Einstieg in Python

Ideal für Programmiereinsteiger

DIE LESEPROBE

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 und 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 während der Entwicklung von Programmen schnell vertraut.
- Klare Strukturen: Python verlangt von Ihnen, 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.
- In die Python-Versionen 3.10 bis 3.12 sind viele Verbesserungen bei den Fehlermeldungen eingeflossen. Fehlermeldungen sind informativer und genauer. Daher helfen sie besser bei der Fehlersuche. Häufig werden bereits passende Vorschläge zur Behebung des Fehlers gemacht.

- Objektbearbeitung: In Python werden alle Daten als Objekte gespeichert. Dies führt zu einer einheitlichen Behandlung von Objekten unterschiedlichen Typs. Die physikalische Speicherung der Objekte von Python erfolgt automatisch. Bei der Entwicklung muss man 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.
- Zahlreiche umfangreiche Bibliotheken sind entweder bereits eingebunden oder lassen sich sehr leicht einbinden. Die Bibliotheken enthalten nützliche, spezialisierte Funktionen aus vielen Bereichen.

1.2 Verbreitung von Python

Aufgrund seiner vielen Vorzüge gehört Python seit vielen Jahren zu den beliebtesten Programmiersprachen. Aktuell (im Dezember 2023) belegt es den ersten Platz in verschiedenen Ranglisten zum Thema Programmiersprachen. Python wird in zahlreichen großen Unternehmen eingesetzt, hier einige Beispiele:

- Die Musik-App Spotify wurde mit Python erstellt.
- Beim Streamingdienst Netflix wird bei der Entwicklung immer mehr zu Python gegriffen, trotz einer Auswahl zwischen verschiedenen Systemen.
- Der Foto-Online-Dienst Instagram verwendet Python in einem wichtigen Teil seiner Anwendung.
- Bei Google wird möglichst häufig Python eingesetzt.
- Der Cloud-Storage-Anbieter Dropbox nutzt ausschließlich Python.

1.3 Aufbau des Buchs

Das vorliegende Buch führt Sie in die Programmiersprache Python in der Version 3.12 ein, die im Oktober 2023 erschienen ist. In meinem Buch lege ich besonderen Wert darauf, 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 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 werden die Programmierkenntnisse in Kapitel 5 vertieft. Kapitel 6 widmet sich der objektorientierten Programmierung mit Python. Einige nützliche Module zur Ergänzung der Programme werden in Kapitel 7 vorgestellt.

In Kapitel 8 und in Kapitel 9 lernen Sie, Daten dauerhaft in Dateien oder Datenbanken zu speichern.

Sowohl Windows als auch Ubuntu Linux und macOS bieten komfortable grafische Benutzeroberflächen (GUIs). Kapitel 10 beschäftigt sich mit der GUI-Erzeugung mithilfe des Moduls `tkinter`. Dieses stellt eine Schnittstelle zwischen dem grafischen Toolkit *Tk* und Python dar. In Kapitel 11 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 Python-Website *https://www.python.org* aus dem Internet laden. Zurzeit (im Dezember 2023) ist das die Datei *python-3.12.0-amd64.exe*. Rufen Sie diese Datei zur Installation auf. Als Erstes müssen Sie bestätigen, dass Sie ein Programm installieren möchten, das nicht aus dem Microsoft Store stammt.

Markieren Sie die Option ADD PYTHON TO PATH, damit Sie später die Möglichkeit haben, Python-Programme auf Ebene der Kommandozeile aus einem beliebigen Verzeichnis heraus zu starten.

Wählen Sie die Option CUSTOMIZE INSTALLATION aus. Lassen Sie ansonsten alle vorgegebenen Optionen gesetzt. Das gilt besonders für das Paketverwaltungsprogramm *pip*, mit dessen Hilfe Sie später zusätzliche Module und Programme installieren können, siehe auch Abschnitt A.1.

Wählen Sie jetzt im Feld CUSTOMIZE INSTALL LOCATION das Installationsverzeichnis *C:\Python*.

Anschließend steht im Startmenü ein Eintrag zu PYTHON 3.12, 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 3.12

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 23.04 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.

Zurzeit (im Dezember 2023) wird unter Ubuntu Linux noch die Python-Version 3.11.6 genutzt. Daher können die im Buch genannten neuen Fähigkeiten der Python-Version 3.12 noch nicht eingesetzt werden. Das wird sich in Kürze geändert haben.

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 Dezember 2023) ist das für macOS die Datei *python-3.12.0-macos11.pkg*.

Nach einem Doppelklick auf diese Datei startet die Installation. Nehmen Sie keine Änderungen vor, landet Python im Verzeichnis *Programme/Python 3.12*. 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 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 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 der Zahl in das duale (oder binäre) 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:

0x1a + 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 zugehörige 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 Bitebene 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 Exponentialoperator **

Der Exponentialoperator ** dient zur Berechnung von Potenzen, also von Ausdrücken 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 4.3 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.

4.1.4 Rundung und Konvertierung

Die eingebaute Funktion `round()` dient zur Rundung einer Zahl auf eine beliebige Stellenzahl vor oder nach dem Komma. Im Unterschied dazu schneidet die bereits bekannte Funktion `int()` die Nachkommastellen einer Zahl ab. Einige Beispiele:

```
x = 1.499999
print("x:", round(x))
x = 1.500000
print("x:", round(x))
print()

x = 12/7
print("x:", x)
print("Gerundet auf sechs Stellen nach dem Komma:", round(x,6))
```

```
print("Gerundet auf null Stellen:", round(x))
print("int(x):", int(x))
print()

x = 12e6/7
print("x:", x)
print("Gerundet auf drei Stellen vor dem Komma:", round(x,-3))
print("Gerundet auf null Stellen:", round(x))
print("int(x):", int(x))
```

Listing 4.4 Datei »zahl_runden.py«

Es wird abgerundet bis `1.499999` und aufgerundet ab `1.500000`.

Das Ergebnis der Division `12 / 7` wird auf drei Arten umgewandelt:

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

Das Gleiche geschieht mit dem Ergebnis der Division `12000000 / 7`:

- Mit `round()` wird das Ergebnis auf drei Stellen vor dem Komma gerundet.
- Mit der gleichen Funktion wird das Ergebnis auf die nächsthöhere bzw. nächstniedrigere ganze Zahl gerundet.
- Mit `int()` wird das Ergebnis in eine ganze Zahl umgewandelt.

Es wird folgende Ausgabe erzeugt:

```
x: 1
x: 2

x: 1.7142857142857142
Gerundet auf sechs Stellen nach dem Komma: 1.714286
Gerundet auf null Stellen: 2
int(x): 1

x: 1714285.7142857143
```

```
Gerundet auf drei Stellen vor dem Komma: 1714000.0
Gerundet auf null Stellen: 1714286
int(x): 1714285
```

4.1.5 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 4.5 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.

4.1.6 Weitere mathematische Funktionen

Ebenfalls im Modul `math` finden Sie eine Reihe von Funktionen und Konstanten. Ein Beispielprogramm:

```
import math

print("Quadratwurzel von 64:", math.sqrt(64))
print("Kubikwurzel von -27:", math.cbrt(-27))
print("Ganzzahlige Quadratwurzel von 80:", math.isqrt(80))
print()

print("Natürlicher Logarithmus von 33:", math.log(33))
print("e hoch 3.5:", math.exp(3.5))
print("2 hoch 10:", math.exp2(10))
print()

print("10er-Logarithmus von 0.001:", math.log10(0.001))
print("10 ** -3:", 10 ** -3)
print()

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

t1 = 3, 2, 5
t2 = 2, 4, 3
print("Produkt:", math.prod(t1))
print("Summe der Produkte:", math.sumprod(t1, t2))

print("Fakultät von 5:", math.factorial(5))
print("Größter gem. Teiler von 60 und 135:", math.gcd(60, 135))
print("Rest:", math.remainder(10.9, 2.5))
print("Rest:", math.remainder(11.9, 2.5))
print()
```

```
if not math.isclose(3, 2.96, rel_tol=0.01):
    print("Nicht nahe dran")

if math.isclose(3, 2.97, rel_tol=0.01):
    print("Nahe dran")
```

Listing 4.6 Datei »zahl_rechner.py«

Es wird die folgende Ausgabe erzeugt:

```
Quadratwurzel von 64: 8.0
Kubikwurzel von 27: -3.0
Ganzzahlige Quadratwurzel von 80: 8

Natürlicher Logarithmus von 33: 3.4965075614664802
e hoch 3.5: 33.11545195869231
2 hoch 10: 1024.0

10er-Logarithmus von 0.001: -3.0
10 ** -3: 0.001

Kreiszahl pi: 3.141592653589793
Eulersche Zahl e: 2.718281828459045

Produkt: 30
Summe der Produkte: 29
Fakultät von 5: 120
Größter gem. Teiler von 60 und 135: 15
Rest: 0.9000000000000004
Rest: -0.5999999999999996

Nicht nahe dran
Nahe dran
```

Die mathematische Quadratwurzel einer positiven Zahl kann mithilfe der Funktion `sqrt()` berechnet werden. Seit Python 3.11. lässt sich mithilfe der Funktion `cbrt()` die kubische Wurzel, also die dritte Wurzel, aus einer Zahl berechnen. 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.

Die Funktion `log()` berechnet den natürlichen Logarithmus einer positiven Zahl zur Basis `e`. Die Funktion `exp()` ermittelt das mathematisch Umgekehrte: den Wert von e^x. Seit Python 3.11. lässt sich mithilfe der Funktion `exp2()` der Wert von 2^x ermitteln.

Die Funktion `log10()` berechnet den Logarithmus einer positiven Zahl zur Basis 10. Anschließend wird das mathematisch Umgekehrte berechnet, und zwar der Wert von 10^x.

Zudem gibt es die mathematischen Konstanten `pi` und `e`.

Seit Python 3.8 lässt sich mithilfe der Funktion `prod()` das Produkt der Elemente eines Iterables ermitteln, hier eines Tupels. Hier wird gerechnet: `3 x 2 x 5 = 30`. Seit Python 3.12 können Sie mithilfe der Funktion `sumprod()` die Summe der Produkte derjenigen Elementpaare von zwei Iterables berechnen, die an derselben Position stehen. Hier wird gerechnet: `3 x 2 + 2 x 4 + 5 x 3 = 29`. Mehr zum Thema »Tupel« finden Sie in Abschnitt 4.4, »Tupel«.

Der Wert der Fakultät darf mathematisch nur von positiven ganzen Zahlen berechnet werden. Dazu dient die Funktion `factorial()`.

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 die beiden 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 `10.9` bzw. `11.9` durch `2.5` geteilt. Das mathematische Ergebnis liegt jeweils zwischen den beiden ganzen Zahlen `4` (`4 x 2.5 = 10`) und `5` (`5 x 2.5 = 12.5`). Im Fall von `10.9` liegt der Wert `10` näher, daher liefert die Funktion `remainder()` den Wert `0.9` (`= 10.9 – 10`). Im Fall von `11.9` liegt der Wert `12.5` näher, daher liefert die Funktion `remainder()` den Wert `-0.6` (`= 11.9 - 12.5`).

Seit Python 3.5 können Sie mithilfe der Funktion `isclose()` feststellen, ob zwei Zahlen einander nahe sind. Die Nähe von zwei Zahlen wird mithilfe einer relativen Toleranz festgelegt, hier `0.01`. Das bedeutet, dass sich die beiden Zahlen um maximal 1 % unterscheiden. Sie können 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.

4.1.7 Genauere Zahlen mit Nachkommastellen

In den restlichen Unterabschnitten von Abschnitt 4.1 behandele ich spezielle Themen zu Zahlen: genauere Zahlen mit Nachkommastellen, komplexe Zahlen, Bitoperatoren und Brüche. Sie können sie zunächst übergehen und bei Bedarf nachschlagen.

Die Klasse `Decimal` aus dem Modul `decimal` bietet Ihnen die Möglichkeit, mit Zahlen zu rechnen, die auf eine größere Anzahl von Nachkommastellen genau sind. Dabei müssen Sie auf einige Besonderheiten achten.

Zunächst ein Beispielprogramm:

```
import decimal

a = 10 / 7
print("float-Wert:  ", a)
a = decimal.Decimal(10) / 7
print("Decimal-Wert:", a)
print("Typ:", type(a))
print()

print("Ganze Zahlen:", a + 2 * 3 - (12 - 2) // 2)
print("Zahlen mit Nachkommastellen:", a + decimal.Decimal(2.5))
print("Division:", a + decimal.Decimal(4 / 2))
print()

a = decimal.Decimal(24)
print("Wert von a:", a)
print("Quadratwurzel von a:", a.sqrt())
print()

print("Natürlicher Logarithmus von a:", a.ln())
print("e hoch 3.178:", decimal.Decimal(3.178).exp())
print()

print("10er-Logarithmus von a:", a.log10())
print("Umdrehung:", 10 ** decimal.Decimal(1.38))
```

Listing 4.7 Datei »zahl_decimal.py«

Zunächst die Ausgabe des Programms:

```
float-Wert:   1.4285714285714286
Decimal-Wert: 1.4285714285714285714285714
Typ: <class 'decimal.Decimal'>

Ganze Zahlen: 2.4285714285714285714285714
Zahlen mit Nachkommastellen: 3.9285714285714285714285714
Division: 3.4285714285714285714285714
```

```
Wert von a: 24
Quadratwurzel von a: 4.898979485566356196394568149

Natürlicher Logarithmus von a: 3.178053830347945619646941601
e hoch 3.178: 23.998708106421155983371646 69

10er-Logarithmus von a: 1.380211241711606022936244587
Umdrehung: 23.999999999999999999999998
```

Es wird das Modul `decimal` importiert, damit die Klasse `Decimal` zur Verfügung steht. Bei `Decimal()` handelt es sich um den Konstruktor der Klasse `Decimal`. Er bietet verschiedene Möglichkeiten, ein Objekt der Klasse `Decimal` zu erzeugen, und liefert eine Referenz auf dieses Objekt zurück.

Hier wird ein `Decimal`-Objekt mit dem ganzzahligen Wert 10 erzeugt. Anschließend wird es durch den ganzzahligen Wert 7 geteilt. Das Ergebnis ist wesentlich genauer als die `float`-Division von 10 / 7.

`Decimal`-Objekte, ganze Zahlen und Klammern können innerhalb eines Ausdrucks gemeinsam verarbeitet werden, solange die beteiligten Operatoren ganzzahlige Ergebnisse erzeugen. Das ist bei den Operatoren zur Addition, Subtraktion, Multiplikation und ganzzahligen Division der Fall.

Sind Zahlen mit Nachkommastellen oder Ergebnisse von Divisionen beteiligt, müssen sie zunächst in `Decimal`-Objekte umgewandelt werden.

Für `Decimal`-Objekte können unter anderem die folgenden Methoden aufgerufen werden:

- `sqrt()` für die Quadratwurzel
- `ln()` für den natürlichen Logarithmus zur Basis `e`
- `exp()` für den Wert von e^x
- `log10()` für den Logarithmus zur Basis 10

Außerdem kann mithilfe des Operators ** potenziert werden.

Hinweis

Eine Methode ist eine Funktion, die nur für Objekte einer bestimmten Klasse aufgerufen werden kann. Klassen, Instanzen, Eigenschaften, Methoden, Konstruktoren und andere Begriffe aus dem Bereich der objektorientierten Programmierung werde ich in Kapitel 6, »Objektorientierte Programmierung«, genauer erläutern.

4.1.8 Komplexe Zahlen

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(f"a = {a}, Typ: {type(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) Typ: <class 'complex'>
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`. 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 den Betrag der komplexen Zahl. Die Methode `conjugate()` liefert die konjugiert komplexe Zahl, also die komplexe Zahl mit geändertem Vorzeichen.

In einem String-Literal können mithilfe von geschweiften Klammern Objekte (hier: `a`), Eigenschaften von Objekten (hier: `a.real` und `a.imag`) oder Funktionsaufrufe (hier: `type(a)`) eingebettet werden.

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 werden 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 3.3.8) wird ein bestimmtes Bit

im Ergebnis nur gesetzt, falls 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 Gr, die sich rechts neben dem Leerzeichen befindet. Ähnlich wie beim logischen Operator `or` (siehe ebenfalls Abschnitt 3.3.8) wird ein bestimmtes Bit im Ergebnis gesetzt, falls dieses Bit in einer der beiden Zahlen oder in beiden Zahlen gesetzt ist. Diese Operation wird auch für jedes einzelne Bit durchgeführt.

Möchten Sie ein einzelnes Bit 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, falls dieses Bit nur in einer der beiden Zahlen gesetzt ist. Ist das Bit in beiden Zahlen gesetzt, wird das Ergebnisbit 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 mit Brüchen rechnen bzw. Informationen über Brüche zur Verfügung stellen. Zu diesem Zweck wird die Klasse `Fraction` aus dem Modul `fractions` (deutsch: Brüche) genutzt.

Es folgt ein Beispielprogramm, in dem einige Berechnungen mit Brüchen gemäß den Regeln der mathematischen Bruchrechnung durchgeführt werden, siehe *https://de.wikipedia.org/wiki/Bruchrechnung*. Am Ende folgen einige Vergleiche zwischen Brüchen:

```
import fractions

b1 = fractions.Fraction(3, -2)
b2 = fractions.Fraction(1, 4)

b3 = b1 * b2
print(f"{b1} * {b2} = {b3}")
print(f"{b1} / {b2} = {b1 / b2}")
print(f"{b1} + {b2} = {b1 + b2}")
print(f"{b1} - {b2} = {b1 - b2}")
print(f"{b1} + {b2} * {b3}= {b1 + b2 * b3}")
print(f"({b1} + {b2}) * {b3}= {(b1 + b2) * b3}")

b4 = fractions.Fraction(-30, 20)
print(f"{b1} == {b4}: {b1 == b4}")
print(f"{b2} > {b4}: {b2 > b4}")
print(f"{b2} < {b4}: {b2 < b4}")
```

Listing 4.11 Datei »zahl_bruch_rechnen.py«

Ein Bruch wird mithilfe von zwei ganzen Zahlen als Zähler und Nenner erstellt und mit einem Forwardslash als Bruchstrich dargestellt.

Die Funktion `Fraction()` aus dem Modul `fractions` bietet verschiedene Möglichkeiten, einen Bruch zu erstellen, genauer gesagt ein `Fraction`-Objekt. Bei `Fraction()` handelt es sich um den Konstruktor der Klasse `Fraction`. Damit wird ein Objekt der Klasse erzeugt und eine Referenz auf dieses Objekt zurückgeliefert.

`Fraction`-Objekte werden bei ihrer Erzeugung automatisch gekürzt. Besitzt der Bruch genau ein negatives Vorzeichen im Zähler oder im Nenner, wird es vor den Zähler gesetzt. Besitzt der Bruch zwei negative Vorzeichen, wird der Bruch mit –1 multipliziert, und die negativen Vorzeichen verschwinden.

Die Ausgabe des Programms:

```
-3/2 * 1/4 = -3/8
-3/2 / 1/4 = -6
-3/2 + 1/4 = -5/4
-3/2 - 1/4 = -7/4
```

```
-3/2 + 1/4 * -3/8= -51/32
(-3/2 + 1/4) * -3/8= 15/32
-3/2 == -3/2: True
1/4 > -3/2: True
1/4 < -3/2: False
```

Im nachfolgenden Programm werden einige Eigenschaften und Methoden des `Fraction`-Objekts verdeutlicht:

```
import fractions

b1 = fractions.Fraction(12, 28)
print("Fraction-Objekt:", b1, type(b1))
print(f"Zähler: {b1.numerator}, Nenner: {b1.denominator}")
print(f"Wert: {b1:.12f}")
print()

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

b3 = fractions.Fraction("350_000 /280_000")
print("Aus String:", b3)
print("Ganzzahlig:", b3.is_integer())
print(f"Formatiert: {b3:.3f}")
```

Listing 4.12 Datei »zahl_bruch_objekt.py«

Die Ausgabe lautet:

```
Fraction-Objekt: 3/7 <class 'fractions.Fraction'>
Zähler: 3, Nenner: 7
Wert: 0.428571428571

Zahl: 2.375
Fraction-Objekt: 19/8

Aus String: 5/4
Ganzzahlig: False
Formatiert: 1.250
```

Zähler und Nenner des Bruchs stehen in den Eigenschaften `numerator` und `denominator` einzeln zur Verfügung. Seit Python 3.12 kann ein Bruch mithilfe eines formatierten String-Literals ausgegeben werden. Hier wird der Wert des Bruchs mit 12 Nachkommastellen dargestellt. Mehr zur Formatierung von Zahlen in Abschnitt 5.2.2.

Ein Bruch kann aus einer Zahl mit Nachkommastellen erstellt werden. Damit wird aus der Zahl `2.375` das `Fraction`-Object `19/8`.

Ein Bruch kann aus einer Zeichenkette erstellt werden, die zwei ganze Zahlen enthält, die mit einem Forwardslash voneinander getrennt sind. Die Zeichenkette kann seit Python 3.12 Leerzeichen enthalten. Seit Python 3.11 können die beiden Zahlen mithilfe von Unterstrichen lesbarer gemacht werden, siehe auch Abschnitt 4.1.2.

Seit Python 3.12 gibt es die Methode `is_integer()`. Sie liefert die Information, ob der Wert eines Bruchs ganzzahlig ist. 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 4.13 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. Das bedeutet, dass derjenige Bruch gesucht wird, 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 $3{,}65684301429 \times 10^{-5}$.

Der relative Fehler 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 4.1.6) und wird im Modul `fractions` als *deprecated* (deutsch etwa: veraltet, ungültig) 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 sind Objekte des Typs `str`. Sie werden in einfache, doppelte oder dreifache doppelte Hochkommata gesetzt. Mithilfe einer `for`-Schleife können Sie auf die Elemente einer Zeichenkette zugreifen.

Ein Beispiel:

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

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

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

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

Listing 4.14 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 Zeichenkette `ta` 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 der ersten `for`-Schleife wird hier auf die Elemente zugegriffen.

Die laufende Nummer eines Elements wird mit *Index* bezeichnet. Auf das Element mit dem gewünschten Index können Sie mithilfe der rechteckigen Klammern `[` und `]` zugreifen. Diese erreichen Sie mithilfe der Taste Alt Gr rechts neben dem Leerzeichen, unter macOS gegebenenfalls mit Alt + 5 und Alt + 6.

Hier geschieht dies in der zweiten `for`-Schleife. Sie wird mithilfe der Schleifenvariablen `i` durchlaufen, die nacheinander die Werte von 0 bis (Länge der Zeichenkette) –1 annimmt. Es werden also nacheinander die Elemente `tx[0]`, `tx[1]` ... ausgegeben, bis zum Element `tx[Länge-1]`.

Die Zeichenkette `tb` wird in einfachen Hochkommata angegeben. Die Zeichenkette `tc` erstreckt sich über mehrere Zeilen. Zu diesem Zweck wird sie in dreifachen doppelten Hochkommata notiert.

Die Zeichenkette `td` 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. Der Operator `not in` stellt das Umgekehrte fest. 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 4.15 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 `linie` gespeichert.

4.2.3 Slices

Bereiche von Sequenzen werden als *Slices* bezeichnet. Den Einsatz von Slices verdeutliche ich am Beispiel eines Strings. 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 am Anfang der Sequenz bei 0. Alternativ können Sie auch einen negativen Index nutzen; er beginnt am Ende der Sequenz mit –1.

Index	0	1	2	3	4	5	6	7	8	9
Element	H	a	l	l	o		W	e	l	t
Negativer Index	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

Tabelle 4.1 Sequenz mit Index

Ein Slice wird, wie ein einzelnes Element, mithilfe von rechteckigen Klammern angegeben. Er kann mit einem Start-Index beginnen, gefolgt von einem Doppelpunkt und einem End-Index. Ein Beispiel:

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

print("[2:7]:", tx[2:7])
print("[:7]:", tx[:7])
print("[2:]:", tx[2:])
print("[:]:", tx[:])
print("[1]:", tx[1])
print("[1:-2]:", tx[1:-2])
print()

print("[2:7:2]:", tx[2:7:2])
s = slice(2, 7, 2)
print("[2:7:2]:", tx[s])
a = 2
b = 7
print("[2:7:2]:", tx[a:b:a])
```

Listing 4.16 Datei »text_slice.py«

Es wird die folgende Ausgabe erzeugt:

```
Text: Hallo Welt
[2:7]: llo W
[:7]: Hallo W
[2:]: llo Welt
[:]: Hallo Welt
[1]: a
[1:-2]: allo We

[2:7:2]: loW
[2:7:2]: loW
[2:7:2]: loW
```

Die Erläuterung der verschiedenen Slices:

- Slice `[2:7]`: Der Bereich erstreckt sich von dem Element, das durch den Start-Index gekennzeichnet wird, bis direkt vor das Element, das durch den End-Index gekennzeichnet wird.
- Slice `[:7]`: 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, ergibt sich kein Slice, sondern nur ein einzelnes Element.
- Slice `[1:-2]`: Wird ein Index mit einer negativen Zahl angegeben, wird für diesen Index vom Ende aus gemessen, beginnend bei –1.
- Slice `[2:7:2]`: Nach dem Start-Index und dem End-Index kann eine Schrittweite angegeben werden, mit der die einzelnen Elemente für den Slice zusammengestellt werden.
- Die eingebaute Funktion `slice()` liefert ein `slice`-Objekt. Dieses Objekt kann zur Ermittlung eines Slice einer Sequenz genutzt werden.
- Start-Index, End-Index und Schrittweite können mithilfe von Variablen angegeben werden.

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 4.17 Datei »text_aenderbarkeit.py«

Die Ausgabe lautet:

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

Es wird versucht, einen Teil der Zeichenkette durch einen Slice zu ersetzen. Das endet in einem Fehler.

Die einzige Möglichkeit zur Änderung einer Variablen, die eine Zeichenkette enthält, ist die Zuweisung einer neuen Zeichenkette an dieselbe Variable. Die neue Zeichenkette 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 4.18 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 in einem analysierten Text vorkommt. Kommt der Suchtext nicht vor, wird `-1` geliefert.

Sie können bei der Methode `find()` optional einen zweiten Parameter angeben, der die Position bestimmt, ab der gesucht werden soll. Das können Sie nutzen, um mithilfe einer `while`-Schleife alle Vorkommen eines Suchtexts zu finden. Die Bedingung der Schleife 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.

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 zur erneuten Zuweisung 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

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

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

Listing 4.19 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.

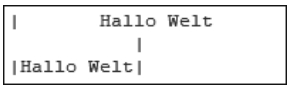

Abbildung 4.1 Löschen von Zeichen am Anfang und am Ende des Textes

4.2.7 Text zerlegen

Die Methode `split()` dient zum Zerlegen eines Texts in eine Liste von Teiltexten. Das nachfolgende Beispiel zeigt zwei Anwendungen:

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

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

Listing 4.20 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
```

Bei der Methode `split()` wird standardmäßig 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. Dieses Zeichen können Sie als Parameter der Methode `split()` verwenden, um den Datensatz in einzelne Elemente 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 4.21 Datei »text_konstanten.py«

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

Eigenschaften und Methoden für Zeichenketten stehen in Python standardmäßig zur Verfügung. Zur Nutzung der Konstanten des Moduls `string` muss dieses importiert werden.

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. Sie können es zunächst übergehen und bei Bedarf nachschlagen.

Die bisher behandelten Zeichenketten sind Objekte des Typs `str`. Sie werden aus dem umfangreichen Zeichensatz der Unicode-Zeichen gebildet.

Objekte des seltener genutzten Typs `bytes` werden aus einem kleineren Zeichensatz gebildet. Ihr Zeichencode liegt nur im Bereich von 0 bis 255. Jedes Zeichen wird nur in einem einzelnen Byte gespeichert. Sie können `bytes`-Objekte mithilfe von Byteliteralen erzeugen oder mithilfe der eingebauten Funktion `bytes()`.

Byteliterale 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 4.22 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 zusammen mit ihrem Typ ausgegeben. Beachten Sie beim Byteliteral das vorangestellte `b`.

Zur Umwandlung eines `str`-Objekts in ein `bytes`-Objekt wird die eingebaute Funktion `bytes()` genutzt. Dabei muss die Codierung des `str`-Objekts angegeben werden, hier UTF-8.

Zur Umwandlung eines `bytes`-Objekts in ein `str`-Objekt wird die Methode `decode()` verwendet.