml ... 306
Import-Csv ... 266, 294
Import-Module ... 143
Import-VM ... 469
IN_CLOSE_WRITE ... 328
info ... 45

Persönliches Exemplar für Thomas Bartholomäus

Information Stream ..... 128
InformationPreference-Variable ..... 129
INI-Dateien ..... 312
input ..... 163
input-Variable ..... 140
insert ..... 167
inspect-Modul ..... 159
Install-Module ..... 245
Internal Field Separator (IFS) ..... 66, 85, 86
Interpreter ..... 47
Invoke-Command ..... 235, 340
Invoke-RestMethod ..... 425
Invoke-Sqlcmd ..... 445
Invoke-WebRequest ..... 292, 417, 425, 458
ip ..... 221
IPv4-Adressen erkennen ..... 278
IPv6-Adressen erkennen ..... 280
items (Python) ..... 183
Iterator (Python) ..... 168

## J

JavaScript (Web Scraping) ..... 414
journalctl ..... 217
JPEG-Dateien konvertieren ..... 389
jq ..... 299
JSON ..... 291, 444
json-Modul ..... 295, 429

## K

Key-Value-Speicher ..... 127, 183
kill ..... 216
Klonen (virtuelle Maschinen) ..... 462, 469
Kommandos ausführen (Bash) ..... 53
Kommandosubstitution ..... 72, 121
Kommentare ..... 113
Konditionelle Kommandoausführung ..... 54
Konstanten ..... 64
KVM ..... 461

## L

Lambda-Funktion (Python) ..... 189
LASTEXITCODE ..... 148
LastWriteTimeUtc ..... 328
Lazy (Regex) ..... 274
len ..... 167
less ..... 221
libvirt ..... 461
like ..... 119
List Comprehension ..... 182
listdir ..... 185
Listen ..... 166
ln ..... 221
load (Python/JSON) ..... 295
Logging ..... 239
  *Logging-Dateien* ..... 217
Lokale Variablen ..... 88, 187
loop ..... 328
ls ..... 209

## M

MAC-Adressen verifizieren ..... 288
mail-Kommando ..... 440
makedirs ..... 408
man ..... 45, 221
Manifest-Datei ..... 145
map ..... 168
mapfile ..... 66
MariaDB ..... 436
match ..... 283, 287
Math-Klasse ..... 117
Matplotlib ..... 433
Measure-Command ..... 244
Measure-Object ..... 106, 241, 266
Mehrzeilige Anweisungen ..... 107
Mehrzeilige Zeichenketten ..... 75, 118, 159
Mengen ..... 169
Mirror-Funktion (wget) ..... 407
mkdir ..... 144, 208
Module ..... 142
  *PowerShell* ..... 244
  *Python* ..... 197
Monitoring ..... 319
Move-Item ..... 224
Mutable Types ..... 175
mv ..... 209
MySQL ..... 436
  *Backup* ..... 384
  *mysql-Kommando* ..... 436
  *mysqldump* ..... 385

## N

Namespace (XML) ..... 305
New-Item ..... 223, 237, 388
New-ItemProperty ..... 333
New-Object ..... 426
New-PSSession ..... 235, 340
New-ScheduledTaskAction ..... 325
New-ScheduledTaskTrigger ..... 325
New-Service ..... 235
Non-greedy (Regex) ..... 274
notlike ..... 119
notmatch ..... 283
NuGet ..... 246
null ..... 123
nullglob-Option ..... 60, 475



© Rheinwerk Verlag, Bonn 2024

## O


open (Python) ... 190
openssh-client ... 332
openssh-server ... 332
openssl ... 451
Operator ... 176
OPTARGS ... 82
Optionale Parameter ... 188
os-Modul ... 185, 309, 408
OSError ... 197
Out-Default ... 242
Out-Gridview ... 100, 243
Out-String ... 131, 230


## P


Paketverwaltung (Linux) ... 219
Parameter ... 113, 139, 188
Parameterauswertung ... 81
Parametersubstitution ... 73
PATH-Variable ... 42
 *Cron-Jobs* ... 317
 *Windows* ... 111
Perl Compatible Regular Expressions (PCRE) ... 270
ping ... 221, 259
pip ... 199
pip-tools ... 202
Pipe-Operator ... 58, 105
pipenv ... 201
Pipes ... 249
Pipfile ... 202
pipreqs ... 201
platform-Modul ... 399
PNG-Dateien konvertieren ... 389
POSIX Regular Expressions ... 270
PowerHTML ... 417
PowerShell ... 22, 91
 *Gallery* ... 244
 *INI-Datei* ... 313
 *Installation, Linux* ... 93, 94
 *Installation, Windows* ... 91
 *ISE* ... 108
 *JSON* ... 291
 *PowerShellGet* ... 244
 *Reguläre Muster* ... 283
 *VSCode* ... 353
 *XML* ... 302
print ... 163
printenv ... 67
process-Block (PowerShell) ... 141
Profile-Datei ... 42, 95
PROFILE-Variable ... 123
ProgressPreference ... 417
Prompt ... 43
 *Python* ... 152
Prozesse verwalten ... 215
ps ... 215
PS1-Dateiendung ... 109
PSCredential ... 426
PSGallery ... 244
PSItem ... 126
PSModulePath-Variable ... 143
pwd ... 208
PWD-Variable ... 123
pwsh ... 94
PyExifTool ... 397
pyinotify ... 328
Python ... 149
 *INI-Datei* ... 312
 *installieren* ... 150
 *interaktiv ausprobieren* ... 152
 *Iterator* ... 168
 *JSON* ... 295
 *Reguläre Muster* ... 287
 *Textdatei verarbeiten* ... 185
 *Tupel* ... 169
 *Variable* ... 172
 *XML* ... 306
 *Zuweisung* ... 173


## Q


qemu-img ... 462
Quantifizierer (Regex) ... 274
Quotierungszeichen
 *Bash* ... 76
 *PowerShell* ... 107


## R


range ... 181
raw-Zeichenkette ... 160
rc.local-Datei ... 464
re-Modul ... 287,408
read ... 72
Read-Host ... 115, 227
Read-S3Object ... 456
reader ... 264
Rechtschreibkorrektur (Script-Beispiel) ... 27
Register-ScheduledTask ... 325
Registrierdatenbank ... 236
Reguläre Ausdrücke ... 269
 *INI-Dateien* ... 313
 *Web Scraping* ... 408
Rekursives Globbing ... 60
Remote-SSH-Erweiterung (VSCode) ... 355
RemoteSigned (ExecutionPolicy) ... 109


Persönliches Exemplar für Thomas Bartholomäus

Remoting ... 235
  *SSH* ... 339
remove ... 167
Remove-Item ... 107, 223, 224, 388
Remove-Module ... 143
Remove-Service ... 235
Remove-VMSnapshot ... 469
Rename-VM ... 471
render ... 414
replace ... 283
Replace ... 459
Repository ... 361
repr ... 164
request-Modul ... 408
requests-html ... 414
requests-Modul ... 410, 429
requirements.txt ... 201
Requires ... 114
REST-API ... 419
Restart-Service ... 235, 339
Restricted (ExecutionPolicy) ... 109
Resume-Service ... 235
return ... 87, 139
rm ... 209
rmdir ... 208
robocopy ... 380
root-Rechte ... 213
RSS-Feed ... 308
rstrip (Python) ... 191
rsync ... 341, 383
run ... 195


## S


S3 (AWS) ... 447
Schleifen ... 83, 132, 180
scp ... 340, 385
  *Beispiel* ... 345
Scripting ... 17
search ... 287
Secure Shell ... 331
sed ... 280, 465
select ... 410
Select-Object ... 106, 225, 239
Select-String ... 228
  *reguläre Muster* ... 284
Select-Xml ... 304
Serielle Kommandoausführung ... 54
set ... 89
Set Comprehension ... 182
Set-AWSCredential ... 454
Set-Content ... 226
Set-DefaultAWSRegion ... 456
Set-ExecutionPolicy ... 109
Set-ItemProperty ... 237
Set-Location ... 99, 223
Set-Service ... 235, 333
Set-StrictMode ... 115, 127, 146
Set-VMNetworkAdapter ... 471
Sets ... 169
Shebang ... 47, 112, 154
Shell ... 20
shift ... 82
shopt ... 475
Signal (bash) ... 89
Signale (SIGKILL, SIGTERM) ... 216
Signierte Scripts ... 109
Simple Storage Service (S3) ... 447
sleep ... 197, 244, 328
Slicing ... 162, 166
Snapshots löschen (Beispiel) ... 33
Software verwalten (Linux) ... 219
sort ... 253
Sort-Object ... 106, 239
Spalten aus Text ausschneiden ... 252
Specified endpoint (Fehlermeldung) ... 456
Speicherplatz ermitteln ... 217
Splatting ... 102
Split ... 133
Split-Path ... 305
splitlines ... 196
SQL Server ... 437
  *Backup* ... 387
  *sqlcmd* ... 437
  *SqlServer (PowerShell-Modul)* ... 387, 445
SSH ... 331
  *Remoting (PowerShell)* ... 339
  *Schlüssel* ... 342
  *Script auf virtuellen Maschinen ausführen* ... 464
  *ssh-copy-id* ... 344
  *ssh-keygen* ... 342
  *sshd_config* ... 339
  *VSCode* ... 355
Standardausgabe ... 56, 195
Standardeingabe ... 56, 140, 195
Standardfehlerkanal ... 56, 195
Start-Job ... 234
Start-Process ... 232
Start-Service ... 235, 333
Start-Sleep ... 244, 328
Stop-Process ... 106, 233
Stop-Service ... 235
str ... 164
Stream Editor ... 280
Streams ... 128
strip (Python) ... 191



© Rheinwerk Verlag, Bonn 2024

Strompreis (REST-API) ... 430
subprocess ... 195
Subshells ... 55
Substitution ... 118
Substitutionsmechanismen ... 72
Subsystem (in sshd_config) ... 339
Success Stream ... 128
Suchen und Ersetzen ... 280
sudo ... 213
Sum ... 106
Suspend-Service ... 235
switch ... 137
sys (Python-Modul) ... 194
systemctl ... 216
Systemdienste ... 216

## T

tail ... 253
tar ... 211, 385
tee ... 59
Tee-Object ... 131
Terminal ... 96
test ... 78
Test-NetConnection ... 243
Test-Path ... 225
Text
- *Auswertung* ... 249
- *Doppelgänger eliminieren* ... 254
- *filtern* ... 226, 250
- *Spalten ausschneiden* ... 252
- *sortieren* ... 253

Textdatei verarbeiten ... 86, 185
then ... 77
timeout ... 90
Toolbox ... 207
touch ... 209
trap ... 89
true ... 123
Tupel (Python) ... 169
Tuple Comprehension ... 182

## U

Umgebungsvariablen ... 67, 125
uniq ... 254
Unrestricted (ExecutionPolicy) ... 109
until ... 86
unzip ... 212
Update-Module ... 245
urljoin ... 412
urllib ... 298, 410
urlopen ... 298, 408

## V

Variablen ... 172
- *Bash* ... 63
- *lokale* ... 88
- *PowerShell* ... 114
- *Python* ... 172
- *Substitution* ... 72
- *vordefinierte* ... 123

venv ... 202
Verbose Stream ... 128
VerbosePreference-Variable ... 129
Verschlüsselung von Dateien ... 450
Verzeichnisse ... 208, 223
- *synchronisieren* ... 341

Verzweigungen ... 77, 135
virsh ... 462
virt-clone ... 462
VirtualBox (XML-Dateien) ... 305
virtualenv ... 202
Virtuelle Maschinen ... 461
- *per SSH auswerten* ... 348

Visual Studio Code ... 351
Vordefinierte Variablen ... 123
Vordergrundprozesse ... 55
VSCode ... 351

## W

Warning Stream ... 128
WarningPreference-Variable ... 129
WatchManager ... 328
wc ... 222, 251
Web Crawling ... 404
Web Scraping ... 31, 403
- *JavaScript* ... 414

Webserver
- *Bilder-Upload* ... 345
- *Monitoring* ... 319

Websites herunterladen ... 405
Wetter (REST-API) ... 32, 428
wget ... 405, 420
Where-Object ... 136, 225, 227, 240
which ... 222
while ... 85, 180
Win32_LogicalDisk ... 135
Windows
- *PowerShell* ... 91
- *SSH-Server installieren* ... 333
- *Terminal* ... 96
- *Windows Management Instrastructure* 135
- *Windows Remote Management* ... 235
- *Windows Task Scheduler* ... 321
- *Windows-Subsystem für Linux (WSL)* ... 40

winget ... 246


Persönliches Exemplar für Thomas Bartholomäus

with (Python) ........ 191
WordPress-Backup ........ 384
Write-Host ........ 130, 227
*in try/catch* ........ 147
Write-Output ........ 108
*in Funktionen* ........ 138
Write-S3Object ........ 456

## X

XML ........ 291, 302
*Namespaces* ........ 305
*xml-Kommando* ........ 311
*xmllint* ........ 310
*XMLStarlet* ........ 311
XPath ........ 304

## Z

Zeichenketten ........ 118, 159
*Bash* ........ 69
*mehrzeilige* ........ 75, 118, 159
zip ........ 212
zipinfo ........ 212
Zsh ........ 45
Zugriffsrechte ........ 214
Zuweisung (Python) ........ 173
Zwischenablage ........ 244
zypper ........ 219


© Rheinwerk Verlag, Bonn 2024

# Die Serviceseiten

Im Folgenden finden Sie Hinweise, wie Sie Kontakt zu uns aufnehmen können.

## Lob und Tadel

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Wenn Sie zufrieden waren, empfehlen Sie das Buch bitte weiter. Wenn Sie meinen, es gebe doch etwas zu verbessern, schreiben Sie direkt an den Lektor dieses Buches: *christoph.meister@rheinwerk-verlag.de*. Wir freuen uns über jeden Verbesserungsvorschlag, aber über ein Lob freuen wir uns natürlich auch!

Auch auf unserer Webkatalogseite zu diesem Buch haben Sie die Möglichkeit, Ihr Feedback an uns zu senden oder Ihre Leseerfahrung per Facebook, Twitter oder E-Mail mit anderen zu teilen. Folgen Sie einfach diesem Link: *https://www.rheinwerk-verlag.de/5672*.

## Zusatzmaterialien

Falls Zusatzmaterialien (Beispielcode, Übungsmaterial, Listen usw.) für dieses Buch verfügbar sind, finden Sie sie in Ihrer Online-Bibliothek sowie auf der Webkatalogseite zu diesem Buch: *https://www.rheinwerk-verlag.de/5672*. Wenn uns sinnentstellende Tippfehler oder inhaltliche Mängel bekannt werden, stellen wir Ihnen dort auch eine Liste mit Korrekturen zur Verfügung.

## Technische Probleme

Im Falle von technischen Schwierigkeiten mit dem E-Book oder Ihrem E-Book-Konto beim Rheinwerk Verlag steht Ihnen gerne unser Leserservice zur Verfügung: *ebooks@rheinwerk-verlag.de*.

## Über uns und unser Programm

Informationen zu unserem Verlag und weitere Kontaktmöglichkeiten bieten wir Ihnen auf unserer Verlagswebsite *https://www.rheinwerk-verlag.de*. Dort können Sie sich auch umfassend und aus erster Hand über unser aktuelles Verlagsprogramm informieren und alle unsere Bücher und E-Books schnell und komfortabel bestellen. Alle Buchbestellungen sind für Sie versandkostenfrei.

# Rechtliche Hinweise

In diesem Abschnitt finden Sie die ausführlichen und rechtlich verbindlichen Nutzungsbedingungen für dieses E-Book.

## Copyright-Vermerk

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Nutzungs- und Verwertungsrechte liegen bei den Autor*innen und beim Rheinwerk Verlag, insbesondere das Recht der Vervielfältigung und Verbreitung, sei es in gedruckter oder in elektronischer Form.
**© Rheinwerk Verlag GmbH, Bonn 2023**

## Nutzungs- und Verwertungsrechte

Sie sind berechtigt, dieses E-Book ausschließlich für persönliche Zwecke zu nutzen. Insbesondere sind Sie berechtigt, das E-Book für Ihren eigenen Gebrauch auszudrucken oder eine Kopie herzustellen, sofern Sie diese Kopie auf einem von Ihnen alleine und persönlich genutzten Endgerät speichern. Zu anderen oder weitergehenden Nutzungen und Verwertungen sind Sie nicht berechtigt.

So ist es insbesondere unzulässig, eine elektronische oder gedruckte Kopie an Dritte weiterzugeben. Unzulässig und nicht erlaubt ist des Weiteren, das E-Book im Internet, in Intranets oder auf andere Weise zu verbreiten oder Dritten zur Verfügung zu stellen. Eine öffentliche Wiedergabe oder sonstige Weiterveröffentlichung und jegliche den persönlichen Gebrauch übersteigende Vervielfältigung des E-Books ist ausdrücklich untersagt. Das vorstehend Gesagte gilt nicht nur für das E-Book insgesamt, sondern auch für seine Teile (z. B. Grafiken, Fotos, Tabellen, Textabschnitte).

Urheberrechtsvermerke, Markenzeichen und andere Rechtsvorbehalte dürfen aus dem E-Book nicht entfernt werden, auch nicht das digitale Wasserzeichen.

## Digitales Wasserzeichen

Dieses E-Book-Exemplar ist mit einem **digitalen Wasserzeichen** versehen, einem Vermerk, der kenntlich macht, welche Person dieses Exemplar nutzen darf. Wenn Sie diese Person nicht sind, liegt ein Verstoß gegen das Urheberrecht vor, und wir bitten Sie freundlich, das E-Book nicht weiter zu nutzen und uns diesen Verstoß zu melden. Eine kurze E-Mail an *service@rheinwerk-verlag.de* reicht schon. Vielen Dank!

## Markenschutz

Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

## Haftungsausschluss

Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor*innen, Herausgeber*innen oder Übersetzer*innen für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen.

# Über den Autor

**Michael Kofler** ist der renommierteste Fachbuchautor im deutschsprachigen Raum und behandelt von Linux über Swift bis zur IT-Security alles, was Sie in der Systemadministration oder der Softwareentwicklung wissen müssen. In diesem Buch zeigt er Ihnen, wie Sie clevere und elegante Scripte erstellen.