# Scripting

Das Praxisbuch für Administratoren
und DevOps-Teams

# 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,

- 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-

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.

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.

**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.)

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-

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-

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

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-

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)

# 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.

### 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
```

### 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'])
```

| 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'))
```

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.

### 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.

**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.

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>
```

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
```

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

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*

# 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

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
```

```
  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

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,

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

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
```

```
  # 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:

```
> 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        ...
```

```
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.)

```
# 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"
```

```
# 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
}
```

```
# 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.

# 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.