Python für alle

Python für alle

Einführung in die Datenanalyse mit Python 3

Charles R. Severance

Deutsche Ausgabe

Fabian Eberts
Heiner Giefers

Originaltitel: *Python for Everybody. Exploring Data Using Python 3.*

Übersetzung und Bearbeitung: Fabian Eberts, Heiner Giefers
Redaktionelle Unterstützung: Elliott Hauser, Sue Blumenberg
Covergestaltung: Aimee Andrion

ISBN 979-8-42547-509-1

1. Auflage 2022

Autorisierte deutsche Übersetzung und Bearbeitung der englischen Ausgabe von *Python for Everybody. Exploring Data Using Python 3.*

Vorwort

Vorwort zur deutschen Ausgabe

Die Deutsche Nationalbibliothek listet über 500 Titel zum Thema Programmierung mit Python, digitale Medien eingeschlossen sind es sogar über 1000. Man kann sich also die Frage stellen, ob das *1001. Buch über Python* wirklich notwendig ist. Wir finden *ja*, denn der Ansatz dieses Buches unterscheidet sich von den meisten anderen zum Thema. *Python für alle* ist Teil eines offenen Curriculums zu Python, das bereits in verschiedene Sprachen übersetzt wurde und in diversen Hochschulen sowie in Onlinekursen Anwendung findet.

Mein großer Dank gilt Charles Severance, der diese Materialsammlung aufgebaut hat und weltweit Menschen dabei unterstützt, seine Inhalte zu verwenden und weiterzuentwickeln. Genau wie der Autor stand auch ich vor der Entscheidung, für einen Kurs zur Einführung in die Programmierung mit Python Lehrmaterialien selbst zu entwickeln oder ein bestehendes Lehrbuch zu verwenden. Viele der vielen Lehrbücher zu Python waren prinzipiell geeignet, aber keines passte genau auf die Anforderungen der Veranstaltung. Der pragmatische Ansatz von *Python for Everybody* eignet sich sehr für Programmieranfänger. Es gibt keinen Anspruch auf Vollständigkeit; vielmehr zielt das Buch darauf ab, die Programmierung mit Python in logisch aufeinander aufbauenden Kapiteln *von Grund auf* zu vermitteln. Dies und die Möglichkeit, das Buch inhaltlich zu erweitern, haben schließlich zum Entschluss geführt, *Python for Everybody* ins Deutsche zu übersetzen.

Die vorliegende Übersetzung orientiert sich weitgehend am englischsprachigen Original. Die Einleitung des Themas in Kapitel 1 wurde etwas verkürzt, in den Kapiteln 2–16 wurden einige Abschnitte aktualisiert und ergänzt, allerdings ohne dabei den Grundaufbau zu verändern. Bei den Programmbeispielen sind die Ausgaben größtenteils ins Deutsche übersetzt worden, Eingabedaten und Bezeichner wurden überwiegend wie in der Originalausgabe belassen.

Neben Charles Severance und allen anderen, die an der Entwicklung der Inhalte dieses Buches beteiligt waren, möchte ich mich auch bei denjenigen bedanken, die so tatkräftig bei der Übersetzung des Buches mitgewirkt haben. Hier sind vor allem Fabian Eberts, Julia Warnke und Sebastian Schmidt zu nennen. Ohne ihren Einsatz hätte sich das Buch nicht in nur wenigen Wochen übersetzen lassen.

Heiner Giefers

Neubearbeitung eines Open-Books

Es ist ganz normal, dass Akademiker, die ständig „publish or perish“ hören, ihre Arbeiten immer von Grund auf neu schaffen wollen. Dieses Buch dagegen versucht, eben nicht bei null anzufangen, sondern stattdessen das Buch *Think Python: How to Think Like a Computer Scientist* von Allen B. Downey, Jeff Elkner und anderen neu zu bearbeiten.

Im Dezember 2009 war ich gerade dabei, mich darauf vorzubereiten, das fünfte Semester in Folge *Networked Programming* an der University of Michigan zu unterrichten, und beschloss, dass es an der Zeit war, ein Python-Lehrbuch zu schreiben, das sich auf die Analyse von Daten konzentriert, anstatt auf das Vermitteln von Algorithmen und Abstraktionen. Mein Ziel in *Networked Programming* ist es, den Lesern Fähigkeiten im Umgang mit Daten mittels mit Python zu vermitteln. Nur wenige meiner Studenten hatten vor, professionelle Computerprogrammierer zu werden. Stattdessen wollten sie Bibliothekare, Manager, Anwälte, Biologen, Wirtschaftswissenschaftler usw. werden, die in ihrem jeweiligen Fachgebiet die Technologie geschickt einsetzen wollten.

Ich konnte nie das perfekte datenorientierte Python-Buch für meinen Kurs finden, also habe ich mich daran gemacht, ein solches Buch zu schreiben. Glücklicherweise zeigte mir Dr. Atul Prakash bei einer Fakultätssitzung drei Wochen bevor ich in den Ferien mit meinem neuen Buch beginnen wollte das Buch *Think Python*, das er in diesem Semester für seinen Python-Kurs verwendet hatte. Es ist ein gut geschriebenes Informatikbuch mit dem Schwerpunkt auf kurzen, direkten Erklärungen und leichter Erlernbarkeit.

Die Gesamtstruktur des Buches wurde geändert, um so schnell wie möglich zu den Problemen der Datenanalyse zu gelangen und von Anfang an eine Reihe von Beispielen und Übungen zur Datenanalyse anzubieten.

Die Kapitel 2–10 ähneln dem Buch *Think Python*, aber es gibt wichtige Änderungen. Zahlenorientierte Beispiele und Übungen sind durch datenorientierte Übungen ersetzt worden. Die Themen werden in der Reihenfolge präsentiert, die für die Erstellung von zunehmend anspruchsvolleren Datenanalyselösungen erforderlich ist. Einige Themen wie `try` und `except` werden vorgezogen und als Teil des Kapitels über Kontrollstrukturen vorgestellt. Funktionen werden nur sehr oberflächlich behandelt, bis sie zur Bewältigung der Programmkomplexität benötigt werden, aber nicht als frühe Lektion in Abstraktion eingeführt. Fast alle benutzerdefinierten Funktionen wurden aus dem Beispielcode und den Übungen außerhalb von Kapitel 4 entfernt. Das Wort *Rekursion*[1] kommt in dem Buch überhaupt nicht vor.

In den Kapiteln 1 und 11–16 ist das gesamte Material brandneu und konzentriert sich auf reale Anwendungen und einfache Beispiele von Python für die Datenanalyse einschließlich regulärer Ausdrücke für die Suche und das Parsing, die Automatisierung von Aufgaben auf Ihrem Computer, das Abrufen von Daten über das Netzwerk, das Scraping von Webseiten nach Daten, objektorientierte Programmierung, die Verwendung von Webdiensten, das Parsing von XML- und JSON-Daten, die Erstellung und Verwendung von Datenbanken mit der Structured Query Language und die Visualisierung von Daten.

Das ultimative Ziel all dieser Änderungen ist es, den Schwerpunkt von der Informatik auf die Datenverarbeitung und -analyse zu verlagern und nur noch Themen aufzunehmen, die auch dann nützlich sein können, wenn man sich entscheidet, kein professioneller Programmierer zu werden.

Studierende, die dieses Buch interessant finden und weiter vertiefen wollen, sollten sich das Buch *Think Python* von Allen B. Downey ansehen. Da es viele Überschneidungen zwischen den beiden Büchern gibt, werden die Studierenden schnell Fähigkeiten in den zusätzlichen Bereichen der technischen Programmierung und des algorithmischen Denkens erwerben, die in *Think Python* behandelt werden. Und da die Bücher einen ähnlichen Schreibstil haben, sollten sie in der Lage sein, *Think Python* mit einem Minimum an Aufwand schnell durchzuarbeiten.

Als Inhaber des Copyrights von *Think Python* hat mir Allen B. Downey die Erlaubnis erteilt, die Lizenz für das Material aus seinem Buch, das in diesem Buch enthalten ist, von der GNU Free Documentation License auf die neuere Creative Commons Attribution-Share Alike Lizenz zu ändern. Dies folgt dem aktuellen Trend der Verschiebung der Lizenzen für offene Dokumentation von der GFDL zur CC-BY-SA (z. B. Wikipedia). Durch die Verwendung der CC-BY-SA-Lizenz wird die starke Copyleft-Tradition des Buches beibehalten, während es für neue Autoren noch einfacher wird, dieses Material nach eigenem Ermessen weiterzuverwenden.

Ich bin der Meinung, dass dieses Buch ein Beispiel dafür ist, warum offene Materialien so wichtig für die Zukunft der Bildung sind, und ich möchte Allen B. Downey und Cambridge University Press für ihre zukunftsweisende Entscheidung danken, das Buch unter einem offenen Copyright zur Verfügung zu stellen. Ich hoffe, dass sie mit dem Ergebnis meiner Bemühungen zufrieden sind und ich hoffe, dass Sie, die Leser, mit *unseren* gemeinsamen Bemühungen zufrieden sind.

Ich möchte Allen B. Downey und Lauren Cowles für ihre Hilfe, Geduld und Beratung bei der Klärung von Urheberrechtsfragen im Zusammenhang mit diesem Buch danken.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
September 9, 2013

Charles Severance ist Professor an der University of Michigan School of Information.

[1] außer natürlich in dieser Zeile!

Inhaltsverzeichnis

Kapitel 1

Warum sollte man Programmieren lernen?

Das Schreiben von Programmen (oder Programmieren) ist eine sehr kreative und lohnende Tätigkeit. Wir können Programme aus vielen Gründen schreiben, angefangen mit dem Ziel, damit den Lebensunterhalt zu verdienen, ein schwieriges Datenanalyseproblem zu lösen, Spaß zu haben oder um jemand anderem bei der Lösung eines Problems zu helfen. Dieses Buch geht davon aus, dass *jeder* wissen sollte, wie man programmiert, und dass man, sobald man die Programmierung beherrscht, herausfindet, was man mit den neugewonnenen Fähigkeiten machen kann.

Wir sind in unserem täglichen Leben von Computern umgeben, von Laptops bis hin zu Handys. Wir nehmen diese Computer als unsere „persönlichen Assistenten“ war, die viele Dinge für uns erledigen können. Die Hardware in unseren heutigen Computern ist im Wesentlichen so gebaut, dass sie uns ständig die Frage stellt: „Was soll ich als Nächstes tun?“

Programmierer fügen der Hardware ein Betriebssystem und eine Reihe von Anwendungen hinzu, und schon haben wir einen persönlichen digitalen Assistenten, der uns bei vielen Problemen des Alltags nützlich sein kann. Unsere Computer sind schnell und haben riesige Mengen an Speicher und könnten uns sehr hilfreich sein, wenn wir nur die Sprache beherrschen würden, um dem Computer zu erklären, was er als Nächstes tun soll. Wenn wir diese Sprache kennen, könnten wir dem Computer sagen, dass er in unserem Namen Aufgaben erledigen soll, die sich wiederholen. Interessanterweise sind die Dinge, die Computer am besten können oft die Dinge, die wir Menschen langweilig und stumpfsinnig finden.

Schauen wir uns zum Beispiel die ersten drei Absätze dieses Kapitels an und finden heraus, welches Wort am häufigsten verwendet wird und wie oft es vorkommt. Während wir in der Lage waren, die Wörter in wenigen Sekunden zu lesen und zu verstehen, ist das Zählen der Wörter fast schmerzhaft, weil es nicht die Art von Problem ist, die der menschliche Verstand einfach lösen kann. Für einen Computer ist das Gegenteil der Fall: Das Lesen und Verstehen von Text auf einem Blatt Papier ist für einen Computer schwer, aber die Wörter zu zählen und Ihnen zu sagen, wie oft das am häufigsten verwendete Wort verwendet wurde, ist für den Computer sehr einfach.

```
python words.py
Enter file:words.txt
Das Wort "die" kommt 9-mal vor
```

Unser Programm sagt uns schnell, dass das Wort „die“ 9 mal im oberen Teil dieses Kapitels verwendet wurde. Genau diese Tatsache, dass Computer Dinge gut können, die Menschen eher nicht gut bzw. schnell können, ist der Grund, warum man die Programmiersprachen beherrschen sollten. Sobald man eine Programmiersprache gelernt hat, kann man viele alltägliche Aufgaben durch den Computer erledigen lassen. So bleibt einem mehr Zeit für die diejenigen Aufgaben, für die wir Menschen einzigartig geeignet sind, nämlich Kreativität, Intuition und Ideenreichtum.

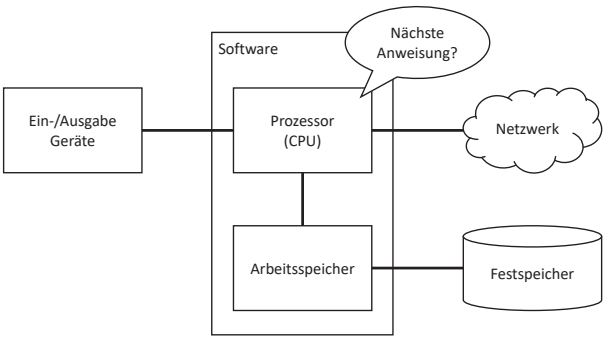

Abbildung 1.1: Aufbau eines Computers

1.1. Kreativität und Motivation

Wenn wir gerade mit dem Programmieren beginnen, werden wir noch einige Erfahrungen sammeln müssen, um professionelle Programme (auch *Software* genannt) entwickeln zu können. Professionelles Programmieren ist allerdings sowohl finanziell als auch persönlich eine sehr lohnende Aufgabe. Nützliche, elegante und clevere Programme zu erstellen, ist eine sehr kreative Aktivität, die durchaus Spaß machen kann. Lukrativ ist die Programmierung vor allem dann, wenn unseren Kunden, bzw. den Nutzern das Programm gefällt und uns einen Nutzen bringt. Dabei stehen wir in Konkurrenz zu anderen Entwicklern. Man sollte also versuchen, dass das eigene Programm besser funktioniert, einen höheren Funktionsumfang hat, sich besser bedienen lässt oder einfach schöner aussieht.

Im Moment besteht unsere Hauptmotivation nicht darin, Geld zu verdienen oder den Endnutzern zu gefallen. Wir möchten unsere eigenen Arbeitsabläufe automatisieren und produktiver mit den Daten und Informationen umgehen, die uns in unserem Leben begegnen. Bei unseren Programmieranfängen sind wir sowohl der Programmierer als auch der Endnutzer unserer Programme. Je mehr Erfahrung wir sammeln und je umfangreicher unsere Programme werden, desto mehr werden wir befähigt, auch Programme für andere zu entwickeln.

1.2. Der Aufbau eines Computers

Bevor wir anfangen, die Sprachen zu lernen, mit der wir Computer *programmieren* können, sollten wir uns ein wenig damit beschäftigen, wie Computer aufgebaut sind. Wenn wir unseren Computer oder unser Handy auseinandernehmen und tief ins Innere schauen würden, würden wir die folgenden Teile finden:

Die wichtigsten Definitionen dieser Teile lauten wie folgt:

- Die *Central Processing Unit* (oder CPU) ist der Teil des Computers, der so gebaut ist, dass er von der Frage „Was kommt als Nächstes?“ besessen ist. Wenn der Computer auf 3,0 Gigahertz eingestellt ist, bedeutet das, dass die CPU drei Milliarden Mal pro Sekunde fragt: „Was kommt als Nächstes?“. Wir müssten lernen, schnell zu sprechen, um mit der CPU Schritt halten zu können.

- Der *Hauptspeicher* wird zum Speichern von Informationen verwendet, die die CPU schnell benötigt. Der Hauptspeicher ist fast so schnell wie die CPU. Aber die im Hauptspeicher gespeicherten Informationen verschwinden, wenn der Computer ausgeschaltet wird.

- Der *Sekundärspeicher* (oder auch *Festspeicher*) wird ebenfalls zum Speichern von Informationen verwendet, ist aber viel langsamer als der Hauptspeicher. Der Vorteil des Sekundärspeichers ist, dass er Informationen auch dann speichern kann, wenn der Computer nicht mit Strom versorgt wird. Beispiele für Sekundärspeicher sind Festplattenlaufwerke oder Flash-Speicher (typischerweise in USB-Sticks und tragbaren Musikplayern zu finden).

- Die *Eingabe- und Ausgabegeräte* sind unser Bildschirm, unsere Tastatur, Maus, Mikrofon, Lautsprecher oder Touchpad und dienen der Interaktion mit dem Computer.

- Heutzutage haben die meisten Computer auch eine *Netzwerkverbindung*, um Informationen über ein Netzwerk abzurufen. Wir können uns das Netzwerk als einen sehr langsamen Ort vorstellen, an dem Daten gespeichert und abgerufen werden, die nicht immer „verfügbar“ sind. In gewissem Sinne ist das Netzwerk also eine langsamere und manchmal unzuverlässige Form des *Sekundärspeichers*.

Die meisten Details über die Funktionsweise dieser Komponenten überlässt man am besten den Computerbauern, aber es ist hilfreich, eine Terminologie zu haben, damit wir beim Schreiben unserer Programme über diese verschiedenen Teile sprechen können.

Als Programmierer ist es unsere Aufgabe, jede dieser Ressourcen zu nutzen und zu koordinieren, um das Problem zu lösen, das wir lösen müssen, und die Daten zu analysieren, die wir aus der Lösung erhalten. Als Programmierer werden wir hauptsächlich mit der CPU „reden“ und ihr sagen, was sie als Nächstes tun soll. Manchmal werden wir der CPU sagen, dass sie den Hauptspeicher, den sekundären Speicher, das Netzwerk oder die Eingabe-/Ausgabegeräte verwenden soll.

Wir müssen jeweils die Person sein, die der CPU die Frage „Was nun?“ beantwortet. Allerdings wäre es sehr ineffizient, wenn wir den Dialog mit der CPU *Live* führen würden. Die CPU kann drei Milliarden Mal pro Sekunde einen Befehl ausführen, wir wären aber lange nicht in der Lage, mit diesem Tempo mitzuhalten. Stattdessen müssen wir unsere Anweisungen im Voraus aufschreiben. Wir nennen diese gespeicherten Anweisungen ein *Programm* und den Akt des Aufschreibens dieser Anweisungen und die korrekte Ausführung der Anweisungen *Programmierung*.

1.3. Programmierung verstehen

Im weiteren Verlauf dieses Buches werden wir versuchen, aus uns Personen zu machen, die die Kunst des Programmierens beherrschen. Am Ende werden wir echte *Programmierer* sein – vielleicht keine professionellen Programmierer, aber zumindest werden wir die Fähigkeit besitzen, ein Daten-/Informationsanalyseproblem zu betrachten und ein Programm zur Lösung des Problems zu entwickeln.

In gewissem Sinne braucht man zwei Fähigkeiten, um ein Programmierer zu sein:

- Erstens müssen wir die Programmiersprache (Python) kennen – wir müssen das Vokabular und die Grammatik kennen. Wir müssen in der Lage sein, die Wörter in dieser neuen Sprache richtig zu schreiben und wissen, wie man wohlgeformte „Sätze“ in dieser neuen Sprache konstruiert.
- Zweitens müssen wir „eine Geschichte erzählen“ können. Beim Schreiben einer Geschichte kombinieren wir Wörter und Sätze, um dem Leser eine Idee zu vermitteln. Es ist eine Kunst, eine Geschichte zu konstruieren, und die Fähigkeit, eine Geschichte zu schreiben, wird verbessert, indem man etwas schreibt und Feedback erhält. Beim Programmieren ist unser Programm die „Geschichte“ und das Problem, das wir zu lösen versuchen, ist die „Idee“.

Wenn man einmal eine Programmiersprache wie Python gelernt hat, wird es einem viel leichter fallen, eine zweite Programmiersprache wie JavaScript oder C++ zu lernen. Die neue Programmiersprache hat einen ganz anderen Wortschatz und eine andere Grammatik, aber die Problemlösungsfähigkeiten sind in allen Programmiersprachen gleich.

Wir werden das „Vokabular“ und die „Sätze“ von Python ziemlich schnell lernen. Es wird länger dauern, bis man in der Lage ist, ein zusammenhängendes Programm zu schreiben, um ein brandneues Problem zu lösen. Wir lehren das Programmieren ähnlich wie das Schreiben. Wir beginnen damit, Programme zu lesen und zu erklären. Dann schreiben wir einfache Programme und mit der Zeit immer komplexere Programme. Durch das wiederholte Schreiben von Programmen schleift sich eine Routine ein und man beginnt bei neuen Problemstellungen geeignete Lösungsmuster von selbst zu erkennen. Wenn man diesen Punkt erreicht hat, wird das Programmieren zu einem sehr angenehmen und kreativen Prozess.

Wenn Sie nun mit dem Erlernen des Vokabulars und der Struktur von (Python-) Programmen beginnen, seien Sie geduldig und bleiben Sie motiviert, auch einfache Beispiele nachzuvollziehen und zu variieren. Denken Sie vielleicht daran wie es war, Lesen und Schreiben zu lernen. Auch dies ist am Anfang mühsam gewesen, hat Ihnen aber schlussendlich das Tor geöffnet, um Wissen zu erlangen und weiterzuentwickeln.

1.4. Wörter und Sätze

Im Gegensatz zu menschlichen Sprachen ist der Wortschatz von Python ziemlich klein. Wir nennen diesen „Wortschatz“ die „reservierten Wörter“. Das sind Wörter, die für Python eine ganz besondere Bedeutung haben. Wenn Python diese Wörter in einem Python-Programm sieht, haben sie eine (und nur eine) Bedeutung für Python. Später, wenn wir Programme schreiben, werden wir unsere eigenen Wörter erfinden, die für uns eine Bedeutung haben und *Bezeichner* genannt werden. Bei der Wahl der Namen für unsere Bezeichner haben wir einen großen Spielraum, aber wir können keines der reservierten Wörter von Python als Namen für eigene Zwecke verwenden.

Zu den reservierten Wörtern in der Sprache Python gehören die folgenden:

```
and     as        assert  break   class   continue  def
del     elif      else    except  False   finally   for
from    global    if      import  in      is        lambda
None    nonlocal  not     or      pass    raise     return
True    try       while   with    yield
```

Wir werden diese reservierten Wörter und ihre Verwendung zu gegebener Zeit lernen, aber jetzt konzentrieren wir uns erst einmal darauf, wie wir unser Python-Programm mit uns *sprechen* lassen können. Da Programme üblicherweise nicht in Form von gesprochener Sprache, sondern eher durch das Anzeigen von Texten und Bildern „reden“, heißt das Kommando `print` also *Drucke*:

```
print('Hello world!')
```

Damit haben wir unseren ersten syntaktisch korrekten Python-Satz geschrieben. Der Satz beginnt mit der Funktion `print`, gefolgt von einer Zeichenfolge unserer Wahl, die in einfachen Anführungszeichen steht. Die Zeichenketten in den `print`-Anweisungen sind in Anführungszeichen eingeschlossen. Einfache Anführungszeichen und doppelte Anführungszeichen haben die gleiche Funktion; die meisten Leute verwenden einfache Anführungszeichen, außer in den Fällen, wo ein einfaches Anführungszeichen (ein Apostroph) in der Zeichenkette selbst erscheint.

Wie Sie der Tabelle oben entnehmen können, ist *print* kein reserviertes Wort, sondern ein *Bezeichner*. In diesem Fall haben aber nicht wir den Bezeichner (also den Namen) eingeführt, sondern es gibt ihn bereits *in Python*. Bezeichner, auf denen im Programm direkt eine öffnende Klammer folgt bezeichnen i. d. R. Funktionen, also soetwas wie Unterprogramme, die eine bestimmte Teilaufgabe erledigen. Python bietet eine Vielzahl von solchen Funktionen, die Ihnen das Leben als Programmierer sehr erleichtern. Wir werden im Verlauf des Buches noch viele dieser Funktionen kennen lernen.

1.5. Konversation mit Python

Nachdem wir nun ein Wort und einen einfachen Satz in Python kennen, müssen wir wissen, wie wir eine Unterhaltung mit Python beginnen können, um unsere neuen Sprachkenntnisse zu testen.

Bevor wir uns mit Python unterhalten können, müssen wir zunächst die Python-Software auf unserem Computer installieren und lernen, wie man Python auf diesem startet. Sie fragen sich nun vielleicht, warum Sie für das Ausführen Ihrer Python-Programme ein anderes Programm auf Ihrem Computer installieren müssen. Das liegt daran, dass Ihre CPU die Sprache Python nicht direkt versteht. Ihr Python-Programm muss also vor – oder besser gesagt *bei* der Ausführung – von der Sprache Python in die Sprache der CPU *übersetzt* werden. Passiert dieses Übersetzten vor dem Starten des Programms (und damit in der Regel einmalig), nennt man den Vorgang *Kompilieren*. Werden Programme unmittelbar bei der Ausführung, und damit jedes Mal erneut übersetzt, so nennt man das *Interpretieren*.

Letzteres ist bei Python der Fall und daher müssen wir einen *Python-Interpreter* auf unserem PC oder Notebook installieren. Leider gibt es hierzu nicht „die eine Anleitung“. Python ist eine *offene* Programmiersprache. Das bedeutet, dass die Regeln, wie Python-Programme geschrieben werden müssen und wie die Anweisungen der Sprache funktionieren, auf einem gemeinschaftsbasiertes Entwicklungsmodell

beruht und vollkommen offengelegt ist. Jeder kann also, nach den vorgegebenen Regeln, einen Python-Interpreter entwickeln und anbieten. Es gibt allerdings eine Standardversion, die man über die Homepage von Python (www.python.org) herunterladen kann. Hier gibt es auch verschiedene Versionen für Windows, MacOS und Linux.

Im Gegensatz zu vielen anderen Programmiersprachen kann man in Python nicht nur ganze Programme starten, sondern man kann auch interaktiv arbeiten und dem Computer einem Befehl nach dem anderen geben. Um dies zu tun, müssen wir auf unserem Computer ein Kommandozeilenfenster öffnen und `python` eingeben. Ist Python korrekt installiert, wird durch diesen Aufruf der *Python-Interpreter* im *interaktiven Modus* gestartet. Das Fenster sollte dann in etwa so aussehen:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Die Eingabeaufforderung `>>>` ist die Art und Weise, wie der Python-Interpreter fragt: „Was soll ich als Nächstes tun?“ Python ist bereit, ein Gespräch mit uns zu führen. Alles, was wir wissen müssen, ist, wie man die Sprache Python spricht.

Nehmen wir an, wir kennen nicht einmal die einfachsten Wörter oder Sätze in Python. Wir könnten den Standardsatz verwenden, den Astronauten verwenden, wenn sie auf einem fernen Planeten landen und versuchen, mit den Bewohnern des Planeten zu sprechen:

```
>>> I come in peace, please take me to your leader
File "<stdin>", line 1
  I come in peace, please take me to your leader
       ^
SyntaxError: invalid syntax
>>>
```

Das läuft nicht so gut. Wenn uns nicht schnell etwas einfällt, werden die Bewohner des Planeten uns wahrscheinlich nicht sonderlich ernst nehmen. Probieren wir doch lieber mal einen Python-Satz, von dem wir bereits wissen, dass er korrekt ist:

```
>>> print('Hello world!')
Hello world!
```

Das sieht schon viel besser aus, also versuchen wir, noch etwas mehr zu kommunizieren:

```
>>> print('You must be the legendary god that comes from the sky')
You must be the legendary god that comes from the sky
>>> print('We have been waiting for you for a long time')
We have been waiting for you for a long time
>>> print('Our legend says you will be very tasty with mustard')
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
File "<stdin>", line 1
  print 'We will have a feast tonight unless you say
                                                    ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

Das Gespräch lief eine Zeit lang sehr gut, und dann haben wir den kleinsten Fehler bei der Verwendung der Sprache Python gemacht, und Python hält uns diesen Fehler gnadenlos vor. An diesem Punkt kann man bereits erkennen, dass Python erstaunlich komplex und mächtig ist und dabei sehr wählerisch ist, was die Syntax angeht, die wir zur Programmierung verwenden. Gleichwohl ist Python aber *nicht* intelligent.

Auch wenn es für uns klar ist, was die Anweisung tun soll, wird Python nicht arbeiten können, solange es einen Fehler bei der Syntax gibt.

Bevor wir nun unser erstes Gespräch mit dem Python-Interpreter beenden, sollten wir noch wissen, wie man sich korrekt von Python „verabschiedet“:

```
>>> good-bye
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined
>>> if you don't mind, I need to leave
File "<stdin>", line 1
  if you don't mind, I need to leave
           ^
SyntaxError: invalid syntax
>>> quit()
```

Man kann feststellen, dass der Fehler bei den ersten beiden Fehlversuchen anders ist. Der zweite Fehler ist anders, weil `if` ein reserviertes Wort ist und Python das reservierte Wort sah und dachte, wir wollten etwas sagen, aber die Syntax des Satzes war falsch.

Der richtige Weg, sich von Python zu verabschieden, ist die Eingabe von `quit()` am interaktiven `>>>`-Prompt. Wir hätten wahrscheinlich eine ganze Weile gebraucht, um das zu erraten, also wird es sich als hilfreich erweisen, ein Buch zur Hand zu haben.

1.6. Interpreter und Compiler

Python ist eine *Hochsprache*, die für Menschen relativ einfach zu lesen und zu schreiben und für Computer zu lesen und zu verarbeiten ist. Andere Hochsprachen sind Java, C++, PHP, Ruby, Basic, Perl, JavaScript und viele mehr. Die eigentliche Hardware in der Central Processing Unit (CPU) versteht keine dieser Hochsprachen.

Die CPU versteht eine Sprache, die wir *Maschinensprache* nennen. Maschinensprache ist sehr einfach und ehrlich gesagt sehr mühsam zu schreiben, weil sie nur aus Nullen und Einsen besteht:

```
00101000111010010010101010000001111
11100110000011101010010101101101
...
```

Die Maschinensprache scheint auf den ersten Blick recht einfach zu sein, da es nur Nullen und Einsen gibt, aber ihre Syntax ist noch komplexer und weitaus komplizierter als Python. Daher schreiben nur sehr wenige Programmierer jemals Maschinensprache. Stattdessen entwickeln wir verschiedene Übersetzer, die es Programmierern ermöglichen, Hochsprachen wie Python oder JavaScript zu schreiben, und diese Übersetzer wandeln die Programme in Maschinensprache um, die dann von der CPU ausgeführt wird.

Da die Maschinensprache an die Computerhardware gebunden ist, ist die Maschinensprache nicht *portabel* über verschiedene Arten von Hardware. In Hochsprachen geschriebene Programme können zwischen verschiedenen Computern übertragen werden, indem ein anderer Interpreter auf dem neuen Computer verwendet wird oder der Code neu kompiliert wird, um eine Maschinensprachversion des Programms für die neue Maschine zu erhalten.

Wie schon im oberen Teil dieses Kapitels angedeutet, lassen sich Programmiersprachenübersetzer in zwei allgemeine Kategorien einteilen: (1) Interpreter und (2) Compiler.

Ein *Interpreter* liest den Quellcode des Programms, wie er vom Programmierer geschrieben wurde, analysiert den Quellcode und interpretiert die Befehle im laufenden Betrieb. Python ist ein Interpreter, und wenn wir Python interaktiv ausführen, können wir eine Zeile (einen Satz) in Python eingeben und Python verarbeitet sie sofort und ist bereit für die Eingabe einer weiteren Python-Zeile.

Bei den Anweisungen in Programmiersprachen kommt es häufig vor, dass wir uns einen Wert für eine spätere Aufgabe merken wollen. In Python können wir das ganz einfach erledigen, indem wir uns einen Namen ausdenken und den Wert mittels eines Gleichheitszeichens dem Namen *zuweisen*.

Wir kennen das Prinzip aus der Mathematik, wo man spätestens in der 5. oder 6. Schulklasse das Rechnen mit *Variablen* erlernt. Höhere Mathematik ohne die Verwendung von Variablen ist praktisch nicht möglich, denn sie sind *das* zentrale Mittel, um Regeln oder Aussagen zu verallgemeinern. Auch in der Programmierung nennt man einen Platzhalter für Werte i. d. R. *Variable*. Dass Python eigentlich keine Variablen verwendet, sondern ausschließlich Namen ist ein technisches Detail. Weil der Begriff der *Variablen* aber so verbreitet ist – in den allermeisten Python Büchern wird von Variablen gesprochen – verwenden wir ihn auch in diesem Buch.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

In diesem Beispiel bitten wir Python, sich den Wert `6` zu merken und dafür den Namen `x` zu verwenden. Wir überprüfen, ob Python sich den Wert tatsächlich gemerkt hat, indem wir `print` verwenden. Dann bitten wir Python, den Wert `x` abzurufen, mit sieben zu multiplizieren und den neu berechneten Wert unter dem Namen `y` zu speichern. Dann möchten wir uns den Wert anzeigen lassen, der sich gerade hinter dem Namen `y` befindet.

Auch wenn wir diese Befehle Zeile für Zeile in Python eingeben, behandelt Python sie als eine geordnete Folge von Anweisungen, wobei spätere Anweisungen Daten abrufen können, die in früheren Anweisungen erstellt wurden. Wir schreiben unseren ersten einfachen Absatz mit vier Sätzen in einer logischen und sinnvollen Reihenfolge.

Es liegt in der Natur eines *Interpreters*, dass er in der Lage ist ein interaktives Gespräch zu führen, wie oben gezeigt. Ein *Compiler* muss das *gesamte* Programm in einer oder mehreren Dateien erhalten. Dann führt er einen Prozess aus, um den High-Level-Quellcode in Maschinensprache zu übersetzen. Danach stellt der Compiler die resultierende Maschinensprache in einer Datei zur späteren Ausführung zur Verfügung.

Wenn wir ein Windows-System haben, haben diese ausführbaren Maschinensprache-Programme die Endungen `.exe` oder `.dll`, welche für „ausführbar“ bzw. „dynamisch gelinkte Bibliothek“ stehen. Unter Linux und Macintosh gibt es kein Suffix, das eine Datei eindeutig als ausführbar kennzeichnet.

Wenn wir eine ausführbare Datei in einem Texteditor öffnen würden, sähe sie völlig verrückt aus und wäre unlesbar:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

Es ist nicht einfach, Maschinensprache zu lesen oder zu schreiben, daher ist es gut, dass wir *Interpreter* und *Compiler* haben, die es uns ermöglichen, in Hochsprachen wie Python oder C zu schreiben.

An diesem Punkt in unserer Diskussion über Compiler und Interpreter, sollte man sich ein wenig über den Python-Interpreter selbst Gedanken machen. In welcher Sprache ist er geschrieben? Ist er in einer kompilierten Sprache geschrieben? Wenn wir `python` eintippen, was genau passiert dann?

Der (Standard-) Python-Interpreter ist in einer Hochsprache namens „C“ geschrieben. Wir können uns den eigentlichen Quellcode des Python-Interpreters ansehen, indem wir www.python.org aufrufen und uns zum Quellcode durcharbeiten. Python ist also selbst ein Programm und wird in Maschinencode kompiliert. Als wir Python auf unserem Computer installiert haben, haben wir eine Maschinencode-Kopie des übersetzten Python-Programms auf unser System geladen. Unter Windows befindet sich der ausführbare Maschinencode für Python wahrscheinlich in einer Datei mit einem Namen wie:

```
C:\Python35\python.exe
```

Das ist mehr, als man wissen muss, um ein Python-Programmierer zu werden, aber manchmal lohnt es sich, diese kleinen, nervigen Fragen gleich zu Beginn zu beantworten.

1.7. Ein Programm schreiben

Das Eingeben von Befehlen in den Python-Interpreter ist ein guter Weg, um mit den Funktionen von Python zu experimentieren, jedoch ist es nicht empfehlenswert für die Lösung komplexer Probleme.

Wenn wir ein Programm schreiben wollen, verwenden wir einen Texteditor, um die Python-Anweisungen in eine Datei zu schreiben, die *Skript* genannt wird. Konventionell haben Python-Skripte die Endung `.py`.

Um das Skript auszuführen, müssen wir dem Python-Interpreter den Namen der Datei mitteilen. In einem Befehlsfenster würden wir `python hallo.py` wie folgt eingeben:

```
$ cat hello.py
print('Hello world!')
$ python hello.py
Hello world!
```

Das `\$` ist die Eingabeaufforderung des Betriebssystems, und das `cat hello.py` zeigt zeigt uns, dass die Datei `hello.py` ein einzeiliges Python-Programm enthält, das eine Zeichenkette druckt.

Wir rufen den Python-Interpreter auf und sagen ihm, dass er den Quellcode aus der Datei `hello.py` lesen soll, anstatt uns interaktiv nach Python-Codezeilen zu fragen.

Man kann feststellen, dass es am Ende des Python-Programms keine Notwendigkeit für `quit()` gibt. Wenn Python den Quellcode aus einer Datei liest, weiß es, dass es aufhören muss, wenn das Ende der Datei erreicht wurde.

1.8. Was ist ein Programm?

Die Definition eines *Programms* ist im Grunde genommen eine Abfolge von Python-Anweisungen, die so gestaltet sind, dass sie etwas tun. Selbst unser einfaches Skript `hello.py` ist ein Programm. Es ist ein einzeiliges Programm, das nicht besonders nützlich ist, aber nach der strengsten Definition ist es ein Python-Programm.

Es ist vielleicht am einfachsten zu verstehen, was ein Programm ist, wenn man an ein Problem denkt, für dessen Lösung ein Programm erstellt werden könnte, und dann ein Programm betrachtet, dass dieses Problem lösen würde.

Nehmen wir an, wir forschen im Bereich Social-Computing über Facebook-Posts und interessieren uns für das am häufigsten verwendete Wort in einer Reihe von Beiträgen. Wir könnten den Datenstrom der Facebook-Posts ausgeben und den Text nach dem häufigsten Wort durchsuchen, aber das würde sehr viel Zeit in Anspruch nehmen und wäre sehr fehleranfällig. Es wäre klug, ein Python-Programm zu schreiben, dass diese Aufgabe schnell und genau erledigt, damit wir das Wochenende mit etwas schönerem verbringen können.

Betrachten wir zum Beispiel den folgenden Text über einen Clown und ein Auto. Sehen wir uns den Text an und finden heraus, welches Wort am häufigsten vorkommt und wie oft es vorkommt.

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

Dann stellen wir uns vor, dass wir diese Aufgabe erledigen, indem wir uns Millionen von Zeilen von Text anschauen. Offen gesagt wäre es schneller, Python zu lernen und ein Python-Programm zu schreiben, um die Wörter zu zählen, als sie manuell durchzusehen.

Die gute Nachricht ist, dass bereits ein einfaches Programm entwickelt wurde, um das häufigste Wort in einer Textdatei zu finden. Es wurde schon geschrieben und getestet, damit wir etwas Zeit sparen können.

```
name = input('Welche Datei?:')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(f'Das Wort "{bigword}" kommt {bigcount}-mal vor')

# Code: https://tiny.one/py4de/code3/words.py
```

Wir müssen nicht einmal Python beherrschen können, um dieses Programm zu nutzen. Als Endanwender benutzen wir einfach das Programm und freuen uns darüber, wie viel manuelle Arbeit wir gespart haben.

Dies ist ein gutes Beispiel dafür, wie Python und die Python-Sprache als Vermittler zwischen uns (dem Endbenutzer) und den Programmierern fungieren. Python bietet uns die Möglichkeit, nützliche Befehlssequenzen bzw. Programme in einer gemeinsamen Sprache auszutauschen, die von jedem verwendet werden kann, der Python auf seinem Computer installiert. Keiner von uns spricht also *mit Python*, sondern wir kommunizieren miteinander *durch* Python.

1.9. Die Bausteine von Programmen

In den nächsten Kapiteln, werden wir mehr über das Vokabular, die Satzstruktur, Absatzstruktur und Erzählstruktur von Python lernen. Uns werden die mächtigen Fähigkeiten von Python näher gebracht werden und wir werden uns aneignen, wie man diese Fähigkeiten zusammensetzen kann, um nützliche Programme zu erstellen.

Es gibt einige konzeptionelle Muster auf niedriger Ebene, die wir zum Erstellen von Programmen benutzen. Diese Konstrukte sind nicht nur für Python-Programme geeignet, sie sind Teil jeder Programmiersprache, von der Maschinensprache bis hin zu den Hochsprachen. Das obige Wortzählprogramm verwendet alle bis auf eines dieser Muster.

Eingabe Abrufen von Daten aus der „Außenwelt“. Dies kann das Lesen von Daten aus einer Datei sein oder sogar aus Sensoren wie einem Mikrofon oder GPS. In unseren ersten Programmen wird die Eingabe durch den Benutzer erfolgen, der Daten über die Tastatur eingibt.

Ausgabe Anzeigen der Ergebnisse des Programms auf einem Bildschirm oder Speichern in einer Datei.

Sequentielle Ausführung Die Ausführung von Anweisungen nacheinander in der Reihenfolge wie sie im Skript vorkommen.

Bedingte Ausführung Prüfung auf bestimmte Bedingungen und anschließende Ausführung oder Überspringen einer Folge von Anweisungen.

Wiederholte Ausführung Wiederholtes Ausführen einer Reihe von Anweisungen, normalerweise mit einer gewissen Variation.

Wiederverwenden Reihe von Anweisungen einmal schreiben und ihnen einen Namen geben. Danach verwenden wir dann diese Anweisungen je nach Bedarf in unserem Programm wieder.

Dass alle Programme fast ausschließlich aus diesen Mustern bestehen, klingt fast zu einfach, um wahr zu sein. Und natürlich ist es nicht ganz so einfach, ein neues Programm zu schreiben. Um einen Vergleich anzustellen, könnte man sagen, dass Gehen einfach „einen Fuß vor den anderen setzen“ bedeutet; und

trotzdem braucht ein Kleinkind sehr viel Übung um richtig Laufen zu können. Beim Programmieren ist das Erlernen der Grundmuster recht einfach. Die „Kunst“ ein Programm zu schreiben besteht aber darin, diese Grundelemente immer wieder neu zusammenzusetzen und zu verweben, um eine nützliche Lösung für ein gegebenes Problem zu schaffen.

1.10. Was kann schon schief gehen?

Wie wir in unseren ersten „Gesprächen“ mit Python gesehen haben, müssen wir sehr genau sein, wenn wir Python-Code schreiben. Die kleinste Abweichung oder der kleinste Fehler führen dazu, dass Python unser Programm nicht ausführen kann.

```
C:\Python> python.exe
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit
(AMD64)]
Type "help", "copyright", "credits" or "license" for more
information.
>>> primt 'Hello World'
  File "<stdin>", line 1
    primt 'Hello World'
          ^
SyntaxError: invalid syntax
>>> primt('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined
>>> Come on, Python
  File "<stdin>", line 1
    Come on, Python
         ^
SyntaxError: invalid syntax
>>>
```

Es ist aussichtslos, mit Python zu verhandeln, doch bitte den Code zu verstehen. Der Interpreter kann nur gültige Python Anweisungen ausführen. Beim kleinsten Fehler wird er abbrechen und ausgeben, an welcher Stelle er beim „Verstehen“ nicht weiter gekommen ist. Wenn Python `SyntaxError: invalid syntax` sagt, bedeutet das: „Ich verstehe einfach nicht was du gemeint hast, aber bitte rede weiter mit mir (`>>>`).“ Ein `NameError` bedeutet, „du scheinst etwas verwenden zu möchten, was ich nicht kenne.“

Wenn die Programme immer anspruchsvoller werden, wird man auf drei allgemeine Arten von Fehlern stoßen:

Syntaxfehler Dies sind die ersten Fehler, die man machen wird, und die am einfachsten zu beheben sind. Ein Syntaxfehler bedeutet, dass wir die „Grammatikregeln“ von Python verletzt haben. Python tut sein Bestes, um genau auf die Zeile und das Zeichen zu zeigen wo es bemerkt hat, dass es verwirrt war. Das einzig tückische an Syntaxfehlern ist, dass der Fehler, der behoben werden muss, manchmal eigentlich früher im Programm liegt als an der Stelle, an der Python *verwirrt* war. Also die Zeile und das Zeichen die Python bei einem Syntaxfehler anzeigt, sind zunächst nur ein Ausgangspunkt für unsere Untersuchungen.

Semantische Fehler Ein semantischer Fehler liegt vor, wenn die Syntax, also die Grammatik des Quellcodes eigentlich korrekt ist, das Programm *so* aber nicht funktioniert. Der `NameError` oben ist ein Beispiel dafür: Es *könnte* eine Funktion namens `primt` existieren und wenn es sie gäbe, wäre das Programm vermutlich korrekt. Es gibt diese Funktion aber im vorliegenden Programm nicht und daher tritt hier bei der Ausführung der Anweisung ein Fehler auf.

Logikfehler Logikfehler sind besonders tückisch, denn sie können von Python so gut wie gar nicht erkannt werden. Es handelt sich um Fehler in der Logik des Programms. Das Programm ist vollkommen korrekt, aber es tut nicht, was wir *beabsichtigt* haben. Ein einfaches Beispiel wäre, wenn wir einer Person den Weg zu einem Restaurant erklären und bei der Wegbeschreibung einmal Links und

Rechts verwechseln. Die Person kann daraufhin die Beschreibung nachvollziehen, biegt aber an einer Kreuzung in die völlig falsche Richtung ab und kommt höchstwahrscheinlich niemals an dem Restaurant an.

Bei allen drei Arten von Fehlern versucht Python lediglich genau das zu tun, was wir verlangt haben. Dass Python die Ausführung abbricht, sobald es einen Fehler erkennt, und nicht einfach versucht zu verstehen, was wir mit der fehlerhaften Anweisung *gemeint* haben, wird sich noch als sehr nützlich herausstellen. Sie werden mit jedem Fehler lernen, wie man korrekten Quellcode schreibt. Gleichzeitig können Sie immer davon ausgehen, dass Python genau das tut, was Sie programmiert haben. Es gibt keinen Interpretationsspielraum beim Abarbeiten der Anweisungen; wenn das Programm nicht funktioniert liegt das immer am Programm selbst und nicht daran, dass Python etwas falsch verstanden hat.

1.11. Debugging

Wenn Python einen Fehler ausspuckt oder sogar ein Ergebnis liefert, dass sich von dem unterscheidet, was wir beabsichtigt hatten, dann beginnt die Suche nach der Fehlerursache. Beim Debugging geht es darum, die Ursache des Fehlers in unserem Code zu finden. Bei der Fehlersuche in einem Programm, insbesondere wenn wir an einem schwerwiegenden Fehler arbeiten, gibt es vier Dinge, die wir versuchen sollten:

Lesen Nochmal den Code durch erneutes Lesen prüfen. Haben wir alles so gesagt, wie wir es sagen wollten?

Ausführen Man sollte einfach nochmal experimentieren, indem man nachverfolgbare Änderungen vornimmt und verschiedene Versionen ausführt. Häufig wir das Problem offensichtlich, wenn wir es an der richtigen Stelle im Programm anzeigen lassen. Manchmal muss man etwas Zeit aufwenden, um ein gutes Gerüst zu bauen.

Nachdenken Man sollte sich Zeit zum Nachdenken nehmen. Was für ein Fehler ist es? Handelt es sich um einen Syntaxfehler, einen Laufzeitfehler oder einen semantischen Fehler? Welche Informationen können wir aus den Fehlermeldungen bzw. der Ausgabe des Programms entnehmen? Welche Art von Fehler könnte die Ursache sein für das Problem, das man sehen kann? Was haben wir zuletzt geändert, bevor das Problem auftrat?

Rückzug Irgendwann ist es am besten sich zurückzuziehen und die letzten Änderungen rückgängig zu machen, bis man wieder ein Programm hat, das funktioniert und das man verstehen kann. Dann kann man mit dem Neuaufbau beginnen.

Programmieranfänger bleiben manchmal bei einer dieser Aktivitäten stecken und vergessen die anderen. Um einen Fehler zu finden, muss man lesen, experimentieren, grübeln und sich manchmal zurückziehen. Wenn man bei einer dieser Aktivitäten nicht weiter kommt, versucht man die anderen. Jede Aktivität hat ihren eigenen Fehlermodus.

Das Lesen des Codes kann zum Beispiel helfen, wenn das Problem ein Tippfehler ist, aber nicht, wenn es sich um ein konzeptionelles Missverständnis handelt. Wenn man nicht versteht, was das Programm tut, kann man es 100 Mal lesen, ohne den Fehler zu sehen.

Experimentieren kann hilfreich sein, vor allem, wenn man kleine, einfache Tests durchführt. Wenn man jedoch Experimente durchführt, ohne dabei nachzudenken oder den Code genau zu lesen, könnte man in ein Muster verfallen, dass man als „Zufallsprogrammierung“ bezeichnen könnte. Dabei handelt es sich um einen Prozess, bei dem zufällige Änderungen vorgenommen werden, bis das Programm (vielleicht) irgendwann das Richtige tut. Es ist unnötig zu erwähnen, dass diese Art der Programmierung sehr lange dauern kann.

Man muss sich Zeit zum Nachdenken nehmen. Fehlersuche ist wie eine experimentelle Wissenschaft. Man sollte mindestens eine Hypothese darüber haben, was das Problem ist. Wenn es zwei oder mehr Möglichkeiten gibt, versucht man, einen Test zu finden, der eine von ihnen ausschließt.

Eine Pause hilft beim Nachdenken. Das gilt auch für das Reden. Wenn man das Problem einer anderen Person (oder sogar sich selbst) erklärt, findet man manchmal die Antwort, bevor man die Frage zu Ende gestellt hat.

Aber selbst die besten Debugging-Techniken versagen, wenn es zu viele Fehler gibt, oder wenn der Code, den wir zu beheben versuchen, zu groß und kompliziert ist. Manchmal ist es am besten, sich zurückzuziehen und das Programm zu vereinfachen, bis man zu einem Ergebnis gelangt, welches funktioniert und man versteht.

Programmieranfänger zögern oft, sich zurückzuziehen, weil sie es nicht ertragen können eine Codezeile zu löschen (selbst wenn sie falsch ist). Wenn man sich dadurch besser fühlt, sollte man einfach das Programm in eine andere Datei sichern, bevor man es zerlegt. Dann kann man die Teile nach und nach wieder Stück für Stück einfügen.

Dass Sie größere Teile Ihres Programms löschen müssen, können Sie meist auch ganz gut verhindern, indem Sie Ihren Lösungsweg für die gesamte Aufgabe in kleinere Teilaufgaben einteilen. Überlegen Sie sich bevor Sie mit dem Schreiben des Codes anfangen, welche logischen Teilfunktionen in Ihrer Aufgabe stecken. Und dann beginnen Sie, die erste Teilaufgabe zu lösen und – ganz wichtig – zu testen. Wenn Sie sich einigermaßen sicher sind, dass die erste Teilaufgabe gelöst ist, nehmen Sie sich die zweite vor, usw. Diese Vorgehensweise hilft sehr, mögliche Fehlerursachen einzukreisen. Wenn Sie stattdessen zu viel „auf einen Schlag“ programmieren, können mehr Fehler auftreten und die Fehlersuche sowie die Beseitigung dauern viel länger.

1.12. Der Lernprozess

Im weiteren Verlauf des Buches sollte man keine Angst haben, wenn die Konzepte beim ersten Mal nicht gut zusammenzupassen scheinen. Als wir sprechen gelernt haben, war es in den ersten Jahren kein Problem, dass man nur niedliche Gluckgeräusche gemacht hat. Es war in Ordnung, wenn es sechs Monate dauerte, um von einfachen Vokabeln zu einfachen Sätzen zu kommen, und 5–6 Jahre brauchte, um von Sätzen zu Absätzen zu kommen, und noch ein paar Jahre, um in der Lage zu sein, selbständig eine interessante, vollständige Kurzgeschichte zu schreiben.

Wir wollen, dass wir Python viel schneller lernen, deshalb lernen wir in den nächsten Kapiteln alles gleichzeitig. Aber es ist wie beim Erlernen einer neuen Sprache, die man sich erst aneignen und verstehen muss, bevor sie sich natürlich anfühlt. Das führt zu einiger Verwirrung, wenn wir Themen lernen und wieder aufgreifen, um das große Ganze zu sehen, während wir die winzigen Fragmente definieren, die dieses große Bild ausmachen. Das Buch ist zwar linear geschrieben und wenn man einen Kurs belegt, wird er auch linear verlaufen. Man sollte jedoch nicht zögern, sich dem Stoff sehr unlinear zu nähern. Egal ob vorwärts und rückwärts oder quer gelesen. Durch Überfliegen von fortgeschrittenem Material, ohne die Details vollständig zu verstehen, erhält man ein besseres Verständnis für das „Warum?“ der Programmierung. Durch Wiederholung früherer Inhalte und sogar Wiederholung früherer Übungen werden wir feststellen, dass wir tatsächlich viel gelernt haben, auch wenn das Material, welches wir gerade anstarren, ein wenig undurchdringlich erscheint.

Wenn wir unsere erste Programmiersprache lernen, gibt es normalerweise ein paar wunderbare „Aha!“-Momente. Dinge, für die Sie vorher Stunden oder Tage benötigt hätten, können automatisiert in Bruchteilen von Sekunden ausgeführt werden. Dazu gehört aber auch, dass man manchmal beim Lernen der Programmiersprache Aufgaben zu lösen hat, die nicht wirklich ein Problem für einen selbst darstellen. Nehmen Sie solche Aufgaben als „Fingerübungen“ hin, die Ihnen helfen, Routine beim Programmieren zu entwickeln.

1.13. Glossar

Bug Ein Fehler im Programmcode.

Central Processing Unit Das Herz eines jeden Computers. Auf ihm läuft die Software, die wir schreiben; es wird auch *CPU* oder *Prozessor* genannt.

Kompilieren Übersetzung eines in einer Hochsprache geschriebenen Programms in eine niedrigere Sprache, um es später ausführen zu können.

Hochsprache Eine Programmiersprache wie Python, die so konzipiert wurde, dass sie für Menschen leicht zu lesen und zu schreiben ist.

Interaktiver Modus Eine Methode zur Verwendung des Python-Interpreters durch Eingabe von Befehlen und Ausdrücken in der Eingabeaufforderung.

Interpretieren Ausführen eines Programms in einer Hochsprache durch zeilenweises Übersetzen.
Low-Level-Sprache Eine Programmiersprache, die so konzipiert ist, dass sie von einem Computer leicht ausgeführt werden kann. Auch *Maschinencode* oder *Assemblersprache* genannt.
Maschinencode Die niedrigste Sprache für Software, also die Sprache, die direkt von der zentralen Recheneinheit (CPU) ausgeführt wird.
Hauptspeicher Speichert Programme und Daten. Der Hauptspeicher verliert seine Informationen, wenn der Strom ausgeschaltet wird.
Parsen Ein Programm untersuchen und die syntaktische Struktur analysieren.
Portabilität Eine Eigenschaft eines Programms, das auf mehr als einer Art von Endgerät laufen kann.
Print-Funktion Eine Anweisung, die den Python-Interpreter veranlasst, einen Wert auf dem Bildschirm anzuzeigen.
Problemlösungsprozess Der Prozess ein Problem zu formulieren, eine Lösung zu finden und die Lösung umzusetzen.
Programm Ein Satz von Anweisungen, der eine Berechnung vorgibt.
Eingabeaufforderung Wenn ein Programm eine Meldung anzeigt und eine Pause macht, damit der Benutzer eine Eingabe im Programm machen kann.
Sekundärspeicher Speichert Programme und Daten und behält die Informationen auch dann bei, wenn der Strom abgeschaltet wird. Im Allgemeinen langsamer als der Hauptspeicher. Beispiele für Sekundärspeicher sind z. B. Festplattenlaufwerke und Flash-Speicher in USB-Sticks.
Semantik Die Bedeutung eines Programms.
Semantischer Fehler Ein Fehler in einem Programm, der dazu führt, dass es etwas anderes tut, als der Programmierer beabsichtigt hat.
Quellcode Der Programmcode eines in einer Hochsprache geschriebenen Programms.

1.14. Übungen

Übung 1: Welche Funktion hat der Sekundärspeicher in einem Computer?

a) Alle Berechnungen und die Logik des Programms ausführen
b) Abruf von Webseiten über das Internet
c) Informationen langfristig zu speichern, auch über einen Stromausfall hinaus
d) Eingaben des Benutzers entgegennehmen

Übung 2: Was ist ein Programm?

Übung 3: Was ist der Unterschied zwischen einem Compiler und einem Interpreter?

Übung 4: Was enthält Maschinencode?

a) Der Python-Interpreter
b) Die Tastatur
c) Die Python-Quelldatei
d) Ein Textverarbeitungsdokument

Übung 5: Was ist falsch an folgendem Code?

```
>>> primt 'Hello world!'
File "<stdin>", line 1
  primt 'Hello world!'
                     ^
SyntaxError: invalid syntax
>>>
```

Übung 6: Wo im Computer wird eine Variable wie `x` gespeichert, nachdem die folgende Python-Zeile beendet ist?

```
x = 123
```

a) CPU
b) Hauptspeicher
c) Sekundärspeicher
d) Eingabegeräte
e) Ausgabegeräte

Übung 7: Was wird das folgende Programm ausgeben:

```
x = 43
x = x + 1
print(x)
```

a) 43
b) 44
c) x + 1
d) Fehler, denn x = x + 1 ist mathematisch nicht möglich

Übung 8: Erläutern Sie die folgenden Punkte anhand eines Beispiels für eine menschliche Fähigkeit: (1) CPU, (2) Hauptspeicher, (3) Sekundärspeicher, (4) Eingabegerät und (5) Ausgabegerät. Zum Beispiel: „Was ist das menschliche Äquivalent zu einer CPU?“

Übung 9: Wie behebt man einen Syntaxfehler?