Thomas Theis

Einstieg in Python

Ideal für Programmiereinsteiger

+ Schritt für Schritt eigene Programme entwickeln
+ Mit vielen Beispielen und Übungen
+ GUI, OOP, Datenbank- und Internetanwendungen u. v. m.

Gedruckt in Deutschland
Ohne Folienkaschierung
Mineralölfreie Druckfarben

Alle Codebeispiele zum Download

Rheinwerk Computing

Kapitel 1
Einführung

In diesem Kapitel stelle ich Ihnen Python kurz vor. Sie lernen die Vorteile von Python kennen und erfahren, wie Sie Python unter Windows, unter Ubuntu Linux und unter macOS installieren.

1.1 Vorteile von Python

Python ist eine sehr einfach zu erlernende Programmiersprache und für den Einstieg in die Welt der Programmierung ideal geeignet. Trotz ihrer Einfachheit bietet diese Sprache auch die Möglichkeit, komplexe Programme für vielfältige Anwendungsgebiete zu schreiben.

Python eignet sich besonders zur schnellen Entwicklung umfangreicher Anwendungen. Diese Technik ist unter dem Stichwort RAD (*Rapid Application Development*) bekannt geworden. Python vereint zu diesem Zweck folgende Vorteile:

- Eine einfache, eindeutige Syntax: Python ist für alle, die in die Programmierung einsteigen, eine ideale Programmiersprache. Sie beschränkt sich auf einfache, klare Anweisungen und häufig auf einen einzigen möglichen Lösungsweg. Dieser prägt sich schnell ein und wird der Entwicklerin bzw. dem Entwickler vertraut.
- Klare Strukturen: Python verlangt vom Entwickler, in einer gut lesbaren Struktur zu schreiben. Die Anordnung der Programmzeilen ergibt zugleich die logische Struktur des Programms.
- Wiederverwendung von Code: Die Modularisierung, also die Zerlegung eines Problems in Teilprobleme und die anschließende Zusammenführung der Teillösungen zu einer Gesamtlösung, wird in Python sehr leicht gemacht. Die vorhandenen Teillösungen können unkompliziert für weitere Aufgabenstellungen genutzt werden, sodass Sie bald über einen umfangreichen Pool an Modulen verfügen.
- Objektbearbeitung: In Python werden alle Daten als Objekte gespeichert. Dies führt zu einer einheitlichen Behandlung für Objekte unterschiedlichen Typs. Andererseits erfolgt die physikalische Speicherung der Objekte von Python automatisch, also

ohne Eingriff des Entwicklers. Dieser muss sich nicht um die Reservierung und Freigabe geeigneter Speicherbereiche kümmern.

- Interpreter/Compiler: Python-Programme werden unmittelbar interpretiert. Sie müssen nicht erst kompiliert und gebunden werden. Dies ermöglicht einen häufigen, schnellen Wechsel zwischen Codierungs- und Testphase.
- Unabhängigkeit vom Betriebssystem: Sowohl Programme, die von der Kommandozeile aus bedient werden, als auch Programme mit grafischen Benutzeroberflächen können auf unterschiedlichen Betriebssystemen (Windows, Linux, macOS) ohne Neuentwicklung und Anpassung eingesetzt werden.

1.2 Verbreitung von Python

Aufgrund seiner vielen Vorzüge gehört Python zu den beliebtesten Programmiersprachen. So wird es zum Beispiel innerhalb des Projekts *100-Dollar-Laptop*, das der Schulausbildung von Kindern in aller Welt dient, für die Benutzeroberfläche verwendet. Aber auch in zahlreichen großen Unternehmen wird Python eingesetzt, hier ein paar Beispiele:

- YouTube wurde zum großen Teil mithilfe von Python entwickelt.
- NASA nutzt Python zur Softwareentwicklung im Zusammenhang mit den Space-Shuttle-Missionen.
- Industrial Light & Magic: Auch Hollywood setzt auf Python – die Produktionsfirma ILM (Star Wars, Indiana Jones, Fluch der Karibik) nutzt es zum Beispiel bei der Entwicklung von Spezialeffekten.
- Honeywell: Python wird weltweit in vielen Firmen zur allgemeinen Hardware- und Softwareentwicklung eingesetzt.

1.3 Aufbau des Buchs

Das vorliegende Buch führt Sie in die Programmiersprache Python in der aktuellen Version 3.10 ein, die im Oktober 2021 erschienen ist. Besonderer Wert wird darauf gelegt, dass Sie selbst praktisch mit Python arbeiten. Daher empfehle ich Ihnen, von Anfang an dem logischen Faden von Erklärungen und Beispielen zu folgen.

Erste Zusammenhänge werden in Kapitel 2, »Erste Schritte«, anhand von einfachen Berechnungen vermittelt. Außerdem lernen Sie, ein Programm einzugeben, zu speichern und es unter den verschiedenen Umgebungen auszuführen.

Sie werden die Sprache spielerisch kennenlernen. Daher begleitet Sie ein selbst programmiertes Spiel durch das Buch. In dem Spiel sollen eine oder mehrere Kopfrechenaufgaben gelöst werden. Es wird mit dem »Programmierkurs« in Kapitel 3 eingeführt und im weiteren Verlauf des Buchs kontinuierlich erweitert und verbessert.

Nach der Vorstellung der verschiedenen Datentypen mit ihren jeweiligen Eigenschaften und Vorteilen in Kapitel 4, »Datentypen«, werden die Programmierkenntnisse in Kapitel 5, »Weiterführende Programmierung«, vertieft. Kapitel 6, »Objektorientierte Programmierung«, widmet sich der objektorientierten Programmierung mit Python. Einige nützliche Module zur Ergänzung der Programme werden in Kapitel 7, »Verschiedene Module«, vorgestellt.

In Kapitel 8, »Dateien«, und in Kapitel 10, »Datenbanken«, lernen Sie, Daten dauerhaft in Dateien oder Datenbanken zu speichern. Python wird zudem in der Internetprogrammierung eingesetzt. Die Zusammenhänge zwischen Python und dem Internet vermittelt Kapitel 9, »Internet«.

Sowohl Windows als auch Ubuntu Linux und macOS bieten komfortable grafische Benutzeroberflächen (GUIs). Kapitel 11, »Benutzeroberflächen«, beschäftigt sich mit der GUI-Erzeugung mithilfe des Moduls `tkinter`. Dieses stellt eine Schnittstelle zwischen dem grafischen Toolkit *Tk* und Python dar. In Kapitel 12 wird die GUI-Erzeugung mithilfe des Moduls `PyQt6` behandelt. Dieses beinhaltet die Elemente von PyQt in der Version 6. PyQt dient als Schnittstelle zwischen der Qt-Bibliothek und Python.

Für die Hilfe bei der Erstellung dieses Buchs bedanke ich mich bei dem gesamten Team des Rheinwerk Verlags, besonders bei Anne Scheibe.

1.4 Übungen

Im Buch finden Sie zahlreiche Übungsaufgaben. Ich empfehle Ihnen, sie unmittelbar zu lösen. Auf diese Weise können Sie Ihre Kenntnisse prüfen, bevor Sie zum nächsten Thema übergehen. Die Lösungen zu den Übungen finden Sie zusammen mit den Beispielprogrammen in den Materialien zum Buch. Beachten Sie dabei Folgendes:

- Es gibt für jedes Problem viele richtige Lösungen. Sieht Ihre Lösung nicht genauso aus wie die angegebene, ist das kein Problem. Betrachten Sie die angegebene Lösung vielmehr als Anregung und als Alternative.
- Bei der eigenen Lösung der Aufgaben wird sicherlich der eine oder andere Fehler auftreten – lassen Sie sich dadurch nicht entmutigen …

- ... denn nur aus Fehlern kann man lernen. Auf die vorgeschlagene Art und Weise werden Sie Python wirklich erlernen – nicht allein durch das Lesen von Programmierregeln.

1.5 Installation unter Windows

Python ist eine frei verfügbare Programmiersprache, die unter verschiedenen Betriebssystemen eingesetzt werden kann. Die jeweils neuesten Python-Versionen können Sie von der offiziellen Python-Website *https://www.python.org* aus dem Internet laden. Zurzeit (im April 2022) sind das die Dateien *python-3.10.4.exe* für ein 32-Bit-System und *python-3.10.4-amd64.exe* für ein 64-Bit-System. Sie können sie unter Windows 8, Windows 10 und Windows 11 installieren.

Rufen Sie zur Installation unter einem 64-Bit-Windows die ausführbare Datei *python-3.10.4-amd64.exe* auf. Als Erstes müssen Sie bestätigen, dass Sie ein Programm installieren möchten, das nicht aus dem Microsoft Store stammt.

Wählen Sie die Option Customize Installation aus. Lassen Sie alle Häkchen bei den Optional Features gesetzt. Das gilt besonders für das Paketverwaltungsprogramm *pip*, mit dessen Hilfe Sie später zusätzliche Module installieren können, siehe auch Abschnitt A.1, »Paketverwaltungsprogramm ›pip‹«. Setzen Sie bei den Advanced Options zusätzlich das Häkchen bei Add Python to environment variables, damit Sie später die Möglichkeit haben, Python-Programme auf Ebene der Kommandozeile aus einem beliebigen Verzeichnis heraus zu starten. Wählen Sie das Installationsverzeichnis *C:\Python*.

Anschließend steht im Startmenü ein Eintrag zu Python 3.10, siehe Abbildung 1.1. Möchten Sie bestimmte Einstellungen der Installation im Nachhinein verändern, können Sie die Installationsdatei erneut aufrufen.

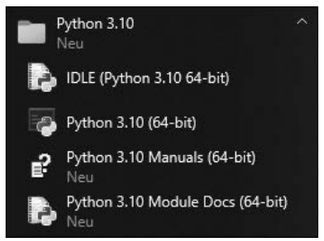

Abbildung 1.1 Startmenü mit Eintrag zu Python 3.10

Bei dem Programm IDLE im Startmenü handelt es sich um eine Entwicklungsumgebung, die selbst in Python geschrieben ist und mit der Sie im Folgenden Ihre Programme schreiben werden. Am besten ziehen Sie eine Verknüpfung zu IDLE auf den Desktop.

1.6 Installation unter Ubuntu Linux

Stellvertretend für andere Linux-Distributionen wird in diesem Buch Ubuntu Linux 21.10 genutzt. Python 3 ist unter Ubuntu Linux bereits installiert und darf auch nicht deinstalliert werden. In einem Terminal können Sie mithilfe des Befehls `python3 -V` (mit großem »V«) die aktuelle Versionsnummer ermitteln.

Zur Installation der Entwicklungsumgebung IDLE geben Sie in einem Terminal den folgenden Befehl ein: `sudo apt install idle3`. Das Programm IDLE können Sie anschließend mit dem Befehl `idle` starten.

1.7 Installation unter macOS

Die jeweils neuesten Python-Versionen können Sie von der offiziellen Python-Website *https://www.python.org* aus dem Internet laden. Zurzeit (im April 2022) ist das für macOS die Datei *python-3.10.4-macos11.pkg*.

Nach einem Doppelklick auf diese Datei startet die Installation. Nehmen Sie keine Änderungen vor, landet Python im Verzeichnis *Programme/Python 3.10*. Darin finden Sie einen Eintrag für die Entwicklungsumgebung IDLE, den Sie als Verknüpfung auf den Desktop ziehen können.

Kapitel 4
Datentypen

In Python werden alle Daten in Objekten gespeichert. Dieses Kapitel beschäftigt sich mit den Eigenschaften und Vorteilen der verschiedenen Datentypen für die Objekte. Operationen, Funktionen und Operatoren für die jeweiligen Datentypen werden vorgestellt. Ein eigener Abschnitt über Objektreferenzen und Objektidentität vervollständigt die Betrachtung.

Es geht zunächst um Zahlen. Anschließend folgen Strings (= Zeichenketten), Listen, Tupel, Dictionarys und Sets. Gemeinsamkeiten und Unterschiede der Datentypen werden erläutert.

4.1 Zahlen

Ganze Zahlen, Zahlen mit Nachkommastellen, Brüche und Operationen mit Zahlen sind Thema dieses Abschnitts. Es gibt einige eingebaute Funktionen für Zahlen. Das Modul `math` enthält eine Reihe von mathematischen Funktionen zur Durchführung von Berechnungen.

4.1.1 Ganze Zahlen

Als Datentyp für ganze Zahlen dient `int` (von englisch *integer* für ganzzahlig). Mit Zahlen dieses Typs wird mathematisch genau gearbeitet.

Üblicherweise wird das dezimale Zahlensystem mit der Basis 10 benutzt. Außerdem stehen in Python die folgenden Zahlensysteme zur Verfügung:

- das duale Zahlensystem (mit der Basis 2)
- das oktale Zahlensystem (mit der Basis 8)
- das hexadezimale Zahlensystem (mit der Basis 16)

Ein Beispiel:

```
a = 27
print("Dezimal:", a)
print("Hexadezimal:", hex(a))
print("Oktal:", oct(a))
print("Dual:", bin(a))

b = 0x1a + 12 + 0b101 + 0o67
print("Summe:", b)
```

Listing 4.1 Datei »zahl_ganz.py«

Folgende Ausgabe wird erzeugt:

```
Dezimal: 27
Hexadezimal: 0x1b
Oktal: 0o33
Dual: 0b11011
Summe: 98
```

Die dezimale Zahl 27 wird in die drei anderen Zahlensysteme umgerechnet und ausgegeben.

Die Funktion `hex()` dient zur Umrechnung und Ausgabe der Zahl in das hexadezimale System. Dieses System nutzt neben den Ziffern 0 bis 9 die Buchstaben »a« bis »f« (oder auch »A« bis »F«) als Ziffern für die Werte von 10 bis 15. Die Zahl 0x1b entspricht dem folgenden Wert:

$1 \times 16^1 + B \times 16^0 = 1 \times 16^1 + 11 \times 16^0 = 16 + 11 = 27$

Zur Umrechnung und Ausgabe der Zahl in das oktale System dient die Funktion `oct()`. Das oktale System nutzt nur die Ziffern 0 bis 7. Die Zahl 0o33 entspricht dem folgenden Wert:

$3 \times 8^1 + 3 \times 8^0 = 24 + 3 = 27$

Die Funktion `bin()` dient zur Umrechnung und Ausgabe der Zahl in das duale System. Dieses System nutzt nur die Ziffern 0 und 1. Die Zahl 0b11011 entspricht dem folgenden Wert:

$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 8 + 2 + 1 = 27$

Sie können auch direkt mit Zahlen in anderen Zahlensystemen rechnen. Die Berechnung der Variablen `b` ergibt:

$0 \times 1a + 12 + 0b101 + 0o67 =$

$1 \times 16^1 + A \times 16^0 + 12 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 6 \times 8^1 + 7 \times 8^0 =$

$16 + 10 + 12 + 4 + 1 + 48 + 7 = 98$

Bei der Eingabe oder Zuweisung muss das Präfix `0x`, `0b` bzw. `0o` vor den weiteren Ziffern stehen, damit das Zahlensystem erkannt wird.

Zahlen setzen sich auf der niedrigsten Ebene aus Bits und Bytes zusammen. In Abschnitt 4.1.9, »Bitoperatoren«, werden Sie noch ein wenig intensiver mit Dualzahlen, der Funktion `bin()` und den sogenannten Bitoperatoren arbeiten, die Ihnen den Zugriff auf Bit-Ebene erleichtern.

4.1.2 Zahlen mit Nachkommastellen

Der Datentyp für Zahlen mit Nachkommastellen heißt `float`. Diese sogenannten Fließkommazahlen werden mithilfe eines Dezimalpunkts und gegebenenfalls in Exponentialschreibweise angegeben.

Dazu ein kleines Beispiel:

```
a = 7.5
b = 2e2
c = 3.5E3
d = 4.2e-3
e = 1_250_000.500_001

print(a, b, c, d, e)
```

Listing 4.2 Datei »zahl_nachkomma.py«

Die Ausgabe lautet:

```
7.5 200.0 3500.0 0.0042 1250000.500001
```

Die Variable `a` erhält den Wert `7.5`. Die Nachkommastellen folgen nach dem Dezimalpunkt. Dies gilt auch für die Eingabe einer Zahl mit Nachkommastellen mithilfe der Funktion `input()`. Die Variable `b` erhält den Wert 200 (= $2 \times 10^2 = 2 \times 100$). Die Variable `c` erhält den Wert 3.500 (= $3{,}5 \times 10^3 = 3{,}5 \times 1.000$), die Variable `d` den Wert 0,0042 (= $4{,}2 \times 10^{-3} = 4{,}2 \times 0{,}001$).

Bei der Zuweisung in Exponentialschreibweise wird mithilfe des Zeichens »e« (oder »E«) ausgedrückt, um wie viele Stellen und in welche Richtung der Dezimalpunkt innerhalb

der Zahl verschoben wird. Diese Schreibweise eignet sich zum Beispiel für sehr große oder sehr kleine Zahlen, da sie die Eingabe vieler Nullen erspart.

Seit Python 3.6 können Sie einen Unterstrich benutzen, um Zahlen mit vielen Ziffern lesbarer zu machen. Es bietet sich an, ihn nach jeder dritten Ziffer einzufügen, wie es bei der Variablen `e` gemacht wurde.

4.1.3 Typ ermitteln

Es ist häufig nützlich zu wissen, ob es sich bei einer Zahl um eine ganze Zahl (Datentyp `int`) oder eine Fließkommazahl (Datentyp `float`) handelt. Die Funktion `type()` gibt den Typ (die Klasse) eines Objekts aus, nicht nur für Zahlentypen. Hierzu ein Programmbeispiel:

```
a = 2
print("Typ:", type(a))

b = 12/6
print("Typ:", type(b))

c = 12//6
print("Typ:", type(c))

d = 12%6 == 0
print("Vergleich liefert:", d)
print("Typ:", type(d))
```

Listing 4.3 Datei »zahl_type.py«

Das Programm erzeugt die folgende Ausgabe:

```
Typ: <class 'int'>
Typ: <class 'float'>
Typ: <class 'int'>
Vergleich liefert: True
Typ: <class 'bool'>
```

Die Variable `a` enthält den Wert 2 und ist vom Typ `int`. Die Variable `b` enthält den gleichen Wert, allerdings als Ergebnis einer mathematischen Division. Es handelt sich um ein Objekt des Typs `float`. Eine ganzzahlige Division ergibt einen Wert des Typs `int`.

Vergleichen Sie das Ergebnis einer Modulo-Operation mit 0, erhalten Sie die Information, ob eine Zahl durch eine andere Zahl glatt teilbar ist. Der Vergleich selbst liefert ein Objekt der Klasse `bool`.

4.1.4 Exponentialoperator **

Der Exponentialoperator ** dient zur Berechnung von Potenzen, also eines Ausdrucks der Form *Basis hoch Exponent*. Es folgen einige Beispiele:

```
z = 5 ** 3
print("5 hoch 3 =", z)
z = -5.2 ** -3.8
print("-5.2 hoch -3.8 =", z)
z = 9 ** 0.5
print("Quadratwurzel aus 9 = 9 hoch 1/2 =", z)
z = 27 ** (1.0/3.0)
print("Kubikwurzel aus 27 = 27 hoch 1/3 =", z)
```

Listing 4.4 Datei »zahl_hoch.py«

Es wird die folgende Ausgabe erzeugt:

```
5 hoch 3 = 125
-5.2 hoch -3.8 = -0.0019018983172844654
Quadratwurzel aus 9 = 9 hoch 1/2 = 3.0
Kubikwurzel aus 27 = 27 hoch 1/3 = 3.0
```

Sowohl bei der Basis als auch beim Exponenten kann es sich um positive oder negative ganze Zahlen oder Zahlen mit Nachkommastellen handeln. Mithilfe des Exponentialoperators lassen sich zum Beispiel auch Quadratwurzeln und Kubikwurzeln berechnen.

4.1.5 Rundung und Konvertierung

Die eingebaute Funktion `round()` dient zur Rundung einer Zahl. Im Unterschied dazu schneidet die bereits bekannte Funktion `int()` die Nachkommastellen einer Zahl ab. Einige Beispiele:

```
x = 12/7
print("x:", x)
print("Gerundet auf drei Stellen:", round(x,3))
print("Gerundet auf null Stellen:", round(x))
print("int(x):", int(x))
```

```
print()

x = -12/7
print("x:", x)
print("Gerundet auf drei Stellen:", round(x,3))
print("Gerundet auf null Stellen:", round(x))
print("int(x):", int(x))
```

Listing 4.5 Datei »zahl_runden.py«

Es wird folgende Ausgabe erzeugt:

```
x: 1.7142857142857142
Gerundet auf drei Stellen: 1.714
Gerundet auf null Stellen: 2
int(x): 1

x: -1.7142857142857142
Gerundet auf drei Stellen: -1.714
Gerundet auf null Stellen: -2
int(x): -1
```

Es werden die beiden Divisionen `12 / 7` und `-12 / 7` ausgeführt. Die beiden Ergebnisse werden jeweils auf drei verschiedene Arten umgewandelt:

1. Mithilfe der eingebauten Funktion `round()` wird das Ergebnis auf drei Stellen nach dem Komma gerundet.
2. Mit der gleichen Funktion wird das Ergebnis auf die nächsthöhere bzw. nächstniedrigere ganze Zahl gerundet.
3. Mithilfe der eingebauten Funktion `int()` wird das Ergebnis in eine ganze Zahl umgewandelt.

4.1.6 Winkelfunktionen

Im Modul `math` finden Sie unter anderem die Winkelfunktionen `sin()`, `cos()` und `tan()` und die inversen Winkelfunktionen `asin()`, `acos()` und `atan()`.

Ein Beispielprogramm:

```
import math

x = 30
xbm = math.radians(x)
print(f"Sinus {x} Grad: {math.sin(xbm)}")
print(f"Kosinus {x} Grad: {math.cos(xbm)}")
print(f"Tangens {x} Grad: {math.tan(xbm)}")

z = 0.5
print(f"Arkussinus {z} in Grad: {math.degrees(math.asin(z))}")
z = 0.866
print(f"Arkuskosinus {z} in Grad: {math.degrees(math.acos(z))}")
z = 0.577
print(f"Arkustangens {z} in Grad: {math.degrees(math.atan(z))}")
```

Listing 4.6 Datei »zahl_winkel.py«

Es wird die folgende Ausgabe erzeugt:

```
Sinus 30 Grad: 0.49999999999999994
Kosinus 30 Grad: 0.8660254037844387
Tangens 30 Grad: 0.5773502691896257
Arkussinus 0.5 in Grad: 30.000000000000004
Arkuskosinus 0.866 in Grad: 30.002910931188026
Arkustangens 0.577 in Grad: 29.984946007397852
```

Nach dem Import des Moduls `math` werden zunächst der Sinus, der Kosinus und der Tangens des Winkels 30 Grad berechnet. Alle Funktionen erwarten einen Winkel im Bogenmaß. Daher findet zuvor eine Umwandlung von Grad in Bogenmaß mithilfe der Funktion `radians()` statt.

Anschließend werden der Arkussinus, der Arkuskosinus und der Arkustangens von bestimmten Werten berechnet. Die Funktionen liefern einen Winkel im Bogenmaß. Dieser wird mithilfe der Funktion `degrees()` in Grad umgerechnet.

4.1.7 Weitere mathematische Funktionen

Ebenfalls im Modul `math` finden Sie einige Funktionen und Konstanten, die Ihnen teilweise von Ihrem Taschenrechner bekannt sind.

Ein Beispielprogramm:

```
import math

a = 4.75
print("Variable a:", a)
print("Quadratwurzel von a:", math.sqrt(a))
```

```
print("Natürlicher Logarithmus von a:", math.log(a))
print("e hoch a:", math.exp(a))
print("10er-Logarithmus von a:", math.log10(a))
print()

b = 34
print("Ganzzahlige Quadratwurzel:", math.isqrt(b))
print()

print("Kreiszahl pi:", math.pi)
print("Eulersche Zahl e:", math.e)
print()

t = 3, 2, -7
print("Produkt:", math.prod(t))
print("Fakultät von 5:", math.factorial(5))
print("Größter gem. Teiler von 60 und 135:", math.gcd(60, 135))
print("Rest:", math.remainder(10.8, 2.5))
print("Rest:", math.remainder(11.8, 2.5))
print()

for x in 2.96, 2.97:
    if math.isclose(3, x, rel_tol=0.01):
        print("Nahe dran")
    else:
        print("Nicht nahe dran")
```

Listing 4.7 Datei »zahl_rechner.py«

Es wird die folgende Ausgabe erzeugt:

```
Variable a: 4.75
Quadratwurzel von a: 2.179449471770337
Natürlicher Logarithmus von a: 1.55814461804655
e hoch a: 115.58428452718766
10er-Logarithmus von a: 0.6766936096248666

Ganzzahlige Quadratwurzel: 5

Kreiszahl pi: 3.141592653589793
Eulersche Zahl e: 2.718281828459045
```



```
Produkt: -42
Fakultät von 5: 120
Größter gem. Teiler von 60 und 135: 15
Rest: 0.8000000000000007
Rest: -0.6999999999999993

Nicht nahe dran
Nahe dran
```

Die Quadratwurzel einer positiven Zahl kann mithilfe der Funktion `sqrt()` berechnet werden, aber auch mithilfe des Exponentialoperators, siehe Abschnitt 4.1.4, »Exponentialoperator **«. Es werden die Funktionen `log()` und `log10()` zur Berechnung der Logarithmen einer positiven Zahl zur Basis e und zur Basis 10 sowie die Funktion `exp()` zur Berechnung von e^x aufgerufen.

Seit Python 3.8 können Sie mithilfe der Funktion `isqrt()` die ganzzahlige Quadratwurzel einer Zahl berechnen. Das ist der größte ganzzahlige Wert, der kleiner ist als die mathematische Quadratwurzel einer Zahl.

Zudem gibt es die mathematischen Konstanten `pi` und `e`. Solche Konstanten stehen für unveränderbare Werte. Sie werden eingesetzt, weil man sich den Namen einer Konstanten meist besser merken kann als ihren Wert.

Seit Python 3.8 lässt sich mithilfe der Funktion `prod()` das Produkt der Elemente eines Iterables ermitteln, hier eines Tupels. Mehr zum Thema »Tupel« finden Sie in Abschnitt 4.4, »Tupel«. Der Wert der Fakultät darf mathematisch und mithilfe der Funktion `factorial()` nur von positiven ganzen Zahlen berechnet werden.

Seit Python 3.5 gibt es die Funktion `gcd()`. Sie ermittelt den größten gemeinsamen Teiler (GGT, englisch: *greatest common divisor*) zweier ganzer Zahlen. Das ist die größte Zahl, durch die sich beide Zahlen ohne Rest teilen lassen.

Seit Python 3.7 lässt sich mithilfe der Funktion `remainder()` der Rest einer Division gemäß dem IEEE-754-Standard berechnen. Dabei handelt es sich um die Differenz zur nächsten ganzen Zahl. Was bedeutet das? Im vorliegenden Beispiel werden 10.8 bzw. 11.8 durch 2.5 geteilt. Das mathematische Ergebnis liegt jeweils zwischen den beiden ganzen Zahlen 4 (4 × 2.5 = 10) und 5 (5 × 2.5 = 12.5). Im Fall von 10.8 liegt der Wert 10 näher, daher liefert die Funktion `remainder()` den Wert 0.8 (= 10.8 – 10). Im Fall von 11.8 liegt der Wert 12.5 näher, daher liefert die Funktion `remainder()` den Wert –0.7 (= 11.8 – 12.5).

Seit Python 3.5 können Sie mithilfe der Funktion `isclose()` feststellen, ob zwei Zahlen einander nahe sind. In den beiden obigen Beispielen wird mithilfe der relativen Toleranz `0.01` festgestellt, ob die beiden Zahlen um maximal 1 % voneinander abweichen. Sie kön-

nen einen von zwei benannten Parametern nutzen. Neben `rel_tol` gibt es auch `abs_tol` für die Messung mit einer absoluten Toleranz. Mehr zu benannten Parametern in Abschnitt 5.6.2, »Benannte Parameter«.

4.1.8 Komplexe Zahlen

In diesem und den beiden nächsten Abschnitten, Abschnitt 4.1.9 und Abschnitt 4.1.10, werden spezielle Themen behandelt: komplexe Zahlen, Bitoperatoren und Brüche. Sie können zunächst übergangen und bei Bedarf nachgeschlagen werden.

Sie haben in Python die Möglichkeit, komplexe Zahlen zu speichern und mit ihnen zu rechnen. Die mathematischen Grundlagen zum Verständnis von komplexen Zahlen sind nicht Thema dieses Buchs. Informationen finden Sie zum Beispiel unter *https://de.wikipedia.org/wiki/Komplexe_Zahl*.

Ein Beispiel dazu:

```
a = 2.5 - 4.2j
print("a =", a)
print(f"Realteil: {a.real}, Imaginärteil: {a.imag}")
print("Betrag:", abs(a))
print("Konjugiert komplex:", a.conjugate())
print()

b = 3.7j
print("b =", b)
print(f"Realteil: {b.real}, Imaginärteil: {b.imag}")
print("Betrag:", abs(b))
print()
...
```

Listing 4.8 Datei »zahl_complex.py«, Teil 1 von 2

Es folgt die Ausgabe des ersten Programmteils:

```
a = (2.5-4.2j)
Realteil: 2.5, Imaginärteil: -4.2
Betrag: 4.887739763939975
Konjugiert komplex: (2.5+4.2j)

b = 3.7j
Realteil: 0.0 Imaginärteil: 3.7
Betrag: 3.7
```


Durch die Zuweisung einer komplexen Zahl wird `a` zu einem Objekt der Klasse `complex`. Ein solches Objekt kann Eigenschaften besitzen. Für ein solches Objekt können Methoden aufgerufen werden. Eine Methode ist eine Funktion, die nur für ein bestimmtes Objekt aufgerufen werden kann. Das trifft hier für die Methode `conjugate()` des Objekts `a` der Klasse `complex` zu. Mehr zu Klassen, Eigenschaften und Methoden erfahren Sie in Kapitel 6, »Objektorientierte Programmierung«.

Eine komplexe Zahl setzt sich aus einem Real- und einem Imaginärteil zusammen. Der Imaginärteil wird durch das Zeichen »j« (oder auch »J«) gekennzeichnet. Wird nur ein Imaginärteil zugewiesen, wird der Realteil auf 0 gesetzt. Komplexe Zahlen werden in runden Klammern ausgegeben. Ausnahme: Es wurde nur ein Imaginärteil zugewiesen.

Die Eigenschaften `real` und `imag` stellen die jeweiligen Teile der komplexen Zahl zur Verfügung. Die Funktion `abs()` liefert ihren Betrag. Die Methode `conjugate()` liefert die konjugiert komplexe Zahl, also die komplexe Zahl mit geändertem Vorzeichen.

Der zweite Teil des Programms:

```
...
print("a + b =", a + b)
print("a - b =", a - b)
print("a * b =", a * b)
print("a / b =", a / b)
print("a ** 2.5 =", a ** 2.5)
print("5.1 + a / 3.2j * 2.8 =", 5.1 + a / 3.2j * 2.8)
print()

c = 2.5 - 4.2j
print("c =", c)
print("a == c:", a == c)
print("b != c:", b != c)
print()

c = 1j
print("c =", c)
print("c * c =", c * c)
```

Listing 4.9 Datei »zahl_complex.py«, Teil 2 von 2

Es folgt die Ausgabe des zweiten Programmteils:

```
a + b = (2.5-0.5j)
a - b = (2.5-7.9j)
```

```
a * b = (15.540000000000001+9.25j)
a / b = (-1.135135135135135-0.6756756756756757j)
a ** 2.5 = (-44.83645966023058-27.915620445612213j)
5.1 + a / 3.2j * 2.8 = (1.4249999999999998-2.1875j)

c = (2.5-4.2j)
a == c: True
b != c: True

c = 1j
c * c = (-1+0j)
```

Sie können gemäß den zugehörigen mathematischen Regeln mithilfe der Operatoren +, -, *, / und ** mit komplexen Zahlen rechnen. Es können auch gemischte Ausdrücke berechnet werden, die neben den komplexen Zahlen auch reelle Zahlen enthalten. Zum Vergleich von komplexen Zahlen können nur die beiden Operatoren == und != genutzt werden. Das Quadrat der komplexen Zahl `j`, also `0 + 1j`, entspricht -1.

4.1.9 Bitoperatoren

Sämtliche Daten, ob nun Zahlen oder Zeichenketten, setzen sich auf der Hardwareebene aus Bits und Bytes zusammen. Auf dieser Ebene können Sie mit Dualzahlen (siehe auch Abschnitt 4.1.1, »Ganze Zahlen«), der Funktion `bin()` und den sogenannten Bitoperatoren arbeiten. Ein Beispiel dazu:

```
# Nur 1 Bit gesetzt
bit0 = 1          # 0000 0001
bit3 = 8          # 0000 1000
print(bin(bit0), bin(bit3))

# Bitweises AND
a = 5             # 0000 0101
erg = a & bit0    # 0000 0001
if erg:
    print(a, "ist ungerade")

# Bitweises OR
erg = 0           # 0000 0000
erg = erg | bit0  # 0000 0001
erg = erg | bit3  # 0000 1001
print("Bits nacheinander gesetzt:", erg, bin(erg))
```



```
# Bitweises Exclusive-OR
a = 21            # 0001 0101
b = 19            # 0001 0011
erg = a ^ b       # 0000 0110
print("Ungleiche Bits:", erg, bin(erg))

# Bitweise Inversion, aus x wird -(x+1)
a = 11            # 0000 1011
erg = ~a          # 1111 0100
print("Bitweise Inversion:", erg, bin(erg))

# Bitweise schieben
a = 11            # 0000 1011
erg = a >> 1      # 0000 0101
print("Um 1 nach rechts geschoben:", erg, bin(erg))
erg = a << 2      # 0010 1100
print("Um 2 nach links geschoben:", erg, bin(erg))
```

Listing 4.10 Datei »operator_bit.py«

Es folgt die Ausgabe:

```
0b1 0b1000
5 ist ungerade
Bits nacheinander gesetzt: 9 0b1001
Ungleiche Bits: 6 0b110
Bitweise Inversion: -12 -0b1100
Um 1 nach rechts geschoben: 5 0b101
Um 2 nach links geschoben: 44 0b101100
```

Zunächst werden die beiden Variablen `bit0` und `bit3` eingeführt, die bei einigen der nachfolgenden Berechnungen benötigt werden. Sie haben die Werte 1 und 8. Am Ende der Programmzeile sehen Sie sie als Dualzahl, also mithilfe von 8 Bit (= 1 Byte) notiert. Das letzte Bit eines Bytes wird Bit 0 genannt, das vorletzte Bit ist Bit 1 usw. Die Werte der beiden Variablen `bit0` und `bit3` sind so gewählt, dass jeweils nur ein Bit gesetzt ist (= 1), die restlichen Bits sind nicht gesetzt (= 0).

Sie können sich auch eine Reihe von acht Leuchtdioden vorstellen, die entweder *an* oder *aus* sind. Diese Information kann innerhalb eines Bytes gespeichert werden. Ist eines seiner Bits gesetzt, ist die betreffende LED *an*, ansonsten *aus*. Zur Verdeutlichung wer-

den die beiden Variablen `bit0` und `bit3` mithilfe der Funktion `bin()` als Dualzahl ausgegeben.

Sie können den Bitoperator `&` zur bitweisen Und-Verknüpfung zweier Zahlen nutzen. Ähnlich wie beim logischen Operator `and` (siehe Abschnitt 3.3.6, »Logische Operatoren«) wird ein bestimmtes Bit im Ergebnis nur gesetzt, wenn dieses Bit in beiden Zahlen gesetzt ist. Diese Operation wird für jedes einzelne Bit durchgeführt.

Möchten Sie wissen, ob ein bestimmtes Bit innerhalb einer Zahl gesetzt ist, verknüpfen Sie diese Zahl mithilfe des Bitoperators `&` mit einer anderen Zahl, in der nur dieses eine gesuchte Bit gesetzt ist. Handelt es sich um das Bit 0, wissen Sie darüber hinaus, ob die Zahl gerade (Bit 0 = 0) oder ungerade (Bit 0 = 1) ist.

Der Bitoperator | dient zur bitweisen Oder-Verknüpfung zweier Zahlen. Das Zeichen für diesen Operator erreichen Sie mithilfe der Taste Alt, die sich rechts neben dem Leerzeichen befindet. Ähnlich wie beim logischen Operator `or` (siehe ebenfalls Abschnitt 3.3.6) wird ein bestimmtes Bit im Ergebnis gesetzt, wenn dieses Bit in einer der beiden Zahlen oder in beiden Zahlen gesetzt ist. Diese Operation wird auch für jedes einzelne Bit durchgeführt.

Möchten Sie einzelne Bits einer Zahl setzen, verknüpfen Sie diese Zahl mithilfe des Bitoperators | mit einer anderen Zahl, in der nur dieses eine gesuchte Bit gesetzt ist.

Der Bitoperator ^ dient zur bitweisen Exklusiv-Oder-Verknüpfung zweier Zahlen. Ein bestimmtes Bit im Ergebnis wird gesetzt, wenn dieses Bit nur in einer der beiden Zahlen gesetzt ist. Ist das Bit in beiden Zahlen gesetzt, wird das Ergebnis-Bit nicht gesetzt. Diese Operation wird ebenfalls für jedes einzelne Bit durchgeführt.

Sie können den Bitoperator ~ zur bitweisen Inversion einer Zahl nutzen. Dabei wird aus der Zahl `x` die Zahl `-(x+1)`, aus 11 wird also -12.

Die beiden Bitoperatoren >> und << dienen zum Schieben von Bits innerhalb einer Zahl:

- Mithilfe von >> werden alle Bits um eine bestimmte Anzahl von Stellen nach rechts geschoben. Die Bits, die dabei nach rechts »hinausfallen«, sind verloren. Eine Verschiebung um n Bit nach rechts entspricht einer ganzzahligen Division durch 2^n. Eine Verschiebung um 1 Bit nach rechts entspricht also einer ganzzahligen Division durch 2.
- Mithilfe von << werden alle Bits um eine bestimmte Anzahl von Stellen nach links geschoben. Eine Verschiebung um n Bit nach links entspricht einer Multiplikation mit 2^n. Eine Verschiebung um 1 Bit nach links entspricht also einer Multiplikation mit 2.

4.1.10 Brüche

Python kann auch mit Brüchen rechnen bzw. Informationen über Brüche zur Verfügung stellen. Dazu wird das Modul `fractions` (deutsch: Brüche) genutzt. Ein Beispiel:

```
import fractions

z = 12
n = 28
print(f"Bruch: {z}/{n}")

b1 = fractions.Fraction(z, n)
print("Fraction-Objekt:", b1)
print(f"Zähler: {b1.numerator}, Nenner: {b1.denominator}")
print("Wert:", b1.numerator / b1.denominator)
print()

x = 2.375
print("Zahl:", x)
b2 = fractions.Fraction(x)
print("Fraction-Objekt:", b2)
```

Listing 4.11 Datei »zahl_bruch.py«

Die Ausgabe lautet:

```
Bruch: 12/28
Fraction-Objekt: 3/7
Zähler: 3, Nenner: 7
Wert: 0.42857142857142855

Zahl: 2.375
Fraction-Objekt: 19/8
```

Zunächst wird ein Beispielbruch in der bekannten Form dargestellt. Er wird aus zwei ganzen Zahlen gebildet, dem Zähler und dem Nenner.

Die Funktion `Fraction()` aus dem Modul `fractions` bietet verschiedene Möglichkeiten, einen Bruch zu erzeugen. Genauer gesagt, handelt es sich bei `Fraction()` um den Konstruktor der Klasse `Fraction`. Damit wird eine Instanz (ein Objekt) der Klasse erzeugt und eine Referenz auf dieses Objekt zurückgeliefert. Klassen, Instanzen, Konstruktoren und andere Begriffe aus der objektorientierten Programmierung werden in Kapitel 6, »Objektorientierte Programmierung«, genauer erläutert.

Das `Fraction`-Objekt, also der Bruch `b1`, der aus 12 und 28 gebildet wird, wird bei der Erzeugung automatisch auf 3/7 gekürzt.

Zähler und Nenner des Bruchs stehen in den Eigenschaften `numerator` und `denominator` einzeln zur Verfügung. Der Wert eines Bruchs lässt sich darüber berechnen: 3/7 = 0,428...

Umgekehrt können Sie auch eine Zahl mit Nachkommastellen in einen Bruch umrechnen. Dazu übergeben Sie die Zahl der Konstruktormethode `Fraction()`: Aus 2.375 wird 19/8.

Im nachfolgenden Programm wird ein Bruch dazu genutzt, eine Zahl mit Nachkommastellen zu approximieren, also anzunähern. Dazu dient die Methode `limit_denominator()`. Das Programm:

```
import fractions

x = 1.84953
print("Zahl:", x)

b3 = fractions.Fraction(x)
print("Fraction-Objekt:", b3)

b4 = b3.limit_denominator(100)
print("Approximiert auf Nenner max. 100:", b4)

wert = b4.numerator / b4.denominator
print("Wert:", wert)
print("rel. Fehler:", abs((x-wert)/x))
```

Listing 4.12 Datei »zahl_bruch_naehern.py«

Die Ausgabe:

```
Zahl: 1.84953
Fraction-Objekt: 8329542618810553/4503599627370496
Approximiert auf Nenner max. 100: 172/93
Wert: 1.8494623655913978
rel. Fehler: 3.656843014286614e-05
```

Es wird die Zahl 1,84953 untersucht. Sie entspricht dem Bruch 184953/100000. Mithilfe der Methode `limit_denominator()` wird der Nenner auf die Zahl 100 begrenzt. Dann wird der Bruch gesucht, der

- einen Nenner mit dem maximalen Wert 100 hat und
- der Zahl 1,84953 am nächsten kommt.

Im vorliegenden Fall ist das der Bruch 172/93. Er hat den Wert 1,8494623655913978 und kommt der ursprünglichen Zahl recht nahe. Der relative Fehler zwischen diesem Wert und der untersuchten Zahl beträgt nur $3{,}65684301429 \times 10^{-5}$.

Er wird mithilfe der eingebauten Funktion `abs()` zur Berechnung des Betrags ermittelt. Beim Betrag handelt es sich um den Absolutwert einer Zahl, also der Zahl ohne das Vorzeichen.

Sollten Sie die Methode `gcd()` zur Ermittlung des größten gemeinsamen Teilers vermissen: Seit Python 3.5 gehört sie als Funktion zum Modul `math` (siehe Abschnitt 4.1.7, »Weitere mathematische Funktionen«) und wird im Modul `fractions` als veraltet bezeichnet.

4.2 Zeichenketten

Zeichenketten sind Sequenzen von einzelnen Zeichen. Auch andere Datentypen gehören zu den Sequenzen. Anhand von Zeichenketten folgt eine Einführung in die Sequenzen.

4.2.1 Eigenschaften

Zeichenketten (Strings) sind Objekte des Datentyps `str`. Sie werden in einfache, doppelte oder dreimal doppelte Hochkommata gesetzt. Mithilfe einer `for`-Schleife können Sie auf die Elemente einer Zeichenkette zugreifen.

Ein Beispiel:

```
tx = "Das"
print("Text:", tx)
print("Typ:", type(tx))
print("Anzahl der Zeichen:", len(tx))
for z in tx:
    print(z)
for i in range(len(tx)):
    print(f"{i}: {tx[i]}")

tx = 'Auch das ist eine Zeichenkette'
print(tx)
```

```
tx = """Diese Zeichenkette
        steht in
        mehreren Zeilen"""
print(tx)

tx = 'Hier sind "doppelte Hochkommata" gespeichert'
print(tx)
```

Listing 4.13 Datei »text_eigenschaft.py«

Es wird die folgende Ausgabe erzeugt:

```
Text: Das
Typ: <class 'str'>
Anzahl der Zeichen: 3
D
a
s
0: D
1: a
2: s
Auch das ist eine Zeichenkette
Diese Zeichenkette
    steht in
    mehreren Zeilen
Hier sind "doppelte Hochkommata" gespeichert
```

Für die erste Zeichenkette wird mithilfe der Funktion `type()` der Datentyp und mithilfe der Funktion `len()` die Anzahl der Elemente ausgegeben, also die Länge der Zeichenkette.

Eine Zeichenkette ist ein Iterable, das aus einzelnen Elementen besteht. Mithilfe einer `for`-Schleife kann auf die Elemente zugegriffen werden. Soll zusätzlich der Index, also die laufende Nummer jedes Elements angegeben werden, benötigen Sie eine Schleifenvariable, die die einzelnen Indizes durchläuft.

Die zweite Zeichenkette wird in einfachen Hochkommata angegeben. Die dritte Zeichenkette wird in dreifachen doppelten Hochkommata angegeben, erstreckt sich über mehrere Zeilen und wird auch so ausgegeben.

Die letzte Zeichenkette verdeutlicht den Vorteil, den das Vorhandensein mehrerer Alternativen bietet: Die doppelten Hochkommata sind hier Bestandteil des Texts und werden auch ausgegeben.

4.2.2 Operatoren

Der Operator `+` dient zur Verkettung mehrerer Sequenzen, der Operator `*` zur Vervielfachung einer Sequenz. Mithilfe des Operators `in` stellen Sie fest, ob ein bestimmtes Element in einer Sequenz enthalten ist. Betrachten Sie das folgende Beispiel für diese Operatoren, angewendet für Strings:

```
kreis = "-oooo-"
stern = "***"
linie = stern + kreis * 3 + stern
print(linie)

tx = "Hallo"
print("Text:", tx)
if "a" in tx:
    print("a ist enthalten")
if "z" not in tx:
    print("z ist nicht enthalten")
```

Listing 4.14 Datei »text_operator.py«

Die Ausgabe lautet:

```
***-oooo--oooo--oooo-***
Text: Hallo
a ist enthalten
z ist nicht enthalten
```

Die Zeichenkette `linie` wird mithilfe des Verkettungsoperators `+` und des Vervielfachungsoperators `*` zusammengesetzt. Der Text `"-oooo-"` wird dabei dreimal hintereinander in `tlinie` gespeichert.

Mithilfe des Operators `in` wird festgestellt, ob das Element `a` in der Sequenz enthalten ist. Der logische Operator `not` dient (zusammen mit `in`) der Feststellung, ob das Element `z` nicht enthalten ist.

4.2.3 Slices

Bereiche von Sequenzen werden als *Slices* bezeichnet. Der Einsatz von Slices wird am Beispiel eines Strings verdeutlicht. Auf die gleiche Art und Weise sind Slices auch auf andere Sequenzen anwendbar.

Als Beispiel für eine Sequenz wird die Zeichenkette `Hallo` gespeichert. Tabelle 4.1 stellt die einzelnen Elemente mit dem zugehörigen Index dar. Der Index beginnt bei 0; alternativ können Sie auch einen negativen Index nutzen, der mit -1 endet (siehe unterste Zeile der Tabelle).

Index	0	1	2	3	4
Element	H	a	l	l	o
Negativer Index	-5	-4	-3	-2	-1

Tabelle 4.1 Sequenz mit Index

Ein Slice wird durch die Angabe eines Bereichs in rechteckigen Klammern (`[]`) erzeugt. Diese erreichen Sie mithilfe der Taste `Alt`, die sich rechts neben dem Leerzeichen befindet.

Ein Slice beginnt mit einem Start-Index, gefolgt von einem Doppelpunkt und einem End-Index. Ein Slice, der nur aus einem einzelnen Element besteht, wird durch die Angabe eines einzelnen Index erzeugt.

```
tx = "Hallo"
print("Text:", tx)

print("[1:4]:", tx[1:4])
print("[:4]:", tx[:4])
print("[2:]:", tx[2:])
print("[:]:", tx[:])
print("[1]:", tx[1])
print("[1:-2]:", tx[1:-2])
```

Listing 4.15 Datei »text_slice.py«

Es wird die folgende Ausgabe erzeugt:

```
Text: Hallo
[1:4]: all
[:4]: Hall
[2:]: llo
[:]: Hallo
[1]: a
[1:-2]: al
```


Die Erläuterung der verschiedenen Slices:

- Slice `[1:4]`: Der Bereich erstreckt sich von dem Element, das durch den Start-Index gekennzeichnet wird, bis vor das Element, das durch den End-Index gekennzeichnet wird.
- Slice `[:4]`: Wird der Start-Index weggelassen, beginnt der Bereich bei `0`.
- Slice `[2:]`: Wird der End-Index weggelassen, endet der Bereich am Ende der Sequenz.
- Slice `[:]`: Werden beide Indizes weggelassen, erstreckt sich der Bereich über die gesamte Sequenz. Eine solche Angabe wird zur Zerlegung von mehrdimensionalen Sequenzen benötigt.
- Slice `[1]`: Wird nur ein Index angegeben, besteht der Bereich nur aus einem einzelnen Element.
- Slice `[1:-2]`: Wird ein Index mit einer negativen Zahl angegeben, wird für diesen Index vom Ende aus gemessen, beginnend bei -1.

Ein Slice einer Sequenz hat denselben Datentyp wie die Sequenz: Ein Slice einer Zeichenkette ist eine Zeichenkette, ein Slice einer Liste ist eine Liste.

4.2.4 Änderbarkeit

Anhand des nachfolgenden Beispiels sehen Sie, dass eine Zeichenkette nicht veränderbar ist. Es können keine Zeichen oder Slices durch andere Zeichen oder Slices ersetzt werden:

```
tx = "Das ist ein Text"
print(tx)

try:
    tx[4:6] = "war"
except:
    print("Fehler")

tx = tx[:4] + "war" + tx[7:]
print(tx)
```

Listing 4.16 Datei »text_aenderbar.py«

Die Ausgabe lautet:

```
Das ist ein Text
Fehler
Das war ein Text
```

Die einzige Möglichkeit zur Änderung einer Variablen, die eine Zeichenkette enthält, ist die Zuweisung einer neuen Zeichenkette in derselben Variablen. Diese kann aus Teilen der alten Zeichenkette zusammengesetzt werden.

4.2.5 Suchen und Ersetzen

Anhand des folgenden Beispiels werden einige Methoden zum Suchen und Ersetzen in Texten verdeutlicht:

```
tx = "Das ist ein Beispielsatz"
print("Text:", tx)

such = "ei"
print("Suchtext:", such)
print()

anz = tx.count(such)
print(f"count: Der String {such} kommt {anz} mal vor")

pos = tx.find(such)
while pos != -1:
    print("An Position", pos)
    pos = tx.find(such, pos+1)

pos = tx.rfind(such)
if pos != -1:
    print("rfind: Zum letzten Mal an Position", pos)
print()

if tx.startswith("Das"):
    print("Text beginnt mit Das")
if not tx.endswith("Das"):
    print("Text endet nicht mit Das")

tx = tx.replace("ist", "war")
print("Nach Ersetzen:", tx)

z = 48.2
tx = str(z)
```



```
tx = tx.replace(".", ",")
print("Zahl mit Komma:", tx)
```

Listing 4.17 Datei »text_suchen.py«

Folgende Ausgabe wird erzeugt:

```
Text: Das ist ein Beispielsatz
Suchtext: ei

count: Der String ei kommt 2 mal vor
An Position 8
An Position 13
rfind: Zum letzten Mal an Position 13

Text beginnt mit Das
Text endet nicht mit Das
Nach Ersetzen: Das war ein Beispielsatz
Zahl mit Komma: 48,2
```

Die Methode `count()` ergibt die Anzahl der Vorkommen eines Suchtexts innerhalb des analysierten Texts.

Die Methode `find()` liefert die Position, an der ein Suchtext innerhalb eines analysierten Texts vorkommt. Kommt der gesuchte Text nicht vor, wird -1 geliefert. Sie können optional einen zweiten Parameter angeben, der die Position bestimmt, ab der gesucht werden soll. Das können Sie nutzen, um mithilfe einer Schleife alle Vorkommen eines Suchtexts zu finden.

Die Methode `rfind()` ergibt die Position des letzten Vorkommens des Suchtexts innerhalb des analysierten Texts.

Mithilfe der Methoden `startswith()` und `endswith()` untersuchen Sie, ob eine Zeichenkette mit einem bestimmten Text beginnt oder endet. Beide Methoden liefern einen Wahrheitswert, daher kann der Rückgabewert zur Steuerung der Verzweigung genutzt werden.

Die Methode `replace()` ersetzt einen gesuchten Teiltext durch einen anderen und liefert den geänderten Text zurück.

Die eingebaute Funktion `str()` erstellt eine Zeichenkette aus einem Objekt. Das können Sie zum Beispiel nutzen, um eine Zahl mit einem Dezimalkomma auszugeben, indem Sie die Zahl zunächst in eine Zeichenkette umwandeln und anschließend den Dezimalpunkt mithilfe der Methode `replace()` durch ein Komma ersetzen.

4.2.6 Leerzeichen entfernen

Nach dem Einlesen von Texten aus einer Datei können sich Leerzeichen, Tabulatorzeichen (\t) und Zeilenende-Zeichen (\n) am Anfang oder am Ende eines Texts befinden. Sie können mithilfe der Methode `strip()` entfernt werden. Ein Beispiel:

```
tx = "   \tHallo Welt\n\t   "
print(f"|{tx}|")
print(f"|{tx.strip()}|")
```

Listing 4.18 Datei »text_leerzeichen.py«

Zur Verdeutlichung wird zusätzlich das Zeichen | vor und nach dem Text ausgegeben. Die Ausgabe des Programms sehen Sie in Abbildung 4.1.

```
|       Hallo Welt
          |
|Hallo Welt|
```

Abbildung 4.1 Löschen von Zeichen am Anfang und am Ende des Textes

4.2.7 Text zerlegen

Die Methode `split()` dient zum Zerlegen von Texten in Teiltexte. Das nachfolgende Beispiel zeigt zwei Anwendungen:

```
tx = "Das ist ein Satz"
print("Text:", tx)
wortliste = tx.split()
for i in range(0, len(wortliste)):
    print("Element:", i, wortliste[i])
print()

tx = "Maier;Hans;6714;3.500,00;15.03.62"
print("Text:", tx)
wortliste = tx.split(";")
for i in range(0, len(wortliste)):
    print("Element:", i, wortliste[i])
```

Listing 4.19 Datei »text_zerlegen.py«

Die Ausgabe lautet:

```
Text: Das ist ein Satz
Element: 0 Das
Element: 1 ist
Element: 2 ein
Element: 3 Satz

Text: Maier;Hans;6714;3.500,00;15.03.62
Element: 0 Maier
Element: 1 Hans
Element: 2 6714
Element: 3 3.500,00
Element: 4 15.03.62
```


Die Methode `split()` zerlegt einen Text in einzelne Teile, die in einer Liste gespeichert werden. Standardmäßig wird das Leerzeichen als Trennzeichen angesehen. Die eingebaute Funktion `len()` liefert auch für eine Liste die Anzahl der Elemente. Mehr Informationen zu Listen gibt es in Abschnitt 4.3, »Listen«.

Die einzelnen Teile eines Datensatzes werden häufig mit einem Semikolon voneinander getrennt. Sie können es als Parameter der Methode `split()` verwenden, um den Datensatz aufzuteilen.

4.2.8 Konstanten

Das Modul `string` stellt einige nützliche Zeichenketten-Konstanten bereit, zum Beispiel alle Buchstaben, alle Ziffern oder alle Interpunktionszeichen, wie Sie in folgendem Programm sehen:

```
import string
print("Klein:", string.ascii_lowercase)
print("Groß:", string.ascii_uppercase)
print("Buchstaben:", string.ascii_letters)
print("Ziffern:", string.digits)
print("Interpunktionszeichen:", string.punctuation)
```

Listing 4.20 Datei »text_konstanten.py«

Diese Konstanten können zum Beispiel für eine zufällige Auswahl von Zeichen für ein Passwort dienen, siehe Abschnitt 5.4, »Verschlüsselung«.

Die Ausgabe sieht wie folgt aus:

```
Klein: abcdefghijklmnopqrstuvwxyz
Groß: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Buchstaben: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Ziffern: 0123456789
Interpunktionszeichen: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~

4.2.9 Datentyp »bytes«

Beim Datentyp »bytes« handelt es sich um ein spezielles Thema. Es kann zunächst übergangen und bei Bedarf nachgeschlagen werden.

Die bisher behandelten Zeichenketten, also Objekte des Datentyps `str`, werden aus dem umfangreichen Zeichensatz der Unicode-Zeichen gebildet.

Objekte des seltener genutzten Datentyps `bytes` entsprechen Zeichen, deren Zeichencode nur im Bereich von 0 bis 255 liegt. Jedes Zeichen kann mithilfe eines Bytes gespeichert werden. Sie können `bytes`-Objekte mithilfe von Byte-Literalen erzeugen oder mithilfe der eingebauten Funktion `bytes()`.

Byte-Literale beginnen mit einem »b« oder einem »B«. Dies ist bei der Eingabe oder Zuweisung zu beachten. Bei der Ausgabe wird ein `b` vorangestellt. Es folgt ein Beispielprogramm:

```
st = "Hallo"
print(st, type(st))

by = b'Hallo'
print(by, type(by))

by = bytes("Hallo", "UTF-8")
print(by, type(by))

by = b'Hallo'
st = by.decode()
print(st, type(st))
```

Listing 4.21 Datei »bytes.py«

Das Programm erzeugt die folgende Ausgabe:

```
Hallo <class 'str'>
b'Hallo' <class 'bytes'>
b'Hallo' <class 'bytes'>
Hallo <class 'str'>
```


Zunächst werden ein `str`-Objekt und ein `bytes`-Objekt per Zuweisung erzeugt und ausgegeben. Beachten Sie beim Byte-Literal das vorangestellte `b`. Zur Umwandlung eines `str`-Objekts in ein `bytes`-Objekt wird die eingebaute Funktion `bytes()` genutzt. Dabei wird die Codierung des `str`-Objekts angegeben, hier UTF-8. Zur Umwandlung eines `bytes`-Objekts in ein `str`-Objekt wird die Methode `decode()` verwendet.

4.3 Listen

Eine Liste ist eine Sequenz von Objekten in ihrer allgemeinsten Form. Sie kann Elemente unterschiedlicher Datentypen enthalten. Eine Liste bietet vielfältige Möglichkeiten, unter anderem die Funktionalität von ein- und mehrdimensionalen Feldern (Arrays), wie man sie aus anderen Programmiersprachen kennt.

4.3.1 Eigenschaften und Operatoren

Eine Liste ist im Gegensatz zu einem String veränderbar. Davon abgesehen ist ein String, vereinfacht gesagt, nur eine Liste zur Speicherung von einzelnen Zeichen. Einige Beispiele für Listen:

```
import random

z = [3, 6.2, -12]
print("Liste:", z)
print("Element:", z[0])
print("Slice", z[:2])
print()

print("Schleife:")
for element in z:
      print(element)
print()

print("Schleife mit Index:")
for i in range(len(z)):
      print(f"{i}: {z[i]}")
print()

a = ["Paris", "Lyon"]
b = ["Rom", "Pisa"]
```

```
c = a + b * 2
print("Addiert und vervielfacht:", c)

print("Zufälliges Element:", random.choice(c))
random.shuffle(c)
print("Nach dem Mischen:", c)
```

Listing 4.22 Datei »liste_eigenschaft.py«

Eine mögliche Ausgabe:

```
Liste: [3, 6.2, -12]
Element: 3
Slice [3, 6.2]

Schleife:
3
6.2
-12

Schleife mit Index:
0: 3
1: 6.2
2: -12

Addiert und vervielfacht: ['Paris', 'Lyon', 'Rom', 'Pisa', 'Rom', 'Pisa']
Zufälliges Element: Rom
Nach dem Mischen: ['Pisa', 'Pisa', 'Rom', 'Paris', 'Rom', 'Lyon']
```

Eine Liste wird innerhalb von rechteckigen Klammern angegeben. Darin werden die einzelnen Elemente durch Kommata getrennt notiert.

Die Variable `z` enthält eine Liste von Zahlen. Wie bei Strings ermitteln Sie ein einzelnes Element einer Liste durch Angabe eines Index. Einen Bereich erhalten Sie mithilfe eines Slice.

Mithilfe einer `for`-Schleife können alle Elemente einer Liste ausgegeben werden, mit oder ohne Index. Die Länge einer Sequenz, also auch einer Liste, ermitteln Sie mit der Funktion `len()`.

Listen können mithilfe des Operators + verkettet und mithilfe des Operators * vervielfacht werden.

Die Funktion `choice()` aus dem Modul `random` liefert ein zufälliges Element aus der Liste. Die Funktion `shuffle()` aus dem Modul `random` mischt die Elemente der Liste mithilfe eines Zufallsgenerators.

4.3.2 Mehrdimensionale Listen

Eine Liste kann Elemente unterschiedlicher Datentypen enthalten. Diese Elemente können wiederum Listen sein. Das kann man zur Erstellung einer mehrdimensionalen Liste nutzen:

```
z = [["Paris","Fr",3.5], ["Rom","It",4.2], ["Madrid","Sp",3.2]]
print("Liste:", z)
print("Länge:", len(z))
print()

print("Unter-Liste:", z[0])
print("Länge:", len(z[0]))
print("Slice von Unter-Listen:", z[:2])
print()

print("Element:", z[2][0])
print("Länge:", len(z[2][0]))
print("Slice von Elementen:", z[2][0][:3])
print()

for i in range(len(z)):
    print(f"{i}: {z[i][0]} hat {z[i][2]} Mio. Einwohner")
for stadt in z:
    print(f"{stadt[0]} hat {stadt[2]} Mio. Einwohner")
```

Listing 4.23 Datei »liste_mehrdimensional.py«

Die Ausgabe lautet:

```
Liste: [['Paris', 'Fr', 3.5], ['Rom', 'It', 4.2], ['Madrid', 'Sp', 3.2]]
Länge: 3

Unter-Liste: ['Paris', 'Fr', 3.5]
Länge: 3
Slice von Unter-Listen: [['Paris', 'Fr', 3.5], ['Rom', 'It', 4.2]]
```

```
Element: Madrid
Länge: 6
Slice von Elementen: Mad

0: Paris hat 3.5 Mio. Einwohner
1: Rom hat 4.2 Mio. Einwohner
2: Madrid hat 3.2 Mio. Einwohner
Paris hat 3.5 Mio. Einwohner
Rom hat 4.2 Mio. Einwohner
Madrid hat 3.2 Mio. Einwohner
```

Die mehrdimensionale Liste `z` besitzt drei Elemente, und zwar drei Unter-Listen. Diese bestehen jeweils aus zwei Zeichenketten und einer Zahl. Auf eine oder mehrere Unter-Listen greifen Sie mithilfe eines Index oder Slice zu.

Die Länge einer Unter-Liste wird mithilfe der Funktion `len()`, bezogen auf die Unter-Liste, ermittelt. Einzelne Elemente einer Unter-Liste erreichen Sie durch Angabe mehrerer Indizes. Der erste Index steht für die Unter-Liste, der zweite für das Element innerhalb der Unter-Liste.

Handelt es sich bei einem Element wiederum um eine Sequenz, können Sie:

- mithilfe eines Index oder eines Slice auf einzelne Unter-Elemente der Sequenz zugreifen
- die Länge der Sequenz mithilfe der Funktion `len()` ermitteln

Mithilfe von `for`-Schleifen können Sie die mehrdimensionale Liste durchlaufen.

4.3.3 Änderbarkeit

Listen können, im Gegensatz zu Strings, verändert werden. Sie können also ein Element oder einen Slice hinzufügen, ändern oder löschen. Ersetzen Sie ein einzelnes Element durch einen Slice, wird eine neue Unter-Liste erzeugt. Es folgt ein Beispiel:

```
z = ["Paris","Lyon","Reims","Nice"]
print("Liste:", z)

z[2] = "Lens"
print("Element ersetzt:", z)

z[1:3] = ["Nancy","Metz","Gap"]
print("Slice durch Liste ersetzt:", z)
```



```
del z[2:]
print("Slice entfernt:", z)

z[0] = ["Paris-Nord","Paris-Sud"]
print("Element durch Liste ersetzt:", z)
```

Listing 4.24 Datei »liste_aendern.py«

Die Ausgabe der Liste in den verschiedenen Zuständen:

```
Liste: ['Paris', 'Lyon', 'Reims', 'Nice']
Element ersetzt: ['Paris', 'Lyon', 'Lens', 'Nice']
Slice durch Liste ersetzt: ['Paris', 'Nancy', 'Metz', 'Gap', 'Nice']
Slice entfernt: ['Paris', 'Nancy']
Element durch Liste ersetzt: [['Paris-Nord', 'Paris-Sud'], 'Nancy']
```

Die Liste besteht zunächst aus vier Zeichenketten und wird wie folgt geändert:

- Ein einzelnes Element wird durch eine andere Zeichenkette ersetzt.
- Ein Slice wird durch eine Liste ersetzt, die eine andere Länge besitzt.
- Ein Slice wird mithilfe des Schlüsselworts `del` aus der Liste entfernt.
- Ein einzelnes Element wird durch eine Liste ersetzt; dadurch wird eine neue Unter-Liste erzeugt.

4.3.4 Methoden

Im nachfolgenden Programm wird eine Reihe von weiteren Methoden zum Ändern und Untersuchen von Listen verdeutlicht:

```
z = ["Paris","Lyon","Metz"]
print("Liste:", z)

z.insert(1,"Nantes")
print("Einsetzen:", z)

z.sort()
print("Sortieren:", z)

z.reverse()
print("Umdrehen:", z)
```

```
such = "Nantes"
if such in z:
    z.remove(such)
    print("Entfernen:", z)

z.append("Paris")
print("Am Ende hinzu:", z)
print()

such = "Paris"
print("Anzahl gefunden:", z.count(such))

startpos = 0
while such in z[startpos:]:
    pos = z.index("Paris", startpos)
    print("Position:", pos)
    startpos = pos + 1
```

Listing 4.25 Datei »liste_methoden.py«

Die Ausgabe der Liste in den verschiedenen Zuständen:

```
Liste: ['Paris', 'Lyon', 'Metz']
Einsetzen: ['Paris', 'Nantes', 'Lyon', 'Metz']
Sortieren: ['Lyon', 'Metz', 'Nantes', 'Paris']
Umdrehen: ['Paris', 'Nantes', 'Metz', 'Lyon']
Entfernen: ['Paris', 'Metz', 'Lyon']
Am Ende hinzu: ['Paris', 'Metz', 'Lyon', 'Paris']

Anzahl gefunden: 2
Position: 0
Position: 3
```

Die Originalliste, bestehend aus drei Zeichenketten, wird erstellt. Danach wird ein neues Element mit der Methode `insert()` an Position 1 eingefügt.

Die Liste wird mit der Methode `sort()` sortiert. Handelt es sich um eine Liste von Zeichenketten, wird alphabetisch sortiert. Eine Liste von Zahlen wird nach Größe sortiert. Bei anderen Typen von Elementen oder bei gemischten Listen ist der Einsatz der Methode `sort()` nur bedingt sinnvoll.

Die Liste wird mithilfe der Methode `reverse()` intern umgedreht.

Ein bestimmtes Element wird in der Liste gesucht. Ist es mindestens einmal vorhanden, wird das erste Vorkommen mit `remove()` gelöscht. Wäre es nicht vorhanden, würde eine Ausnahme ausgelöst werden.

Ein Element wird am Ende der Liste mithilfe der Methode `append()` angefügt.

Die Anzahl der Vorkommen eines bestimmten Elements wird mithilfe der Methode `count()` ermittelt.

Die Position des Vorkommens eines bestimmten Elements kann mithilfe der Methode `index()` bestimmt werden. Mithilfe eines zweiten Parameters kann festgelegt werden, ab welcher Position die Suche beginnen soll. Kommt das Element nicht vor, wird eine Ausnahme ausgelöst.

Zur Bestimmung aller Positionen eines bestimmten Elements wird eine `while`-Schleife genutzt. Ihre Bedingung lautet: »Solange das Element in der (restlichen) Liste vorkommt«. Beim ersten Mal wird mit dieser Bedingung die gesamte Liste geprüft. Danach wird nur noch derjenige Teil der Liste geprüft, der nach der letzten gefundenen Position liegt.

4.3.5 List Comprehension

Die Technik der *List Comprehension* ermöglicht die schnelle Erzeugung einer Liste aus einer anderen Liste. Dabei können Sie die Elemente der ersten Liste filtern und verändern. Es folgen drei Beispiele:

```
x = [3, -6, -8, 9, 15]
print(x)
y = [2, 13, 4, -8, 4]
print(y)
print()

a = [z+1 for z in x]
print(a)

b = [z+1 for z in x if z>0]
print(b)

c = [x[i]+y[i] for i in range(len(x))]
print(c)
```

```
d = [x[i]+y[i] for i in range(len(x)) if x[i]>0 and y[i]>0]
print(d)
```

Listing 4.26 Datei »liste_comprehension.py«

Die Ausgabe lautet:

```
[3, -6, -8, 9, 15]
[2, 13, 4, -8, 4]

[4, -5, -7, 10, 16]
[4, 10, 16]
[5, 7, -4, 1, 19]
[5, 19]
```

Zunächst werden die beiden Beispiellisten `x` und `y` erstellt und ausgegeben.

Die Liste `a` entspricht der Liste `x`. Der Wert jedes Elements wird allerdings um 1 erhöht. Der Ausdruck `z+1 for z in x` bedeutet: »Übernehme die Elemente der Liste und addiere 1«.

Bei der Liste `b` verhält es sich ähnlich. Allerdings werden nur die Elemente mit positiven Werten herausgefiltert. Der Ausdruck `z+1 for z in x if z>0` bedeutet: »Übernehme die Elemente der Liste und addiere 1, aber nur falls sie größer als 0 sind«.

Für die Liste `c` wird auch der Index einbezogen. Die Elemente der beiden Listen werden paarweise addiert. Der Ausdruck `x[i]+y[i] for i in range(len(x))` bedeutet: »Addiere die i-ten Elemente der beiden Listen über die Länge der Liste x«. Die Liste `y` könnte also auch größer sein.

Bei der Liste `d` verhält es sich wiederum ähnlich. Allerdings werden nur die Paare mit zwei positiven Elementen herausgefiltert. Der Ausdruck `x[i]+y[i] for i in range(len(x)) if x[i]>0 and y[i]>0` bedeutet: »Addiere die i-ten Elemente der beiden Listen über die Länge der Liste x, aber nur falls beide größer als 0 sind«.

4.4 Tupel

Ein Tupel entspricht einer Liste, deren Elemente nicht verändert werden dürfen. Ansonsten gelten die gleichen Regeln, und es können die gleichen Operationen und Methoden auf Tupel wie auf Listen angewendet werden, sofern sie keine Veränderung des Tupels hervorrufen.

Nachfolgend werden einige Besonderheiten von Tupeln verdeutlicht:

```
z = (3, 6, -8)
print("Tupel 1 verpackt:", z)
z = 3, 6, -8
print("Tupel 2 verpackt:", z)

for i in 3, 6, -8:
    print(i)

a, b, c = z
print("Tupel entpackt:", a, b, c)

a, b, c = 3, 6, -8
print("Mehrfache Zuweisung:", a, b, c)

a, b, c = c, a, a+b
print("Auswirkung später:", a, b, c)

a, *b, c = 3, 6, 12, -28, -8
print("Rest in Liste:", a, b, c)
```

Listing 4.27 Datei »tupel.py«

Die Ausgabe des Programms:

```
Tupel 1 verpackt: (3, 6, -8)
Tupel 2 verpackt: (3, 6, -8)
3
6
-8
Tupel entpackt: 3 6 -8
Mehrfache Zuweisung: 3 6 -8
Auswirkung später: -8 3 9
Rest in Liste: 3 [6, 12, -28] -8
```

Ein Tupel enthält mehrere Elemente, die durch Kommata voneinander getrennt sind. Sie können innerhalb von runden Klammern notiert werden. Die Zuweisung eines Tupels zu einer einzelnen Variablen wird auch *Verpacken* eines Tupels genannt.

Die Ihnen bereits bekannte `for`-Schleife arbeitet mit einem Tupel.

Ein Tupel kann mithilfe einer Zuweisung in mehrere Variablen *entpackt* werden. Dabei muss darauf geachtet werden, dass die Anzahl der Variablen mit der Anzahl der Elemente des Tupels übereinstimmt.

Mithilfe einer mehrfachen Zuweisung können mehrere Werte gleichzeitig mehreren Variablen zugewiesen werden. Auch hier muss auf die passende Anzahl geachtet werden.

Werden bei einer mehrfachen Zuweisung die zugewiesenen Werte verändert, wirkt sich das erst nach dem Ende der gesamten Zuweisung aus. Im vorliegenden Beispiel erhält `a` den ursprünglichen Wert von `c`, `b` den ursprünglichen Wert von `a` und `c` die Summe der ursprünglichen Werte von `a` und `b`.

Mithilfe eines *Ausdrucks mit Stern* (englisch: *starred expression*) kann eine mehrfache Zuweisung flexibler gestaltet werden. Sie können vor eine der Variablen den Operator * setzen. Damit verweist diese Variable auf eine Liste, die nach der Zuweisung die überzähligen Werte enthält.

Im vorliegenden Beispiel werden der erste und der letzte Wert den Variablen `a` und `c` zugewiesen. Die mittleren Werte (hier sind es drei) werden in der Liste `b` gespeichert. Werden nur zwei Elemente zugewiesen, ist die Liste `b` leer.

4.5 Dictionarys

Ein Dictionary oder assoziativer Array ist mit einem Wörterbuch zu vergleichen. In einem Wörterbuch finden Sie unter einem Schlüsselbegriff die zugeordnete Information. So steht etwa in einem englisch-deutschen Wörterbuch unter dem Eintrag *house* der zugeordnete deutsche Begriff *Haus*.

4.5.1 Eigenschaften, Operatoren und Methoden

In Python stellen Dictionarys veränderliche Objekte dar und enthalten Paare. Jedes Paar besteht aus einem eindeutigen Schlüssel und einem zugeordneten Wert. Über den Schlüssel greifen Sie auf den Wert zu. Als Schlüssel werden meist Strings verwendet, es können aber auch Zahlen genutzt werden. Die Paare eines Dictionarys sind ungeordnet.

Im folgenden Beispiel werden die Namen und Altersangaben mehrerer Personen in einem Dictionary erfasst und bearbeitet. Der Name dient als Schlüssel. Über diesen Schlüssel kann auf die Altersangabe (also auf den Wert) zugegriffen werden.


```
dc = {"Peter":31, "Julia":28, "Werner":35}
print("Dictionary:", dc)
print("Anzahl:", len(dc))
print()

dc_vergleich = {"Peter":31, "Werner":35, "Julia":28}
if dc == dc_vergleich:
    print("Gleich")

dc["Julia"] = 27
print("Wert ersetzt:", dc)

dc["Moritz"] = 22
print("Element hinzugefügt:", dc)
print()

if "Julia" in dc:
    print(dc["Julia"])
del dc["Julia"]
if "Julia" not in dc:
    print("Nicht enthalten")
print("Element entfernt:", dc)

dc_hinzu = {"Moritz": 18, "Karin": 29}
dc.update(dc_hinzu)
print("Update:", dc)
```

Listing 4.28 Datei »dictionary.py«

Folgende Ausgabe wird erzeugt:

```
Dictionary: {'Peter': 31, 'Julia': 28, 'Werner': 35}
Anzahl: 3

Gleich
Wert ersetzt: {'Peter': 31, 'Julia': 27, 'Werner': 35}
Element hinzugefügt: {'Peter': 31, 'Julia': 27, 'Werner': 35, 'Moritz': 22}

27
Nicht enthalten
Element entfernt: {'Peter': 31, 'Werner': 35, 'Moritz': 22}
Update: {'Peter': 31, 'Werner': 35, 'Moritz': 18, 'Karin': 29}
```

Das Dictionary `dc` wird mit drei Paaren erzeugt und ausgegeben. Dictionarys werden mithilfe von geschweiften Klammern erzeugt. Die Paare werden durch Kommata voneinander getrennt, ein Paar wird in der folgenden Form notiert: *Schlüssel:Wert*.

Die Funktion `len()` liefert die Länge, also die Anzahl der Paare des Dictionarys.

Sie können Dictionarys nur mithilfe der beiden Operatoren `==` und `!=` miteinander vergleichen. Zwei Dictionarys stimmen überein, wenn Sie dieselben Paare enthalten, also dieselben Schlüssel mit denselben Werten. Da die Paare eines Dictionarys ungeordnet sind, ist beim Vergleich die Reihenfolge der Elemente nicht relevant.

Auf einen einzelnen Wert greifen Sie mithilfe des Schlüssels in rechteckigen Klammern zu. Bei der Zuweisung eines Paars wird geprüft, ob der Schlüssel bereits existiert. Ist das der Fall, wird nur der alte Wert durch den neuen Wert ersetzt. Ist das nicht der Fall, wird das Paar zum Dictionary hinzugefügt.

Mithilfe des Operators `in` prüfen Sie, ob ein bestimmter Schlüssel im Dictionary existiert. Das ist zum Beispiel vor der Ausgabe eines Werts oder vor dem Entfernen eines Paars mithilfe des Operators `del` notwendig. Beim Versuch, auf einen nicht existierenden Schlüssel zuzugreifen, tritt eine Ausnahme auf.

Die Methode `update()` dient zum Aktualisieren der Paare eines neuen Dictionarys zu einem vorhandenen Dictionary. Bei der Zuweisung der neuen Paare wird jeweils geprüft, ob der Schlüssel bereits existiert. Ist das der Fall, wird nur der alte Wert durch den neuen Wert ersetzt. Ist das nicht der Fall, wird das Paar zum Dictionary hinzugefügt.

4.5.2 Dynamische Views

Die Methoden `keys()`, `items()` und `values()` erzeugen sogenannte dynamische Views eines Dictionarys. Ändert sich das Dictionary, ändern sich diese Views ebenfalls. Nachfolgend wird mit diesen Views gearbeitet:

```
dc = {"Peter":31, "Julia":28, "Werner":35}
print("Dictionary:", dc)

k = dc.keys()
print("Schlüssel:", k)
for schluessel in k:
    print(schluessel)
if "Werner" in k:
    print("Schlüssel Werner ist enthalten")
print()

v = dc.values()
print("Werte:", v)
for wert in v:
    print(wert)
if 31 in v:
    print("Wert 35 ist enthalten")
print()

i = dc.items()
print("Items:", i)
for k, v in i:
    print(f"Schlüssel {k}, Wert {v}")
```

Listing 4.29 Datei »dictionary_view.py«

Das Programm erzeugt die folgende Ausgabe:

```
Dictionary: {'Peter': 31, 'Julia': 28, 'Werner': 35}
Schlüssel: dict_keys(['Peter', 'Julia', 'Werner'])
Peter
Julia
Werner
Schlüssel Werner ist enthalten

Werte: dict_values([31, 28, 35])
31
28
35
Wert 35 ist enthalten

Items: dict_items([('Peter', 31), ('Julia', 28), ('Werner', 35)])
Schlüssel Peter, Wert 31
Schlüssel Julia, Wert 28
Schlüssel Werner, Wert 35
```

Mithilfe der Methode `keys()` wird eine View der Schlüssel des Dictionarys erzeugt, die anschließend ausgegeben wird. Die einzelnen Schlüssel werden mithilfe einer `for`-Schleife ausgegeben. Mithilfe des Operators `in` wird geprüft, ob ein bestimmter Schlüssel in der View existiert.

Die Methode `values()` dient zur Erzeugung einer View der Werte des Dictionarys, die anschließend ausgegeben wird. Die einzelnen Werte werden mithilfe einer `for`-Schleife

ausgegeben. Mithilfe des Operators `in` wird geprüft, ob ein bestimmter Wert in der View existiert.

Mithilfe der Methode `items()` wird eine View der Schlüssel-Wert-Paare des Dictionarys erzeugt, die anschließend ausgegeben wird. Die einzelnen Paare werden mithilfe einer `for`-Schleife ausgegeben. Dabei wird jeweils das Tupel aus Schlüssel und Wert in zwei Variablen entpackt.

4.6 Sets, Mengen

Sets (deutsch: Mengen) unterscheiden sich von Listen und Tupeln dadurch, dass jedes Element nur einmal existiert.

Sets sind ungeordnet, daher ist auch die Reihenfolge bei der Ausgabe eines Sets nicht festgelegt. Einzelne Elemente können nicht anhand eines Slice bestimmt werden. Allerdings können Sie mit Sets einige interessante Operationen durchführen, die aus der Mengenlehre bekannt sind.

4.6.1 Eigenschaften, Operatoren und Methoden

Im folgenden Beispiel wird ein Set erzeugt und bearbeitet:

```
st = set([8, 5, 5, 2, 5])
print("Set:", st)
print("Anzahl:", len(st))

for x in st:
    print(x)
if 5 in st:
    print("Wert ist enthalten")

su = set([2, 8])
if su < st:
    print("Echte Teilmenge")
sv = set([2, 8, 5])
if sv <= st:
    print("Teilmenge")

st.add(-12)
st.add(-12)
```



```
print("Element hinzu:", st)

st.discard(5)
print("Element entfernt:", st)

sk = st.copy()
print("Kopie:", sk)

sk.clear()
print("Geleert:", sk)

sf = frozenset([8, 5, 5, 2, 5])
print("Frozenset:", sf)
```

Listing 4.30 Datei »set_eigenschaft.py«

Die Ausgabe lautet:

```
Set: {8, 2, 5}
Anzahl: 3
8
2
5
Wert ist enthalten
Echte Teilmenge
Teilmenge
Element hinzu: {8, 2, -12, 5}
Element entfernt: {8, 2, -12}
Kopie: {8, 2, -12}
Geleert: set()
Frozenset: frozenset({8, 2, 5})
```

Ein Set erzeugen Sie mithilfe der Funktion `set()`. Ihr wird als einziger Parameter eine Liste oder ein anderes Iterable übergeben. In der hier übergebenen Liste sind Werte mehrfach enthalten, im Set nur noch einmal.

Die Funktion `len()` liefert die Länge, also die Anzahl der Elemente eines Sets. Mithilfe einer `for`-Schleife können Sie das Set durchlaufen. Mit `in` prüfen Sie, ob ein bestimmtes Element im Set enthalten ist.

Mit den vier Operatoren `<`, `<=`, `>` und `>=` stellen Sie fest, ob ein Set eine Teilmenge oder eine echte Teilmenge eines anderen Sets ist. Dabei ist die Reihenfolge der Elemente im Set unerheblich:

- Das Set su ist eine echte Teilmenge des Sets st: Alle Elemente von su sind in st enthalten, und su hat weniger Elemente als st.
- Das Set sv ist eine einfache Teilmenge des Sets st: Alle Elemente von sv sind in st enthalten, aber sv hat ebenso viele Elemente wie st.

Mithilfe der Methode add() fügen Sie ein Element hinzu, falls es noch nicht enthalten ist. Die Methode discard() dient zum Löschen eines bestimmten Elements. Es stellt kein Problem dar, falls es nicht enthalten ist. Die Methode copy() erzeugt ein neues Set als Kopie des alten Sets. Die Methode clear() entfernt alle Elemente aus dem Set.

Der englische Begriff *frozen* bedeutet »eingefroren«. Ein Frozenset stellt die unveränderliche Variante eines Sets dar. Es wird mithilfe der Funktion frozenset() erzeugt und bei der Ausgabe durch den Begriff frozenset. Mithilfe einer for-Schleife kann es durchlaufen und mithilfe des Operators in geprüft werden. Der Versuch, es zu verändern, führt zu einer Ausnahme.

4.6.2 Mengenlehre

Im nachfolgenden Programm werden mithilfe der Operatoren |, &, - und ^ einige Operationen aus der mathematischen Mengenlehre durchgeführt:

```
st = set([8, 15, "x"])
su = set([4, "x", "abc", 15])
print("st:", st)
print("su:", su)

print("Vereinigungsmenge:", st | su)
print("Schnittmenge:", st & su)
print("Differenzmenge st-su:", st - su)
print("Differenzmenge su-st:", su - st)
print("Symmetrische Differenzmenge:", st ^ su)
```

Listing 4.31 Datei »set_mengenlehre.py«

Die Ausgabe lautet:

```
st: {8, 'x', 15}
su: {'x', 4, 15, 'abc'}
Vereinigungsmenge: {'x', 'abc', 4, 8, 15}
Schnittmenge: {'x', 15}
Differenzmenge st-su: {8}
```



```
Differenzmenge su-st: {'abc', 4}
Symmetrische Differenzmenge: {4, 8, 'abc'}
```

Der Operator | dient zur Vereinigung zweier Sets. Das entstehende Set enthält alle Elemente, die in einem der beiden Sets enthalten sind. Auch im neuen Set kommt jedes Element nur einmal vor.

Die Elemente, die in beiden Sets enthalten sind, bilden die Schnittmenge. Sie wird mithilfe des Operators & ermittelt.

Bei einer Differenzmenge müssen Sie beachten, welches Set von welchem anderen Set abgezogen wird. Mithilfe des Operators - werden zwei verschiedene Differenzmengen erstellt. Die Operation st - su zieht vom Set st alle Elemente ab, die auch in su enthalten sind. Bei der Operation su - st verhält es sich umgekehrt.

Bei der symmetrischen Differenzmenge werden mit dem Operator ^ die Elemente ermittelt, die nur in einem der beiden Sets enthalten sind.

4.7 Wahrheitswerte und Nichts

Objekte und Ausdrücke können wahr oder falsch sein, außerdem gibt es auch das Nichts-Objekt. Betrachten wir einige Zusammenhänge.

4.7.1 Wahrheitswerte »True« und »False«

Alle Objekte in Python besitzen einen Wahrheitswert: True oder False. Bei Vergleichen zur Steuerung von Verzweigungen und Schleifen wird ein solcher Wahrheitswert ermittelt, siehe auch Abschnitt 3.3.1, »Vergleichsoperatoren«. Wahrheitswerte können in Variablen des Datentyps bool gespeichert werden. Die Funktion bool() liefert den Wahrheitswert eines Ausdrucks oder eines Objekts.

Folgende Objekte ergeben True:

- eine Zahl ungleich 0, also größer als 0 oder kleiner als 0
- eine nicht leere Sequenz (String, Liste, Tupel)
- ein nicht leeres Dictionary
- eine nicht leere Menge

Folgende Objekte ergeben False:

- eine Zahl, die den Wert 0 hat
- eine leere Sequenz (String "", Liste [], Tupel ())

- ein leeres Dictionary: `{}`
- eine leere Menge: `set()`, `frozenset()`
- eine Objektsammlung `x`, für die gilt `len(x) == 0`
- die Konstante `None` (siehe Abschnitt 4.7.2, »Nichts, ›None‹«)

Für Liste, Dictionary und Set gilt: Solange die Objektsammlung Elemente enthält, ergibt sich der Wahrheitswert `True`. Werden die Elemente entfernt, ergibt sich der Wahrheitswert `False`. Im folgenden Programm wird der Wahrheitswert verschiedener Objekte überprüft und ausgegeben.

```
print("5>3:", 5>3)
print("5<3:", 5<3)
print("Typ von 5>3:", type(5>3))

print("Zahl -0.1:", bool(-0.1))
print("Zahl 0:", bool(0))

print("String 'Hallo':", bool("Hallo"))
print("String '':", bool(""))

print("Liste [2,8]:", bool([2,8]))
print("Liste []:", bool([]))

print("Tupel (2,8):", bool((2,8)))
print("Tupel ():", bool(()))

print("Dictionary {'Julia':28, 'Werner':32}:",
      bool({"Julia":28, "Werner":32}))
print("Dictionary {}:", bool({}))

print("Set (2,8,2):", bool(set([2,8,2])))
print("Set ():", bool(set([])))
```

Listing 4.32 Datei »wahrheitswert.py«

Das Programm erzeugt die Ausgabe:

```
5>3: True
5<3: False
Typ von 5>3: <class 'bool'>
Zahl -0.1: True
```



```
Zahl 0: False
String 'Hallo': True
String '': False
Liste [2,8]: True
Liste []: False
Tupel (2,8): True
Tupel (): False
Dictionary {'Julia':28, 'Werner':32}: True
Dictionary {}: False
Set (2,8,2): True
Set (): False
```

String, Liste, Tupel, Dictionary und Set ergeben `False`, wenn sie leer sind, und `True`, wenn sie nicht leer sind. Dies können Sie zur Prüfung der betreffenden Objekte nutzen.

4.7.2 Nichts, »None«

Das Schlüsselwort `None` bezeichnet das Nichts-Objekt. `None` ist das einzige Objekt des Datentyps `NoneType`. Funktionen ohne Rückgabewert liefern `None` zurück. Dies kann auf Folgendes hinweisen:

- dass Sie eine Funktion falsch einsetzen, bei der Sie einen Rückgabewert erwarten
- dass eine Funktion kein Ergebnis liefert, obwohl dies erwartet wird.

Ein Beispiel:

```
def summe(a, b):
    c = a + b

# Programm
erg = summe(7,12)
if not erg:
    print("Fehler")
if erg is None:
    print("Fehler")
print("Ergebnis:", erg)
print("Wahrheitswert:", bool(erg))
print("Typ:", type(erg))
```

Listing 4.33 Datei »nichts.py«

Die Ausgabe lautet:

Fehler
Fehler
Ergebnis: None
Wahrheitswert: False
Typ: <class 'NoneType'>

Bei der Definition der Funktion `summe()` wird die Rückgabe des Ergebnisses »vergessen«. Daher erhält `erg` den Wert `None`. Das Nichts-Objekt hat den Wahrheitswert `False`. Damit können Sie feststellen, dass ein Fehler vorliegt.

4.8 Referenz, Identität und Kopie

In diesem Abschnitt erläutere ich den Zusammenhang zwischen Objekten und Referenzen. Wir untersuchen die Identität von Objekten und erzeugen Kopien von Objekten.

4.8.1 Referenz und Identität

Der Name eines Objekts entspricht einer Referenz auf das Objekt.

Weisen Sie diese Referenz einem anderen Namen zu, erzeugen Sie eine zweite Referenz auf dasselbe Objekt. Mithilfe des Identitätsoperators `is` stellen Sie fest, dass beide Referenzen auf dasselbe Objekt verweisen.

Wird das Objekt über die zweite Referenz geändert, zeigt sich eine der beiden folgenden Verhaltensweisen:

- Im Fall eines einfachen Objekts wie Zahl oder String wird ein zweites Objekt erzeugt, in dem der neue Wert gespeichert wird. Die beiden Referenzen verweisen danach auf zwei verschiedene Objekte.
- Im Fall eines nicht einfachen Objekts wie Liste, Tupel, Dictionary, Set usw. wird das Originalobjekt geändert. Es gibt nach wie vor ein Objekt mit zwei verschiedenen Referenzen.

Mithilfe des Operators `==` stellen Sie fest, ob zwei Objekte den gleichen Inhalt haben, ob also zum Beispiel zwei Listen die gleichen Elemente enthalten.

Im folgenden Beispiel werden nacheinander eine Zahl, ein String und eine Liste erzeugt und zweimal referenziert. Anschließend wird der zweiten Referenz jeweils ein neuer Inhalt zugewiesen. Identität und Inhalt werden anhand der beiden Operatoren `is` und `==` festgestellt.


```
print("Zahl:")
x = 12.5
y = x
print("dasselbe Objekt:", x is y)
y = 15.8
print("dasselbe Objekt:", x is y)
print("gleicher Inhalt:", x == y)
print()

print("String:")
x = "Robinson"
y = x
print("dasselbe Objekt:", x is y)
y = "Freitag"
print("dasselbe Objekt:", x is y)
print("gleicher Inhalt:", x == y)
print()

print("Liste:")
x = [23,"hallo",-7.5]
y = x
print("dasselbe Objekt:", x is y)
y[1] = "welt"
print("dasselbe Objekt:", x is y)
```

Listing 4.34 Datei »referenz.py«

Es wird die folgende Ausgabe erzeugt:

Zahl:
dasselbe Objekt: True
dasselbe Objekt: False
gleicher Inhalt: False

String
dasselbe Objekt: True
dasselbe Objekt: False
gleicher Inhalt: False

Liste:
dasselbe Objekt: True
dasselbe Objekt: True

Die Ausgabe zeigt, dass die Objekte zunächst jeweils identisch sind.

Durch die Zuweisung eines neuen Werts wird bei einer Zahl oder einem String ein neues Objekt erzeugt. Die Inhalte sind danach unterschiedlich.

Die Liste (hier stellvertretend auch für andere Objekte) existiert insgesamt nur einmal, auch wenn einzelne Elemente der Liste verändert werden. Sie können über beide Referenzen auf dieselbe Liste zugreifen.

4.8.2 Ressourcen sparen

Python spart gern Ressourcen. Dies kann zu einem ungewöhnlichen Verhalten führen: Wird einem Objekt über eine Referenz ein Wert zugewiesen *und* wird auf denselben Wert bereits von einer anderen Referenz verwiesen, kann es geschehen, dass die beiden Referenzen anschließend auf dasselbe Objekt verweisen. Python spart also Speicherplatz.

Das Schlüsselwort `del` dient zur Löschung von nicht mehr benötigten Referenzen. Ein Objekt, auf das zwei Referenzen verweisen, wird durch das Löschen der ersten Referenz nicht gelöscht.

Ein Beispiel:

```
x = 42
y = 56
print(f"x:{x}, y:{y}, identisch:{x is y}")

y = 42
print(f"x:{x}, y:{y}, identisch:{x is y}")

del y
print("x:", x)

del x
try:
    print("x:", x)
except:
    print("Fehler")
```

Listing 4.35 Datei »ressourcen.py«

Es wird die folgende Ausgabe erzeugt:


```
x:42, y:56, identisch:False
x:42, y:42, identisch:True
x: 42
Fehler
```

Zunächst erhalten die Variablen `x` und `y` unterschiedliche Werte. Damit handelt es sich bei ihnen um zwei Referenzen auf zwei verschiedene Objekte. Anschließend erhält die Variable `y` denselben Wert wie die Variable `x`. Nun gibt es nur noch ein Objekt, auf das zwei Referenzen verweisen.

Die Referenz `y` wird gelöscht. Das Objekt existiert weiterhin und kann über die Referenz `x` erreicht werden. Anschließend wird die Referenz `x` gelöscht. Die Ausgabe über diese Referenz führt zu einem Fehler, da der Name nicht mehr existiert.

4.8.3 Objekte kopieren

Echte Kopien von nicht einfachen Objekten wie Liste, Tupel, Dictionary, Set usw. können Sie wie folgt erzeugen: Sie erzeugen ein leeres Objekt und fügen die einzelnen Elemente hinzu.

Für umfangreiche Objekte, die wiederum andere Objekte enthalten, kann das sehr aufwendig werden. Sie können daher auch die Funktion `deepcopy()` aus dem Modul `copy` verwenden.

Beide Vorgehensweisen werden in folgendem Programm gezeigt.

```
import copy

x = [23,"hallo",-7.5]
y = []
for i in x:
    y.append(i)
print("dasselbe Objekt:", x is y)
print("gleicher Inhalt:", x == y)
print()

x = [23,["Berlin","Hamburg"],-7.5,12,67]
y = copy.deepcopy(x)
print("dasselbe Objekt:", x is y)
print("gleicher Inhalt:", x == y)
```

Listing 4.36 Datei »kopieren.py«

Das Programm erzeugt die Ausgabe:

```
dasselbe Objekt: False
gleicher Inhalt: True

dasselbe Objekt: False
gleicher Inhalt: True
```

Die Ausgabe zeigt, dass in beiden Fällen jeweils ein neues Objekt erzeugt wird. Die Inhalte der beiden Objekte sind allerdings gleich.

Inhalt

12 Benutzeroberflächen mit PyQt 397

Anhang 431

Materialien zum Buch

Auf der Webseite zu diesem Buch stehen folgende Materialien für Sie zum Download bereit:

- **alle Beispielprogramme**
- **Lösungen der Übungsaufgaben**

Gehen Sie auf *https://www.rheinwerk-verlag.de/5472*. Klicken Sie auf den Reiter MATERIALIEN. Sie sehen die herunterladbaren Dateien samt einer Kurzbeschreibung des Dateiinhalts. Klicken Sie auf den Button HERUNTERLADEN, um den Download zu starten. Je nach Größe der Datei (und Ihrer Internetverbindung) kann es einige Zeit dauern, bis der Download abgeschlossen ist.