GANZ EINFACH

Programmieren lernen mit Python

Ohne Kopfschmerzen vom ersten Skript bis zur eigenen Automatisierung

Ein unabhängiges Praxisbuch



Danilo Sieren

Danilo Sieren

Ganz Einfach

Python

Programmieren verstehen und anwenden
Vom ersten Skript zum eigenen Projekt

Überarbeitete und erweiterte
Neuauflage 2026

Willkommen

Python hilft dir, wiederkehrende Aufgaben zu automatisieren, Daten auszuwerten und eigene Anwendungen zu entwickeln. Dieses Buch erklärt die Sprache von den ersten Ausdrücken bis zu Programmen mit Dateien, Datenbanken, Netzwerkzugriffen und einer Benutzeroberfläche. Du brauchst keine Programmiererfahrung. Wichtig sind ein eigener Rechner, etwas Zeit zum Ausprobieren und die Bereitschaft, Ergebnisse zu überprüfen.

Du lernst an kleinen vollständigen Programmen. Ein Beispiel zeigt jeweils eine überschaubare Aufgabe; danach untersuchst du seine Grenzen und veränderst es. Ein Dateiinventar begleitet mehrere Kapitel. Es zählt und beschreibt Dateien, speichert Ergebnisse und macht daraus Berichte. Das ist eine reale Verwaltungsaufgabe, ohne eine fertige Cloudplattform vorzutäuschen. Eine Ausgabe mit print ist noch kein Upload, ein Hashwert noch keine Verschlüsselung und ein lokaler Webdienst noch kein öffentlich betreibbares Produkt.

Die ersten zehn Kapitel behandeln Sprache und Datenstrukturen. Danach folgen Funktionen, Module, Typen, Fehlerbehandlung und Tests. Die zweite Hälfte verbindet diese Kenntnisse mit Betriebssystem, HTTP, SQLite, Nebenläufigkeit, Oberflächen und Auslieferung. Die Anhänge enthalten ein zusammenhängendes Projekt, vertiefende Übungen, Lösungen und Hilfen zur Fehlersuche. Du kannst die Kapitel nacheinander durcharbeiten oder später über Inhaltsverzeichnis und Sachregister gezielt nachschlagen.

W.1 Arbeitsweise und Voraussetzungen

Lege für die Übungen einen eigenen lokalen Ordner an. Verwende zunächst ausschließlich selbst erzeugte Testdateien.

Ändere eine Sache auf einmal, starte das Programm erneut und notiere, was du erwartet hast. Vergleiche das Ergebnis mit deiner Vermutung. Dieses Vorgehen hilft dir auch dann weiter, wenn du eine Fehlermeldung noch nicht vollständig verstehst.

Codeblöcke enthalten entweder ein vollständiges kleines Programm, eine ausdrücklich benannte Projektdatei oder Terminalbefehle. Terminalbefehle gehören in die Kommandozeile deines Betriebssystems. Pythoncode gehört in eine Datei mit der Endung .py oder in die interaktive Pythonkonsole. Die Bezeichnung vor dem Beispiel nennt den Zweck. Verwende im Editor normale gerade Anführungszeichen; typografische Anführungszeichen aus Textverarbeitungen verändern den Quelltext.

In einem größeren Projekt sind mehrere Dateien sinnvoll. Jede abgedruckte Datei ist vollständig; die zugehörigen Dateien stehen im selben beschriebenen Projektordner. Die Trennung folgt den Aufgaben im Programm. Sie soll dir helfen, Einlesen, Berechnen, Speichern und Anzeigen unabhängig voneinander zu verstehen.

W.2 Versionsstand und Lernpfad

Redaktionsstand ist der 2. November 2026. Die offizielle Veröffentlichungsseite nennt Python 3.15.2 vom 25. Oktober 2026. Für neue Installationen verwendest du eine aktuelle, unterstützte Wartungsversion. Die Grundbeispiele sind bewusst mit Python 3.12 und neuer formuliert. Abschnitte über Neuerungen kennzeichnen ausdrücklich, wenn sie 3.14 oder 3.15 voraussetzen. So kannst du auch mit einem unterstützten, etwas älteren Interpreter sinnvoll lernen. [D01]

Bei Zusatzpaketen zählt die Kombination aus Pythonversion, Betriebssystem und Paketversion. Eine neue Pythonhauptversion kann erscheinen, bevor jedes Erweiterungspaket passende Binärdateien anbietet. Falls eine benötigte Bibliothek die neueste

Reihe noch nicht unterstützt, ist eine getrennte Projektumgebung mit einer unterstützten älteren Reihe vernünftig. Installiere keine zufälligen Ersatzpakete, nur um eine Fehlermeldung zu umgehen.

Die kurzen Grundübungen brauchen keine Cloudkonten und keine bezahlten Dienste. Für HTTP verwendest du zuerst einen lokalen Testserver. SQLite speichert lokal. Die Einführung in FastAPI und Datenanalyse ist eine Erweiterung, für die du zusätzliche Pakete in einer eigenen Umgebung installierst. Welche Bibliothek zu deinem späteren Projekt passt, entscheidest du anhand der Aufgabe und ihrer dokumentierten Anforderungen.

Inhalt

Willkommen 2
1 Python einrichten und Programme starten 7
2 Projektumgebungen und Zusatzpakete 11
3 Namen Werte und Datentypen 15
4 Mit Zahlen zuverlässig rechnen 19
5 Texte Unicode und formatierte Ausgabe 23
6 Bedingungen und Eingabeprüfung 29
7 Schleifen und begrenzte Wiederholungen 34
8 Listen Tupel und gemeinsame Referenzen 38
9 Dictionaries Mengen und Datenmodelle 42
10 Iteratoren Generatoren und Datenströme 46
11 Funktionen und klare Schnittstellen 51
12 Module Pakete und Projektstruktur 58
13 Typanmerkungen und lesbare Datenverträge 62
14 Klassen Verhalten und Komposition 67
15 Ausnahmen und verlässliche Fehlergrenzen 71
16 Debugging und aussagekräftige Protokolle 75
17 Tests die Verhalten absichern 79
18 Dateien Pfade und kontrollierte Änderungen 85
19 CSV JSON Archive und sichere Datenformate 91
20 Kommandozeile Prozesse und Konfiguration 97
21 Datum Zeit und wiederholbare Abläufe 101
22 HTTP und verlässliche API Clients 104
23 SQLite und dauerhafte Daten 110
24 Asyncio und strukturierte Nebenläufigkeit 120
25 Threads Prozesse und gemeinsam verwendeter Zustand 125

26 Eine bedienbare Oberfläche mit Tkinter 129
27 Webschnittstellen mit FastAPI 134
28 Daten auswerten und ehrlich darstellen 140
29 Algorithmen und gemessene Verbesserung 148
30 Ausliefern Warten und Python 2026 einordnen 152
A Praxisprojekt Dateiinventar 157
B Vertiefende Werkstatt 171
C Fehler gezielt diagnostizieren 181
D Von der Übung zur eigenen Anwendung 187
E Nachschlagen und Aufgaben auswählen 196
F Lösungen und begründete Erwartungen 202
Detailliertes Inhaltsverzeichnis 210
Sachregister 219
Quellen und weiterführende Dokumentation 225
Impressum 231

1 Python einrichten und Programme starten

1.1 Interpreter Editor und Terminal

Ein Pythonprogramm ist zunächst Text. Der Interpreter liest diesen Text und führt die darin beschriebenen Operationen aus. CPython ist die verbreitete Referenzimplementierung. Sie übersetzt Quelltext in Bytecode und verarbeitet diesen mit ihrer Laufzeitumgebung. Die häufige Erklärung, Python führe jede Textzeile sofort einzeln aus, ist deshalb zu grob. Schon ein Syntaxfehler am Ende einer Datei kann verhindern, dass ihr vorheriger Inhalt ausgeführt wird.

Der Editor bearbeitet Dateien. Das Terminal startet Programme und zeigt deren Ausgaben an. Eine Entwicklungsumgebung verbindet diese Aufgaben, ergänzt automatische Vorschläge und einen Debugger. Du kannst mit IDLE aus einer passenden Pythoninstallation oder einem anderen Editor beginnen. Entscheidend ist, dass der Editor die Datei als UTF-8 speichert und du weißt, welcher Interpreter sie startet.

Die interaktive Konsole eignet sich zum Prüfen kurzer Ausdrücke. Eine Eingabe wie 2 + 3 zeigt unmittelbar 5. Ein Skript eignet sich für wiederholbare Abläufe und mehrere Funktionen. Speichere es vor dem Start. Sonst untersuchst du möglicherweise den alten Dateistand, obwohl im Editor bereits eine Änderung sichtbar ist.

1.2 Installation unter Windows

Die offizielle Dokumentation empfiehlt den Python install manager. Du erhältst ihn über python.org oder den Microsoft Store; die beiden dort genannten Varianten sind gleichwertig. Nach der Installation öffnest du ein neues Terminal. Prüfe

zuerst, ob python und py auf die gewünschte Installation zeigen. Bereits vorhandene Installationen und ältere Startprogramme können die Befehle beeinflussen. Lösche solche Installationen nicht blind, sondern prüfe die Pfade und die Hinweise zur Fehlerbehebung. [D02]

Terminal unter Windows mit installiertem Python install manager:

```
py list
py install 3.15
py -V:3.15 --version
py -V:3.15 -m venv .venv
.venv\Scripts\python.exe --version
```

Die Versionsangabe 3.15 wählt die Reihe, keine für immer unveränderliche Wartungsversion. Prüfe die tatsächlich installierte Ausgabe. Die letzte Zeile verwendet den Interpreter der neuen Projektumgebung direkt. Dafür musst du keine Aktivierungsskripte freigeben und keine systemweite PowerShellrichtlinie ändern.

Ein Terminal mit Administratorrechten ist für normale Lernprojekte nicht erforderlich. Wenn der Rechner zentral verwaltet wird, beachtest du dessen freigegebenen Installationsweg. Ein Fehler wegen fehlender Rechte ist kein Anlass, Schutzmechanismen pauschal auszuschalten.

1.3 Installation unter macOS und Linux

Unter macOS verwendest du eine für dein System geeignete Installation von python.org oder eine bewusst gewählte Distribution. Verändere keine Pythonbestandteile, die das Betriebssystem oder andere Werkzeuge verwalten. Prüfe python3 --version und anschließend den Interpreterpfad. Der macOS-Installer und seine Begleitdateien erläutern auch die Einrichtung der Zertifikate. [D03]

Unter Linux installierst du Python und gegebenenfalls das venv-Paket über die Paketverwaltung deiner Distribution. Deren Namen unterscheiden sich. Eine Meldung über eine extern verwaltete Umgebung bedeutet, dass du nicht mit pip in das Systempython schreiben sollst. Erzeuge eine Projektumgebung. Ein generelles sudo pip install ist für dieses Buch kein Installationsweg.

Terminal unter macOS oder Linux:

```
python3 --version
python3 -m venv .venv
.venv/bin/python --version
.venv/bin/python -m pip --version
```

Wenn python3 eine ältere Reihe startet, kannst du einen gezielt installierten Interpreter wie python3.15 verwenden. Entscheidend ist der tatsächliche Pfad. Ein Alias im Terminal und die Auswahl im Editor können unterschiedliche Programme bezeichnen.

1.4 Das erste Skript

Erzeuge im Projektordner die Datei start.py. Vermeide Dateinamen wie json.py, typing.py oder sqlite3.py. Solche Namen können später gleichnamige Standardmodule verdecken. Ein sachlicher Name wie start.py oder inventar.py ist leichter zuzuordnen.

Vollständige Datei start.py:

```
import sys

print("Hallo Python")
print("Version:", sys.version.split()[0])
print("Interpreter:", sys.executable)
```

Starte die Datei unter Windows mit .venv\Scripts\python.exe start.py, unter macOS und Linux mit .venv/bin/python start.py. Die erste Zeile ist deine Programmausgabe. Danach siehst du

Version und Interpreterpfad. Diese beiden Informationen sind bei Installationsproblemen hilfreicher als die Vermutung, dass „Python installiert“ sei.

Fehlt die Datei, prüfe den aktuellen Ordner. Gibt es einen SyntaxError, vergleiche Klammern und Anführungszeichen. Wenn ein Textverarbeitungsprogramm die Datei als start.py.txt gespeichert hat, korrigiere den tatsächlichen Namen einschließlich der Endung. Eine sichtbare Bezeichnung im Dateimanager muss nicht die vollständige Endung zeigen.

1.5 Werkstatt für einen verlässlichen Start

Ergänze eine zweite Begrüßung und starte das Skript erneut. Öffne anschließend die interaktive Konsole und probiere denselben print-Aufruf. Beende sie mit exit(). Du solltest jetzt den Unterschied zwischen Dateiausführung und interaktiver Eingabe benennen können.

Notiere Betriebssystem, Pythonversion und Projektpfad. Prüfe nach einer Änderung im Editor, ob die neue Ausgabe erscheint. Wenn das gelingt, hast du einen reproduzierbaren Ausgangspunkt. Du brauchst zunächst keine Erweiterungen, Konten, Container oder komplexe Konfiguration.

2 Projektumgebungen und Zusatzpakete

2.1 Warum jedes Projekt eine eigene Umgebung bekommt

Zwei Programme können verschiedene Versionen derselben Bibliothek benötigen. Ohne Trennung überschreibt die Installation für das zweite Programm möglicherweise eine Voraussetzung des ersten. Eine virtuelle Umgebung ordnet einem Projekt einen Interpreter und eigene installierte Pakete zu. Sie ist keine virtuelle Maschine und kein Schutzraum für unbekannten Code. Programme darin können weiterhin Dateien lesen, Netzverbindungen öffnen und mit deinen Benutzerrechten arbeiten. [D04]

Der Ordner .venv ist ein austauschbares Arbeitsmittel. Du kopierst ihn nicht auf einen anderen Rechner und speicherst ihn nicht als alleinige Projektsicherung. Die Umgebung enthält Pfade und plattformabhängige Dateien. Bewahre stattdessen Quelltext, Konfiguration, Tests und die Beschreibung der benötigten Pakete auf. Daraus erzeugst du später eine neue Umgebung.

Für ein Lernprojekt ist ein lokaler Ordner zweckmäßig. Eine aktive Umgebung und eine geöffnete SQLite-Datenbank gehören nicht in einen unbedacht synchronisierten Ordner. Synchronisationsprogramme können gleichzeitig geänderte Dateien auseinanderlaufen lassen. Eine Synchronisation ersetzt außerdem keine unabhängige, versionierte Sicherung.

2.2 Aktivierung und direkte Interpreterpfade

Eine Aktivierung verändert vor allem die Suchpfade der aktuellen Terminalsitzung. Danach verweist python üblicherweise auf die Umgebung. Das ist bequem, aber keine

Pflicht. Der direkte Aufruf der Umgebungsdatei macht die Auswahl ausdrücklich. In diesem Buch nennen Installationsbefehle häufig python -m pip. Dabei steht python für den zuvor überprüften Projektinterpreter.

Falls du eine Umgebung aktiviert hast, kontrollierst du die Auswahl trotzdem mit sys.executable. Die Anzeige (.venv) im Terminal ist eine Hilfe, kein Beweis. Ein anderes Terminalfenster hat seine eigene Umgebungsauswahl. Ein Editor kann wiederum einen anderen Interpreter gespeichert haben.

Eine Anwendung kann während einer Aktualisierung noch alte Module geladen haben. Beende sie und starte sie nach der Paketänderung neu. Erscheint ein Fehler nur im Editor, aber nicht im Terminal, vergleiche zuerst deren Interpreterpfade. Installiere das Paket nicht wahllos mehrfach.

2.3 Pakete bewusst installieren

Ein Importname und ein Paketname können verschieden sein. Die Installation erfolgt anhand der offiziellen Projektdokumentation, nicht anhand eines geratenen Namens. Prüfe Schreibweise, Herausgeber, unterstützte Pythonversionen und Herkunft. Ähnlich benannte Pakete müssen nicht zusammengehören. Die Python Packaging Authority beschreibt den üblichen Ablauf mit pip und venv. [D05]

Terminal in der überprüften Projektumgebung:

```
python -m pip --version
python -m pip install --upgrade pip
python -m pip list
python -m pip check
```

Der letzte Befehl prüft deklarierte Abhängigkeitskonflikte. Er beweist weder Sicherheit noch fachliche Eignung eines Pakets. Für die ersten Sprachkapitel brauchst du überhaupt kein Zusatzpaket. Installiere Erweiterungen erst, wenn das entsprechende Beispiel sie tatsächlich verwendet.

Bei einer Fehlermeldung liest du den ersten relevanten Fehler, nicht nur die letzte Zusammenfassung. Ein fehlendes Binärpaket für eine neue Pythonversion ist etwas anderes als eine unterbrochene Verbindung. Ein Installationsversuch aus Quelltext kann zusätzliche Compiler voraussetzen. Prüfe die offiziell unterstützte Kombination, bevor du dein System umbaust.

2.4 Abhängigkeiten nachvollziehbar festhalten

Für eine kleine Anwendung kann eine requirements-Datei die geprüften Pakete mit Versionen festhalten. python -m pip freeze zeigt den installierten Zustand. Diese Ausgabe enthält oft auch indirekte Abhängigkeiten und ist keine universelle, plattformunabhängige Sperrdatei. Sie ersetzt nicht die Beschreibung, welche Pakete dein Projekt unmittelbar benötigt.

Ein späteres Paketprojekt beschreibt Metadaten und direkte Abhängigkeiten in pyproject.toml. Eine Anwendung braucht zusätzlich einen kontrollierten Weg, ihren getesteten Gesamtzustand wiederherzustellen. Werkzeuge für Sperrdateien lösen dabei mehr Aufgaben als ein bloßer Name mit einer Versionsuntergrenze. Kapitel 30 erläutert den Unterschied zwischen Bibliotheksanforderungen und einer konkret ausgelieferten Anwendung.

Aktualisiere Abhängigkeiten in einer Kopie oder einem gesonderten Entwicklungsstand. Führe Tests aus und überprüfe Änderungen der öffentlichen Schnittstellen. Wenn eine neue Version Fehler verursacht, muss der vorherige geprüfte Zustand wiederherstellbar sein. „Immer die neueste Version“ ist ohne diesen Rückweg keine Wartungsstrategie.

2.5 Werkstatt mit einer frischen Umgebung

Lege einen zweiten leeren Projektordner an und erzeuge darin eine neue .venv. Vergleiche die Paketliste mit der ersten

Umgebung. Ergänze ein kurzes Skript, das sys.executable ausgibt. Starte es ausdrücklich mit beiden Interpretern. Du solltest anhand der Ausgabe erkennen, welche Umgebung arbeitet.

Erkläre danach in eigenen Worten, welche Dateien unersetzlich sind und welche neu erzeugt werden können. Der Quelltext und eigene Daten sind unersetzlich. Eine sauber beschriebene Umgebung lässt sich neu aufbauen. Diese Unterscheidung wird bei Sicherung und Auslieferung wichtig.

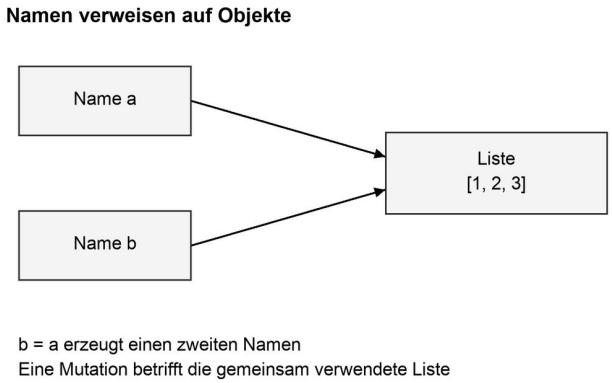
3 Namen Werte und Datentypen

3.1 Namen binden Objekte

Mit groesse = 120 bindest du den Namen groesse an ein Ganzzahlobjekt. Der Name ist kein Behälter mit unveränderlichem Datentyp. Eine spätere Zuweisung kann ihn an ein anderes Objekt binden. Das Objekt selbst besitzt seinen Typ. Diese Sicht erklärt später auch, warum zwei Namen dieselbe Liste verändern können.

Eine Zuweisung erzeugt nicht automatisch eine unabhängige Kopie. Bei einfachen unveränderlichen Werten fällt das zunächst wenig auf. Bei Listen und Dictionaries ist die Unterscheidung entscheidend. Schreibe beim Nachdenken auf, welche Namen auf dasselbe Objekt zeigen. Das ist genauer als die Vorstellung, jede Variable enthalte stets einen eigenen Datenblock.

Dynamische Typisierung bedeutet nicht, dass beliebige Werte zusammenpassen. Die Addition einer Zahl und eines Textes führt ohne geeignete Umwandlung zu einem TypeError. Python prüft die konkrete Operation während der Ausführung. Typanmerkungen können zusätzliche Werkzeuge informieren, ändern diese Grundregel aber nicht.



3.2 Die ersten Typen kennenlernen

int repräsentiert ganze Zahlen, float Gleitkommazahlen, str Text und bool Wahrheitswerte. None bezeichnet einen ausdrücklich fehlenden Wert. Ein leerer Text, die Zahl null und None sind verschiedene Dinge. Für ein optionales Änderungsdatum kann None sinnvoll sein; eine fehlende Dateigröße sollte nicht stillschweigend zu null werden, wenn null bereits „leere Datei“ bedeutet.

Vollständiges Programm typen.py:

```
name = "bericht.txt"
anzahl = 3
anteil = 0.25
fertig = False
datum = None

for wert in (name, anzahl, anteil, fertig, datum):
    print(repr(wert), type(wert).__name__)
```

repr stellt einen Wert so dar, dass etwa Anführungszeichen und bestimmte Steuerzeichen sichtbar werden. Das hilft bei der Fehlersuche. Ein Text "3" und die Zahl 3 sehen in einer schlichten Ausgabe ähnlich aus, gehören aber zu verschiedenen Typen. Für Fachlogik brauchst du deshalb klare Erwartungen an Eingaben.

3.3 Zuweisung Vergleich und Identität

Ein einzelnes Gleichheitszeichen weist zu. Zwei Gleichheitszeichen vergleichen Werte. is vergleicht Identität: Es fragt, ob beide Ausdrücke dasselbe Objekt bezeichnen. Verwende is None und is not None für fehlende Werte. Verwende == für fachliche Gleichheit wie Dateinamen oder Messwerte.

Ob kleine Ganzzahlen oder bestimmte Texte intern wiederverwendet werden, ist kein zuverlässiger Bestandteil deiner Anwendungslogik. Ein Vergleich wie name is "Anna" ist daher falsch, auch wenn er in einer Sitzung zufällig das gewünschte Ergebnis liefert. Du willst Textinhalte vergleichen, keine interne Objektwiederverwendung.

Ein Name wird beim Zugriff in den passenden Gültigkeitsbereichen gesucht. Ein Tippfehler kann einen NameError auslösen. Zwei ähnlich aussehende Unicodezeichen können verschiedene Namen ergeben. Verwende für Projektbezeichner eine klare, konsistente Schreibweise. Deutsche Texte in Ausgaben sind selbstverständlich möglich; einfache ASCII-Bezeichner erleichtern oft Zusammenarbeit und Tastatureingabe.

3.4 Änderbarkeit und Lebensdauer

Zahlen, Texte und Tupel sind unveränderlich. Wenn du einen Text mit einem weiteren Text verbindest, erhältst du einen neuen Wert. Listen und Dictionaries können ihren Inhalt verändern, ohne dass du den Namen neu bindest. Änderbarkeit ist damit eine Eigenschaft des Objekttyps, nicht des Variablennamens.

Die Lebensdauer eines Objekts hängt davon ab, ob es noch erreichbar ist. Ein return beendet einen Funktionsaufruf; sein zurückgegebenes Objekt kann danach weiterverwendet werden. Die vereinfachte Aussage, alle lokalen Daten verschwänden sofort vollständig aus dem Speicher, trifft deshalb nicht zu. Schließbare Ressourcen wie Dateien behandelst du unabhängig davon ausdrücklich mit with.

Verlasse dich für Dateien, Sperren oder Netzwerkverbindungen nicht auf einen vermuteten Zeitpunkt der Speicherbereinigung. Unterschiedliche Pythonimplementierungen können Objekte verschieden verwalten. Eine klar begrenzte Ressourcennutzung ist auch leichter zu testen.

3.5 Werkstatt zur Zustandsverfolgung

Notiere vor jedem Programmschritt die erwarteten Werte. Weise zunächst a = 10 und dann b = a zu. Binde a anschließend an 20. b bleibt an 10 gebunden. Wiederhole den Versuch später mit einer Liste und einer Änderung durch append. Der Unterschied zeigt, warum Zuweisung und Mutation zwei verschiedene Vorgänge sind.

Wenn ein Ergebnis überrascht, gib Typ und repr des betroffenen Wertes aus. Damit klärst du zuerst, welche Daten tatsächlich vorliegen. Ein komplizierter Lösungsversuch auf Grundlage eines falsch verstandenen Typs kostet meist mehr Zeit als diese kurze Prüfung.

4 Mit Zahlen zuverlässig rechnen

4.1 Ganzzahlen und Rechenoperationen

Python unterstützt Addition, Subtraktion, Multiplikation und Potenzen. Eine Division mit / liefert bei zwei gewöhnlichen Ganzzahlen einen float. Die Ganzzahldivision // rundet dagegen nach unten. Bei negativen Ergebnissen ist das etwas anderes als das bloße Abschneiden der Nachkommastellen. -7 // 3 ergibt -3 und -7 % 3 ergibt 2. Zusammen gilt a = (a // b) * b + a % b, solange b nicht null ist.

int hat in Python keine feste Grenze von 32 oder 64 Bit. Sehr große Zahlen benötigen aber Rechenzeit und Speicher. Auch Ein- und Ausgabeumwandlungen können Schutzgrenzen besitzen. Für gewöhnliche Dateigrößen ist int deshalb passend, aber fremde Eingaben dürfen trotzdem nicht unbeschränkt groß werden.

Klammern machen die beabsichtigte Rechnung sichtbar. Schreibe eine Prozentrechnung lieber als belegt / gesamt * 100 und prüfe gesamt vorher auf einen gültigen positiven Wert. Die Division durch null ist kein sinnvoller Sonderwert, sondern ein Fehler im Vertrag der Funktion.

4.2 Gleitkommazahlen und Toleranzen

Viele Dezimalbrüche besitzen keine endliche Darstellung im binären Zahlensystem. Ein float speichert deshalb häufig einen nahe gelegenen Wert. 0.1 + 0.2 ist im üblichen CPython-float nicht exakt gleich 0.3. Das ist keine gelegentliche Laune des Rechners, sondern eine Folge der Darstellung. [D06]

Für Messwerte vergleichst du häufig innerhalb einer fachlich gewählten Toleranz. math.isclose unterstützt relative und

absolute Toleranzen. Nahe null ist eine absolute Toleranz wichtig. Eine Toleranz ist keine universelle Fehlerkorrektur: Du musst begründen, welche Abweichung für deine Daten zulässig ist. [D07]

Vollständiges Programm rechnen.py:

```
from decimal import Decimal, ROUND_HALF_UP
from math import isclose

print(0.1 + 0.2 == 0.3)
print(isclose(0.1 + 0.2, 0.3))

preis = Decimal("19.99")
steuer = Decimal("0.19")
brutto = preis * (1 + steuer)
cent = Decimal("0.01")
print(brutto.quantize(cent, rounding=ROUND_HALF_UP))
```

Das Programm zeigt False, True und 23.79. Die Dezimalwerte werden aus Text erzeugt. Decimal(0.1) würde bereits den zuvor erzeugten binären Näherungswert übernehmen. Bei Geldbeträgen legst du außerdem fest, wann gerundet wird und welche Rundungsregel gilt. Das Beispiel verwendet eine ausdrücklich gewählte Rechenregel, keine allgemeine steuerrechtliche Vorschrift. [D08]

4.3 Einheiten eindeutig führen

Dateigrößen speicherst du im Programm möglichst als ganze Bytezahl. Erst für die Anzeige rechnest du um. 1 kB sind 1000 Byte, 1 KiB sind 1024 Byte. Entsprechend unterscheiden sich MB und MiB sowie GB und GiB. Vermische diese Einheiten nicht in derselben Formel. Eine Festplatte mit einer dezimalen Kapazitätsangabe erscheint in einer binär rechnenden Anzeige mit einem anderen Zahlenwert, ohne dass dadurch Speicher verschwunden wäre.