Premature optimization is the root of all evil (or at least most of it) in programming.

Donald Ervin Knuth

2 Einführung in Python

Bevor wir mit der Bearbeitung von physikalischen Fragestellungen mithilfe des Computers beginnen können, müssen wir uns zunächst einige grundlegende Arbeitstechniken aneignen. Damit Sie mit diesem Buch arbeiten können, sollten Sie mit den grundlegenden Funktionen Ihres Computers und des Betriebssystems vertraut sein. Sie sollten insbesondere wissen, wie man Programme auf dem Computer startet und wie Dateien in der Verzeichnis- oder Ordnerstruktur des Computers abgelegt werden. Wenn Sie schon Erfahrungen im Umgang mit Python haben, sollten Sie die folgenden Abschnitte dennoch lesen, weil wir neben den Grundlagen der Sprache Python auch einige Konventionen besprechen, die uns durch das gesamte Buch begleiten werden. Wenn Sie schon Erfahrungen mit einer anderen Programmiersprache haben, sollten Sie beim Lesen der folgenden Abschnitte genau darüber nachdenken, an welchen Stellen sich die Arbeit mit Python von der Ihnen bekannten Programmiersprache unterscheidet.

Es ist wahrscheinlich unmöglich, in nur einem Kapitel eine umfassende Einführung in die Programmiersprache Python zu geben. Wir können daher nur eine kleine Teilmenge der Programmiersprache Python besprechen. Darüber hinaus kann man leicht auch ein ganzes Buch nur über die Erzeugung von grafischen Ausgaben mit der Bibliothek Matplotlib füllen, und ein weiteres Buch kann man sicher über das Numerikpaket NumPy schreiben. Nun soll es nicht so sein, dass Sie erst drei Bücher lesen müssen, bevor Sie damit anfangen können, physikalische Aufgaben mit dem Computer zu bearbeiten. Wir werden daher in diesem Kapitel zunächst die wichtigsten Aspekte der Programmiersprache Python an ganz konkreten Beispielen kennen lernen, ohne dabei für jeden Befehl eine formale Definition aller möglichen Parameter und syntaktischen Regeln zu geben. Auf die wichtigen Bibliotheken NumPy und Matplotlib gehen wir dann anschließend in Kap. 3 ein.

Ich halte es für hilfreich, beim Programmieren darauf zu achten, dass die verwendeten Programmierkonzepte auch über die einfachsten Beispiele hinaus anwendbar sind. Einige Einführungen in das wissenschaftliche Rechnen mit Python starten mit dem Befehl `from pylab import *`. Dieser Befehl bewirkt, dass man viele mathematische, grafische und numerische Funktionen direkt aufrufen kann, sorgt aber auf der anderen Seite in größeren Programmen schnell für recht unübersichtlichen Code. Es lohnt sich meiner Meinung nach auf jeden Fall, von Anfang an etwas strukturierter zu arbeiten. Vielleicht müssen Sie dadurch an der einen oder anderen Stelle ein paar Zeichen mehr

O. Natt, *Physik mit Python*, https://doi.org/10.1007/978-3-662-66454-4_2

Code eintippen, aber dafür haben Sie die Sicherheit, dass Sie das Erlernte auch für etwas aufwendigere Programme sinnvoll einsetzen können.

Wenn Sie vielleicht schon etwas programmieren können, wird Ihnen an der folgenden Einführung auffallen, dass wir ein Thema komplett auslassen, das in nahezu allen Programmierbüchern eine ganz zentrale Rolle spielt: die Ein- und Ausgabe von Dateien sowie die Verarbeitung von Benutzereingaben. Der Grund dafür ist darin zu sehen, dass Sie mit diesem Buch nicht vorrangig lernen sollen, Anwendungsprogramme zu schreiben, die von anderen Personen benutzt werden können. Vielmehr werden Sie hauptsächlich Programme schreiben, die von Ihnen selbst benutzt werden, um physikalische Probleme zu lösen. Sie werden im Laufe der Zeit merken, dass es in den meisten Fällen völlig ausreichend ist, wenn Sie die notwendigen Eingabedaten direkt im Programm hinterlegen.

2.1 Installation einer Python-Distribution

Um in der Programmiersprache Python zu programmieren, benötigen Sie einen Computer, auf dem Python installiert ist. Geeignet ist nahezu jeder Rechner, auf dem ein aktuelles Betriebssystem läuft. Python ist für Microsoft-Windows, Apple macOS und Linux verfügbar. Auf der Webseite der Python-Foundation [1] finden sich viele wertvolle Hinweise rund um die Programmiersprache Python. Insbesondere findet man dort die vollständige Dokumentation der Standardbibliothek, die in der Programmiersprache Python enthalten ist. Als Anfänger sollte man jedoch davon Abstand nehmen, das Installationspaket direkt von dieser Webseite auf seinem Rechner zu installieren. Der Grund dafür ist, dass man für die Bearbeitung von physikalischen Fragestellungen eine Reihe von Bibliotheken[1] benötigt, die für das Erstellen von grafischen Ausgaben oder numerischen Berechnungen notwendig sind. Diese Bibliotheken alle von Hand zu installieren, ist leider recht aufwendig.

Glücklicherweise gibt es sogenannte **Python-Distributionen**. Es handelt sich dabei um Zusammenstellungen von Python mit den wichtigsten Bibliotheken, die man als ein Paket auf dem Rechner installieren kann. Besonders zu empfehlen ist die Distribution **Anaconda**, da diese sehr aktuell und umfangreich ist und sich trotzdem einfach installieren lässt. Auf der Webseite von Anaconda [2] werden Installationspakete für Windows, macOS und Linux zum Herunterladen bereitgestellt. Für Windows wird eine 64-Bit- und eine 32-Bit-Version angeboten. Wählen Sie hier die Variante aus, die zu dem von Ihnen verwendeten Betriebssystem passt.

Achtung!

Stellen Sie sicher, dass Sie die Programmiersprache Python in einer Version 3.8 oder höher installiert haben, damit Sie die Beispiele in diesem Buch unverändert übernehmen können.

Falls Sie anstelle der Python-Distribution Anaconda lieber das Installationspaket der Python-Foundation [1] verwenden möchten, so finden Sie auf der Webseite zu

[1] Eine Bibliothek ist eine Sammlung von Werkzeugen, die das Programmieren erleichtern oder bestimmte Funktionen zur Verfügung stellen.

Tab. 2.1: Pakete, die unter Linux installiert werden sollten. *Die folgenden Pakete sollten Sie über die Paketverwaltung Ihrer Linux-Distribution installieren, wenn Sie keine Python-Distribution verwenden möchten. Die genauen Namen der Pakete können unter Umständen bei einzelnen Linux-Distributionen abweichend sein.*

Paket	Erläuterung	Version
python	Grundpaket der Programmiersprache	≥ 3.8
ipython	Interaktive Python-Shell	
matplotlib	Bibliothek für grafische Darstellungen	≥ 3.1
numpy	Bibliothek für numerische Berechnungen	≥ 1.17
scipy	Bibliothek mit wissenschaftlichen Funktionen	≥ 1.3
spyder	Interaktive Entwicklungsumgebung	≥ 3.3

diesem Buch unter dem Punkt »Fragen« eine detaillierte Installationsanleitung für die benötigten Bibliotheken und Zusatzprogramme.

2.2 Installation von Python unter Linux

Wenn Sie Linux als Betriebssystem verwenden, können Sie neben der oben erwähnten Python-Distribution Anaconda auch das Python verwenden, das über die Paketverwaltung Ihrer Linux-Distribution installiert werden kann. Dieses Vorgehen hat den Vorteil, dass Python über den Update-Mechanismus des Betriebssystems aktualisiert wird. Wie man Pakete unter Linux installiert, hängt stark von der jeweiligen Linux-Distribution ab. Bitte informieren Sie sich dazu in der Anleitung Ihrer Linux-Distribution. Auf der Webseite zu diesem Buch finden Sie unter dem Punkt »Fragen« eine detaillierte Installationsanleitung für einige gängige Linux-Distributionen. Generell sollten Sie darauf achten, dass die in Tab. 2.1 aufgelisteten Pakete installiert sind und diese mindestens in der jeweils angegebenen Version vorliegen. Wenn diese Pakete in Ihrer Linux-Distribution nicht in diesen Versionen angeboten werden, sollten Sie auch unter Linux die Python-Distribution Anaconda [2] installieren.

2.3 Installation eines Texteditors

Neben der Programmiersprache Python benötigen Sie noch einen geeigneten Texteditor. Mit dem Texteditor erstellen Sie Dateien, die einzelne Befehle enthalten, die dann von Python abgearbeitet werden. Es gibt einige Texteditoren, die das Programmieren aktiv unterstützen, indem beispielsweise bestimmte Teile des Programmtextes farblich hervorgehoben werden. Der mit Microsoft-Windows mitgelieferte Texteditor ist aus diesem Grund nur sehr bedingt zum Erstellen von Computerprogrammen geeignet. Einige geeignete Editoren sind in Tab. 2.2 aufgelistet. Bei der Auswahl habe ich mich auf solche Editoren beschränkt, die auch für Anfänger gut zu bedienen sind.

Tab. 2.2: Texteditoren. *Einige Texteditoren, die sich für das Erstellen von Python-Programmen gut eignen.*

Name	Webadresse
Visual Studio Code	code.visualstudio.com
Notepad++	notepad-plus-plus.org
Geany	geany.org
Kate	kate-editor.org
gedit	projects.gnome.org/gedit

2.4 Installation einer Entwicklungsumgebung

Eine **integrierte Entwicklungsumgebung** (auch IDE für engl. integrated development environment) ist ein Programm, das Sie beim Programmieren weitaus stärker unterstützt als ein einfacher Texteditor das tun kann. Eine integrierte Entwicklungsumgebung ermöglicht es, den programmierten Code direkt auszuführen und unterstützt darüber hinaus beim Testen und bei der Fehlersuche. Es gibt für die Programmiersprache Python viele empfehlenswerte Entwicklungsumgebungen, die sich aber eher an Menschen richten, die häufig größere Programme schreiben. Falls Sie bereits Erfahrung im Programmieren haben und vielleicht auch schon in anderen Sprachen mit integrierten Entwicklungsumgebungen gearbeitet haben, dann kann ich Ihnen für Python die Entwicklungsumgebung PyCharm [3] empfehlen.

Zum Einstieg in das Programmieren mit Python empfehle ich die Programmierumgebung **Spyder** [4]. Insbesondere für die Arbeit im naturwissenschaftlichen Umfeld ist es häufig notwendig, Programme interaktiv zu erstellen, und Spyder unterstützt genau diese Art des interaktiven Programmierens. Das Programm ist in der Python-Distribution Anaconda bereits enthalten.

Neben Spyder ist auch der Editor **Visual Studio Code** sehr empfehlenswert, da er viele Funktionen bietet, die man sonst nur in den großen integrierten Entwicklungsumgebungen findet. Auch dieser Editor unterstützt das interaktive Programmieren, erfordert aber etwas mehr Einarbeitungszeit als Spyder. Auf der Webseite zu Visual Studio Code [5] gibt es eine ausführliche Anleitung, wie man diesen Editor für die Arbeit mit Python einrichtet.

2.5 Starten einer interaktiven Python-Sitzung

Es gibt prinzipiell zwei Möglichkeiten, mit Python zu arbeiten. Die erste Möglichkeit besteht darin, eine Textdatei zu erstellen, in der die einzelnen Befehle niedergeschrieben sind. Diese Textdatei bezeichnet man dann als ein **Python-Programm** oder auch als **Python-Skript**. Dieses Python-Programm wird dann von Python ausgeführt. Die zweite Möglichkeit besteht darin, eine interaktive Sitzung zu starten, in der man die Python-Befehle eintippt. Jeder Befehl wird dann unmittelbar von Python ausgeführt. Die Software, in der man die Befehle eingeben kann, bezeichnet man als eine Shell. Wie Sie eine **Python-Shell** starten, hängt vom Betriebssystem und der verwendeten Python-Distribution ab. Falls Sie beispielsweise Windows mit der Distribution Anaconda verwenden, dann können Sie im Startmenü den Punkt **Anaconda-Prompt** auswählen und dort den Befehl `python` eingeben. Wenn Sie Linux verwenden, dann öffnen Sie

bitte ein Terminalfenster und geben dort den Befehl `python` oder `python3` ein.[2] Sie sollten ein Fenster erhalten, das ungefähr den folgenden Inhalt hat:

```
(base) C:\Users\natt>python
Python 3.9.13 (main, Aug 25 2022, 23:51:50)
[MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

Bitte kontrollieren Sie noch einmal, dass Sie mindestens die Version 3.8 der Programmiersprache Python installiert haben. Im oben angegebenen Beispiel wird die Python-Version 3.9.13 verwendet.

Die drei größer-Zeichen `>>>` sind der sogenannte **Python-Prompt**, der Sie zur Eingabe von Befehlen auffordert. Sie können nun Python-Befehle eingeben, und diese werden direkt ausgeführt, nachdem Sie die Eingabetaste gedrückt haben. Um Python zu beenden, geben Sie bitte den Befehl

```
>>> exit()
```

ein. Bei der Eingabe des Befehls müssen Sie streng auf die Groß- und Kleinschreibung achten. Beachten Sie bitte auch, dass zwischen dem Wort `exit` und den Klammern `()` kein Leerzeichen steht. Weiterhin dürfen Sie keine Leerzeichen am Zeilenanfang einfügen, da diese in Python eine spezielle Bedeutung haben, auf die wir in Abschn. 2.16 eingehen werden. Falls Sie sich einmal bei einem Befehl vertippt haben sollten, ist das nicht weiter schlimm. Sie erhalten dann vielleicht eine Fehlermeldung, die mehr oder weniger gut verständlich ist. Wenn Sie einen vorher eingegebenen Befehl noch einmal bearbeiten wollen, dann können Sie mit der Pfeiltaste nach oben zu diesem Befehl zurückblättern.

Alternativ kann man statt des Befehls `python` auch den Befehl `ipython` verwenden. Das »i« in `ipython` steht dabei für »interaktiv«. Der Prompt sieht dort etwas anders aus, und es gibt eine Reihe von Funktionen, die die Eingabe etwas erleichtern. Insbesondere erhält man häufig durch Drücken der TAB-Taste sinnvolle Vorschläge zur Vervollständigung der Python-Befehle. Für die Beispiele in diesem Buch ist es irrelevant, ob Sie die Python-Shell oder die IPython-Shell benutzen. Im Buch wird der Übersichtlichkeit halber immer nur der Python-Prompt dargestellt. Eine ausführliche Dokumentation der Funktionen der IPython-Shell finden Sie online [6].

2.6 Python als Taschenrechner

Die einfachste Anwendung einer interaktiven Python-Sitzung besteht darin, Python als einen erweiterten Taschenrechner zu benutzen. Geben Sie einfache Rechenaufgaben, die die Grundrechenarten Addition `+`, Subtraktion `-`, Multiplikation `*` und Division `/` benutzen, direkt hinter dem Prompt an und führen Sie die Berechnung durch Drücken der Eingabetaste aus.

[2] Bei einigen Linux-Distributionen ruft der Befehl `python` die veraltete Version Python 2.7 auf. Probieren Sie in diesem Fall stattdessen den Befehl `python3`.

Achtung!

Um einen Befehl auszuführen, müssen Sie diesen mit der Eingabetaste bestätigen.

Es ist für die bessere Lesbarkeit empfehlenswert, jeweils ein Leerzeichen zwischen den Zahlen und dem Rechenzeichen zu lassen. Die Rechenzeichen nennt man auch **Operatoren**, während man die Zahlen, die vom Rechenzeichen verarbeitet werden, als **Operanden** bezeichnet. Nachfolgend sind einige Beispiele angegeben:

```
>>> 17 + 4
21
>>> 23 - 5
18
>>> 8 / 5
1.6
>>> 175 * 364
63700
>>> 32 / 1000000
3.2e-05
```

Sie erkennen an diesen Beispielen schon einige Besonderheiten: Dezimalbrüche werden mit einem Dezimalpunkt anstelle des im Deutschen üblichen Kommas dargestellt. Bei sehr großen oder sehr kleinen Zahlen wird die wissenschaftliche Notation benutzt. Die Angabe `3.2e-05` ist als $3{,}2 \cdot 10^{-5}$ zu lesen. Dementsprechend sollte man sehr große oder sehr kleine Zahlen auch in dieser Schreibweise angeben.

Sie können selbstverständlich auch komplizierte Rechnungen ausführen. Dabei müssen Sie beachten, dass Python die übliche Operatorrangfolge der Mathematik »Punktrechnung vor Strichrechnung« übernimmt. Selbstverständlich können Sie Klammern einsetzen, um eine andere Reihenfolge der Auswertung zu erzwingen. Wenn Sie sich bei der Operatorrangfolge einmal unsicher sind, setzen Sie lieber ein Klammerpaar zu viel.

```
>>> 3 + 4 * 2
11
>>> 3 + (4 * 2)
11
>>> (3 + 4) * 2
14
```

Zum Klammern von Ausdrücken dürfen Sie allerdings nur die üblichen runden Klammern `( )` verwenden. Die eckigen Klammern `[ ]` und die geschweiften Klammern `{ }` haben in Python eine andere Bedeutung, wie man an dem folgenden Beispiel sieht.

```
>>> [(3 + 4) * 2 + 7] * 3
[21, 21, 21]
```

Das erwartete Ergebnis erhält man, wenn man die eckigen Klammern durch runde Klammern ersetzt.

```
>>> ((3 + 4) * 2 + 7) * 3
63
```

Wir werden bei der Behandlung von Listen in Abschn. 2.9 verstehen, warum der Ausdruck `[(3 + 4) * 2 + 7] * 3` ein so sonderbares Ergebnis produziert.

Vielleicht haben Sie schon mehr Rechnungen ausprobiert als die in den oben angegebenen Beispielen, und vielleicht sind Sie darüber gestolpert, dass Python anscheinend nicht immer genau rechnet. Bei der einfachen Rechnung $4{,}01 - 4 = 0{,}01$ stellt man fest, dass Python ein unerwartetes Ergebnis liefert:

```
>>> 4.01 - 4
0.009999999999999787
```

Der Grund hierfür ist darin zu sehen, dass die Nachkommastellen einer Gleitkommazahl im Computer im Binärsystem dargestellt werden und es bei der Umwandlung von Dezimalbrüchen in die Binärdarstellung zu unvermeidlichen Rundungsfehlern kommt. Bitte beachten Sie aber, dass der Fehler tatsächlich klein ist.

```
>>> (4.01 - 4) - 0.01
-2.1337098754514727e-16
```

Die Differenz des berechneten Ergebnisses vom erwarteten Ergebnis beträgt gerade einmal $-2 \cdot 10^{-16}$. Die relative Abweichung[3] liegt also in der Größenordnung von $-2 \cdot 10^{-14}$ und ist somit außerordentlich klein. Dennoch muss man sich beim Rechnen mit Dezimalzahlen, die im Computer als sogenannte Gleitkommazahlen dargestellt werden, stets über solche Rundungsfehler bewusst sein.

Um Potenzen zu bilden, wird in Python der Operator `**` verwendet. Dabei kann der Exponent eine beliebige Gleitkommazahl sein. Man kann mit dem Operator `**` insbesondere auch negative Potenzen und gebrochene Potenzen berechnen.

```
>>> 2 ** 8
256
>>> 4 ** -5
0.0009765625
>>> 4 ** -1
0.25
>>> 16 ** 0.25
2.0
```

Wenn man Potenzen von negativen Zahlen bildet, muss man geeignet klammern, wie das folgende Beispiel zeigt:

```
>>> -2 ** 4
-16
```

Das Ergebnis ist -16, denn der Ausdruck wird als $-(2^4)$ interpretiert und nicht als $(-2)^4$. Um Letzteres zu berechnen, muss man die -2 in Klammern setzen:

```
>>> (-2) ** 4
16
```

[3] Wenn man eine Abweichung Δx von einer Größe x betrachtet, dann ist die relative Abweichung das Verhältnis $\Delta x/x$. In dem dargestellten Beispiel ist die betrachtete Größe die tatsächliche Differenz von $x = 0{,}01$, und die Abweichung beträgt $\Delta x = 2 \cdot 10^{-16}$.

Man sagt auch, dass der Operator `**` stärker bindet als das Vorzeichen `-`. Der Potenz-Operator bindet auch stärker als die Division. Beachten Sie dies bitte, wenn Sie gebrochene Potenzen berechnen:

```
>>> 9 ** 1 / 2
4.5
>>> 9 ** (1 / 2)
3.0
```

Achtung!

Potenzen werden in Python mit dem Operator `**` gebildet. Der Operator `^`, der in vielen anderen Programmiersprachen eine Potenz anzeigt, führt in Python eine bitweise XOR-Verknüpfung aus.

Gelegentlich kommt es vor, dass man eine **Division mit Rest** durchführen möchte. Dazu gibt es in Python den Operator `//`, der das Ergebnis der ganzzahligen Division liefert, und den Operator `%`, der den Rest der ganzzahligen Division zurückgibt.

```
>>> 17 // 3
5
>>> 17 % 3
2
```

Neben den Grundrechenarten und der Potenzbildung benötigt man zur Bearbeitung physikalischer Probleme oft weitere mathematische Funktionen wie die Wurzelfunktion, die Exponentialfunktion, Logarithmen und die trigonometrischen Funktionen. Diese sind in Python in einem **Modul** enthalten, das zunächst importiert werden muss. Man **importiert** ein Modul mit einer `import`-Anweisung. Wie man Funktionen aus dem Modul `math` benutzt, wird am einfachsten an Beispielen deutlich.

```
>>> import math
>>> math.sqrt(64)
8.0
>>> math.exp(4)
54.598150033144236
>>> math.sin(math.pi / 4)
0.7071067811865475
>>> math.atan(1)
0.7853981633974483
```

Sie erkennen, dass man den Modulnamen und den Funktionsnamen mit einem Punkt trennt. Das Argument der Funktion wird, wie in der Mathematik üblich, in runde Klammern gesetzt. Die Funktion `math.sqrt` berechnet die Quadratwurzel. Die Funktion `math.exp` ist die Exponentialfunktion. Vielleicht ist Ihnen an den Beispielen schon aufgefallen, dass Python bei trigonometrischen Funktionen immer im Bogenmaß rechnet. Es gibt zwei hilfreiche Funktionen, mit denen man zwischen dem Bogen- und dem Gradmaß umrechnen kann:

```
>>> import math
>>> math.radians(45)
0.7853981633974483
>>> math.degrees(math.atan(1))
45.0
```

Eine vollständige Liste aller Funktionen, die im Modul `math` enthalten sind, finden Sie in der Dokumentation der Python-Standardbibliothek [7]. Alternativ können Sie auch direkt in der Python-Shell eine Hilfe erhalten, indem Sie nach dem Importieren des Moduls `math` den Befehl `help(math)` ausführen.[4] Analog können Sie mit dem Befehl `help(math.sin)` auch die Hilfe einer einzelnen Funktion aufrufen.[5]

2.7 Importieren von Modulen

Wir haben gesehen, wie man mit der Anweisung `import math` das Modul `math` importiert. Anschließend kann man Funktionen oder Konstanten aus dem Modul aufrufen, indem man den Modulnamen und den Funktionsnamen mit einem Punkt trennt. Die Programmiersprache Python bietet noch weitere Möglichkeiten, Funktionen aus einem Modul zu importieren, die man am einfachsten an Beispielen verstehen kann.

```
>>> from math import sin, cos, pi
>>> sin(pi / 2)
1.0
>>> cos(pi / 2)
6.123233995736766e-17
>>> tan(pi / 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'tan' is not defined
```

Der Befehl `from math import sin, cos, pi` importiert die Funktionen `math.sin` und `math.cos` sowie die Konstante `math.pi` so, dass man sie ohne das vorangestellte `math.` verwenden kann. Sie können auch erkennen, dass die Funktion `math.tan` nicht importiert wurde. Der Python-Interpreter gibt beim Versuch, die Funktion `tan` aufzurufen, die Fehlermeldung aus, dass `tan` nicht definiert ist. Sie können auch gleich alle Definitionen aus einem Modul importieren:

```
>>> from math import *
>>> tan(pi / 4)
0.9999999999999999
```

[4] Wenn der Hilfetext nicht komplett in das Fenster passt, können Sie durch Drücken der Leertaste eine Seite weiter blättern. Durch Drücken der Taste »q« verlassen Sie die Hilfefunktion.

[5] Wenn Sie die Hilfefunktion innerhalb von Python benutzen, wird Ihnen auffallen, dass dort in den Argumenten der Funktionen häufig ein Schrägstrich (`/`) auftaucht, wie zum Beispiel in `sin(x, /)`. Dieser Schrägstrich bedeutet, dass die Argumente vor dem Schrägstrich reine Positionsargumente sind. Wir werden auf die unterschiedlichen Arten von Funktionsargumenten in Abschn. 2.17 näher eingehen.

Man bezeichnet diese Art, ein Modul zu importieren, als einen **Stern-Import**. Der Stern-Import wirkt auf den ersten Blick besonders bequem. Er bringt allerdings einige Nachteile mit sich.

- Wir werden später sehen, dass es noch andere Module gibt, die ebenfalls eine Funktion mit dem Namen `sin` enthalten. So besitzt beispielsweise das Modul `numpy` eine solche Funktion, die dazu dient, den Sinus von vielen Zahlen gleichzeitig zu berechnen. Es ist offensichtlich, dass es zu Verwechslungen kommen kann, wenn man beide Module verwenden möchte.

- Etwas längere Programme sind bei der Verwendung des Stern-Imports manchmal schwer zu verstehen. Das ist nicht nur ein Problem, wenn irgendwann eine andere Person das Programm weiter bearbeiten möchte, sondern oft auch, wenn Sie selbst nach längerer Zeit eines Ihrer Programme wieder anschauen.[6] Die Schwierigkeit bei der Verwendung von Stern-Importen besteht darin, dass oft gar nicht klar ist, aus welchem Modul eine Funktion überhaupt kommt.

- Man verliert leicht den Überblick, was eigentlich alles importiert wird. Stellen Sie sich vor, dass Sie in einer analytischen Rechnung eine Zeitdauer mit dem griechischen Buchstaben τ bezeichnen. Anschließend setzen Sie Ihre Rechnung in ein Python-Programm um. Die zugehörige Variable heißt dann wahrscheinlich `tau`. Wenn Sie nun das Modul `math` über `from math import *` importieren, sind Fehler vorprogrammiert, denn im Modul `math` ist `tau` bereits als 2π definiert.

2.8 Variablen

Wenn Sie mathematische oder physikalische Aufgaben händisch bearbeiten, ist es Ihnen bereits vertraut, mit Variablen zu rechnen. Die Namen von Variablen, die sogenannten **Bezeichner**, dürfen in Python aus beliebigen Klein- oder Großbuchstaben, dem Unterstrich `_` und den Ziffern 0–9 bestehen. Dabei darf der Name nicht mit einer Ziffer anfangen. Im Prinzip können Sie alle Buchstaben verwenden, die der Unicode-Zeichensatz[7] zur Verfügung stellt. Ich empfehle Ihnen allerdings, sich auf die lateinischen Buchstaben a–z und A–Z sowie die Ziffern und den Unterstrich zu beschränken. Bitte beachten Sie, dass in Python stets zwischen Groß- und Kleinschreibung unterschieden wird. Es gibt einige **Schlüsselwörter** in Python, die als Bezeichner von Variablen und Funktionen verboten sind, weil sie eine spezielle Bedeutung haben (siehe Kasten).

Jedes Ding, das irgendwie im Speicher des Computers gespeichert werden kann, bezeichnen wir als ein **Objekt**. Wir werden in Abschn. 2.9 sehen, dass es sich dabei nicht notwendigerweise um Zahlen, sondern auch um Zeichenketten (Strings), Listen oder andere Dinge handeln kann. Eine Variable ist ein Name für ein bestimmtes Objekt.

[6] Der Begriff »längere Zeit« ist aus meiner Erfahrung ein Euphemismus. Mir persönlich passiert es durchaus, dass ich ein eigenes Programm schon nach wenigen Stunden nicht mehr verstehe, wenn ich nicht sauber programmiert habe.

[7] Ein Computer verarbeitet ausschließlich Zahlen. Um einen Text im Computer darzustellen, benötigt man eine Konvention, die aussagt, welcher Buchstabe durch welche Zahl repräsentiert wird. Der Unicode-Zeichensatz ist eine solche Zuordnungstabelle, die neben dem englischen Zeichensatz auch viele sprachspezifische Zeichen enthält.

Schlüsselwörter in Python

Die folgenden Wörter sind in Python reserviert und dürfen nicht als Bezeichner verwendet werden.

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

Man sagt auch, dass die Variable eine **Referenz**, also ein Bezug, auf ein bestimmtes Objekt sei. In Python dient das Gleichheitszeichen `=` als Zuweisungsoperator. Eine Zuweisung der Form `a = 9.81` wird meistens als »Weise der Variablen `a` den Wert 9,81 zu« gelesen, und wir verwenden diese Sprechweise ebenfalls. Allerdings werden wir im Folgenden sehen, dass man sich eine solche Zuweisung eher in der Form »Gib der Zahl 9,81 den Namen `a`« einprägen sollte, da dies die Funktionsweise der Sprache Python eigentlich besser wiedergibt.

Die Verwendung des Zuweisungsoperators wird an dem folgenden Beispiel gezeigt:

```
>>> a = 9.81
>>> t = 0.5
>>> s = 1/2 * a * t**2
>>> print(s)
1.22625
```

Die erste Zeile weist der Variablen `a` den Wert 9,81 zu. Die zweite Zeile weist der Variablen `t` den Wert 0,5 zu. In der dritten Zeile wird der Variablen `s` das Ergebnis von $\frac{1}{2}at^2$ zugewiesen. In der vierten Zeile benutzen wir die in Python eingebaute Funktion `print`, um die Variable `s` auszugeben. Sie können mit Python-Variablen also (fast) genauso rechnen, wie man das auf Papier auch tun würde. Für das bessere Verständnis sollten Sie eine Zuweisung der Art

```
>>> a = 9.81
```

wie folgt verstehen: Die rechte Seite vom Gleichheitszeichen erzeugt ein Objekt. In diesem Fall ist das eine Gleitkommazahl, die den Zahlenwert 9,81 hat. Dieses Objekt muss irgendwo im Hauptspeicher des Computers abgelegt werden. Stellen Sie sich den Platz im Hauptspeicher des Computers als eine Kiste vor (siehe Abb. 2.1). Die Zuweisung `a = 9.81` bewirkt nun, dass man dieser Kiste den Namen »a« gibt. Das kann man sich anschaulich so vorstellen, dass ein Schild mit der Aufschrift »a« an der Kiste angebracht wird. Wenn man anschließend der Variablen `a` durch

```
>>> a = 1.62
```

einen neuen Wert zuweist, sollten Sie sich das wie folgt vorstellen: Es wird eine neue Kiste hergenommen. In diese wird die Zahl 1,62 gelegt. Anschließend wird das Schild mit der Aufschrift »a« von der Kiste mit der Zahl 9,81 abgenommen und an der neuen Kiste angebracht (siehe zweite Zeile von Abb. 2.1). Wenn Sie sich nun fragen,

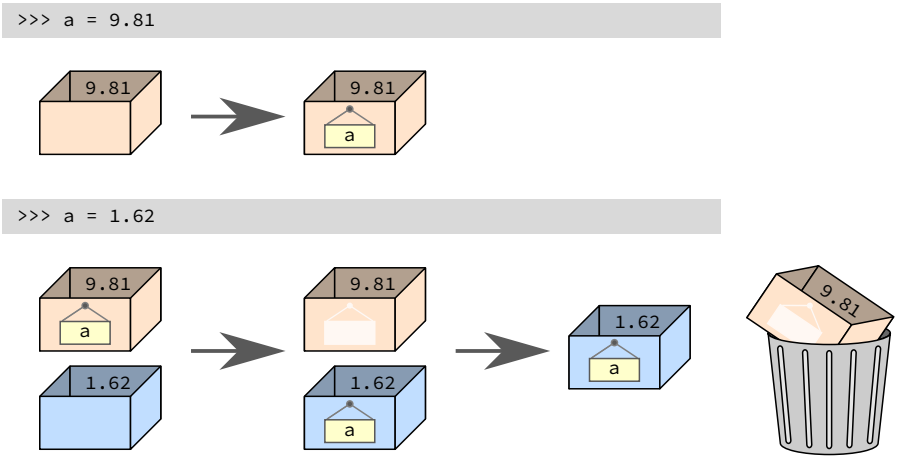

Abb. 2.1: Variablenzuweisung in Python. *Die Zuweisung* `a = 9.81` *bewirkt, dass ein Objekt mit dem Zahlenwert* 9,81 *erzeugt wird. Dieses Objekt erhält den Namen* `a`*. In der Illustration ist das Objekt eine Kiste und der Name ein daran angebrachtes Schild. Wenn anschließend die Zuweisung* `a = 1.62` *ausgeführt wird, so wird ein neues Objekt mit dem Zahlenwert* 1,62 *erstellt. Dieses Objekt hat nun den Namen* `a`*. In der Illustration wird das Schild von der Kiste mit dem alten Objekt entfernt und an der Kiste mit dem neuen Objekt angebracht. Es verbleibt ein Objekt, das überhaupt nicht mehr benannt ist. Dieses Objekt wird von der Müllabfuhr entsorgt.*

was mit der alten Kiste und deren Inhalt passiert, dann ist das eine sehr berechtigte Frage. Python nimmt Ihnen hier sehr viel Arbeit ab. Es gibt eine Art Müllabfuhr (engl. **garbage collection**), die nach und nach alle Kisten, die nicht mehr benötigt werden, weil beispielsweise kein Schild mit einem Namen mehr daran angebracht ist, entsorgt. Das geschieht ganz automatisch im Hintergrund, ohne dass Sie sich darum kümmern müssen.

2.9 Datentypen und Klassen

In Python hat jedes Objekt einen Datentyp. Man sagt auch, dass es sich um ein Objekt einer bestimmten **Klasse** handelt. Den Typ eines Objekts kann man in Python mit der Funktion `type` bestimmen:

```
>>> a = 2
>>> b = 2.0
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
```

Sie können an diesem Beispiel sehen, dass Zahlen vom Typ »ganze Zahl« `int` (von engl. integer) oder vom Typ Gleitkommazahl `float` (von engl. floating point number) sein können. Weiterhin erkennt man an dem Beispiel, dass sich die Zahlen `2` und `2.0`

unterscheiden. Das Erste ist eine ganze Zahl mit dem Zahlenwert 2 und das Zweite eine Gleitkommazahl mit dem gleichen Zahlenwert.

In Python haben Variablen im Gegensatz zu anderen Programmiersprachen wie Java oder C keinen vordefinierten Datentyp. In Java oder C bezeichnet die Variable einen bestimmten Speicherplatz im Hauptspeicher des Computers. In Python ist die Variable dagegen ein Name, der auf einen bestimmten Speicherplatz verweist.

Bei den Datentypen muss man zwischen **veränderlichen** (engl. mutable) und **unveränderlichen** (engl. immutable) Typen unterscheiden. Intuitiv gehen Sie vermutlich davon aus, dass eine Zahl in Python eine veränderliche Größe ist, denn eine Anweisung der Art

```
>>> x = 5
>>> x = x + 1
>>> print(x)
6
```

verändert augenscheinlich den Wert der Variablen `x`. Um verstehen zu können, was mit einer unveränderlichen Größe gemeint ist, schauen wir uns noch einmal in kleinen Schritten an, was die drei Anweisungen bewirken.

Die erste Zeile sagt aus: »Gib der Zahl 5 den Namen `x`«. Die zweite Zeile sagt aus: »Addiere zu der Zahl mit dem Namen `x` die Zahl 1 und nenne das Ergebnis von nun an `x`«. Wichtig dabei ist, dass der Name `x` nach Zeile zwei auf eine andere Stelle im Speicher des Computers verweist als vor der Zeile zwei. Man hat also nicht den Inhalt einer Speicherstelle verändert, sondern der Name verweist auf eine andere Speicherstelle.

Es erscheint Ihnen vielleicht sehr umständlich oder als Wortklauberei, dass wir Zahlen in Python als unveränderliche Größen bezeichnen. Diese Eigenschaft ist aber überaus nützlich, da sie eine Reihe von Programmierfehlern vermeidet. Wir werden darauf noch einmal zurückkommen, wenn wir den Datentyp der Liste besprechen, der ein veränderlicher Datentyp ist.

`int` Ganze Zahlen

Der Datentyp `int` stellt ganze Zahlen dar. Es gibt keine vordefinierte maximale Größe der Zahlen, wie es in anderen Programmiersprachen üblich ist. Die größte darstellbare Zahl ist nur durch den Hauptspeicher Ihres Computers begrenzt.

`float` Gleitkommazahlen

Der Datentyp `float` stellt sogenannte Gleitkommazahlen dar. Sie können Gleitkommazahlen als eine Art Darstellung der reellen Zahlen der Mathematik auffassen. Dabei muss allerdings klar sein, dass der Computer nur mit einer bestimmten endlichen Genauigkeit rechnet. Der genaue Wertebereich der Gleitkommazahlen hängt von der Rechnerarchitektur ab. Sie können diesen Wertebereich wie folgt herausfinden:

```
>>> import sys
>>> sys.float_info
sys.floatinfo(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
```

```
min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.2204460492503131e-16, radix=2, rounds=1)
```

Auf dem in diesem Beispiel verwendeten Rechner ist die größte darstellbare Gleitkommazahl $1{,}79 \cdot 10^{308}$. Die kleinste positive darstellbare Gleitkommazahl ist $2{,}22 \cdot 10^{-308}$. Die Angabe `epsilon`, die hier den Zahlenwert $2{,}22 \cdot 10^{-16}$ hat, ist der kleinste relative Unterschied zweier Gleitkommazahlen. Die Bedeutung dieser Größe erkennt man an den folgenden Beispielen:

```
>>> (1 + 1e-16) - 1
0.0
>>> (1 + 2e-16) - 1
2.220446049250313e-16
```

Im ersten Beispiel weicht die Zahl $1 + 1 \cdot 10^{-16}$ so wenig von 1 ab, dass die Differenz zur Zahl 1 bei der Rechnung mit Gleitkommazahlen exakt 0 ergibt. Im zweiten Beispiel ist die Differenz nicht wie vielleicht erwartet $2 \cdot 10^{-16}$, sondern $2{,}22 \cdot 10^{-16}$. Dies spiegelt genau die Bedeutung der Größe `epsilon` wider: Es gibt im Rahmen der Rechengenauigkeit des Computers keine Gleitkommazahl, die zwischen `1` und `1 + epsilon` liegt.

`complex` Komplexe Gleitkommazahlen

Python kann auch mit komplexen Zahlen rechnen. Wenn Sie mit komplexen Zahlen noch nicht vertraut sind, können Sie diesen Abschnitt zunächst überspringen.

Eine komplexe Zahl wird durch ihren Real- und Imaginärteil dargestellt. In der Mathematik schreibt man komplexe Zahlen meistens in der Form $a + \mathrm{i}b$, wobei man a als den Realteil und b als den Imaginärteil der Zahl bezeichnet. Die Größe i nennt man die **imaginäre Einheit** mit $\mathrm{i}^2 = -1$. In Python wird die imaginäre Einheit mit dem Buchstaben `j` dargestellt. Sie können mit komplexen Zahlen genauso rechnen wie mit Gleitkommazahlen. Dies wird an den folgenden Beispielen verdeutlicht:

```
>>> z1 = 3 + 4j
>>> type(z1)
<class 'complex'>
>>> z2 = 3.5 * z1
>>> print(z2)
(10.5+14j)
>>> z3 = z1 * z2
>>> print(z3)
(-24.5+84j)
>>> z3.real
-24.5
>>> z3.imag
84.0
```

Die Beispiele zeigen einen weiteren Aspekt der Programmiersprache Python: Eine komplexe Zahl hat zwei Eigenschaften, die man auch als **Attribute** bezeichnet: den Realteil `real` und den Imaginärteil `imag`. Man kann auf die Attribute zugreifen, indem man den Namen des Attributs mit einem Punkt vom Namen des Objekts trennt. Den

Betrag einer komplexen Zahl kann man in Python mit der Funktion `abs` berechnen. In dem Modul `cmath` ist eine Reihe mathematischer Funktionen für komplexe Zahlen enthalten.

`bool` Wahrheitswerte

In Python gibt es einen speziellen Datentyp für Wahrheitswerte, die man auch als **boolesche Werte** bezeichnet. Dieser Datentyp kann nur die Werte `True` für wahr und `False` für falsch annehmen. Wir werden die Verwendung dieses Datentyps bei den bedingten Anweisungen ausführlich diskutieren.

`str` Zeichenketten (Strings)

Strings sind unveränderliche Aneinanderreihungen von Zeichen, die im Unicode-Standard dargestellt werden. Strings können also Buchstaben aus verschiedenen Alphabeten, Sonderzeichen, Ziffern, Leerzeichen und so weiter enthalten. Einen String stellt man in Python dar, indem man die Zeichenketten in einfache Anführungszeichen einschließt.[8] Alternativ kann man die Zeichenkette auch in doppelte Anführungszeichen einschließen. Das ist hilfreich, wenn Sie innerhalb des Strings einfache Anführungszeichen verwenden wollen:

```
>>> x1 = 'Das ist eine Zeichenkette.'
>>> print(x1)
Das ist eine Zeichenkette.
>>> x2 = "Das ist auch eine Zeichenkette."
>>> print(x2)
Das ist auch eine Zeichenkette.
>>> x3 = "Ich sage: 'Das ist eine Zeichenkette.'"
>>> print(x3)
Ich sage: 'Das ist eine Zeichenkette.'
>>> x4 = 'Du sagtest: "Das ist eine Zeichenkette."'
>>> print(x4)
Du sagtest: "Das ist eine Zeichenkette."
```

Strings lassen sich addieren und mit einer ganzen Zahl multiplizieren. Die Bedeutung der Operationen können Sie sich an dem folgenden Beispiel klarmachen:

```
>>> a = 'Oh '
>>> b = 'la '
>>> c = '!'
>>> a + 2 * b + c
Oh la la !
```

Addition von Strings bewirkt also ein Hintereinanderhängen (Verketten) von Strings. Dabei werden nicht automatisch Leerzeichen oder andere Trennzeichen eingefügt. Die Leerzeichen im oben angegebenen Beispiel sind bereits in den Strings `a` und `b` enthalten.

[8] Das einfache Anführungszeichen ist das Zeichen, das auf der deutschen Computertastatur auf der gleichen Taste liegt wie die Raute #. Bitte verwenden Sie nicht die Akzente, die auf der deutschen Tastatur rechts neben dem Fragezeichen zu finden sind.

Die Multiplikation eines Strings mit einer natürlichen Zahl erzeugt eine entsprechende Wiederholung des Strings. Bitte beachten Sie auch hier, dass bei einer Zuweisung der folgenden Art

```
>>> a = 'Hallo'
>>> a = a + '!'
>>> print(a)
Hallo!
```

die ursprüngliche Zeichenkette `'Hallo'` nicht verändert wird. Der Befehl `a = a + '!'` erzeugt vielmehr eine neue Zeichenkette mit dem entsprechenden Inhalt.

Man kann auf einzelne Zeichen eines Strings zugreifen, indem man die Position des gewünschten Zeichens in der Zeichenkette als Index in eckigen Klammern anhängt. Dabei ist darauf zu achten, dass das erste Zeichen eines Strings den Index null hat.

```
>>> s = 'Dies ist ein kleiner Text'
>>> s[0]
'D'
>>> s[1]
'i'
>>> s[23]
'x'
```

Python gibt eine Fehlermeldung aus, wenn man versucht, auf ein Zeichen zuzugreifen, das nicht existiert:

```
>>> s = 'Dies ist ein kleiner Text'
>>> s[25]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Eine sehr praktische Eigenschaft von Python ist, dass man mit negativen Indizes arbeiten kann, um die Zeichen eines Strings von hinten abzuzählen:

```
>>> s = 'Dies ist ein kleiner Text'
>>> s[-1]
't'
>>> s[-2]
'x'
```

Die Länge eines Strings kann man mit der Python-Funktion `len` bestimmen.

```
>>> s = 'Dies ist ein kleiner Text'
>>> len(s)
25
```

Darüber hinaus gibt es eine ganze Reihe **Methoden** in Python, mit denen man Strings bearbeiten kann. Eine Methode ist gewissermaßen eine Funktion, die mit einem Objekt verbunden ist. Eine Methode ruft man auf, indem man den Methodennamen und den Objektnamen mit einem Punkt verbindet. Die Methoden der Klasse `str` werden in der Dokumentation der Standardbibliothek von Python [7] erklärt. Alternativ können Sie

Bedeutung des Punktes . in Python

Der Punkt hat in Python zwei Bedeutungen:

1. als Dezimalpunkt bei der Angabe von Zahlen, wie in `3.1415`,
2. als Trennzeichen zwischen einem Objekt und einer Eigenschaft des Objektes wie in `z.real`, bzw. einer Methode wie bei `s.replace()`.

Darüber hinaus haben wir den Punkt noch als Trennzeichen zwischen einem Modul und einer im Modul enthaltenen Funktion wie bei `math.sin()` oder einer im Modul enthaltenen Konstante wie bei `math.pi` kennengelernt. Das ist aber kein Widerspruch, denn in Python ist auch ein importiertes Modul ein Objekt, nämlich ein Objekt, das den Datentyp `module` hat.

in Python auch `help(str)` aufrufen. Als Beispiel probieren wir einmal die Methode `replace`, die eine Ersetzung von Textteilen vornimmt.

```
>>> s = 'Dies ist ein kleiner Text'
>>> s1 = s.replace('kleiner', 'kurzer')
>>> print(s)
Dies ist ein kleiner Text
>>> print(s1)
Dies ist ein kurzer Text
```

Es würde den Rahmen dieses Buches sprengen, wenn wir alle Methoden, die in Python definiert sind, ausführlich besprechen. Bitte schauen Sie sich aber einmal die Dokumentation der Klasse `str` an, damit Sie wissen, welche Methoden es überhaupt gibt.

Neben den Rechenoperatoren `+` und `*` gibt es für Strings in Python noch den Operator `in`. Dieser Operator überprüft, ob ein bestimmter String in einem anderen String enthalten ist. Auch das kann man wieder am besten an einem Beispiel erkennen:

```
>>> s = 'Dies ist ein kleiner Text'
>>> 'ie' in s
True
>>> 'ren' in s
False
```

Die Zeichenkette »ie« ist im String `s` also enthalten, die Zeichenkette »ren« dagegen nicht. Der Operator `in` gibt ein Objekt vom Typ `bool` zurück, wie das folgende Beispiel zeigt:

```
>>> s = 'Dies ist ein kleiner Text'
>>> b = 'er' in s
>>> type(b)
<class 'bool'>
```

`tuple` Ketten von Objekten (Tupel)

Ein Tupel ist in Python eine Aneinanderreihung oder Kette von beliebigen Objekten. Ein Tupel wird gebildet, indem man mehrere Objekte mit Kommata getrennt aufzählt. In vielen Fällen wird die Lesbarkeit des Programmcodes besser, wenn man bei der Definition eines Tupels runde Klammern um die Aufzählung der Objekte schreibt, auch wenn dies nicht notwendig ist. Die einzelnen Elemente eines Tupels können von völlig unterschiedlichen Datentypen sein:

```
>>> t = (1, 2.0, 4+3j, 'Klaus')
>>> type(t)
<class 'tuple'>
```

Tupel haben große Ähnlichkeit mit Strings. Strings sind Aneinanderreihungen von Zeichen, während Tupel Aneinanderreihungen von beliebigen Objekten sind. Dementsprechend funktioniert die Indizierung von Tupeln genauso wie die von Strings. Ein bestimmtes Element eines Tupels erhält man, indem man an den Variablennamen den Index des Elements in eckigen Klammern anhängt. Auch hier ist zu beachten, dass man in Python bei null anfängt zu zählen.

```
>>> t = (1, 2.0, 4+3j, 'Klaus')
>>> t[0]
1
>>> t[1]
2.0
```

Genau wie bei Strings kann man auch bei Tupeln mit negativen Indizes arbeiten, um das letzte, vorletzte etc. Element anzusprechen:

```
>>> t = (1, 2.0, 4+3j, 'Klaus')
>>> t[-1]
Klaus
>>> t[-2]
(4+3j)
```

Genau wie Strings kann man Tupel addieren und mit einer ganzen Zahl multiplizieren:

```
>>> t1 = (1, 'zwei', 3)
>>> t2 = ('one', 2, 'three')
>>> t1 + 2 * t2
(1, 'zwei', 3, 'one', 2, 'three', 'one', 2, 'three')
```

Während bei den Strings die Funktion `len` die Anzahl der Zeichen im String zurückgibt, gibt sie bei einem Tupel die Anzahl der Elemente im Tupel an.

```
>>> t = (1, 2.0, 4+3j, 'Klaus')
>>> len(t)
4
```

Der Operator `in` überprüft, ob ein bestimmtes Objekt in einem Tupel enthalten ist.

```
>>> t = (1, 2.0, 4+3j, 'Klaus')
>>> 3 in t
False
>>> 'Klaus' in t
True
```

Es gibt Fälle, in denen wir ein Tupel mit nur einem Element erzeugen müssen. Dies kann man erreichen, indem man an das Element ein Komma anhängt.

```
>>> t = (1, )
>>> type(t)
<class 'tuple'>
```

`list` Listen

Eine Liste ist, ähnlich wie ein Tupel, eine Aneinanderreihung von beliebigen Python-Objekten. Der wesentliche Unterschied zum Tupel besteht darin, dass eine Liste ein veränderliches Objekt ist, während das Tupel unveränderlich ist. Wir können bei einer Liste also nachträglich Objekte durch andere Objekte ersetzen und Objekte hinzufügen oder entfernen. Eine Liste erzeugt man, indem man die Objekte in eckigen Klammern mit Kommata trennt.

```
>>> liste = [1, 2.0, 4+3j, 'Klaus']
>>> type(liste)
<class 'list'>
```

Alle Operationen, die wir oben für Tupel besprochen haben, funktionieren genauso auch für Listen. Probieren Sie die Beispiele aus dem Abschnitt über Tupel für die entsprechenden Listen aus.

Listen sind veränderlich. Wir können nachträglich Objekte mit der Methode `append` an die Liste anhängen, wir können Objekte ersetzen, und wir können Elemente mit dem Befehl `del` aus der Liste entfernen, wie die folgenden Beispiele zeigen:

```
>>> liste = [1, 2, 3, 4, 5]
>>> liste[2] = 'drei'
>>> print(liste)
[1, 2, 'drei', 4, 5]
>>> liste.append('six')
>>> print(liste)
[1, 2, 'drei', 4, 5, 'six']
>>> del liste[1]
>>> print(liste)
[1, 'drei', 4, 5, 'six']
```

Neben `append` gibt es noch eine Reihe weiterer Methoden der Klasse `list`, die Sie in der Dokumentation der Standardbibliothek [7] unter dem Stichpunkt »Sequence Types« finden. In dem Python-Tutorial [8] gibt es unter dem Stichwort »Data-Types« eine sehr übersichtliche Darstellung der Methoden der Klasse `list`. So kann man beispielsweise mit `count` zählen, wie oft ein bestimmtes Objekt in der Liste vorkommt, oder man kann mit `index` den Index eines bestimmten Objektes suchen.

Vielleicht finden Sie es gewöhnungsbedürftig, dass die Datentypen, die wir vor der Liste kennengelernt haben, unveränderlich sind, und Sie denken sich, warum man denn überhaupt Tupel verwenden möchte, wenn es Listen gibt. Das folgende Beispiel zeigt, dass veränderliche Python-Objekte nicht ganz unproblematisch sind:

```
1 >>> liste = [1, 2, 3, 4]
2 >>> liste1 = liste
3 >>> liste[1] = 'zwei'
4 >>> print(liste)
5 [1, 'zwei', 3, 4]
6 >>> print(liste1)
7 [1, 'zwei', 3, 4]
```

Wenn Sie bisher mit einer Programmiersprache wie C oder Java gearbeitet haben, haben Sie wahrscheinlich erwartet, dass der letzte `print`-Befehl die Ausgabe `[1, 2, 3, 4]` erzeugt. Warum erzeugt Python hier diese Ausgabe? Dazu müssen wir uns daran erinnern, dass Variablen in Python Referenzen auf Objekte sind. Denken Sie bitte wieder an das Bild mit den Kisten. Die Variable `liste` ist ein Schild, das an der Kiste mit der Liste angebracht ist. Die zweite Zeile des Beispiels oben bewirkt, dass an diese Kiste ein zweites Schild mit der Aufschrift `liste1` gehängt wird. Die Kiste trägt jetzt zwei Schilder: eines mit der Aufschrift `liste` und eines mit der Aufschrift `liste1`. In Zeile 3 wird der Inhalt der Kiste verändert. Wenn wir nun den Inhalt der Kiste ausgeben lassen, so erhalten wir den veränderten Inhalt, ganz gleichgültig, ob wir die Kiste mit dem Namen `liste` ansprechen oder mit dem Namen `liste1`.

Achtung!

Variablen in Python sind Referenzen auf Objekte. Eine Variable ist also ein Name, den man dem Objekt gibt. Ein Objekt kann durch mehrere Namen angesprochen werden. Eine Zuweisung der Form `variable2 = variable1` kopiert das Objekt nicht, sondern erzeugt nur einen zweiten Namen, unter dem das Objekt angesprochen (referenziert) werden kann.

Wenn Sie eine Kopie einer Liste erstellen wollen, dann können Sie die Methode `copy` verwenden.

```
1 >>> liste = [1, 2, 3, 4]
2 >>> liste1 = liste.copy()
3 >>> liste[1] = 'zwei'
4 >>> print(liste)
5 [1, 'zwei', 3, 4]
6 >>> print(liste1)
7 [1, 2, 3, 4]
```

Man hat damit also erreicht, dass Änderungen in der Liste `liste` nun keinen Einfluss auf die Liste `liste1` haben, weil sich beide Variablen auf unterschiedliche Objekte beziehen.

`dict` Wörterbücher

Ein Dictionary `dict` ist ein Datentyp, der es erlaubt, eine Zuordnung von bestimmten Objekten zu anderen Objekten vorzunehmen. Sie können sich unter einem Dictionary am einfachsten ein Telefonbuch vorstellen, wie das folgende Beispiel zeigt.

```
>>> telefon = {'Klaus': 1353, 'Manfred': 5423, 'Petra': 3874}
>>> telefon['Klaus']
1353
```

Sie können erkennen, dass ein Dictionary durch ein Paar von geschweiften Klammern `{}` erzeugt wird. Jeder Eintrag besteht aus einem **Schlüssel** und einem bestimmten **Wert**, die durch einen Doppelpunkt voneinander getrennt sind. Die Schlüssel sind in unserem Fall die Namen und die Werte die Telefonnummern. Das Dictionary ordnet also jedem Schlüssel genau einen Wert zu.[9]

`set` Mengen

Ebenfalls mit geschweiften Klammern werden in Python Mengen (Datentyp `set`) gebildet. Unter einer Menge sollten Sie sich genau das vorstellen, was man in der mathematischen Mengenlehre unter einer Menge versteht. In einer Menge kann jedes Element nur maximal einmal vorkommen, und die Reihenfolge der Elemente spielt keine Rolle, wie das folgende Beispiel zeigt.

```
>>> a = {4, 3, 5, 4, 4, 3}
>>> print(a)
{3, 4, 5}
```

Auf Mengen kann man in Python die üblichen Operationen der Mengenlehre wie Vereinigung (`union`), Schnittmenge (`intersection`) und Differenz (`difference`) anwenden sowie die Frage stellen, ob eine Menge eine Teilmenge (`issubset`) oder eine Obermenge (`issuperset`) einer anderen Menge ist. Die Anwendung dieser Methoden wird im folgenden Beispiel demonstriert.

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3, 4, 6, 7}
>>> a.union(b)
{1, 2, 3, 4, 6, 7}
>>> a.intersection(b)
{2, 3, 4}
>>> a.difference(b)
{1}
>>> a.issubset(b)
False
>>> a.issuperset(b)
False
```

[9] Der Schlüssel muss dabei nicht unbedingt ein String sein. Es kann sich auch um einen anderen unveränderlichen Datentyp handeln.

Mengen benutzt man häufig dazu, um aus einer Liste von Zahlen die doppelten Einträge zu entfernen, indem man eine Liste erst mit der Funktion `set` in eine Menge umwandelt und anschließend wieder mit der Funktion `list` in eine Liste, wie das folgende Beispiel zeigt:

```
>>> liste = [1, 2, 3, 2, 3, 2, 5, 7, 5]
>>> liste2 = list(set(liste))
>>> liste2
[1, 2, 3, 5, 7]
```

Wenn man anstelle der Liste ein Tupel erzeugen möchte, kann man statt der Funktion `list` die Funktion `tuple` verwenden.

2.10 Arithmetische Zuweisungsoperatoren

Wir haben bereits Zuweisungen der Form

```
variable = wert
```

kennengelernt. Bei einer solchen Zuweisung wird einer Variablen ein bestimmter Wert zugeordnet. Es kommt relativ häufig vor, dass man den Wert einer Variablen verändern möchte, indem man die Variable zum Beispiel um eine bestimmte Zahl erhöht oder mit einem bestimmten Faktor multipliziert. Wenn wir die Variable `x` um zwei erhöhen möchten, dann kann man das in Python durch

```
x = x + 2
```

ausdrücken. Da Zuweisungen dieser Art sehr häufig vorkommen, gibt es in Python dafür eine spezielle Kurzschreibweise. Um die Variable `x` um zwei zu erhöhen, kann man auch die Anweisung

```
x += 2
```

benutzen. Man bezeichnet den Operator `+=` als einen **arithmetischen Zuweisungsoperator**, da er eine Rechenoperation mit einer Zuweisung verknüpft. Völlig analog kann man den Operator `-=` verwenden, um eine Variable um einen bestimmten Wert zu verringern. Der Operator `*=` multipliziert eine Variable mit einem bestimmten Faktor und `/=` teilt eine Variable durch einen bestimmten Faktor.

2.11 Mehrfache Zuweisungen (Unpacking)

Wir haben einige Datentypen kennengelernt, die aus einer Sammlung von Objekten bestehen: Strings (`str`), Tupel (`tuple`), Listen (`list`) und Mengen (`set`). Bei diesen Datentypen kann man eine besondere Form der Zuweisung verwenden, die in vielen Fällen sehr praktisch ist. Dabei werden die einzelnen Elemente getrennten Variablen zugewiesen. Wir demonstrieren dies am Beispiel einer Liste:

```
>>> a = [3, 7, 5]
>>> x, y, z = a
```

```
>>> print(x)
3
>>> print(y)
7
>>> print(z)
5
```

Es gibt einige Funktionen in Python, die zwei oder mehr Werte als Ergebnis zurückgeben. In diesen Fällen kann man die Ergebnisse direkt unterschiedlichen Variablen zuordnen. Ein Beispiel ist die Funktion `math.modf`, die eine Gleitkommazahl in den ganzzahligen Anteil und den gebrochenen Anteil zerlegt. Diese Funktion liefert ein Tupel der beiden Anteile zurück.

```
>>> import math
>>> a = math.modf(2.25)
>>> print(a)
(0.25, 2.0)
```

Mithilfe der mehrfachen Zuweisung kann man die beiden Teile direkt zwei unterschiedlichen Variablen zuweisen:

```
>>> import math
>>> x, y = math.modf(2.25)
>>> print(x)
0.25
>>> print(y)
2.0
```

Diese Art der mehrfachen Zuweisung wird auch als **Unpacking** bezeichnet, da das Tupel gewissermaßen ausgepackt wird und seine Bestandteile auf mehrere Variablen verteilt werden. Wichtig ist, dass die Anzahl der Variablen exakt mit der Anzahl der auszupackenden Elemente übereinstimmt. Wenn das Tupel (oder die Liste) mehr oder weniger Elemente enthält, als Sie Variablen angeben, erhalten Sie eine Fehlermeldung.

2.12 Indizierung von Ausschnitten (Slices)

Für die Datentypen, die aus einzelnen Elementen bestehen, die man mit einem Index ansprechen kann, gibt es eine weitere Variante der Indizierung, die man als **Slicing** bezeichnet. Dies betrifft Strings (`str`), Tupel (`tuple`) und Listen (`list`). Nehmen wir an, dass `a` eine Liste, ein Tupel oder ein String ist. Dann kann man mit

```
a[Anfang:Ende:Schrittweite]
```

eine bestimmte Teilmenge der Elemente von `a` auswählen. Dabei wird mit dem Index mit `Anfang` begonnen. Dieser Index wird in der Schrittweite `Schrittweite` hochgezählt. Das geschieht so lange, wie der Index kleiner als `Ende` ist. Wir demonstrieren dies an einem einfachen Beispiel mit einer Liste:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> a[4:12:2]
[4, 6, 8, 10]
```

Beginnend mit dem Index 4 wird in 2er-Schritten hochgezählt, solange der Index kleiner als 12 ist. Beachten Sie bitte, dass das Element `a[12]` *nicht* im Ergebnis auftaucht. Wenn die Schrittweite nicht angegeben wird, so wird eine Schrittweite von 1 angenommen. Wenn der Anfang und das Ende nicht angegeben werden, so wird vom Anfang der ursprünglichen Liste bzw. bis zum Ende der ursprünglichen Liste gezählt. Es sind auch negative Schrittweiten erlaubt. In diesem Fall wird die Liste rückwärts durchlaufen. Dazu muss dann natürlich `Anfang` größer sein als `Ende`. Wir demonstrieren die verschiedenen Möglichkeiten der Slices wieder an einigen Beispielen:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> a[5:]
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> a[:10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[::2]
[0, 2, 4, 6, 8, 10, 12, 14]
>>> a[::-1]
[14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> a[12:4:-2]
[12, 10, 8, 6]
```

2.13 Formatierte Strings

Wenn man mit Python eine physikalische Aufgabe löst, möchte man oft das Ergebnis in Form eines Antwortsatzes mit einer sinnvollen Anzahl von Dezimalstellen angeben. Es gibt dazu in Python verschiedene Möglichkeiten. Die am einfachsten zu benutzende besteht aus den sogenannten **formatierten Strings** oder kurz **f-Strings**. Ein f-String ist ein String, der mit dem Buchstaben `f` vor dem einleitenden Anführungszeichen markiert ist. Innerhalb des Strings können mehrere Python-Ausdrücke in geschweiften Klammern vorkommen. Die Ausdrücke in den geschweiften Klammern werden ausgewertet, und an ihrer Stelle wird das Ergebnis der Auswertung eingesetzt. Wir demonstrieren das an einem Beispiel:

```
>>> import math
>>> a = 2
>>> print(f'Die Quadratwurzel von {a} ist {math.sqrt(a)}.')
'Die Quadratwurzel von 2 ist 1.4142135623730951.'
```

Innerhalb der geschweiften Klammen kann man noch genauer angeben, wie der Ausdruck formatiert werden soll. Am häufigsten werden wir Gleitkommazahlen ausgeben müssen. Dazu kann man hinter den Ausdruck einen Doppelpunkt angeben. Nach dem Doppelpunkt kann angegeben werden, wie viele Zeichen für die Ausgabe der Zahl reserviert werden sollen. Dies ist wichtig, wenn man Zahlen untereinander so ausgeben möchte, dass zusammengehörige Dezimalstellen untereinander stehen. Nach dieser

Angabe folgt ein Punkt und danach die Angabe der Nachkommastellen. Es empfiehlt sich, danach noch den Buchstaben `f` anzuhängen. Diese Angabe erzwingt, dass die Zahl als Gleitkommazahl angegeben wird. Das obige Beispiel kann dann wie folgt aussehen, wobei wir auf die Angabe einer vorgegebenen Anzahl von Zeichen verzichtet haben.

```
>>> import math
>>> a = 2
>>> f'Die Quadratwurzel von {a:.3f} ist {math.sqrt(a):.3f}.'
'Die Quadratwurzel von 2.000 ist 1.414.'
```

Die f-Strings bieten noch viel mehr Formatierungsmöglichkeiten, die hier nicht im Detail besprochen werden können. Diese sind ausführlich unter dem Stichwort »Format Specification Mini-Language« der Dokumentation der Python-Standardbibliothek [7] dokumentiert.

2.14 Vergleiche und boolesche Ausdrücke

Aus der Schulmathematik kennen Sie die Vergleichsoperatoren gleich =, ungleich ≠, kleiner als <, größer als >, kleiner oder gleich ≤ und größer oder gleich ≥. Diese Vergleichsoperatoren werden in Python durch die Operatoren `==`, `!=`, `<`, `>`, `<=`, `>=` ausgedrückt und liefern jeweils einen Wahrheitswert vom Typ `bool` zurück, wie das folgende Beispiel zeigt.

```
>>> 3 < 4
True
>>> 3 > 5
False
>>> 6 == 2 * 3
True
```

Wahrheitswerte lassen sich mit den Operatoren `and` und `or` verknüpfen. Der Ausdruck `x and y` ist genau dann wahr, wenn sowohl `x` als auch `y` den Wert `True` haben. Der Ausdruck `x or y` ist genau dann wahr, wenn mindestens einer der beiden Ausdrücke `x` bzw. `y` wahr ist. Der Operator `not` negiert einen Wahrheitswert: Aus `True` wird `False` und umgekehrt. Wir zeigen dies wieder an einigen Beispielen:

```
>>> 3 < 4 or 5 > 2
True
>>> 3 < 4 or 5 < 2
True
>>> not 3 < 4
False
>>> 3 < 4 and 3 > 1
True
```

Wenn Sie beispielsweise überprüfen wollen, ob eine Variable `x` einen Wert zwischen 0 und 10 hat, dann können Sie dies wie folgt überprüfen:

```
>>> x = 3
>>> 0 <= x and x <= 10
True
```

Ausdrücke dieser Art, bei der mehrere Vergleichsoperatoren mit `and` verknüpft sind, lassen sich in Python verkürzt in der folgenden Form darstellen, die sich stark an die übliche Notation in der Mathematik anlehnt:

```
>>> x = 3
>>> 0 <= x <= 10
True
```

Neben der Gleichheit von zwei Objekten, die mit dem Operator `==` festgestellt wird, gibt es noch die Identität von zwei Objekten, die mit dem Operator `is` festgestellt wird. Der Unterschied zwischen `==` und `is` entspricht in etwa dem Bedeutungsunterschied zwischen »das Gleiche« und »dasselbe« in der deutschen Sprache. Um den Unterschied zu verdeutlichen, erzeugen wir zwei Listen mit dem gleichen Inhalt:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```

Die Listen `a` und `b` haben zwar den gleichen Inhalt, es sind aber unterschiedliche Objekte. Wir können zum Beispiel nachträglich ein Element der Liste `b` verändern und damit bewirken, dass sich die Inhalte der Listen unterscheiden. Anders sieht es im folgenden Beispiel aus:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a == b
True
>>> a is b
True
```

Die Operation `a is b` liefert nun `True` zurück. Das bedeutet, dass `a` und `b` ein und dasselbe Objekt sind. Selbst wenn wir nachträglich ein Element von `b` ändern, sind trotzdem beide Listen völlig identisch, da `a` und `b` nur zwei unterschiedliche Namen für dasselbe Objekt sind.

Vergleiche mit Gleitkommazahlen liefern gelegentlich überraschende Ergebnisse, wie das folgende Beispiel demonstriert. Wir betrachten dazu die offensichtlich erfüllte Gleichung $3 + 0{,}4 - 0{,}2 = 3{,}2$. Wenn wir diese Gleichung in Python aufschreiben, erhalten wir allerdings die folgende Ausgabe:

```
>>> 3 + 0.4 - 0.2 == 3.2
False
```

Aufgrund der unvermeidlichen Rundungsfehler weicht das Ergebnis der Berechnung `3 + 0.4 - 0.2` minimal von 3,2 ab, und daher liefert der Vergleich ein `False`. In

vielen Fällen kann man die Überprüfung der Gleichheit zweier Gleitkommazahlen vollständig vermeiden. Wenn Sie beispielsweise eine Schleife abbrechen möchten, wenn die Zahl `x` gleich null ist, sollten Sie überlegen, ob es nicht sogar logischer ist, die Schleife so lange auszuführen, wie die Zahl `x` größer als null ist. Wenn es unumgänglich ist, zwei Gleitkommazahlen auf (näherungsweise) Gleichheit zu überprüfen, sollten Sie den Vergleich `a == b` durch `abs(a - b) < fehler` ersetzen, wobei `fehler` eine hinreichend kleine Zahl ist und die eingebaute Funktion `abs` den Betrag einer Zahl berechnet.

Achtung!

Vergleichen Sie nie Gleitkommazahlen mit den Operatoren `==` oder `!=`. In vielen Fällen verursachen derartige Vergleiche schwer auffindbare Programmfehler.

Eine Besonderheit von Python ist, dass fast jedem Objekt eine Wahrheitswert zugewiesen wird.[10] Diesen findet man mit der Funktion `bool` heraus:

```
1  >>> bool(5.0)
2  True
3  >>> bool(0.0)
4  False
5  >>> bool([1, 2, 3])
6  True
7  >>> bool([])
8  False
9  >>> bool([0])
10 True
```

Die dahinterstehenden Regeln lauten:

- Die Zahl Null wird als `False` interpretiert.
- Leere Sammlungen von Objekten (Listen, Tupel, Mengen, Dictionaries, Strings) werden als `False` interpretiert.
- Die Objekte `False` und `None` werden als `False` interpretiert.
- Alle anderen Objekte werden als `True` interpretiert.

Im obigen Beispiel ergibt sich in den Zeilen 7 und 8 beispielsweise der Wert `False` für eine leere Liste. Eine Liste, die die Zahl Null enthält, wird dagegen als `True` interpretiert, wie in den Zeilen 9 und 10 zu erkennen ist.

2.15 Erstellen von Python-Programmen

Bisher haben wir die Befehle immer in der Python-Shell eingegeben. Sobald die Aufgaben etwas komplexer werden, lohnt es sich, alle Anweisungen in eine Datei zu schreiben und diese Anweisungen dann von Python ausführen zu lassen. Dazu muss man eine Textdatei erstellen, deren Name mit der Dateiendung `.py` endet. In dieser Datei werden

[10] Eine Ausnahme werden wir in Abschnitt 3.10 kennen lernen.

die Befehle zeilenweise aufgeschrieben. Anschließend kann man die Datei ausführen. Dazu muss man beim Starten von Python den Dateinamen hinter den Befehl `python` schreiben.

Erstellen Sie mit einem Texteditor (siehe Abschn. 2.3) eine Datei mit dem Namen `programm.py` und speichern Sie diese Datei in einem Ordner Ihrer Wahl. Der Inhalt der Datei ist in dem Listing Programm 2.1 angegeben.

Programm 2.1: *`Grundlagen/programm.py`*

```
1  """Berechnung der Quadratwurzel von 17."""
2
3  import math
4
5  # Weise a den Wert 17 zu.
6  a = 17
7  # Berechne die Quadratwurzel von a.
8  b = math.sqrt(a)
9  # Gib das Ergebnis aus.
10 print(b)
```

Um das Programm unter Windows mit der Distribution Anaconda laufen zu lassen, öffnen Sie über das Startmenü einen Anaconda-Prompt. Dort müssen Sie zunächst in das Verzeichnis wechseln, in dem die Datei gespeichert ist. Das geschieht mit dem Befehl `cd`[11], hinter dem Sie den Namen des Ordners angeben müssen. Falls Sie das Programm unter Windows auf einem anderen Laufwerk gespeichert haben, müssen Sie zunächst dieses Laufwerk als aktuelles Laufwerk auswählen. Dazu geben Sie bitte den Laufwerksbuchstaben gefolgt von einem Doppelpunkt ein und betätigen die Eingabetaste. Um das Programm ausführen zu lassen, geben Sie beim Starten von Python den Dateinamen hinter dem Befehl `python` an. Sie sollten im Anschluss folgende Ausgabe erhalten:[12]

```
(base) C:\Users\natt>cd Documents\Python
(base) C:\Users\natt\Documents\Python>python programm.py
4.123105625617661
(base) C:\Users\natt\Documents\Python>
```

Um das Programm unter einem anderen Betriebssystem oder einer anderen Python-Distribution zu starten, können Sie ganz analog vorgehen.

In dem Programm gibt es einige Besonderheiten: In der ersten Zeile steht ein Text, der die Funktion des Programms erläutert. Dieser Text ist in ein Paar von jeweils drei doppelten Anführungszeichen eingeschlossen. Man bezeichnet diesen Text als einen **Docstring**. Die Zeilen 5, 7 und 9 beginnen mit einer Raute `#`. Alles hinter dem Rautezeichen ist ein **Kommentar**, der nur dazu dient, das Programm besser verständlich zu machen. Der Inhalt des Kommentars wird von Python ignoriert. Dennoch sind die Kommentare wichtig, damit Sie später Ihre eigenen Programme noch verstehen.

Das Eingeben des Programmcodes und das Starten von Python ist etwas komfortabler, wenn Sie die Entwicklungsumgebung Spyder verwenden. In Abb. 2.2 ist eine

[11] Der Befehl `cd` steht für »change directory«.

[12] In dem dargestellten Beispiel wurde die Datei `programm.py` in einem Unterordner `Python` des Ordners `Documents` des Benutzers `natt` gespeichert.

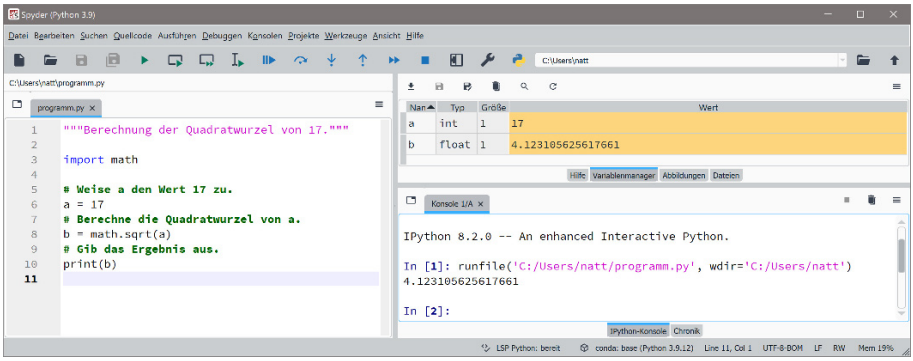

Abb. 2.2: Integrierte Entwicklungsumgebung Spyder. *In dem Fensterbereich auf der linken Seite ist das Programm* `programm.py` *geöffnet. Verschiedene Bestandteile des Programmcodes werden automatisch farblich hervorgehoben. Mit dem grünen Pfeil in der Werkzeugleiste oben kann man das Programm ausführen. Die Programmausgabe ist in dem Fensterbereich unten rechts zu sehen. In dem Fensterbereich oben rechts werden die definierten Variablen tabellarisch angezeigt.*

typische Sitzung zu sehen. Beim Arbeiten mit dem Programm werden Sie feststellen, dass Ihnen der Editor an vielen Stellen eine Auswahl von möglichen Befehlen anbietet. Die Funktionen des Programms Spyder sind zu umfangreich, als dass sie hier ausführlich beschrieben werden können. Es gibt allerdings im Hilfe-Menü ein ausführliches Tutorium, das Sie in die Funktionen dieser Entwicklungsumgebung einführt.

Wir werden im Folgenden immer wieder Beispiele besprechen, die mehrere Zeilen umfassen. Sie können diese natürlich trotzdem in der Python-Shell interaktiv eingeben. Praktischer ist es allerdings, wenn Sie für diese Beispiele jeweils eine entsprechende Datei anlegen. Um Ihnen eine Orientierung zu geben, welches Vorgehen Sie am besten wählen sollten, werden die Beispiele, die Sie besser in eine Datei schreiben sollten, ohne einen Python-Prompt `>>>` abgedruckt.

2.16 Kontrollstrukturen

In den meisten Fällen besteht ein Programm nicht einfach nur aus einer Liste von Befehlen, die der Reihe nach abgearbeitet werden. Vielmehr muss ein Programm flexibel auf ein Ergebnis einer Rechnung oder auf eine andere Bedingung reagieren, sodass bestimmte Programmteile nur bedingt oder wiederholt ausgeführt werden. Dazu gibt es bestimmte Konstrukte, die den Ablauf des Programms beeinflussen. Diese Konstrukte werden wir in den folgenden Abschnitten besprechen. Dabei spielen sogenannte **Anweisungsblöcke** eine wichtige Rolle, die in Python durch eine **Einrückung** gekennzeichnet sind. Hierdurch unterscheidet sich Python ganz wesentlich von vielen anderen Programmiersprachen (siehe Kasten).

Blöcke in Python

In vielen Programmiersprachen werden Blöcke von Anweisungen durch Klammern (geschweifte Klammern in C, C++, Java, C#) oder durch bestimmte Schlüsselwörter (begin und end in Pascal) gekennzeichnet. In Python wird ein Block immer durch einen Doppelpunkt eingeleitet. Die Zeilen, die zu einem Block gehören, werden durch die gleichmäßige Einrückung der Anweisungen gekennzeichnet.

Es ist üblich (aber nicht zwingend), dass zum Einrücken immer vier Leerzeichen verwendet werden. Es wird dringend davon abgeraten, zum Einrücken von Blöcken das Tabulatorzeichen zu verwenden, da es vom Texteditor abhängt, durch wie viele Leerzeichen ein Tabulator dargestellt wird. Viele Texteditoren lassen sich so konfigurieren, dass sie beim Drücken der Tabulatortaste automatisch eine vorgegebene Anzahl von Leerzeichen einfügen.

2.17 Funktionen

Eine Funktion ist eine Art Unterprogramm. Wir haben schon einige Funktionen wie die `print`-Funktion, die Funktion `len` oder die trigonometrischen Funktionen verwendet. Funktionen erlauben es, Code wiederzuverwenden. Nehmen Sie an, Sie schreiben ein Programm, in dem mehrfach zwischen Temperaturangaben in Grad Celsius und Grad Fahrenheit umgerechnet werden muss. Damit Sie nicht bei jeder Umrechnung wieder die gleiche Formel eintippen müssen, ist es praktisch, eine Funktion zu definieren, die diese Umrechnung vornimmt. Dies geschieht, wie im nächsten Beispiel gezeigt wird, mit dem Schlüsselwort `def`.

```
1 def fahrenheit_von_celsius(grad_c):
2     grad_f = grad_c * 1.8 + 32
3     return grad_f
4
5 x = 35.0
6 y = fahrenheit_von_celsius(x)
7 print(f'{x} Grad Celsius entsprechen {y} Grad Fahrenheit')
```

In der ersten Zeile wird die Funktion mit dem Namen `fahrenheit_von_celsius` definiert. In den runden Klammern steht die Liste von Argumenten. In diesem Beispiel haben wir nur ein Argument, dem wir den Namen `grad_c` geben. Nach der schließenden Klammer folgt ein Doppelpunkt, der unbedingt erforderlich ist, weil er einen **Anweisungsblock** einleitet. Die nächsten beiden Zeilen sind mit der gleichen Anzahl Leerstellen eingerückt. Die **Einrückung** zeigt an, dass diese Zeilen zusammen einen Anweisungsblock bilden. Dieser Block wird ausgeführt, wenn die Funktion aufgerufen wird. Das Ende des Blocks wird dadurch erkannt, dass die Zeile 5 nicht mehr eingerückt ist. Die Leerzeilen werden ignoriert. Es ist gängige Praxis, dass vor und nach einer Funktionsdefinition zwei Leerzeilen gelassen werden. Bei den sehr kurzen Beispielen in diesem Abschnitt werden wir aus Platzgründen jedoch darauf verzichten. In Zeile 2 erfolgt die Umrechnung von °C in °F. In Zeile 3 wird das Ergebnis mit einer `return`-Anweisung zurückgegeben. Die Zeilen 5 bis 7 zeigen die Anwendung der neu

definierten Funktion. Man kann erkennen, dass der Aufruf der Funktion als Ergebnis den Wert hat, der von der `return`-Anweisung zurückgegeben worden ist.

Beim Umgang mit Python-Funktionen sind einige Punkte zu beachten, die wir im Folgenden jeweils an einem möglichst kurzen Beispiel darstellen. Wenn eine Funktion eines ihrer Argumente verändert, dann ist das außerhalb der Funktion nicht sichtbar. Das folgende Programm liefert also die Ausgabe `7, 12` und nicht `12, 12`.

```
1 def f(x):
2     x = x + 5
3     return x
4
5 y = 7
6 z = f(y)
7 print(y, z)
```

Eine Funktion kann sehr wohl ein *veränderliches* Objekt, das ihr als Argument übergeben wurde, verändern. Dies zeigt das folgende Beispiel. Als Ausgabe erhalten wir `[1, 5, 3, 4]`, weil innerhalb der Funktion `f` die Liste verändert worden ist.

```
1 def f(x):
2     x[1] = 5
3     return x[2]
4
5 lst = [1, 2, 3, 4]
6 y = f(lst)
7 print(lst)
```

Eine Funktion kann lesend auf eine Variable zugreifen, die außerhalb der Funktion definiert wurde. Das folgende Programm liefert somit die Ausgabe `27`.

```
1 def f(x):
2     return x + q
3
4 q = 17
5 print(f(10))
```

Eine Funktion kann aber keine Variable *verändern*, die außerhalb der Funktion definiert worden ist. Das folgende Programm liefert zunächst die Ausgabe `12` und anschließend `17`. Die Zuweisung `q = 2` erzeugt eine neue Variable innerhalb der Funktion. Diese Variable hat zwar den gleichen Namen wie die Variable `q`, die außerhalb der Funktion definiert worden ist, es sind aber dennoch zwei unterschiedliche Variablen.

```
1 def f(x):
2     q = 2
3     return x + q
4
5 q = 17
6 print(f(10))
7 print(q)
```

Wenn man innerhalb einer Funktion eine Variable definiert, die es auch außerhalb der Funktion gibt, so wird die Variable außerhalb der Funktion verdeckt. Wir betrachten dazu das folgende Beispiel:

```
1 def f(x):
2     a = q
3     q = 7
4     return x + q
5
6 q = 17
7 print(f(10))
```

Man könnte erwarten, dass die Ausgabe `17` erzeugt wird. Stattdessen wird eine Fehlermeldung ausgegeben:

```
UnboundLocalError: local variable 'q' referenced before assignment
```

Da in Zeile 3 innerhalb der Funktion eine neue Variable `q` angelegt wird, ist die ursprüngliche Variable `q`, die außerhalb der Funktion definiert wurde, unsichtbar. In Zeile 2 wird also versucht, auf eine Variable zuzugreifen, der bisher noch kein Wert zugewiesen wurde.

Wenn eine Funktion mehrere Argumente hat, so werden diese normalerweise anhand ihrer Position innerhalb der Argumentliste zugewiesen und man bezeichnet die Argumente dann als **Positionsargumente**. Alternativ kann man aber auch die Namen der Argumente beim Aufruf der Funktion angeben, man spricht in diesem Fall von **Schlüsselwortargumenten**. Wir betrachten dazu wieder ein Beispiel:

```
1 def f(x, y, z):
2     return 100 * x + 10 * y + z
3
4 print(f(3, 5, 7))
5 print(f(y=5, z=7, x=3))
```

Die Ausgabe dieses Programms besteht zweimal aus dem Wert `357`. Im ersten Aufruf der Funktion in Zeile 4 werden die drei Argumente über die Position zugewiesen und im zweiten Aufruf in Zeile 5 über die Namen der Argumente. Sie können erkennen, dass in diesem Fall die Reihenfolge der Argumente ohne Bedeutung ist.

2.18 Funktionen mit optionalen Argumenten

Man kann ein Argument einer Funktion auch mit einem **Vorgabewert** belegen und dieses Argument damit zu einem **optionalen Argument** machen. Wir verdeutlichen das anhand der folgenden Definition einer Funktion, die die n-te Wurzel einer positiven Zahl berechnet:

```
1 def wurzel(x, n=2):
2     return x ** (1 / n)
```

Die Zuweisung `n=2` in der Argumentliste der Funktion sorgt dafür, dass das Argument `n` den Vorgabewert `2` hat, falls kein anderer Wert beim Aufruf der Funktion angegeben

wird. Man kann die Funktion nun mit einem oder mit zwei Argumenten aufrufen, wie die folgenden Zeilen zeigen:

```
3 print(f'Die Quadratwurzel von 4 ist {wurzel(4)}.')
4 print(f'Die Kubikwurzel von 27 ist {wurzel(27, 3)}.')
5 print(f'Die vierte Wurzel von 625 ist {wurzel(625, n=4)}.')
```

2.19 Bedingte Ausführung von Anweisungen

Es kommt häufig vor, dass man Anweisungen nur ausführen lassen möchte, wenn eine bestimmte Bedingung erfüllt ist. Dies wird in Python mit der `if`- und der `else`-Anweisung erreicht.

```
1 import math
2
3 a = 2
4 if a >= 0:
5     b = math.sqrt(a)
6     print(f'Die Quadratwurzel von {a} ist {b}.')
7 else:
8     print(f'Die Quadratwurzel von {a} ist keine reelle Zahl.')
9 print('Das Programm ist beendet.')
```

In Zeile 3 wird die Variable `a` definiert. In Zeile 4 wird überprüft, ob die Bedingung $a \geq 0$ erfüllt ist. Der Doppelpunkt am Ende der Bedingung leitet, ähnlich wie bei der Definition einer Funktion, wieder einen Anweisungsblock ein, der aus den Zeilen 5 und 6 besteht, die mit der gleichen Anzahl Leerstellen eingerückt sind. Dieser Block wird ausgeführt, wenn die Bedingung $a \geq 0$ erfüllt ist. Das Ende des Blocks wird dadurch erkannt, dass die Zeile 7 nicht mehr eingerückt ist. Nach der Anweisung `else:` kommt wieder ein Block, der diesmal nur aus einer Zeile besteht. Dieser Block wird ausgeführt, wenn die Bedingung $a \geq 0$ nicht erfüllt ist. Das Ende des Blocks wird wieder daran erkannt, dass die Zeile 9 nicht mehr eingerückt ist.

Man kann `if-else` Anweisungen in Python beliebig schachteln. Welches `else` zu welchem `if` gehört, ergibt sich daraus, dass die Befehle um die gleiche Anzahl Leerzeichen eingerückt sind. Selbstverständlich kann man bei einer bedingten Anweisung den Teil ab `else` auch komplett weglassen, wenn er nicht benötigt wird.

2.20 Bedingte Wiederholung von Anweisungen

Neben den einfachen bedingten Anweisungen möchte man oft bestimmte Anweisungen so lange wiederholen, wie eine bestimmte Bedingung erfüllt ist. Dazu dient in Python die `while`-Anweisung.

```
1 import math
2
3 a = 1
```

Tipp: Abbrechen von Python-Programmen

Wenn Sie mit Schleifen arbeiten, wird es Ihnen irgendwann sicher einmal passieren, dass Sie ein Python-Programm starten, das aufgrund eines Programmierfehlers sehr lange läuft oder vielleicht nie beendet wird. In diesem Fall können Sie das Programm jederzeit mit der Tastenkombination Strg+C abbrechen.

```
4 while a < 6:
5     b = math.sqrt(a)
6     print(f'Die Quadratwurzel von {a} ist {b}.')
7     a += 1
8 print('Das Programm ist beendet.')
```

Die Zeile 4 bildet zusammen mit dem Block, der aus den Zeilen 5 bis 7 besteht, eine Schleife. Beim ersten Durchlauf wird überprüft, ob die Bedingung $a < 6$ wahr ist. Da vorher $a = 1$ gesetzt wurde, werden die Zeilen 5 bis 7 ausgeführt und dabei a um eins erhöht. Am Ende von Zeile 7 wird wieder in Zeile 4 gesprungen und erneut überprüft, ob die Bedingung erfüllt ist. Das geschieht so lange, bis die Bedingung nicht mehr erfüllt ist. Danach wird die Ausführung des Programms in Zeile 8 fortgesetzt. Die Ausgabe des Programms sieht wie folgt aus:

```
Die Quadratwurzel von 1 ist 1.0.
Die Quadratwurzel von 2 ist 1.4142135623730951.
Die Quadratwurzel von 3 ist 1.7320508075688772.
Die Quadratwurzel von 4 ist 2.0.
Die Quadratwurzel von 5 ist 2.23606797749979.
Das Programm ist beendet.
```

Es gibt gelegentlich den Fall, dass es ungünstig ist, die Bedingung für den Abbruch einer Schleife am Anfang der Schleife zu überprüfen. In diesem Fall wird häufig die folgende Technik angewendet: Man programmiert zunächst mit `while` eine Endlosschleife, indem man für die Bedingung einfach den Ausdruck `True` einsetzt. An einer Stelle innerhalb der Schleife erfolgt ein `if`-Befehl, der die Schleife unter der gewünschten Bedingung mit dem Befehl `break` abbricht. Das folgende Programm verwendet diese Technik, um alle Quadratzahlen auszugeben, die kleiner oder gleich 20 sind.

```
1 a = 1
2 while True:
3     b = a**2
4     if b > 20:
5         break
6     print(f'Das Quadrat von {a} ist {b}.')
7     a += 1
```

Die Bedingung in einer `while`- oder in einer `if`-Anweisung kann in Python ein beliebiges Objekt sein. Die Bedingung wird ausgewertet, indem der Wahrheitswert des Objekts mithilfe der Funktion `bool` ermittelt wird, wie es am Ende von Abschn. 2.14 erläutert wurde. Dies wird am folgenden Code verdeutlicht:

```
a = 10
while a:
    print(a)
    a -= 1
```

Solange `a` einen Wert ungleich null hat, wird `a` gemäß den in Abschnitt 2.14 angegebenen Regeln als `True` interpretiert. Das Programm gibt also nacheinander die Zahlen von 10 bis 1 auf dem Bildschirm aus. Danach hat die Variable `a` den Wert null. Dieser wird als `False` interpretiert und die Schleife wird beendet.

Ich rate von dieser Art Code ab. Seien Sie lieber explizit und geben Sie die Abbruchbedingung in Form eines Vergleichs mit einem Vergleichsoperator an. Der Code

```
a = 10
while a > 0:
    print(a)
    a -= 1
```

lässt sich wesentlich einfacher verstehen und nachvollziehen als der Code mit der Bedingung `while a:`.

2.21 Schleifen über eine Aufzählung von Elementen

Nicht alle Wiederholungen von Anweisungen lassen sich mit der `while`-Schleife elegant ausdrücken. Häufig möchte man eine Reihe von Anweisungen für eine bestimmte Menge von Elementen ausführen. Dazu dient die `for`-Anweisung.

```
import math

liste = [1, 2, 3, 4, 5, 6, 7, 8, 9]
for a in liste:
    b = math.sqrt(a)
    print(f'Die Quadratwurzel von {a} ist {b}.')
```

In Zeile 3 wird eine Liste definiert, die die Zahlen 1 bis 9 enthält. In Zeile 4 befindet sich die `for`-Anweisung. Die `for`-Anweisung wählt das erste Element der Liste aus und weist es der Variablen `a` zu. Anschließend wird der Block, der aus den Zeilen 5 und 6 besteht, ausgeführt. Als Nächstes wird `a` das zweite Element der Liste zugewiesen, und die Zeilen 5 und 6 werden erneut ausgeführt. Dies wird wiederholt, bis die Variable `a` alle Elemente der Liste durchlaufen hat. Man sagt auch, die `for`-Schleife **iteriert** über die Elemente der Liste. Genauso kann man über die Elemente eines Tupels iterieren oder sogar über die Zeichen eines Strings. Man sagt, dass diese Objekte **iterierbar** sind. Probieren Sie doch einmal das folgende Beispiel aus:

```
for i in 'Hallo':
    print(i)
```

Der String `'Hallo'` ist eine Aneinanderreihung von Zeichen, und demzufolge iteriert die `for`-Schleife über alle Zeichen des Strings.

Häufig möchte man eine Reihe von Anweisungen für eine vorgegebene Anzahl von ganzen Zahlen ausführen. Dafür kann man das `range`-Objekt verwenden. Ein Objekt, das man durch den Aufruf `range(10)` erzeugt, liefert nacheinander die zehn Zahlen von 0 bis 9. Ähnlich wie bei der Slice-Operation kann man bei einem `range`-Objekt auch die Anfangszahl, die Endzahl und die Schrittweite angeben. Dabei ist, wie auch bei den Slices, zu beachten, dass die Endzahl nicht mehr mit ausgegeben wird.

```
range(10)              # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
range(2, 12, 3)        # 2, 5, 8, 11
range(2, 12, 2)        # 2, 4, 6, 8, 10
range(12, 4, -1)       # 12, 11, 10, 9, 8, 7, 6, 5
range(2, 10)           # 2, 3, 4, 5, 6, 7, 8, 9
```

Wir wollen die `for`-Schleife mit dem `range`-Objekt für ein Anwendungsbeispiel verwenden: In der Physik werden häufig Näherungen benutzt. Eine Näherung, die sehr häufig verwendet wird, ist durch

$$\sin(x) \approx x \qquad \text{für} \qquad x \ll 1$$

gegeben. Wir wollen uns die Frage stellen, wie groß der prozentuale Fehler ist, den man durch diese Näherung einführt. Diese Frage wird durch das Programm 2.2 beantwortet, wobei der mehrzeilige Docstring hier aus Platzgründen weggelassen wurde.

Programm 2.2: *Grundlagen/naeherung_sin1.py*

```
 9 import math
10
11 for winkel in range(5, 95, 5):
12     x = math.radians(winkel)
13     fehler = 100 * (x - math.sin(x)) / math.sin(x)
14     print(f'Winkel: {winkel:2} Grad, Fehler: {fehler:4.1f} %')
```

In einem späteren Beispiel wollen wir den Fehler der Näherung grafisch auftragen. Dazu benötigen wir eine Wertetabelle, die die Winkel und die zugehörigen Fehler enthält. Diese Aufgabe kann man zum Beispiel mit zwei Listen lösen. Das zugehörige Programm 2.3 werden wir im Folgenden abschnittsweise besprechen. Nach dem hier nicht mit abgedruckten Docstring und dem Importieren des Moduls `math` wollen wir eine Liste mit den Winkeln im Gradmaß erzeugen. Dies geschieht, indem wir ein passendes `range`-Objekt definieren und dieses mit der Funktion `list` in eine Liste umwandeln:

Programm 2.3: *Grundlagen/naeherung_sin2.py*

```
12 liste_winkel = list(range(5, 95, 5))
```

Als Nächstes wird eine zweite Liste angelegt, die zunächst noch leer ist. Diese Liste soll die berechneten Werte aufnehmen.

```
15 liste_fehler_prozentual = []
```

Wir iterieren nun mit einer `for`-Schleife über die Elemente der Liste `liste_winkel`. Für jeden Winkel berechnen wir den prozentualen Fehler der Näherung $\sin(x) \approx x$ und hängen diesen an die Liste `liste_fehler_prozentual` an.

```
18 for winkel in liste_winkel:
19     x = math.radians(winkel)
20     fehler = 100 * (x - math.sin(x)) / math.sin(x)
21     liste_fehler_prozentual.append(fehler)
```

Abschließend geben wir den Inhalt der beiden Listen auf dem Bildschirm aus.

```
24 print(liste_winkel)
25 print(liste_fehler_prozentual)
```

Wir werden in Abschn. 3.1 sehen, wie man die gleiche Aufgabe mithilfe der numerischen Bibliothek NumPy sehr viel kürzer erledigt.

2.22 Schleifen mit zip und enumerate

Gelegentlich kommt es vor, dass man über mehrere Listen, Tupel oder Ähnliches gleichzeitig iterieren möchte. Nehmen wir einmal an, wir haben zwei Listen von Zahlen und möchten diese tabellarisch ausgeben. Eine Möglichkeit, dies zu bewerkstelligen, ist der folgende Ansatz:

Programm 2.4: `Grundlagen/for_index.py`

```
3 liste1 = [2, 4, 6, 8, 10]
4 liste2 = [3, 5, 7, 9, 11]
5 for i in range(len(liste1)):
6     a = liste1[i]
7     b = liste2[i]
8     print(f'{a:3d}    {b:3d}')
```

Die Variable `i` durchläuft nacheinander die Indizes 0 bis 4, und in den Zeilen 6 und 7 werden über die Indizierung die entsprechenden Werte aus den Listen ausgelesen.

Sehr viel eleganter kann man das Problem mit der Python-Funktion `zip` lösen. An die Funktion `zip` kann man mehrere iterierbare Objekte (z.B. Listen) übergeben, und man erhält ein neues iterierbares Objekt. Der erste Wert dieses Objekts besteht aus einem Tupel der jeweils ersten Elemente aller Argumente. Der zweite Wert besteht aus einem Tupel der jeweils zweiten Elemente aller Argumente und so fort. Um sich die Funktion von `zip` zu veranschaulichen, kann man das Ergebnis in eine Liste umwandeln:

```
>>> a = [1, 2, 3]
>>> b = ['A', 'B', 'C']
>>> list(zip(a, b))
[(1, 'A'), (2, 'B'), (3, 'C')]
```

Mithilfe der Funktion `zip` lässt sich das Programm 2.4 wie folgt abwandeln:

Programm 2.5: `Grundlagen/for_zip.py`

```
3 liste1 = [2, 4, 6, 8, 10]
4 liste2 = [3, 5, 7, 9, 11]
5 for a, b in zip(liste1, liste2):
6     print(f'{a:3d}    {b:3d}')
```

In der Zeile 5 wird über die Elemente des `zip`-Objektes iteriert. In jeder Iteration wird ein Tupel zurückgegeben, das aus je einem Element der beiden Listen besteht. Dieses Tupel wird mit einer Mehrfachzuweisung auf die beiden Variablen `a` und `b` verteilt.

Eine ähnliche Funktionalität bietet die Funktion `enumerate`. Das Argument dieser Funktion muss ein iterierbares Objekt sein. Die Funktion liefert ein neues iterierbares Objekt. Mit jeder Iteration über dieses Objekt wird ein Tupel zurückgegeben, das einen Index und das entsprechende Element des Arguments von `enumerate` enthält. Mit dieser Funktion kann man das kleine Programm 2.6

Programm 2.6: *Grundlagen/for_ohne_enumerate.py*

```
3 primzahlen = [2, 3, 5, 7, 11, 13, 17, 19, 23]
4 for i in range(len(primzahlen)):
5     p = primzahlen[i]
6     print(f'Die {i+1}-te Primzahl ist {p}.')
```

etwas kürzer und eleganter in der folgenden Form schreiben.

Programm 2.7: *Grundlagen/for_mit_enumerate.py*

```
3 primzahlen = [2, 3, 5, 7, 11, 13, 17, 19, 23]
4 for i, p in enumerate(primzahlen):
5     print(f'Die {i+1}-te Primzahl ist {p}.')
```

2.23 Styleguide PEP 8

In vielen alltäglichen Bereichen gibt es Konventionen, die uns das Leben erleichtern. Wenn man in einer fremden Küche das Essbesteck sucht, findet man es meist sehr schnell, weil es sich in einer der Schubladen direkt unter der Arbeitsfläche befindet. Es gibt natürlich keinerlei Vorschrift, die den Aufbewahrungsort des Essbestecks in einer Küche regelt, aber es ist üblich und praktisch, dieses an dem oben beschriebenen Ort zu lagern.

Genauso wie es Konventionen im täglichen Leben gibt, gibt es auch Konventionen für die Verwendung einer Programmiersprache. Man spricht dann von sogenannten **Styleguides**. Für die Programmiersprache Python gibt es einen offiziellen Styleguide mit der Bezeichnung »PEP 8« [9]. Ich empfehle, dass Sie sich beim Erstellen von Programmen weitgehend an diesem Styleguide orientieren, da dies dafür sorgt, dass Ihr Code einheitlichen Konventionen folgt und damit leichter zu verstehen ist. Gerade beim Bearbeiten physikalischer Probleme mit dem Computer gibt es allerdings einige Ausnahmen zu beachten, auf die wir in Kapitel 4 noch genauer eingehen werden. An dieser Stelle möchte ich auf einige Punkte des Styleguides hinweisen, die ich für besonders wichtig halte:

- Benutzen Sie zum Einrücken von Anweisungsblöcken immer vier Leerzeichen.
- Beschränken Sie die Zeilenlänge auf maximal 79 Zeichen, da zu lange Zeilen die Lesbarkeit des Codes verschlechtern.

- Verwenden Sie vor und nach Operatoren jeweils ein Leerzeichen. Der Ausdruck `grad_f = grad_c * 1.8 + 32` lässt sich wesentlich einfacher lesen als der Ausdruck `grad_f=grad_c*1.8+32`.
- Verwenden Sie Kommentare, um den Zweck Ihres Codes zu erklären und um Ihren Code ggf. zu strukturieren. Versuchen Sie aber vor allem selbsterklärenden Code zu schreiben. Wenn Sie einen langen Kommentar schreiben, um eine sehr komplizierte Codezeile zu erklären, sollten Sie darüber nachdenken, ob man an dieser Stelle nicht vielleicht den Code selbst vereinfachen kann.
- Fügen Sie am Anfang jedes Python-Programms und bei jeder Definition einer Funktion einen Docstring ein, um Ihre Programme zu dokumentieren.

Die Dokumentation von Code mit Docstrings wird leider oft vernachlässigt. Bei sehr einfachen Funktionen, bei denen die Bedeutung der Argumente unmittelbar einleuchtend ist, kann der Docstring nur aus einer Zeile bestehen. Diese Zeile soll den Zweck der Funktion erklären, wie das folgende Beispiel zeigt:

```
1 def mittelwert(x):
2     """Berechne den Mittelwert der Elemente von x."""
3     return sum(x) / len(x)
```

Wenn die Erklärung des Zwecks einer Funktion mehr Platz benötigt, dann sollte dennoch eine Kurzbeschreibung der Funktion in der ersten Zeile stehen. Danach können Sie mit einer Zeile Abstand weitere Erklärungen zu der Funktion geben. Insbesondere sollten dann auch die Bedeutung und die Datentypen der Argumente erläutert werden. Ich empfehle dazu die Formatierung zu verwenden, die in der folgenden Beispielfunktion gezeigt wird. Bei dieser Art von Docstring handelt es sich um die von der Firma Google vorgeschlagene Formatierung, die ausführlich online dokumentiert ist [10], und die sich in der Praxis gut bewährt hat.

Programm 2.8: `Grundlagen/temperatur.py`

```
4  def fahrenheit_von_celsius(grad_c):
5      """Wandle eine Temperaturangabe von °C in °F um.
6
7      Die Berechnung erfolgt auf Grundlage der beiden Fixpunkte
8      der Temperaturskalen: 0 °C = 32 °F und 100 °C = 212 °F
9
10     Args:
11         grad_c (float):
12             Temperaturangabe in °C.
13
14     Returns:
15         float: Temperaturangabe in °F.
16     """
```

Viele Python-Entwicklungsumgebungen verarbeiten intern die Docstrings und geben Ihnen beim Programmieren eine aktive Unterstützung. Öffnen Sie doch einmal das Programm 2.8 in der Entwicklungsumgebung Spyder und setzen Sie den Cursor an eine Textstelle, an der die Funktion `fahrenheit_von_celsius` verwendet wird. Wenn Sie

nun die Tastenkombination Strg+i drücken, erhalten Sie direkt eine schön formatierte Ausgabe der Dokumentation zu dieser Funktion.

Zusammenfassung

Installation der benötigten Software: Wir haben besprochen, welche Software Sie auf Ihrem Computer installieren müssen, um mit diesem Buch arbeiten zu können.

Die Programmiersprache Python: Die grundlegenden Elemente der Programmiersprache Python wurden vorgestellt, soweit wir diese für die Arbeit mit diesem Buch benötigen. Die Sprache Python umfasst noch viel mehr, als dargestellt werden konnte. Für einen systematischen und ausführlicheren Einstieg empfehle ich das Buch von Langtangen [11], das einen besonderen Schwerpunkt auf das wissenschaftliche Programmieren setzt, sowie das Python-Tutorial [8], das von der Python-Foundation angeboten wird. Darüber hinaus gibt es eine unüberschaubare Vielfalt an einführenden Python-Büchern. Sie sollten bei der Wahl eines Buches aber darauf achten, dass das Buch sich explizit auf die Version 3 der Programmiersprache Python bezieht.

Dokumentation: In diesem Buch werden häufig Funktionen benutzt, ohne dass alle Argumente erklärt werden. Ich halte es nicht für sinnvoll, in einem Buch mit dem Schwerpunkt »physikalische Simulationen« eine komplette Dokumentation aller Funktionen abzudrucken. Sie sollten sich angewöhnen, bei allen Funktionen die entsprechende Hilfe in Python mit dem `help`-Befehl oder die entsprechende Onlinedokumentation zu Python [12] und der dazugehörigen Standardbibliothek [7] zu lesen.

Aufgaben

Aufgabe 2.1: Die Fibonacci-Folge ist eine rekursiv definierte Zahlenfolge. Die ersten beiden Zahlen sind eins, und jede weitere Zahl ergibt sich aus der Summe der beiden Vorgänger. Das Bildungsgesetz für die Folge lautet also: $a_n = a_{n-1} + a_{n-2}$ mit $a_0 = a_1 = 1$. Schreiben Sie eine Funktion `fibonacci`, die die ersten n Glieder der Fibonacci-Folge berechnet und als Liste zurückgibt. Benutzen Sie diese Funktion, um die ersten 20 Folgenglieder auszugeben.

Aufgabe 2.2: Erstellen Sie eine Funktion `ist_schaltjahr`. Diese Funktion soll als Argument eine Jahreszahl als ganze Zahl übergeben bekommen und einen booleschen Wert zurückgeben, der `True` ist, wenn es sich um ein Schaltjahr nach dem gregorianischen Kalender handelt. Benutzen Sie diese Funktion, um eine Liste der Schaltjahre von 1900 bis 2200 auszugeben.

Aufgabe 2.3: Das Collatz-Problem, das auch als die $(3n + 1)$-Vermutung bekannt ist, ist ein bislang ungelöstes mathematisches Problem, das sich auf die Collatz-Folge bezieht. Die Collatz-Folge zur Startzahl n ist wie folgt definiert: Starte mit einer natürlichen Zahl n. Wenn n eine gerade Zahl ist, dann wähle als

nächstes Folgenglied $n/2$. Wenn n eine ungerade Zahl ist, dann wähle als nächstes Folgenglied $3n + 1$.

a) Schreiben Sie eine Funktion `collatz`, die für eine gegebene Startzahl die Collatz-Folge in Form einer Liste zurückgibt. Brechen Sie die Liste ab, wenn die Folge die Zahl 1 erreicht hat. Die Folge wiederholt ab dann den Zyklus $1 \to 4 \to 2 \to 1 \ldots$ fortwährend.

b) Finden Sie mithilfe Ihrer Funktion `collatz` aus allen Startzahlen bis zu einer gegebenen maximalen Größe diejenige Startzahl, die die meisten Schritte in der Collatz-Folge benötigt, bis die Folge bei der Zahl eins endet.

Aufgabe 2.4: Schreiben Sie eine Funktion `quersumme`, die die Quersumme einer natürlichen Zahl berechnet. Verwenden Sie dabei die Operatoren `//` und `%`, um auf die Ziffern der Zahl zuzugreifen. Testen Sie Ihre Funktion, indem Sie diese auf eine vorgegebene Liste von Zahlen anwenden und das Ergebnis ausgeben lassen.

Aufgabe 2.5: Erstellen Sie eine Funktion `ggt`, die zwei positive ganze Zahlen als Argument erwartet und den größten gemeinsamen Teiler zurückgibt. Verwenden Sie dabei den klassischen euklidischen Algorithmus. Bei diesem wird sukzessive die größere der beiden Zahlen durch die Differenz der beiden Zahlen ersetzt, bis eine der beiden Zahlen null ist. Die andere Zahl ist dann der größte gemeinsame Teiler. Testen Sie Ihre Funktion, indem Sie für eine Reihe von zufälligen ganzen Zahlen das Ergebnis Ihrer Funktion `ggt` mit der Bibliotheksfunktion `math.gcd` vergleichen. Zur Erzeugung von Zufallszahlen können Sie die Funktion `random.randint` aus dem Modul `random` verwenden.

Aufgabe 2.6: Ein pythagoreisches Zahlentripel besteht aus drei natürlichen Zahlen $a \leq b \leq c$ mit $a^2 + b^2 = c^2$. Schreiben Sie ein Programm, das alle pythagoreischen Zahlentripel findet, für die zusätzlich die Bedingung $a + b + c = 1000$ erfüllt ist.

Aufgabe 2.7: Das Sieb des Eratosthenes ist ein Algorithmus zur Bestimmung von Primzahlen. Informieren Sie sich, wie dieser Algorithmus funktioniert, und implementieren Sie diesen in Python. Hinweis: Erstellen Sie eine Liste mit booleschen Werten. Ein Eintrag `True` bedeutet, dass die Zahl noch ein möglicher Primzahlkandidat ist. Alle Zahlen, die durch das Sieb fallen, werden als `False` markiert.

Aufgabe 2.8: Sie möchten ein Haus bauen und dazu ein Darlehen über 350 000 € aufnehmen. Der jährliche Zinssatz beträgt 1,9 % und Sie zahlen eine konstante monatliche Rate von 1800 €. Schreiben Sie ein Programm, das eine Tabelle ausgibt, aus der Sie für jeden Monat entnehmen können, wie viel Zinsen Sie bezahlen und wie groß die Restschuld ist. Das Programm soll außerdem ausgeben, wie lange es dauert, bis Sie Ihr Darlehen komplett zurückgezahlt haben und wie viel Zinsen Sie insgesamt bezahlt haben. Benutzen Sie die eingebaute Funktion `round`, um die berechneten Zinsen jeweils auf ganze Cent zu runden.

Aufgabe 2.9: Das Ziegenproblem ist in den 1990er Jahren in Deutschland durch eine Fernseh-Spielshow der breiten Öffentlichkeit bekannt geworden. Hinter einem von drei verschlossenen Toren befindet sich ein Hauptgewinn und hinter den anderen Toren ein Trostpreis. Der Kandidat entscheidet sich für eines der drei Tore. Anschließend wird eines der beiden anderen Tore geöffnet, wobei es sich dabei natürlich nicht um das Tor mit dem Hauptgewinn handelt. Der Kandidat kann nun entweder bei seiner ursprünglichen Wahl bleiben, oder sich für das andere noch geschlossene Tor entscheiden. Anschließend werden alle Tore geöffnet und der Kandidat erhält als Gewinn den Preis, der sich hinter dem von ihm zuletzt gewählten Tor befindet.

a) Schreiben Sie eine Funktion, die eine Runde dieses Spiels simuliert. Als Argument soll die Funktion einen booleschen Wert erwarten, der angibt, ob der Kandidat seine ursprüngliche Wahl immer ändert oder immer beibehält. Ein weiteres optionales boolesches Argument der Funktion soll es ermöglichen, den genauen Spielverlauf als Text auszugeben. Die Funktion soll einen booleschen Wert zurückgeben, der anzeigt, ob der Kandidat gewonnen hat oder nicht.

   Das Problem lässt sich besonders elegant lösen, wenn man ein Tupel mit den möglichen Toren definiert. Mithilfe der Funktion `random.choice` kann man ein zufälliges Tor auswählen. Danach kann man das Tupel in eine Menge umwanden und mithilfe der Mengenoperationen die Spielregeln besonders elegant implementieren.

b) Benutzen Sie diese Funktion, um für eine große Anzahl (z.B. 100 000) von Spieldurchgängen die relative Häufigkeit der Gewinne bei den beiden Spielstrategien »Ich bleibe stets bei meiner ursprünglichen Wahl« und »Ich wechsle immer« zu berechnen.

Literatur

[1] Welcome to Python. Python Software Foundation. https://www.python.org.

[2] Anaconda – The World's Most Popular Data Science Platform. https://www.anaconda.com.

[3] PyCharm. The Python IDE for Professional Developers. https://www.jetbrains.com/pycharm.

[4] Spyder: The Scientific Python Development Environment. https://spyder-ide.org.

[5] Visual Studio Code – Code editing. Redefined. https://code.visualstudio.com.

[6] IPython Documentation. https://ipython.readthedocs.io.

[7] The Python Standard Library. Python Software Foundation. https://docs.python.org/3/library.

[8] Python Tutorial. Python Software Foundation. https://docs.python.org/tutorial/index.html.

[9] PEP 8 – Style Guide for Python Code. Python Software Foundation. https://www.python.org/dev/peps/pep-0008.

[10] Example Google Style Python Docstrings. https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html.

[11] Langtangen HP. A Primer on Scientific Programming with Python. Berlin, Heidelberg: Springer, 2016. DOI:10.1007/978-3-662-49887-3.

[12] Python documentation. Python Software Foundation. https://docs.python.org.