Bausteine Computergestützter Datenanalyse

Lukas Arnold Simone Arnold Florian Bagemihl
Matthias Baitsch Marc Fehr Maik Poetzsch
Sebastian Seipel

2025-03-14

Inhaltsverzeichnis

Werkzeugbaustein Python **5**
Voraussetzungen . . . 5
Lernziele . . . 6

1 Einleitung: Datenanalyse mit Python **7**
1.1 Grundbegriffe der objektorientierten Programmierung . . . 8
1.1.1 Klassen, Typen, Objekte, Attribute . . . 8
1.2 Programmcode formatieren . . . 11
1.3 Ausgabe formatieren . . . 13
1.4 ganze Zahlen . . . 14
1.5 Gleitkommazahlen . . . 14
1.6 Zeichenfolgen . . . 15
1.7 Aufgaben . . . 15

2 Datentypen **16**
2.1 Zahlen . . . 16
2.1.1 Ganzzahlen . . . 16
2.1.2 Gleitkommazahlen . . . 17
2.2 Arithmetische Operatoren . . . 18
2.3 Aufgaben Zahlen . . . 19
2.4 Boolsche Werte . . . 20
2.5 Logische Operatoren . . . 21
2.6 Aufgaben boolsche Werte . . . 23
2.7 Zeichenfolgen . . . 23
2.8 Operationen mit Zeichenfolgen . . . 24
2.9 Aufgaben Zeichenfolgen . . . 25
2.10 Variablen . . . 25
2.10.1 Weitere Zuweisungsoperatoren . . . 27
2.10.2 Benennung von Variablen . . . 28
2.11 Aufgaben Variablen . . . 29

3 Funktionen: Grundlagen **31**
3.1 Funktionen und Methoden . . . 32
3.1.1 Funktionen . . . 32
3.1.2 Methoden . . . 33

3.2 Parameter . . . 35
3.3 Aufgaben Funktionen . . . 38
3.4 Flusskontrolle . . . 38
3.4.1 Abzweigungen . . . 38
3.4.2 Schleifen . . . 41
3.4.3 Ausnahmebehandlung . . . 45
3.5 Aufgaben Flusskontrolle . . . 47

4 Sammeltypen **49**
4.1 Listen . . . 49
4.1.1 Slicing: der Zugriffsoperator [] . . . 50
4.1.2 Listenmethoden . . . 52
4.1.3 Aufgaben Listen . . . 56
4.2 Tupel . . . 56
4.2.1 Tupel kopieren . . . 57
4.3 Mengen . . . 58
4.3.1 Mengen kopieren . . . 59
4.4 Dictionaries . . . 60
4.4.1 Dictionaries kopieren . . . 60
4.5 Übersicht Sammeltypen . . . 61
4.6 Löschen: das Schlüsselwort del . . . 62
4.7 Funktionen . . . 63
4.8 Operationen: Verwendung von Schleifen . . . 63
4.9 Aufgaben Sammeltypen . . . 63

5 Eigene Funktionen definieren **67**
5.1 Syntax . . . 67
5.2 Optionale Parameter . . . 68
5.3 Rückgabewert(e) . . . 70
5.4 Aufgaben Funktionen definieren . . . 70

6 Dateien lesen und schreiben **73**
6.1 Dateiobjekte . . . 73
6.1.1 Dateipfad . . . 73
6.1.2 Zugriffsmodus . . . 74
6.1.3 Dateiinhalt ausgeben . . . 77
6.2 Dateien einlesen . . . 77
6.3 Aufgabe Dateien einlesen . . . 80
6.4 Daten interpretieren . . . 81
6.5 for-Schleife mit break . . . 82
6.6 Methode dateiobjekt.readline() . . . 83
6.7 Aufgabe Daten interpretieren . . . 86
6.8 Einlesen als Liste . . . 87

6.9 Dateien schreiben 89
6.10 Aufgabe Dateien schreiben 89

7 Module und Pakete importieren **90**
7.1 import as 92
7.2 Kleine Modulübersicht 93

Werkzeugbaustein Python

Bausteine Computergestützter Datenanalyse von Lukas Arnold, Simone Arnold, Florian Bagemihl, Matthias Baitsch, Marc Fehr, Maik Poetzsch und Sebastian Seipel. Werkzeugbaustein Python von Maik Poetzsch ist lizensiert unter CC BY 4.0. Das Werk ist abrufbar auf GitHub. Ausgenommen von der Lizenz sind alle Logos Dritter und anders gekennzeichneten Inhalte. 2025

Zitiervorschlag

Arnold, Lukas, Simone Arnold, Matthias Baitsch, Marc Fehr, Maik Poetzsch, und Sebastian Seipel. 2025. „Bausteine Computergestützter Datenanalyse. Werkzeugbaustein Python“. https://github.com/bausteine-der-datenanalyse/w-python.

BibTeX-Vorlage

```
@misc{BCD-w-python-2025,
 title={Bausteine Computergestützter Datenanalyse. Werkzeugbaustein Python},
 author={Arnold, Lukas and Arnold, Simone and Baitsch, Matthias and Fehr, Marc and Poetzsch,
 year={2025},
 url={https://github.com/bausteine-der-datenanalyse/w-python}}
```

Voraussetzungen

Keine Voraussetzungen, hilfreich ist der w-Pseudocode

Querverweis auf:

- w-NumPy
- w-Pandas

Lernziele

In diesem Bausteine werden die Grundzüge der Programmierung mit Python vermittelt. In diesem Baustein lernen Sie …

- Grundbegriffe der objektorientierten Programmierung kennen.
- Python-Code zu schreiben, Variablen zu erstellen, Operationen durchzuführen und die Ausgabe zu formatieren.
- die Dokumentation zu lesen und zu verwenden
- die exklusive Zählweise von Python kennen.
- den Unterschied zwischen Funktionen und Methoden kennen und wie eigene Funktionen geschrieben werden.
- Module und Pakete laden

Querverweis auf:

- Methodenbaustein Einlesen strukturierter Datensätze

1 Einleitung: Datenanalyse mit Python

Platzhalter Einleitungsvideo: Stärken und Schwächen von Python

Stärken:
Einstieg: sehr einfach, da die Sprache genau dafür entworfen wurde und mit wenigen Strukturen und Funktionalitäten auskommt
Generalität: wird weltweit in allen Disziplinen eingesetzt ohne domänenspezifische Vereinfachungen zu nutzen
Erweitbarkeit: riesige Anzahl bereits verfügbarer Module mit sehr umfangreichen Funktionsumfängen. Die Installation klappt in der Regel reibungslos und schnell
Verfügbarkeit: freie Interpreter für alle populären Computersysteme, sowohl der Interpreter als auch fast alle Module sind quelloffen und frei verfügbar
Schwächen:
Python wurde nicht speziell für die Datenanalyse gemacht. Die Sprache und die verfügbaren Module entwickeln sich dynamisch, sodass viele Dinge nicht einheitlich umgesetzt ist. Die Dokumentation ist sehr umfangreich, aber nicht sonderlich zugänglich

Die Erzeugung und Auswertung von Daten ist ein zentraler Bestandteil wissenschaftlicher Forschung. Die computergestützte Datenanalyse ermöglicht es, große Datenmengen (teil-)automatisiert auszuwerten. Gut lesbare Skriptsprachen wie Python sorgen für eine nachvollziehbare Datenverarbeitung und ermöglichen es, Analysen “auf Knopfdruck” zu wiederholen oder anzupassen.

Abbildung 1.1: Logo der Programmiersprache Python

Python Logo von Python Software Foundation steht unter der GPLv3. Die Wort-Bild-Marke ist markenrechtlich geschützt: https://www.python.org/psf/trademarks/. Das Werk ist abrufbar auf wikimedia. 2008

Python kommt als schlichte Konsole daher. Python-Code wird in die Konsole eingegeben oder in einer reinen Textdatei, dem Skript, gespeichert. Der Programmcode wird von einem sogenannten Interpreter ausgeführt. Der Interpreter übersetzt die Programmanweisungen des Skripts in Maschinencode für das jeweilige Computersystem. Dadurch kann das Skript auf verschiedenen Computersystemen ausgeführt werden. Moderne Python-Interpreter sind nicht auf durch ASCII darstellbare Zeichen limitiert und können auch mit Zeichen aus dem Format UTF-8 umgehen, das das ASCII-Format z. B. um deutsche Sonderzeichen erweitert.

Zahlreiche Funktionen wie Codeformatierung, Codevervollständigung und Fehleranalyse werden durch eine sogenannte integrierte Entwicklungsumgebung (Integrated Development Environment, IDE) bereitgestellt.

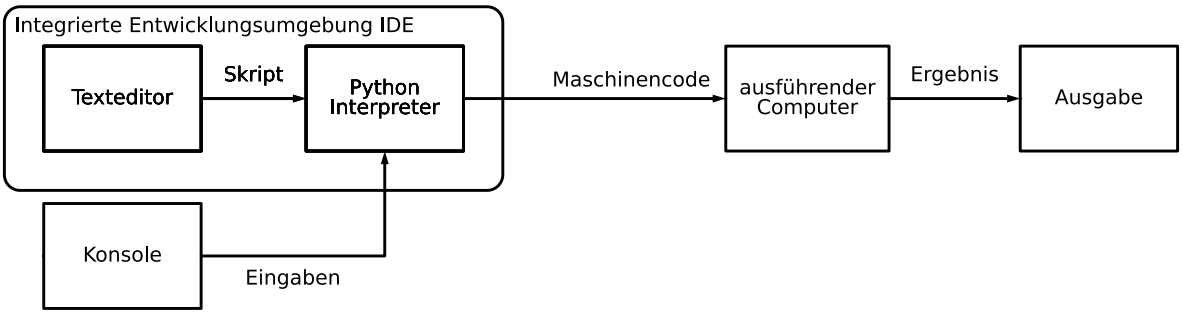


Abbildung 1.2: Programmentwicklung mit Python

1.1 Grundbegriffe der objektorientierten Programmierung

Python ist eine objektorientierte Programmiersprache. Die objektorientierte Programmierung ist ein System, um Ordnung in komplexe Computerprogramme zu bringen. In diesem Abschnitt werden die Grundbegriffe der objektorientierten Programmierung mit Python vermittelt. Sie erfahren, was der Unterschied zwischen einem Objekt, einer Klasse und dem Datentyp ist.

1.1.1 Klassen, Typen, Objekte, Attribute

Ein Pythonprogramm besteht aus verschiedenen Elementen: Operatoren und Operanden, Funktionen und Methoden, Werten und Variablen und vielem mehr. Alles in Python ist ein Objekt.

Jedes Objekt gehört zu einer Klasse, beispielsweise zur Klasse der Ganzzahlen. Die Klasse bestimmt als Blaupause die *Eigenschaften* und das *Verhalten* des Objekts - etwa welche Daten gespeichert und welche Operationen ausgeführt werden können. Ein kurzes Beispiel: Abhängig von ihrer Klasse, verhalten sich Objekte anders mit dem Operator `+`.

```
print(type(2), 2 + 2, "Ganzzahlen werden addiert.")
print(type('a' and '2'), 'a' + '2', "Zeichen werden verkettet.")
print(type(True), True + True, "Wahrheitswerte werden addiert.")
```

```
<class 'int'> 4 Ganzzahlen werden addiert.
<class 'str'> a2 Zeichen werden verkettet.
<class 'bool'> 2 Wahrheitswerte werden addiert.
```

Das liegt daran, dass das Verhalten des Operators `+` für die Klassen Ganzzahlen ('int'), Zeichenfolgen ('str') und Boolesche Werte ('bool') definiert ist. Anders verhält es sich mit `None`, einer Klasse, mit der nicht existente Werte verarbeitet werden:

```
print(None + None)
```

```
unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

Python kennt sehr viele Klassen. In Python werden Klassen (class) auch Typen (type) genannt. In früheren Versionen von Python waren Klassen und Typen noch verschieden. Inzwischen gibt es diesen Unterschied nicht mehr, beide Begriffe kommen aber noch in der Sprache vor.

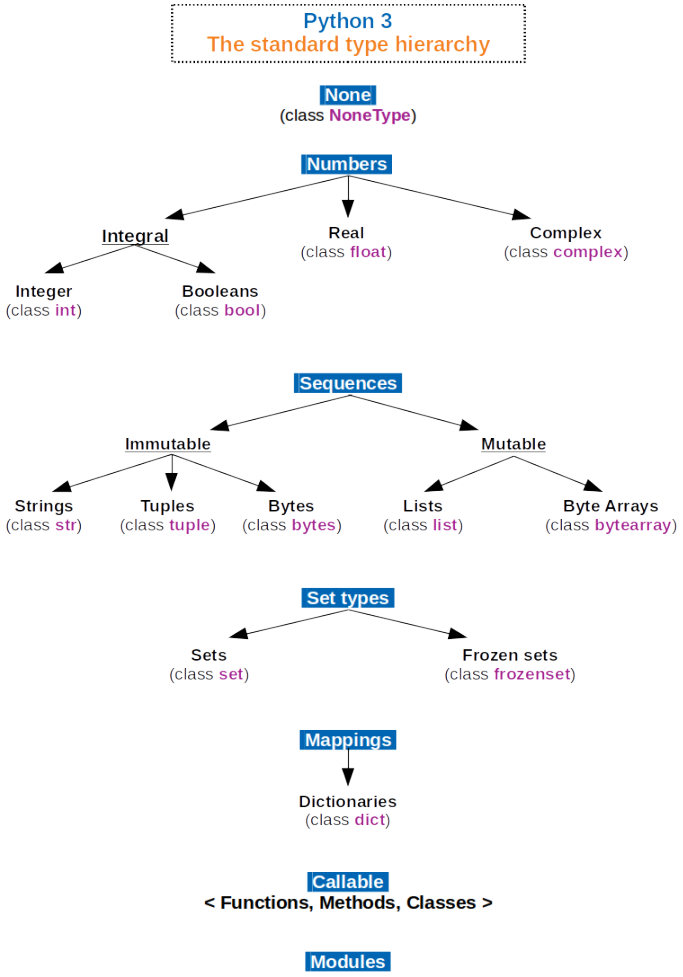


Abbildung 1.3: Datentypen in Python

Python 3. The standard type hierarchy. von ist lizensiert unter CC BY SA 4.0 und abrufbar auf wikimedia. 2018

Zu welcher Klasse bzw. zu welchem Typ ein Objekt gehört, kann mit der Funktion `type()` ermittelt werden.

```
print(type(print))
```

```
<class 'builtin_function_or_method'>
```

Attribute speichern Eigenschaften eines Objekts. Sie treten in der Form `objekt.attribut` auf und werden ohne nachfolgende Klammern aufgerufen. Attribute haben an dieser Stelle der Einführung keine praktische Bedeutung, werden uns aber später wieder begegnen. Eine zweite Form der Attribute ist die Methode. Methoden sind Funktionen, die zu einer bestimmten Klasse gehören. Methoden haben die Form `objekt.methode()`, werden also mit nachfolgenden Klammern aufgerufen. Die Benutzung von Funktionen und Methoden lernen wir in den kommenden Kapiteln kennen. Wie Sie die verfügbaren Attribute und Methoden eines Objekts bestimmen, erfahren Sie in Beispiel 2 und Beispiel 1.

1.2 Programmcode formatieren

Bei der Formatierung von Python-Code müssen nur wenige Punkte beachtet werden. Um mit Python eine Ausgabe zu erzeugen, wird die Funktion `print(eingabe)` verwendet. Diese Funktion gibt das Argument eingabe aus.

1. Zahlen und Operatoren können direkt eingegeben werden. Text, genauer eine Zeichenfolge, muss in einfache oder doppelte Anführungszeichen gesetzt werden, andernfalls interpretiert Python diesen als Namen eines Objekts. Zeichenfolgen können neben Buchstaben, Sonderzeichen und Zahlen enthalten.

```
print(1 + 2)
print('123: Hallo Welt!')
text_variable = 'Hallo Python!'
print(text_variable)
```

```
3
123: Hallo Welt!
Hallo Python!
```

2. Kommentare werden mit einer vorangestellten Raute `#` gekennzeichnet. Kommentare markieren Code, der nicht ausgeführt werden soll, oder Erläuterungen.

```
# Ein reiner Kommentar
# print("Python ist großartig!") # auskommentierter Code, gefolgt von einem Kommentar
print("Python ist ziemlich gut.") # auszuführender Code, gefolgt von einem Kommentar
```

```
Python ist ziemlich gut.
```

3. Ausdrücke müssen in einer Zeile stehen. Längere Ausdrücke können mit dem Backslash `\` über mehrere Zeilen fortgesetzt werden (hinter `\` darf keine `#` stehen). Innerhalb von Funktionen wie zum Beispiel `print()` können Zeilen nach jedem Komma fortgesetzt werden.

```
variable1 = 15
variable2 = 25

# Zeilenfortsetzung mit \
summe = variable1 + \
    variable2

# Zeilenfortsetzung innerhalb einer Funktion
print(variable1,
 variable2,
  summe)
```

```
15 25 40
```

In dem obenstehenden Beispiel werden Variablen angelegt. Mit Variablen beschäftigten wir uns im nächsten Kapitel. Trotzdem möchte ich Sie bitten, sich variable1 und variable2 nocheinmal kurz anzusehen. Wir kommen später darauf zurück.

4. Die Anzahl der Leerzeichen zwischen Operanden und Operatoren kann beliebig sein.

```
print(1+0, 1 + 1, 1 +                  2)
```

```
1 2 3
```

5. Die Einrückung mit Leerzeichen kennzeichnet einen zusammengehörigen Code-Block. Innerhalb eines Code-Blocks muss immer die gleiche Anzahl Leerzeichen verwendet werden. Es muss mindestens ein Leerzeichen gesetzt werden, ansonsten ist die Anzahl der Leerzeichen beliebig. Üblich sind 2 oder 4 Leerzeichen.
   Die folgende for-Schleife führt alle Anweisungen im eingerückten Ausführungsblock aus. Die anschließende, nicht eingerückte Zeile markiert den Beginn einer neuen, nicht zur Schleife gehörigen Anweisung.

```
for i in range(2):
    print(variable1)
    print(variable2)
print(summe)
```

```
15
25
15
25
40
```

1.3 Ausgabe formatieren

Mit sogenannten `f-Strings` können formatierte Zeichenfolgen erstellt werden. Formatierte Zeichenfolgen werden mit einem den Anführungsstrichen vorangestellten `f` erstellt. Werte und Variablen können durch Platzhalter eingesetzt werden, die mit geschweiften Klammern `{}` angegeben und mit Formattierungsinformationen versehen werden. Das Formatierungsformat innerhalb der geschweiften Klammer ist vereinfacht dargestellt:

```
{Variablenname:PlatzbedarfAusgabetyp}
```

Ein f-String mit Platzhaltern ohne Formatierungsinformationen:

```
zahl1 = 5
zahl2 = 7
verhältnis = zahl1 / zahl2
print(f"Das Verhältnis von {zahl1} zu {zahl2} ist {verhältnis}.")
```

```
Das Verhältnis von 5 zu 7 ist 0.7142857142857143.
```

Die Anzahl der darzustellenden Nachkommastellen kann wie folgt festgelegt werden: `{verhältnis:.2f}`.

- `:` leitet die Formatierungsbefehle ein.
- `.` gibt an, dass Formatierungsinformationen für die Darstellung hinter dem Dezimaltrennzeichen folgen.
- `2` ist der Wert für die darzustellenden Nachkommastellen.
- `f` spezifiziert die Darstellung einer Gleitkommazahl 'float'.

```
print(f"Das Verhältnis von {zahl1} zu {zahl2} ist {verhältnis:.2f}.")
```

```
Das Verhältnis von 5 zu 7 ist 0.71.
```

Das gleiche ist mit einem Wert möglich:

```
print(f"Das Verhältnis ist genauer {0.7142857142857143:.3f}.")
```

```
Das Verhältnis ist genauer 0.714.
```

Ein Wert für die insgesamt darzustellenden Stellen wird vor dem Dezimaltrennzeichen übergeben `{verhältnis:7.2f}` bzw. inklusive führender Nullen `{verhältnis:07.2f}`:

```
print(f"Das Verhältnis von {zahl1} zu {zahl2} ist {verhältnis:7.2f}.")
print(f"Das Verhältnis ist genauer {0.7142857142857143:07.3f}.")
```

```
Das Verhältnis von 5 zu 7 ist    0.71.
Das Verhältnis ist genauer 000.714.
```

Das Dezimaltrennzeichen zählt als eine Stelle.

Häufig verwendete Formatierung sind:

1.4 ganze Zahlen

Ganze Zahlen haben den Ausgabetyp `d`.

Formatierung	Ausgabe
`nd`	n-Stellen werden für die Ausgabe verwendet
`0nd`	Ausgabe von n-Stellen, wobei die Leerstellen mit Nullen aufgefüllt werden.
`+d`	Ausgabe des Vorzeichens auch bei positiven Zahlen

1.5 Gleitkommazahlen

Gleitkommazahlen haben die Ausgabetypen `f` und `e`.

Formatierung	Ausgabe
`.mf`	m-Stellen werden für die Nachkommastellen genutzt.
`n.mf`	Insgesamt werden n-Stellen verwendet, wobei m-Stellen für die Nachkommastellen genutzt werden.
`n.me`	Genauso, aber die Ausgabe erfolgt in exponentieller Schreibweise.

1.6 Zeichenfolgen

Zeichenfolgen haben den Ausgabetyp `s`.

Formatierung	Ausgabe
`ns`	Insgesamt werden n-Stellen verwendet.
`<ns`, `>ns`, `^ns`	Genauso, jedoch wird die Zeichenfolge linksbündig, rechtsbündig bzw. zentriert platziert.

Eine Auflistung aller verfügbaren Ausgabetypen findet sich in der Python Dokumentation.

1.7 Aufgaben

1. Gleitkommazahlen können natürlich, z. B. {1015.39:12.4f}, oder wissenschaftlich, {1015.39:e}, dargestellt werden.
   - Verändern Sie die natürliche Schreibweise so, dass nur noch eine Stelle nach dem Komma angezeigt wird. Was fällt auf?
   - Verändern Sie die wissenschaftliche Schreibweise so, dass anstelle von `e` die Zehnerbasis als `E` geschrieben wird.
2. Wandeln Sie die Zahl 1015.39 in eine Zeichenfolge um und stellen Sie diese mit 12 Stellen rechtsbündig dar.
3. Geben Sie mit Hilfe der formatierten Zeichenfolge eine Tabelle aus, welche die Spalten x, x^2 und x^3 für ganze Zahlen zwischen -2 und 2 auflistet.

Die Musterlösung kann Marc machen.

> Tip 1: Musterlösung Ausgabe

(Arnold 2023a)

(Arnold 2023b)

(Arnold 2023c)

2 Datentypen

Objekte, die Daten speichern, haben einen Datentyp. Der Datentyp gibt an, wie die gespeicherten Werte von Python interpretiert werden sollen. Beispielsweise kann der Wert "1" in Python ein Zeichen, eine Ganzzahl, einen Wahrheitswert, den Monat Januar, den Wochentag Dienstag oder die Ausprägung einer kategorialen Variablen repräsentieren.

In diesem Abschnitt werden die für die Datenanalyse wichtigsten Datentypen vorgestellt.

2.1 Zahlen

Zu den Zahlen gehören Ganzzahlen, boolsche Werte (Wahrheitswerte), Gleitkommazahlen sowie komplexe Zahlen (die hier nicht näher vorgestellt werden).

2.1.1 Ganzzahlen

Ganzzahlen werden standardmäßig im Dezimalsystem eingegeben und können positiv oder negativ sein.

```
print(12, -8)
```

```
12 -8
```

Darüber hinaus können Ganzzahlen auch in anderen Basen angegeben werden:

- Dualsystem: Ziffern `0` und `1` mit dem Präfix `0b`

```
print(0b1000, "plus", 0b0000, "plus", 0b0010, \
      "plus", 0b0001, "ist", 0b1011)
```

```
8 plus 0 plus 2 plus 1 ist 11
```

- Oktalsystem: Ziffern `0` bis `7` mit dem Präfix `0o`

```
print(0o7000, "plus", 0o0700, "plus", 0o0020, \
      "plus", 0o0000, "ist", 0o7720)
```

```
3584 plus 448 plus 16 plus 0 ist 4048
```

- Hexadezimalsystem: Ziffern `0` bis `F` mit dem Präfix `0x`

```
print(0xF000, "plus", 0x0200, "plus", 0x00A0, \
      "plus", 0x0001, "ist", 0xF2A1)
```

```
61440 plus 512 plus 160 plus 1 ist 62113
```

2.1.2 Gleitkommazahlen

Gleitkommazahlen werden entweder mit dem Dezimaltrennzeichen `.` oder in Exponentialschreibweise angegeben. Gleitkommazahlen haben den Typ `float`.

```
print(120.6, 1206e-1, 12060e-2, "\n")

print("Beim Lottogewinn in Exponentialschreibeweise zählt das Vorzeichen.")
print(1e-3, "oder", 1e+3)
```

```
120.6 120.6 120.6

Beim Lottogewinn in Exponentialschreibeweise zählt das Vorzeichen.
0.001 oder 1000.0
```

Da Computer im Binärsystem arbeiten, können Dezimalzahlen nicht exakt gespeichert werden. Beispielsweise ist die Division von 1 durch 10 dezimal gleich 0.1. Binär ist 1_2 durch 1010_2 aber ein periodischer Bruch:

$$\frac{1_2}{1010_2} = 0,000\overline{1100}_2$$

Dezimalzahlen müssen deshalb als Bruch zweier Ganzzahlen approximiert werden (Der Binärbruch, der 0.1 annähert, ist in Dezimalschreibweise 3602879701896397 / 2^{55}). Dadurch kommt es vor, dass mehrere Gleitkommazahlen durch die selbe Binärapproximation repräsentiert werden. Python gibt zwar die jeweils kürzeste Dezimalzahl aus, da Berechnungen aber binär durchgeführt werden, kann sich bei Berechnungen die nächste Binärapproximation und damit die zugehörige kürzeste Dezimalzahl ändern (weitere Informationen in der Python Dokumentation).

```
print(0.1) # Die kürzeste Dezimalzahl zur Binärapproximation
print(format(0.1, '.17g')) # Die nächstlängere Dezimalzahl zur selben Binärapproximation
print(0.3 - 0.2) # binär gerechnet, ändert sich die Binärapproximation
```

```
0.1
0.10000000000000001
0.09999999999999998
```

In der praktischen Arbeit mit Python kommen deshalb gelegentlich auf den ersten Blick ungewöhnlich wirkende Ergebnisse vor.

```
print(0.1 + 0.2)
print(0.01 + 0.02)
print(0.001 + 0.002)
print(0.0001 + 0.0002)
print(0.00001 + 0.00002)
```

```
0.30000000000000004
0.03
0.003
0.00030000000000000003
3.0000000000000004e-05
```

2.2 Arithmetische Operatoren

Mit arithmetischen Operatoren können die Grundrechenarten verwendet werden. Das Ergebnis ist meist vom Typ `float`, außer, wenn beide Operanden vom Typ `int` sind und das Ergebnis als ganze Zahl darstellbar ist.

Operator	Beschreibung
`+`, `-`	Addition / Subtraktion
`*`, `/`	Multiplikation / Division
`//`, `%`	Ganzzahlige Division / Rest
`**`	Potenzieren

Werden mehrere Operatoren kombiniert, so muss deren Reihenfolge beachtet bzw. durch die Verwendung von Klammern `(1 + 2) * 3` hergestellt werden. Es gelten die gleichen Regeln

wie beim schriftlichen Rechen. Die vollständige Übersicht der Reihenfolge der Ausführung ist in der Pythondokumentation aufgeführt. Für die arithmethischen Operatoren gilt folgende, absteigende Reihenfolge.

Operator
`**`
`*` , `/` , `//` , `%`
`+` , `-`

Bei gleichrangigen Operationen werden diese von links nach rechts ausgeführt.

2.3 Aufgaben Zahlen

Lösen Sie die folgenden Aufgaben mit Python.

1. $4 + 2 * 4 = ?$
2. 2 hoch 12 = ?
3. Was ist der Rest aus 315 geteilt durch 4?
4. $\quad + \ 6 \ / \quad = ?$
5. Welche Dezimalzahl ist 11111101001_2 ?
6. $11111101001_2 \ / \ 101_2 = ?$
7. Welcher Kapitalertrag ist größer, wenn 1000 Euro angelegt werden?
   a) 20 Jahre Anlagedauer mit 3 Prozent jährlicher Rendite
   b) 30 Jahre Anlagedauer mit 2 Prozent jährlicher Rendite

Die Musterlösung kann Marc machen.

> Tip 2: Musterlösung Zahlen

2.4 Boolsche Werte

Die boolschen Werte `True` und `False` sind das Ergebnis logischer Abfragen, die wir später genauer kennenlernen. Sie nehmen auch die Werte `1` und `0` an und gehören in Python deshalb zu den Zahlen.

```
print("Ist 10 größer als 9?", 10 > 9)
print("Ist 11 kleiner als 10?", 11 < 10)
print("Ist 10 genau 10.0?", 10 == 10.0, "\n")

print("True und False können mit + addiert:", True + False)
print("... und mit * multipliziert werden:", True * False, "\n")
```

```
Ist 10 größer als 9? True
Ist 11 kleiner als 10? False
Ist 10 genau 10.0? True

True und False können mit + addiert: 1
... und mit * multipliziert werden: 0
```

Die Multiplikation von Wahrheitswerten ist nützlich, um mehrere logische Abfragen zu einem logischen UND zu kombinieren:

```
print("Ist 10 > 9 UND 10 > 8?", (10 > 9) * (10 > 8))
```

```
Ist 10 > 9 UND 10 > 8? 1
```

Die Funktion `bool()` gibt den Wahrheitswert eines Werts zurück.

```
print("Ist 10 > 9 UND > 8?", bool((10 > 9) * (10 > 8)))
```

```
Ist 10 > 9 UND > 8? True
```

Die meisten Werte in Python haben den Wahrheitswert `True`.

```
print(bool(1), bool(2), bool(2.4))
print(bool('a'), bool('b'), bool('ab'))
```

```
True True True
True True True
```

Neben `False` und `0` haben leere und nicht existierende Werte oder Objekte den Wahrheitswert `False`.

```
print(bool(False), bool(0))
print(bool("")) # eine leere Zeichenfolge
print(bool([])) # eine leere Liste
print(bool(())) # eine leeres Tupel
print(bool({})) # ein leeres Dictionary
print(bool(None)) # None deklariert einen nicht existenten Wert
```

```
False False
False
False
False
False
False
```

Die Sammeltypen Liste, Tupel und Dictionary lernen wir in den folgenden Kapiteln kennen. Boolsche Werte können die Ausführung von Programmcode steuern, indem sie wie `an` und `aus` wirken. So kann Programmcode mit einer `if`-Anweisung nur dann ausgeführt werden, wenn ein Sammeltyp auch Werte enthält.

```
meine_Liste = ['Äpfel', 'Butter']
if meine_Liste:
    print(f"Wir müssen {meine_Liste} einkaufen.")

meine_Liste = [] # eine leere Liste
if meine_Liste:
    print(f"Wir müssen {meine_Liste} einkaufen.")
```

```
Wir müssen ['Äpfel', 'Butter'] einkaufen.
```

2.5 Logische Operatoren

Zu den logischen Operatoren gehören die logischen Verknüpfungen `and`, `or` und `not`. Darüber hinaus können auch vergleichende Operatoren wie `>`, `>=` oder `==` verwendet werden. Das Ergebnis dieser Operationen ist vom Typ `bool`. Die Operatoren werden in folgender Reihenfolge ausgeführt. Gleichrangige Operatoren werden von links nach rechts ausgeführt.

Operator	Beschreibung
`&`	bitweises UND
`^`	bitweises XOR
`|`	bitweises ODER
`<`, `<=`, `>`, `>=`, `!=`, `==`	kleiner / kleiner gleich / größer als / größer gleich / ungleich / gleich
`not`	logisches NICHT
`and`	logisches UND
`or`	logisches ODER

Warning 1: Bitweise Operatoren

Besondere Vorsicht ist mit den bitweisen Operatoren geboten. Diese vergleichen Zahlen nicht als Ganzes, sondern stellenweise (im Binärsystem). Zu beachten ist, dass die bitweisen Operatoren Ausführungspriorität vor Vergleichsoperationen haben.

```
print(10 > 5 and 10 > 6)
print(10 > 5 & 10 > 6)
print(5 & 10)
print(10 > False > 6)
print((10 > 5) & (10 > 6))
```

```
True
False
0
False
True
```

Tipp

Im Allgemeinen werden die bitweisen Operatoren für die Datenanalyse nicht benötigt. Vermeiden Sie unnötige Fehler: Vermeiden Sie die bitweisen Operatoren `&`, `^` und `|`.
Die Operatoren `&`, `^` und `|` haben jedoch für Mengen (die wir später kennenlernen werden) eine andere Bedeutung. Auch in anderen Modulen kommt den Operatoren syntaktisch eine andere Bedeutung zu, bspw. im Paket Pandas bei der Übergabe mehrerer Slicing-Bedingungen `df = df[(Bedingung1) & (Bedingung2) | (Bedingung3)]`.

2.6 Aufgaben boolsche Werte

Lösen Sie die Aufgaben mit Python.

1. Ist das Verhältnis aus 44 zu 4.5 größer als 10?
2. Ist es wahr, dass 4.5 größer als 4 aber kleiner als 5 ist?
3. Ist 2 hoch 10 gleich 1024?
4. Sind die Zahlen 3, 4 und 5 ganzzahlig durch 2 teilbar ODER ungleich 10?
5. Prüfen Sie, ob eine Person den Vollpreis bezahlen muss, wenn Sie Ihr Alter angibt. Kinder unter 14 Jahren fahren kostenlos, Jugendliche zwischen 14 und 18 Jahren und Senior:innen ab 65 Jahren erhalten einen Rabatt.

Die Musterlösung kann Marc machen

Tip 3: Tipp

2.7 Zeichenfolgen

Zeichenfolgen (Englisch string) werden in Python in einfache oder doppelte Anführungszeichen gesetzt.

```
print('eine Zeichenfolge')
print("noch eine Zeichenfolge")
```

```
eine Zeichenfolge
noch eine Zeichenfolge
```

Innerhalb einer Zeichenfolge können einfache oder doppelte Anführungszeichen verwendet werden, solange diese nicht den die Zeichenfolge umschließenden Anführungszeichen entsprechen.

```
print('A sophisticated heap beam, which we call a "LASER".')
print("I've turned the moon into what I like to call a 'Death Star'.")
```

```
A sophisticated heap beam, which we call a "LASER".
I've turned the moon into what I like to call a 'Death Star'.
```

Das Steuerzeichen \ (oder Fluchtzeichen, escape character) erlaubt es, bestimmte Sondernzeichen zu verwenden.

```
print("Das Steuerzeichen \\ ermöglicht die gleichen \"Anführungszeichen\" in der Ausgabe von
print("Erst ein\tTabstopp, dann eine\nneue Zeile.")
```

```
Das Steuerzeichen \ ermöglicht die gleichen "Anführungszeichen" in der Ausgabe von print.
Erst ein    Tabstopp, dann eine
neue Zeile.
```

Ein vorangestelltes `r` bewirkt, dass das Steuerzeichen \ nicht verarbeitet wird (raw string literal). Dies ist beispielsweise bei der Arbeit mit Dateipfaden praktisch.

```
print("Die Daten liegen unter: C:\tolle_daten\nordpol\weihnachtsmann")
print(r"Die Daten liegen unter: C:\tolle_daten\nordpol\weihnachtsmann")
```

```
Die Daten liegen unter: C:  olle_daten
ordpol\weihnachtsmann
Die Daten liegen unter: C:\tolle_daten\nordpol\weihnachtsmann
```

2.8 Operationen mit Zeichenfolgen

Einige Operatoren funktionieren auch mit Daten vom Typ string.

```
# string + string
print('a' + 'b')

# string + Zahl
print(15 * 'a')

# logische Operatoren
print('a' < 'b', 'a' >= 'b', 'a' != 'b')
print('a' or 'b', 'a' and 'b')
```

```
ab
aaaaaaaaaaaaaaa
True False True
a b
```

2.9 Aufgaben Zeichenfolgen

Lösen Sie die Aufgaben mit Python.

1. Was passiert, wenn Sie die Zeichenfolge “Python” mit “for beginners” addieren?
2. Erzeugen Sie eine Zeichenfolge, die 10 mal die Zeichenfolge “tick tack”.
3. Welche Zeichenfolge ist kleiner, “Aachen” oder “Bern”. Warum ist das so, wie werden Zeichenfolgen verglichen?
4. Geben Sie den Dateipfad aus: “~\home\tobi\neue_daten”

Die Musterlösung kann Marc machen.

Tip 4: Musterlösung Aufgaben Zeichenfolgen

2.10 Variablen

Variablen sind Platzhalter bzw. Referenzen auf Daten. Die Zuweisung wird durch den Zuweisungsoperator = dargestellt. Der Name einer Variablen darf nur aus Buchstaben, Zahlen und Unterstrichen bestehen. Dabei darf das erste Zeichen keine Zahl sein.

```
var_1 = 'ABC'
var_2 = 26
var_3 = True

print("Das", var_1, "hat", var_2, "Buchstaben. Das ist", var_3)
```

```
Das ABC hat 26 Buchstaben. Das ist True
```

Variablen müssen in Python nicht initialisiert werden. Der Datentyp der Variablen wird durch die Zuweisung eines entsprechenden Werts festgelegt.

```
print("Die Variable var_1 hat den Typ", type(var_1))
print("Die Variable var_2 hat den Typ", type(var_2))
print("Die Variable var_3 hat den Typ", type(var_3))
```

```
Die Variable var_1 hat den Typ <class 'str'>
Die Variable var_2 hat den Typ <class 'int'>
Die Variable var_3 hat den Typ <class 'bool'>
```

Der Datentyp einer Variable ändert sich, wenn ihr ein neuer Wert eines anderen Datentyps zugewiesen wird.

```
var_1 = 100
print("Die Variable var_1 hat den Typ", type(var_1))
```

```
Die Variable var_1 hat den Typ <class 'int'>
```

Ebenso kann sich der Datentyp einer Variable ändern, um das Ergebnis einer Operation aufnehmen zu können. In diesem Beispiel kann das Ergebnis nicht in Form einer Ganzzahl gespeichert werden. Python weist dem Objekt var_1 deshalb den Datentyp 'float' zu.

```
var_1 = var_1 / 11
print("Die Variable var_1 hat den Typ", type(var_1))
```

```
Die Variable var_1 hat den Typ <class 'float'>
```

Python enthält Funktionen, um den Datentyp einer Variablen zu bestimmen und umzuwandeln. Die Funktion `type()` wurde in den Code-Beispielen bereits benutzt, um den Datentyp (bzw. die Klasse) von Objekten zu bestimmen. Die Umwandlung des Datentyps zeigt der folgende Code-Block.

```
a = 67
print(a, type(a))

b = a + 1.8
print(b, type(b), "\n")

print(f"Beachten Sie das Abschneiden der Nachkommastelle:\nUmwandlung in Ganzzahlen mit int()

print(f"Umwandlung in ASCII-Zeichen mit chr(): {( a := chr(a) ), ( b := chr(b) )}\n")

print(f"Umwandlung in eine ASCII-Zahl mit ord(): {( a := ord(a) ), ( b := ord(b) )}\n")

print(f"Umwandlung in Fließkommazahlen mit float(): {( a := float(a) ), ( b := float(b) )}\n

print(f"Umwandlung in Zeichen mit str(): {( a := str(a) ), ( b:= str(b) )}\n")

print(f"Umwandlung in Wahrheitswerte mit bool(): {bool(a), bool(b)}")
```

```
67 <class 'int'>
68.8 <class 'float'>

Beachten Sie das Abschneiden der Nachkommastelle:
Umwandlung in Ganzzahlen mit int(): (67, 68)

Umwandlung in ASCII-Zeichen mit chr(): ('C', 'D')

Umwandlung in eine ASCII-Zahl mit ord(): (67, 68)

Umwandlung in Fließkommazahlen mit float(): (67.0, 68.0)

Umwandlung in Zeichen mit str(): ('67.0', '68.0')

Umwandlung in Wahrheitswerte mit bool(): (True, True)
```

2.10.1 Weitere Zuweisungsoperatoren

In dem obenstehenden Code-Block wurde der sogenannte Walross-Operator `:=` verwendet. Dieser erlaubt es, Zuweisungen innerhalb eines Ausdrucks (hier innerhalb der Funktion `print()`) vorzunehmen. Python kennt eine ganze Reihe weiterer Zuweisungsoperatoren (weitere Operatoren siehe Python-Dokumentation oder übersichtlicher hier).

Operator	entspricht der Zuweisung
a += 2	a = a + 2
a -= 2	a = a - 2
a *= 2	a = a * 2
a /= 2	a = a / 2
a %= 2	a = a % 2
a //= 2	a = a // 2
a **= 2	a = a ** 2

Lesbare Zuweisungen

Die in der Tabelle gezeigten Zuweisungsoperatoren sind für jemanden, der*die Ihren Code liest, gut zu lesen und nachzuvollziehen, da Zuweisungen immer am Beginn einer Zeile, also ganz links, stehen.
Dagegen kann der Walross-Operator an einer beliebigen Stelle in Ihrem Code stehen. Das mag für Sie beim Schreiben ein Vorteil sein, der Lesbarkeit ist das aber abträglich. Wenn Sie den Walross-Operator verwenden, achten Sie deshalb auf die Nachvollziehbarkeit Ihres

Codes.

2.10.2 Benennung von Variablen

Für die Benennung von Variablen gibt es (meist) nur wenige Vorgaben. Trotzdem ist es besser, einen langen, aber ausführlichen Variablennamen zu vergeben, als einen kurzen, der sich schnell schreiben lässt. Denn Programmcode wird deutlich häufiger gelesen als geschrieben. Können Sie sich erinnern? Welcher Wert ist in der Variablen `Var_3` gespeichert, und welche Werte sind in `variable1` und `a` gespeichert? Es reicht schon, wenn Sie sich an den richtigen Datentyp erinnern können.

Falls Sie sich nicht erinnern können, dann ist dieses Beispiel gelungen: Die Namensgebung dieser Variablen ist alles andere als gut. Die Auflösung steht im folgenden Aufklapper.

Auflösung Variablen

```
print(var_3, type(var_3))
print(variable1, type(variable1))
print(a, type(a))
```

```
True <class 'bool'>
15 <class 'int'>
67.0 <class 'str'>
```

Deshalb empfiehlt es sich "sprechende", das heißt selbsterklärende, Variablennamen zu vergeben. Unter selbsterklärenden Variablennamen versteht sich, dass der Variablenname den Inhalt der Variable beschreibt. Wird bspw. in einer Variable der Studienabschluss gespeichert, so kann diese mit `academic_degree` oder `studienabschluss` bezeichnet werden. Werden Daten aus verschiedenen Jahren verarbeitet, kann das Jahr zu besseren Unterscheidbarkeit in den Variablennamen einfließen, etwa: `academic_degree_2023` oder `studienabschluss2024`. Dies verbessert die Lesbarkeit des Codes und vereinfacht die Benutzung der Variable. Mehr Informationen finden sich in diesem Wikipedia Abschnitt.

Schlüsselwörter und Funktionsnamen

In Python reservierte Schlüsselwörter und Funktionsnamen sind ungeeignete Variablennamen. Während Python die Wertzuweisung zu Schlüsselwörtern wie `True` oder `break` mit einem Syntaxfehler quittiert, lassen sich Funktionsnamen neue Werte zuweisen, beispielsweise mit `print = 6`. Wenn Sie die Funktion print dann aufrufen, funktioniert diese natürlich nicht mehr. In diesem Fall müssen Sie die Zuweisung aus dem Skript entfernen und Python neu starten.

Folgende Schlüsselwörter gibt es in Python:

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

2.11 Aufgaben Variablen

1. Schreiben Sie ein Skript, welches eine gegebene Zeit in Sekunden in die Anzahl Tage, Stunden, Minuten und Sekunden aufteilt und diese Aufteilung ausgibt.
   Berechnen Sie die Aufteilung für folgende Zeiten:

   a) 79222 s

   b) 90061 s

   c) 300000 s

2. Die Position eines Fahrzeugs zur Zeit t, welches konstant mit der Beschleunigung a beschleunigt, ist gegeben durch:

$$x(t) = x_0 + v_0 \cdot t + \frac{1}{2} \cdot at^2$$

Dabei ist x_0 die Anfangsposition und v_0 die Anfangsgeschwindigkeit. Erstellen Sie Variablen für x_0, v_0 und a und weisen Sie ihnen Werte zu. Da die Variablen nur Werte, aber keine Einheiten abbilden, überlegen Sie sich die ggf. notwendigen Umrechnungen. Folgende Werte können sie als Beispiel verwenden:

$$x_0 = 10\,km \;\; v_0 = 50\frac{km}{h} \;\; a = 0.1\frac{m}{s^2}$$

Erzeugen Sie eine Variable für den Zeitpunkt t, z. B.: $t = 10\,min$. Berechnen Sie mit obiger Gleichung und mit Hilfe der Variablen die Position $x(t)$. Geben Sie nicht nur den Wert von $x(t)$ in Kilometer aus, sondern betten ihn in einen ganzen Antwortsatz (einschließlich Einheiten) ein.

Die Musterlösung kann Marc machen.

Tip 5: Musterlösung Aufgaben Variablen

(Arnold 2023a)

(Arnold 2023b)

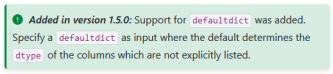
3 Funktionen: Grundlagen

Funktionen sind Unterprogramme, die Programmanweisungen bündeln, damit Programmteile mehrfach verwendet werden können. Auf diese Weise kann ein Programm schneller geschrieben werden und ist auch leichter lesbar. Python bringt, wie Sie der Dokumentation entnehmen können, eine überschaubare Anzahl von grundlegenden Funktionen mit. In diesem Kapitel wird die allgemeine Verwendung der in Python enthaltenen Funktionen vermittelt.

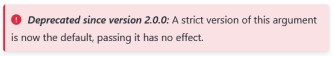
Python wird dynamisch weiterentwickelt: regelmäßig erscheinen neue Versionen mit neuen Eigenschaften. In diesem Kapitel wird deshalb mit einer Reihe von Tipps auch vermittelt, wie die Dokumentation von Python zu lesen ist. Dies erfolgt auch in Hinblick auf die Möglichkeit, Python umfangreich durch Module zu erweitern. So haben beispielsweise die Funktionen des Moduls Pandas nicht selten dutzende dokumentierte Parameter.

Tip 6: Dokumentation

Der wichtigste Tipp zuerst: **Benutzen Sie die Dokumentation!** Auch wenn Sie eine Funktion kennen: Vergewissern Sie sich regelmäßig, dass Sie noch auf dem neuesten Stand sind. Auf diese Weise erhalten Sie einen vollständigen Überblick über standardmäßig gesetzte und optional verfügbare Parameter. Außerdem erkennen Sie Änderungen in der Programmausführung und vermeiden so unerwartete Fehler.



(a) Neuerung in Python



(a) Abkündigung in Python

Achten Sie auf die korrekte Version der Dokumentation.

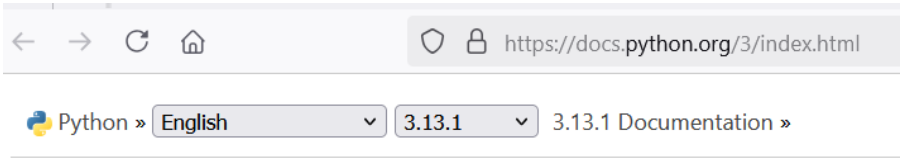


Abbildung 3.3: Versionsauswahl der Dokumentation

3.1 Funktionen und Methoden

In Python gibt es zwei Arten von Funktionen: Funktionen und Methoden.

3.1.1 Funktionen

Funktionen können Objekte unabhängig von ihrem Datentyp übergeben werden. Funktionen werden über ihren Funktionsnamen gefolgt von runden Klammern `()` aufgerufen. Ein Beispiel ist die Funktion `print()`:

```
var_str = 'ABC'
var_int = 26
var_bool = True

print("Die Variable var_1 hat den Typ", type(var_str))
print("Die Variable var_2 hat den Typ", type(var_int))
print("Die Variable var_3 hat den Typ", type(var_bool))
```

```
Die Variable var_1 hat den Typ <class 'str'>
Die Variable var_2 hat den Typ <class 'int'>
Die Variable var_3 hat den Typ <class 'bool'>
```

Funktionen müssen immer einen Wert zurückgeben. Wenn Funktionen keinen Wert zurückgeben können oder sollen, wird der Wert `None` zurückgegeben, der nicht existente Werte kennzeichnet.

```
res = print( 15 )
print(res)
```

```
15
None
```

Funktionen können verschachtelt und so von innen nach außen nacheinander ausgeführt werden. In diesem Code-Beispiel wird zunächst die Summe zweier Zahlen und anschließend der Wahrheitswert des Ergebnisses gebildet. Dieser wird anschließend mit der Funktion print ausgegeben.

```
print(bool(sum([1, 2])))
```

```
True
```

3.1.2 Methoden

Methoden sind eine Besonderheit objektorientierter Programmiersprachen. Im vorherigen Kapitel wurde erläutert, dass in Python Objekte zu einem bestimmten Typ bzw. zu einer Klasse gehören und abhängig von den in ihnen gespeicherten Werte einen passenden Datentyp erhalten. Methoden sind Funktionen, die zu einer bestimmten Klasse gehören und nur für Objekte dieser Klasse verfügbar sind. Methoden können auch für mehrere Klassen definiert sein. Methoden werden getrennt durch einen Punkt `.` hinter Objekten mit ihrem Namen aufgerufen: `variable.methode` bzw. `(wert).methode`. Beispielsweise sind `.upper()`, `.lower()` und `.title` für Zeichenfolgen definierte Methoden.

```
toller_text = "Python 3.12 ist großartig."

print(toller_text.upper())
print(toller_text.lower())
print(toller_text.title(), "\n")

print(("Mit in Klammern gesetzten Werten klappt es auch.").upper())
```

```
PYTHON 3.12 IST GROSSARTIG.
python 3.12 ist großartig.
Python 3.12 Ist Großartig.

MIT IN KLAMMERN GESETZTEN WERTEN KLAPPT ES AUCH.
```

Für Objekte mit einem unpassenden Datentyp sind Methoden wie `.lower()` nicht verfügbar.

```
print((1).upper())
```

```
'int' object has no attribute 'upper'
```

Methoden können verkettet und so nacheinander ausgeführt werden. In diesem Beispiel wird die Zeichenfolge ‘Katze’ klein geschrieben, dann die Häufigkeit des Buchstabens ‘k’ gezählt.

```
print('Katze'.lower().count('k'))
```

```
1
```

Welche Methoden für ein Objekt verfügbar sind, kann mit der Funktion `dir(objekt)` bestimmt werden. Die Ausgabe der Funktion ist aber häufig sehr umfangreich. Um die relevanten Einträge auszuwählen, muss die Ausgabe gefiltert werden. Notwendig ist das aber nicht - Interessierte schauen in Beispiel 1.

Note 1: Methoden eines Objekts bestimmen

Mit der Funktion `dir(objekt)` können die verfügbaren Methoden eines Objekts ausgegeben werden. Dabei werden jedoch auch die Attribute und die Methoden der Klasse des Objekts ausgegeben, sodass die Ausgabe oft sehr umfangreich ist. Zum Beispiel für die Ganzzahl 1:

```
print(dir(1))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__di
```

Um die Ausgabe auf Methoden einzugrenzen, kann folgende Funktion in Listenschreibweise verwendet werden:

```
objekt = 1

attribute = [attr for attr in dir(objekt) if callable (getattr(objekt, attr))]
print(attribute)
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__di
```

Mit doppelten Unterstrichen umschlossene Methoden sind für die Klasse definierte Methoden. Folgende Funktion entfernt Methoden mit doppelten Unterstrichen aus der Ausgabe:

```
objekt = 1

attribute = [attr for attr in dir(objekt) if (callable(getattr(objekt, attr)) and not attr
print(attribute)
```

```
['as_integer_ratio', 'bit_length', 'conjugate', 'from_bytes', 'to_bytes']
```

Im Fall einer Ganzzahl können Methoden (zur Abgrenzung von Gleitkommazahlen in umschließenden Klammern) wie folgt aufgerufen werden:

```
(1).as_integer_ratio()
```

```
(1, 1)
```

Die Methoden des Objekts ‘toller_text’:

```
objekt = toller_text

attribute = [attr for attr in dir(objekt) if (callable(getattr(objekt, attr)) and not attr
print(attribute)
```

```
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
```

3.2 Parameter

Vielen Funktionen und Methoden können getrennt durch Kommata mehrere Parameter übergeben werden. Die Werte, die als Parameter übergeben werden, werden Argumente genannt (Python-Dokumentation). Parameter steuern die Programmausführung. Die für die Funktion `print()` verfügbaren Parameter stehen in der Dokumentation der Funktion:

```
print(*objects, sep=' ', end='\n', file=None, flush=False)
```

`*objects`, `sep`, `end`, `file` und `flush` sind die Parameter der Funktion `print()`.

- Parameter ohne Gleichheitszeichen `=` müssen beim Funktions- bzw. Methodenaufruf übergeben werden. Parameter mit Gleichheitszeichen `=` können beim Aufruf übergeben werden, es handelt sich um optionale Parameter.
- Die Werte hinter dem Gleichheitszeichen zeigen die Standardwerte (default value) der Parameter an. Diese werden verwendet, wenn ein Argument nicht explizit beim Aufruf übergeben wird.

Tip 7: Ausnahmen bei Standardwerten

Bei den in der Funktionsdefinition genannten Werten handelt es sich nicht immer um die tatsächlichen Standardwerte. Es empfiehlt sich deshalb, wenn eine Funktion verwendet wird, die Beschreibung der Parameter zu lesen.
Einige Funktionen verwenden das Schlüsselwort `None` zur Kennzeichnung des Standardwerts. Der Wert `None` dient dabei als Platzhalter. Ein Beispiel ist die NumPy-Funktion numpy.loadtxt().

```
numpy.loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None,
              skiprows=0, usecols=None, unpack=False, ndmin=0, encoding=None, max_rows=None
              *, quotechar=None, like=None)
```

- Für den Parameter `delimiter` ist als Standardwert das Schlüsselwort `None` eingetragen. Wie der Funktionsbeschreibung zu entnehmen ist, ist der Standartwert tatsächlich das Leerzeichen: “The default is whitespace.”
- Auch der Parameter `usecols` hat den Standarwert `None`: “The default, None, results in all columns being read.”

Ein weiteres Beispiel ist die Funktion pandas.read_csv(). Einige Argumente haben den Standardwert `<no_default>`. (Im Folgenden werden nur ausgewählte Parameter gezeigt).

```
pandas.read_csv(sep=<no_default>, verbose=<no_default>)
```

Aus der Beschreibung können die tatsächlichen Standardwerte abgelesen werden:
sep : str, default ‘,’
verbose : bool, default False

- Argumente können in Python entweder als positionales Argument übergeben werden Das heißt, Python erwartet Argumente in einer feststehenden Reihenfolge entsprechend der Parameter der Funktionsdefinition. Alternativ können Argumente als Schlüsselwort übergeben werden, die Zuordnung von Eingaben erfolgt über den Namen des Parameters. Standardmäßig können Argumente positional oder per Schlüsselwort übergeben werden. Abweichungen davon werden durch die Symbole `*` und `/` gekennzeichnet (siehe folgenden Tipp).

Tip 8: Positionale und Schlüsselwortargumente, *args und **kwargs

Die Symbole `*` und `/` zeigen an, welche Parameter positional und welche per Schlüsselwort übergeben werden können bzw. müssen.

Linke Seite	Trennzeichen	Rechte Seite
nur positionale Argumente	/	positionale oder Schlüsselwortargumente
positionale oder Schlüsselwortargumente	*	nur Schlüsselwortargumente

(https://realpython.com/python-asterisk-and-slash-special-parameters/)

Ein Beispiel für das Trennzeichen `*` ist die Funktion `glob` aus dem gleichnamigen Modul. Der Parameter `pathname` kann positional (an erster Stelle) oder als Schlüsselwort übergeben werden. Die übrigen Parameter müssen als Schlüsselwortargumente übergeben werden.

```
glob.glob(pathname, *, root_dir=None, dir_fd=None, recursive=False, include_hidden=False)
```

Beide Steuerzeichen können innerhalb einer Funktionsdefinition vorkommen, allerdings nur in der Reihenfolge `/` und `*`. Im umgekehrten Fall wäre es unmöglich, Argumente zu übergeben. Ein Beispiel ist die Funktion `sorted`. Der erste Parameter muss positional übergeben werden, die Parameter `key` und `reverse` müssen als Schlüsselworte übergeben werden.

```
sorted(iterable, /, *, key=None, reverse=False)¶
```

Ausnahmen

Einige Funktionen weichen von der Systematik ab, beispielsweise die Funktionen `min()` und `max()`. Diese sind (u. a.) in der Form definiert:

```
min(iterable, *, key=None)
max(iterable, *, key=None)
```

Beide Funktionen akzeptieren den Parameter `iterable` aber nicht als Schlüsselwort.

Vielen Funktionen können beliebig viele Argumente positional oder als Schlüsselwort übergeben werden. Im Allgemeinen wird dies durch die Schlüsselwörter `*args` (positionale Argumente) und `**kwargs` (key word arguments, Schlüsselwortargumente) angezeigt. Der Unterschied wird durch das eine bzw. die beiden Sternchen markiert, die Schlüsselwörter selbst sind austauschbar (wie bei der Funktion `print(*objects)`). Das Schlüsselwort `*args` entspricht zugleich dem Symbol `*` in der Funktionsdefinition, d. h. rechts davon dürfen nur Schlüsselwortargumente stehen. Weitere Informationen dazu finden Sie hier.

In der Funktionsdefinition von `print()` ist `*objects` also ein positionaler Parameter (dieser steht immer an erster Stelle), der keinen Standardwert hat und dem beliebig viele Argumente übergeben werden können (n Eingaben stehen an den ersten n-Stellen). Die weiteren Parameter der Funktion `print()` sind optional und müssen als Schlüsselwort übergeben werden.

3.3 Aufgaben Funktionen

1. Richtig oder falsch: Methoden stehen abhängig vom Datentyp eines Werts oder eines Objekts zur Verfügung.
2. Geben Sie die drei Werte 1, 2 und 3 mit `print()` aus. Parametrisieren Sie die Funktion so, dass ihre Ausgabe wie folgt aussieht:

```
1_x_2_x_3
```

3. Schlagen Sie in der Dokumentation die Funktion bool() nach.

- Welche Parameter nimmt die Funktion entgegen und welche davon sind optional?
- Welche Argumente werden positional und welche als Schlüsselübergeben? Ist die Art der Übergabe wählbar oder festgelegt?

Lösungen

Aufgabe 1: richtig
Aufgabe 2

```
print(1, 2, 3, sep = "_x_")
```

Aufgabe 3: Die Funktion bool() hat ein optionales Argument object mit dem Standardwert False. Das Argument muss positional übergeben werden.

3.4 Flusskontrolle

Die Flusskontrolle ermöglicht es, die Ausführung von Programmteilen zu steuern. Anweisungen können übersprungen oder mehrfach ausgeführt werden.

3.4.1 Abzweigungen

Abzweigungen ermöglichen eine Fallunterscheidung, bei der abhängig von einer oder mehreren Bedingungen verschiedene Teile des Skripts ausgeführt werden.

In Python werden Abzweigungen mit dem Schlüsselwort `if` eingeleitet. Dieses wird von der Abzweigbedingung gefolgt und mit einem Doppelpunkt `:` abgeschlossen. Falls die Abzweigbedingung wahr ist, wird der eingerückte Anweisungsblock ausgeführt.

```
if Bedingung:
    Anweisungsblock
```

```
# Beispiel: Zahl kleiner als ein Schwellwert

a = 7
if a < 10:
    print( 'Die Zahl', a, 'ist kleiner als 10.')
```

```
Die Zahl 7 ist kleiner als 10.
```

Es ist auch möglich einen alternativen Anweisungsblock zu definieren, welcher ausgeführt wird, wenn die Bedingung falsch ist. Dieser wird mit dem `else` Schlüsselwort umgesetzt.

```
if Bedingung:
    # Bedingung ist wahr
    Anweisungsblock
else:
    # Bedingung ist falsch
    Anweisungsblock
```

```
# Beispiel: Zahl kleiner als ein Schwellwert mit alternativer Ausgabe

a = 13
if a < 10:
    print( 'Die Zahl', a, 'ist kleiner als 10.')
else:
    print( 'Die Zahl', a, 'ist nicht kleiner als 10.')
```

```
Die Zahl 13 ist nicht kleiner als 10.
```

Es können auch mehrere Bedingungen übergeben werden.

```
# Beispiel: Zahl im Wertebereich zwischen 10 und 20

a = 1
if a < 20 and a > 10:
    print( 'Die Zahl', a, 'liegt zwischen 10 und 20.')
else:
    print( 'Die Zahl', a, 'liegt nicht zwischen 10 und 20.')
```

```
Die Zahl 1 liegt nicht zwischen 10 und 20.
```

Schließlich können mehrere alternative Bedingungen geprüft werden. Dies ist zum einen durch das Verschachteln von Abzweigungen möglich.

```
# Beispiel: Zahl im Wertebereich zwischen 10 und 20 mit verschachtelten Abzweigungen

a = 12
if a > 10:
    print( 'Die Zahl', a, 'ist größer als 10.' )

    if a < 20:
        print( 'Die Zahl', a, 'ist kleiner als 20.' )
        print( 'Damit liegt die Zahl zwischen 10 und 20.')
    else:
        print( 'Die Zahl', a, 'ist größer als 20 und liegt nicht im gesuchten Wertebereich.'
else:
        print( 'Die Zahl', a, 'ist kleiner als 10 und liegt nicht im gesuchten Wertebereich.
```

```
Die Zahl 12 ist größer als 10.
Die Zahl 12 ist kleiner als 20.
Damit liegt die Zahl zwischen 10 und 20.
```

Zum anderen ist dies mit dem Schlüsselwort `elif` möglich.

```
# Beispiel: Zahl im Wertebereich zwischen 10 und 20 mit elif

a = 112
if a < 20 and a > 10:
  print('Die Zahl', a, 'liegt zwischen 10 und 20.')
elif a < 10:
  print('Die Zahl', a, 'ist kleiner als 10 und liegt nicht im gesuchten Wertebereich.' )
elif a > 20 and a <= 100:
  print('Die Zahl', a, 'ist größer als 20, aber nicht größer als 100.')
elif a > 20 and a <= 1000:
  print('Die Zahl', a, 'ist größer als 20, aber nicht größer als 1000.')
else:
  print('Die Zahl', a, 'liegt nicht zwischen 10 und 20 und ist größer als 1000.')
```

```
Die Zahl 112 ist größer als 20, aber nicht größer als 1000.
```

3.4.2 Schleifen

Schleifen ermöglichen es, Anweisungen zu wiederholen. In Python können `while`- und `for`-Schleifen definiert werden. Beide benötigen:

- einen **Schleifenkopf**, welcher die Ausführung des Anweisungsblocks steuert, und
- einen **Anweisungsblock**, also eine Gruppe von Anweisungen, welche bei jedem Schleifendurchlauf ausgeführt werden.

Die `while`-Schleife kommt mit nur einer Bedingung im Schleifenkopf aus und ist die allgemeinere von beiden. Jede `for`-Schleife kann zu einer `while`-Schleife umgeschrieben werden (indem ein Zähler in den Anweisungsblock integriert wird.) Welcher der beiden Typen verwendet wird, hängt von der jeweiligen Aufgabe ab.

while-Schleifen

Eine `while`-Schleife führt den Anweisungsblock immer wieder aus, solange die Ausführbedingung wahr ist. Die Schleife wird mit dem Schlüsselwort `while` eingeleitet, gefolgt von der Ausführbedingung. Dieser Schleifenkopf wird mit einem Doppelpunkt `:` abgeschlossen. Darunter wird der eingerückte Anweisungsblock definiert.

```
while Bedingung:
    Anweisungsblock
```

Beim Beginn der Schleife und nach jedem Durchlauf wird die Bedingung geprüft. Ist sie wahr, so wird der Anweisungsblock ausgeführt, wenn nicht, ist die Schleife beendet und die nächste Anweisung außerhalb der Schleife wird ausgeführt.

```
# Beispiel: Erhöhen eines Variablenwertes

# Setze Startwert
a = 5

# Definiere Schleife, welche solange ausgeführt
# wird, wie a kleiner als oder gleich 10 ist
while a <= 10:
    # Anweisungsblock der Schleife:

    # 1. Ausgabe des aktuellen Werts von a
    print('aktueller Wert von a', a)
```

```
    # 2. Erhöhung von a um Eins
    a += 1

# Ausgabe des Wertes nach der Schleife
print('Wert von a nach der Schleife', a)
```

```
aktueller Wert von a 5
aktueller Wert von a 6
aktueller Wert von a 7
aktueller Wert von a 8
aktueller Wert von a 9
aktueller Wert von a 10
Wert von a nach der Schleife 11
```

Warning 2: Endlosschleife

`while`-Schleifen führen zu einer Endlosschleife, wenn die Abbruchbedingung nicht erreicht werden kann. Beispielsweise fehlt in der folgenden Schleife eine Möglichkeit für die Laufvariable x den Wert 5 zu erreichen.

```
x = 1

while x < 5:
  print(x)
```

In diesem Fall können Sie die Programmausführung durch Drücken von `Strg`+ `C` beenden.

for-Schleifen

Während die `while`-Schleife ausgeführt wird, solange eine Bedingung erfüllt ist, wird die `for`-Schleife über eine Laufvariable gesteuert, die eine Sequenz durchläuft. Die Syntax sieht wie folgt aus:

```
for Laufvariable in Sequenz:
  Anweisungsblock
```

Zur Definition des Schleifenkopf gehören die beiden Schlüsselworte `for` und `in` und der Kopf wird mit einem Doppelpunkt `:` abgeschlossen. Auch hier wird der Anweisungsblock eingerückt.

Die Sequenz wird mit einem range-Objekt erstellt, das mit der Funktion `range(start = 0, stop, step = 1)` erzeugt wird. `range()` nimmt ganzzahlige Werte als *positionale Argumente* entgegen und erzeugt Ganzzahlen von `start` bis *nicht einschließlich* `stop` mit der Schrittweite `step`. Dabei ist wichtig, dass Python **exklusiv zählt**, das heißt, Python beginnt standarmäßig bei `0` zu zählen und der als Argument `stop` übergebene Wert wird nicht mitgezählt.

Die Funktion `range()` gibt ein range-Objekt zurück, das mit `print()` nicht unmittelbar die erwartete Ausgabe erzeugt.

```
# range(start = 1, stop = 5) - step wird nicht übergeben, es gilt der Standardwert step = 1
print(range(1, 5), type(range(1, 5)))
```

```
range(1, 5) <class 'range'>
```

Dieses Verhalten wird faule Auswertung (lazy evaluation) genannt: Die Werte des range-Objekts werden erst erzeugt, wenn Sie benötigt werden. Im Folgenden Code wird das range-Objekt mit einer Schleife durchlaufen und für jeden Durchlauf der Wert der Laufvariable `i` ausgegeben.

```
for i in range(1, 5):
  print(i)
```

```
1
2
3
4
```

Mit dem Parameter `step` kann die Schrittweite gesteuert werden.

```
for i in range(1, 15, 3):
  print(i)
```

```
1
4
7
10
13
```

Nützlich ist die Ausgabe des range-Objekts in eine Liste oder in ein Tupel, Sammeltypen, die im nächsten Kapitel behandelt werden.

```
# Ausgabe der geraden Zahlen 1-10 in eine Liste
print("Liste:", list(range(2, 11, 2)))

# Ausgabe der ungeraden Zahlen 1-10 in ein Tupel
print("Tupel:", tuple(range(1, 11, 2)))
```

```
Liste: [2, 4, 6, 8, 10]
Tupel: (1, 3, 5, 7, 9)
```

`start` und `stop` können auch negativ sein, `step` muss immer größer 0 sein.

```
for i in range(-5, -1):
  print(i)
```

```
-5
-4
-3
-2
```

`stop` muss immer größer als `start` sein. Um eine absteigende Zahlenfolge zu erzeugen, wird die Funktion `reversed(sequenz)` verwendet.

```
# Die Ausgabe bleibt leer
print(list(range(5, 0)))

# Mit der Funktion reversed geht es
print(list(reversed(range(0, 5))))
```

```
[]
[4, 3, 2, 1, 0]
```

Listennotation

Die sogenannte Listennotation ist eine Kurzschreibweise für for-Schleifen. In Listennotation geschriebene Schleifen müssen in einer Zeile stehen und haben die folgende Syntax:

```
quadratzahlen = [wert ** 2 for wert in range(10, 0, -1)]
print(quadratzahlen)
```

```
[100, 81, 64, 49, 36, 25, 16, 9, 4, 1]
```

(Matthes (2017), S. 71)

Die Schlüsselwörter break und continue

Manchmal kann es notwendig sein, den Anweisungsblock einer Schleife vorzeitig zu verlassen. Dafür können die Schlüsselwörter `break` und `continue` benutzt werden. Das Schlüsselwort `break` bewirkt, dass die Schleife sofort verlassen wird. Dagegen führt das Schlüsselwort `continue` dazu, dass der aktuelle Schleifendurchlauf beendet und der nächste Durchlauf begonnen wird.

```
x = 0
while x < 10:

  x += 1

  # keine geraden Zahlen ausgeben
  if x % 2 == 0:
    continue

  # Schleife bei x == 7 beenden
  if x == 7:
    break

  print(x)
```

```
1
3
5
```

3.4.3 Ausnahmebehandlung

Die Ausnahmebehandlung erlaubt es, Python alternative Anweisungen zu geben, die beim Auftreten eines Fehlers ausgeführt werden sollen. Dies ist beispielsweise beim Einlesen von Datensätzen nützlich, um sich die Ursache von Fehlermeldungen ausgeben zu lassen - eine Technik, die im Methodenbaustein Einlesen strukturierter Datensätze vorgestellt wird.

In Python gibt es zwei Arten von Fehlern. Dies sind erstens Syntaxfehler, die Python mit einer Fehlermeldung ähnlich wie der folgenden quitiert. Syntaxfehler werden durch das Schreiben von syntaktisch korrektem Programmcode behoben.

```
print(1}
```

```
closing parenthesis '}' does not match opening parenthesis
```

Die zweite Art von Fehlern sind Ausnahmen (exceptions), die auch bei syntaktisch korrektem Programmcode auftreten können. Ausnahmen führen auch zu Fehlermeldungen.

```
# Beispiel 1: Division durch Null
print(1 / 0)
```

```
division by zero
```

```
# Beispiel 2: undefinierte Variable
print(undefinierte_variable)
```

```
name 'undefinierte_variable' is not defined
```

Fehlermeldungen wie diese können in Python mit der Ausnahmebehandlung abgefangen werden. Diese wird mit dem Schlüsselwort `try` eingeleitet, das mit dem Doppeltpunkt `:` abgeschlossen wird. In der nächsten Zeile folgt eingrückt der Anweisungblock, der auf Ausnahmen getestet werden soll. *Hinweis: Der Anweisungsblock wird tatsächlich ausgeführt, Änderungen an Daten oder Dateien sind also möglich.* Anschließend wird mit dem Schlüsselwort `except`, das von einem Doppelpunkt `:` und in der nächsten Zeile von einem eingerückten Anweisungsblock gefolgt wird, festgelegt, was beim Aufkommen einer Ausnahme passieren soll. Optional kann mit dem Schlüsselwort `else` nach dem gleichen Schema ein weiterer Anweisungsblock definiert werden, der bei einer erfolgreichen Ausführung des Anweisungsblocks unter `try` *zusätzlich* ausgeführt wird. Der allgemeiner Aufbau lautet wie folgt:

```
try:
  Anweisungsblock_1
except:
  Anweisungsblock falls Anweisungblock_1 eine Ausnahme erzeugt
else:
  optionaler Anweisungsblock falls Anweisungsblock_1 keine Ausnahme erzeugt
```

Mithilfe der Ausnahmebehandlungen können die Elemente angezeigt werden, die zu einer Fehlermeldung führen.

```
a = 1
b = 2

try:
  differenz = a - b
except:
```

```
  print(f"Die Differenz aus {a} und {b} konnte nicht gebildet werden.")
else:
  print(f"Die Differenz aus {a} und {b} ist {differenz}.")
```

```
Die Differenz aus 1 und 2 ist -1.
```

```
a = 1
b = 'abc'

try:
  differenz = a - b
except:
  print(f"Die Differenz aus {a} und {b} konnte nicht gebildet werden.")
else:
  print(f"Die Differenz aus {a} und {b} ist {differenz}.")
```

```
Die Differenz aus 1 und abc konnte nicht gebildet werden.
```

Auch ist es möglich, die Fehlermeldung abzufangen und auszugeben. Dafür wird die Zeile `except:` wie folgt modifiziert `except Exception as error:`

```
a = 1
b = 'abc'

try:
  differenz = a - b
except Exception as error:
  print(f"Die Differenz aus {a} und {b} konnte nicht gebildet werden.")
  print(error)
else:
  print(f"Die Differenz aus {a} und {b} ist {differenz}.")
```

```
Die Differenz aus 1 und abc konnte nicht gebildet werden.
unsupported operand type(s) for -: 'int' and 'str'
```

3.5 Aufgaben Flusskontrolle

1. Schreiben Sie ein Programm, dass von 1 bis 25 und von 38 bis 50 zählt und jeden Wert, der ganzzahlig durch 7 teilbar ist, mit `print()` ausgibt.

2. Roulette: Schreiben Sie ein Programm, dass für eine Zahl prüft, ob diese im Wertebereich des Spieltischs liegt. Falls nein, soll eine Fehlermeldung ausgegeben werden. Falls ja, soll das Programm ausgeben, ob die Zahl

- gerade oder ungerade ist,
- rot oder schwarz ist,
- niedrig (1-18) oder hoch (19-36) ist und
- im 1., 2. oder 3. Dutzend liegt.

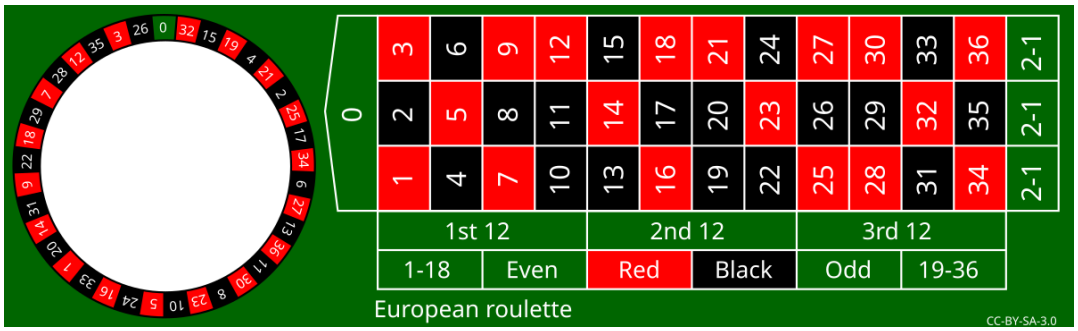


Abbildung 3.4: Roulette Tableau

European roulette von Betzaar.com ist lizensiert unter CC 3.0 BY-SA und verfügbar auf wikimedia.org. 2010

Die Musterlösung kann Marc machen.

> Musterlösung Aufgaben Flusskontrolle

(Arnold (2023d))

4 Sammeltypen

Sammeltypen werden benutzt, um mehrere Werte in einer Variablen zu speichern und zu verarbeiten. In Python gibt es vier Sammeltypen, die jeweils eine eigene Klasse sind:

- Listen enthalten eine flexible Anzahl von Elementen beliebigen Typs.
- Tupel können wie Listen Elemente beliebigen Typs enthalten, sind aber unveränderlich.
- Mengen sind ungeordnete Sammlungen, die jedes Element nur einmal enthalten können.
- assoziative Arrays oder Dictionaries sind Zuordnungstabellen, d. h. sie bestehen aus Schlüssel-Wert-Paaren.

In diesem Kapitel werden die vier Sammeltypen zunächst kurz vorgestellt. Anschließend wird die Arbeitsweise insbesondere mit Listen erläutert.

4.1 Listen

Wie alle Typen in Python werden Listen durch Zuweisung erstellt. Bei der Definition einer Liste werden die Elemente durch eckige Klammern `[]` eingeklammert und mit Kommata `,` getrennt. Listen können mit dem `+`-Operator verkettet werden. `*` verkettet eine Liste n-mal.

```
text_variable = 'abc'

liste1 = [1, 'xy', True, text_variable]
print(liste1)

# Listen können auch Listen enthalten
liste2 = [None, liste1]
print(liste2)

# Listen können mit + und * verkettet werden
print(liste1 + liste2)
print(liste1 * 2)
```

```
[1, 'xy', True, 'abc']
[None, [1, 'xy', True, 'abc']]
[1, 'xy', True, 'abc', None, [1, 'xy', True, 'abc']]
[1, 'xy', True, 'abc', 1, 'xy', True, 'abc']
```

Eine leere Liste kann durch Zuweisung von `[]` erstellt werden.

```
leere_liste = []
print(leere_liste)
```

```
[]
```

4.1.1 Slicing: der Zugriffsoperator []

Der Zugriff auf einzelne oder mehrere Elemente einer Liste (und andere Sammeltypen) erfolgt über den Zugriffsoperator []. Ein Ausschnitt aus einem Objekt wird Slice genannt, der Operator heißt deshalb auch Slice Operator.

Zugriff auf einzelne Elemente

Elemente werden über ihren Index, bei 0 beginnend, angesprochen.

```
print(liste1)
print(liste1[0])
print(liste1[3])
```

```
[1, 'xy', True, 'abc']
1
abc
```

Auf verschachtelte Listen kann mit zwei aufeinanderfolgenden Zugriffsoperatoren zugegriffen werden. Die Liste `liste2` enthält an Indexposition 1 eine Liste mit 4 Elementen.

```
print(liste2)
print(liste2[1])
print(liste2[1][0], liste2[1][1], liste2[1][2], liste2[1][3])
```

```
[None, [1, 'xy', True, 'abc']]
[1, 'xy', True, 'abc']
1 xy True abc
```

Mit negativen Indizes können Elemente vom Ende aus angesprochen werden. So entspricht z. B. die -1 dem letzten Element.

```
print(liste1)
print(liste1[-1], liste1[-3])
```

```
[1, 'xy', True, 'abc']
abc xy
```

Zugriff auf mehrere Elemente

Indexbereiche können in der Form `[start:stop:step]` angesprochen werden. `start` ist das erste adressierte Element, `stop` *das erste nicht mehr adressierte Element* und `step` die Schrittweite.

Zugriffsoperator	Ausschnitt
liste[start:stop]	Elemente von start bis stop - 1
liste[:]	Alle Elemente der Liste
liste[start:]	Elemente von start bis zum Ende der Liste
liste[:stop]	Elemente vom Anfang der Liste bis stop - 1
liste[::3]	Auswahl jedes dritten Elements

Negative Werte für `start`, `stop` oder `step` bewirken eine Rückwärtsauswahl von Elementen.

Zugriffsoperator	Ausschnitt
liste[-1]	das letzte Element der Liste
liste[-2:]	die letzten beiden Elemente der Liste
liste[:-2]	alle bis auf die beiden letzten Elemente
liste[::-1]	alle Elemente in umgekehrter Reihenfolge
liste[1::-1]	die ersten beiden Elemente in umgekehrter Reihenfolge
liste[:-3:-1]	die letzten beiden Elemente in umgekehrter Reihenfolge
liste[-3::-1]	alle außer die letzten beiden Elemente in umgekehrter Reihenfolge

(Beispiele von Greg Hewgill unter der Lizenz CC BY-SA 4.0 verfügbar auf stackoverflow. 2009)

Zeichenfolgen

Auch aus Zeichenfolgen können mit dem Slice Operator Ausschnitte ausgewählt werden.

```
print('Ich bin ein string'[::2])
print('Hallo Welt'[0:6])
print('abc'[::-1])
```

```
Ihbnensrn
Hallo
cba
```

4.1.2 Listenmethoden

Für den Listentyp sind verschiedene Methoden definiert.

Elemente bestimmen

- `list.index(x, start, stop)` gibt die Indexposition des ersten Elements x aus. Die optionalen Argumente `start` und `stop` erlauben es, den Suchbereich einzuschränken.
- `list.count(x)` gibt die Häufigkeit von x in der Liste aus.
- `list.reverse()` kehrt die Reihenfolge der Listenelemente um (die Liste wird dadurch verändert!).
- `list.sort(reverse = False)` sortiert die Liste, mit dem optionalen Argument reverse = True absteigend (die Liste wird dadurch verändert!). Die Datentypen innerhalb der Liste müssen sortierbar sein (d. h. alle Elemente sind numerisch oder Zeichen).

```
print(liste1)

liste1.reverse()
print(liste1)

# True wird als 1 gezählt
print("True wird als 1 gezählt:", liste1.index(1), liste1.count(1))
```

```
[1, 'xy', True, 'abc']
['abc', True, 'xy', 1]
True wird als 1 gezählt: 1 2
```

Elemente einfügen

- `list.append(x)` hängt ein einzelnes Element an das Ende der Liste an.
- `list.extend(sammeltyp)` hängt alle mit `sammeltyp` übergebenen Elemente an das Ende der Liste an. Der Sammeltyp kann eine Liste, ein Tupel, eine Menge oder ein Dictionary sein.
- `list.insert(i, x)` fügt an der Position i Element x ein.

```
print(liste1, "\n")

liste1.append('Hallo')
liste1.extend(['Hallo', 'Welt!'])
liste1.insert(2, '12345')

print(liste1)
```

```
['abc', True, 'xy', 1]

['abc', True, '12345', 'xy', 1, 'Hallo', 'Hallo', 'Welt!']
```

Elemente entfernen

- `list.remove(x)` entfernt *das erste* Element x in der Liste und gibt einen ValueError zurück, wenn x nicht in der Liste enthalten ist.
- `liste.pop(i)` entfernt das Element an der Indexposition i. Wird kein Index angegeben, wird das letzte Element entfernt. Die Methode `liste.pop(i)` gibt die entfernten Elemente zurück.
- `liste.clear()` entfernt alle Elemente einer Liste.

```
liste1.remove('Hallo')
print(liste1)

liste1.pop(2)
```

```
['abc', True, '12345', 'xy', 1, 'Hallo', 'Welt!']


'12345'
```

Listen und Listenelemente kopieren

In Python enthalten Listen Daten nicht direkt, sondern bestehen aus Zeigern auf die Speicherorte der enthaltenen Elemente. Wird eine Liste durch Zuweisung einer anderen Liste angelegt, dann werden nicht die Elemente der Liste kopiert, sondern beide Listen greifen dann auf den selben Speicherort zu.

```
# Kopieren durch Zuweisung
liste1 = [1, 'xy', True, text_variable]
print("liste1:", liste1, "\n")
liste2 = liste1

## Ändern eines Elements in liste2
liste2[0] = 'ABC'
print("Auch liste1 hat sich durch die Zuweisung in liste2 verändert:", liste1, "\n")
```

```
liste1: [1, 'xy', True, 'abc']

Auch liste1 hat sich durch die Zuweisung in liste2 verändert: ['ABC', 'xy', True, 'abc']
```

Um eine Liste zu kopieren und ein neues Objekt im Speicher anzulegen, kann die Methode `liste.copy()` verwendet werden. Auch durch die Verwendung des Zugriffsoperators `[:]` wird eine neue Liste im Speicher angelegt.

```
# Verwendung der Methode liste.copy()
liste1 = [1, 'xy', True, text_variable]
liste2 = liste1.copy()

## Ändern eines Elements in liste2
liste2[0] = 'ABC'
print("liste1 bleibt durch die Zuweisung in liste2 unverändert:", liste1, "\n")

# Verwendung des Slice Operators
liste1 = [1, 'xy', True, text_variable]
liste2 = liste1[:]

## Ändern eines Elements in liste2
liste2[0] = 'ABC'
print("liste1 bleibt durch die Zuweisung in liste2 unverändert:", liste1)
```

```
liste1 bleibt durch die Zuweisung in liste2 unverändert: [1, 'xy', True, 'abc']
```

```
liste1 bleibt durch die Zuweisung in liste2 unverändert: [1, 'xy', True, 'abc']
```

Die Kopie von Listenelementen ist in dieser Hinsicht unproblematisch.

```
# Verwendung des Slice Operators
liste1 = [1, 'xy', True, text_variable]
liste2 = liste1[0:2]

# Ändern eines Elements in liste2
liste2[0] = 'ABC'
print("liste1 bleibt durch die Zuweisung in liste2 unverändert:", liste1)
```

```
liste1 bleibt durch die Zuweisung in liste2 unverändert: [1, 'xy', True, 'abc']
```

Um zu überprüfen, ob sich zwei Objekte den Speicherbereich teilen, kann die Objekt-ID mit der Funktion `id()` verglichen oder die Operatoren `is` bzw. `is not` verwendet werden, die die Funktion `id()` aufrufen.

```
liste1 = [1, 'xy', True, text_variable]
liste2 = liste1

print("ID liste1:", id(liste1))
print("ID liste2:", id(liste2))
print("ID liste1 gleich ID list2:", liste1 is liste2)
```

```
ID liste1: 4478709696
ID liste2: 4478709696
ID liste1 gleich ID list2: True
```

> **Identität vs. Wertgleichheit**
>
> Der Operator `is` prüft die Identität zweier Objekte und unterscheidet sich dadurch vom logischen Operator `==`, der auf Wertgleichheit prüft. Da liste1 und liste2 die gleichen Elemente enthalten, liegen sowohl Identität und Wertgleichheit vor. Der Unterschied von Identität und Wertgleichheit kann anhand eines Werts verdeutlicht werden (Im Code-Beispiel wird eine Syntax-Warnung unterdrückt.).

```
# Wertgleichheit
print(1 == 1.0)
print(liste1 == liste2, "\n")

# Identität
print(1 is 1.0)
print(liste1 is liste2)
```

```
True
True

False
True
```

4.1.3 Aufgaben Listen

1. Erstellen Sie eine Liste 'wochentage', die die sieben Tage der Woche enthält. Verwenden Sie den Slice-Operator, um eine neue Liste 'wochenende' mit den Tagen des Wochenendes zu erstellen. Entfernen Sie die Tage des Wochenendes aus der Liste 'wochentage'.
2. 4-Tage-Woche: Verwenden Sie Listenmethoden, um den Freitag aus der Liste 'wochentage' zu entfernen und der Liste 'wochenende' vor dem Samstag hinzuzufügen.
3. Bestimmen Sie in der Liste `zahlen = [34, 12, 0, 67, 23]` die Position des Werts 0. Entfernen Sie den Wert aus der Liste und geben Sie die Liste aufsteigend sortiert aus.
4. Geben Sie nun mit Hilfe des Zugriffsoperators `[]` die Indexpositionen 1 und 3 der sortierten Liste 'zahlen' aus.

Musterlösung kann Marc machen.

Tip 9: Musterlösung

4.2 Tupel

Tupel sind Listen sehr ähnlich, jedoch sind Tupel unveränderbare Datenobjekte. Das heißt, die Elemente eines angelegten Tupels können weder geändert, noch entfernt werden. Auch können keine neuen Elemente zum Tupel hinzugefügt werden.

Tupel werdem mit runden Klammern `()` erzeugt, die Elemente werden mit einem Komma `,` getrennt. Ein Tupel mit einem Wert wird mit einem Komma in der Form `(wert, )` angelegt. Der Zugriff auf die Elemente eines Tupels ist mit dem Slice-Operator `[start:stop:step]` möglich. Tupel können mit den Operatoren `+` und `*` verkettet werden.

```
tupel1 = (2, 7.8, 'Feuer', True, text_variable)
tupel2 = (1, )

print(tupel1)
print(tupel1[2:4])
print(tupel1[::2])
print(tupel1[-1])
print(tupel1[2:4] + tupel2)
print(3 * tupel2)
```

```
(2, 7.8, 'Feuer', True, 'abc')
('Feuer', True)
(2, 'Feuer', 'abc')
abc
('Feuer', True, 1)
(1, 1, 1)
```

4.2.1 Tupel kopieren

Tupel verhalten sich beim Kopieren gegensätzlich zu Listen. Für Tupel ist die Methode `.copy()` nicht definiert. Dagegen bewirkt die Kopie mittels dem Zugriffsoperator `[:]` zwar, dass zwei Tupel auf den selben Speicherplatz zugreifen. Bei der Neuzuweisung eines Tupels legt Python, wie für jedes Objekt, ein neues Objekt im Speicher an.

```
# Kopieren durch Zuweisung
tupel1 = (1, 2, 3)
tupel2 = tupel1

## Neuzuweisung der Werte von tupel1
tupel1 = (4, 5, 6)
print(f"Die in tupel2 gespeicherten Werte sind unverändert:\n{tupel1} {tupel2}\n")

# Kopieren mit Slice Operator
tupel1 = (1, 2, 3)
tupel2 = tupel1[:]
print(tupel2 is tupel1)
```

```
## Neuzuweisung der Werte von tupel1
tupel1 = (4, 5, 6)
print(tupel1, tupel2)
```

```
Die in tupel2 gespeicherten Werte sind unverändert:
(4, 5, 6) (1, 2, 3)

True
(4, 5, 6) (1, 2, 3)
```

4.3 Mengen

In Python können Mengen mit der `set()` Funktion z. B. aus einer Liste oder aus einem Tupel erzeugt oder durch geschweiften Klammern `{}` erstellt werden (eine leere Menge kann nur mit `set()` erzeugt werden, da `{}` ein leeres Dictionary anlegt). Mengen sind ungeordnete Sammelung, dementsprechend haben die Elemente keine Reihenfolge.

```
liste = [1, 1, 5, 3, 3, 4, 2, 'a', 123, 1000, ('tupel', 5)]
print("Das Objekt liste als Menge:\n", set(liste))

menge = {1, 2, 3, 4, 5, 1000, ('tupel', 5), 'a', 123}
print("Die Menge kann auch mit geschweiften Klammern erzeugt werden:", menge)
```

```
Das Objekt liste als Menge:
 {1, 2, 3, 4, 5, ('tupel', 5), 1000, 'a', 123}
Die Menge kann auch mit geschweiften Klammern erzeugt werden: {1, 2, 3, 4, 5, ('tupel', 5),
```

Mengen können beispielsweise für Vergleichsoperationen verwendet werden.

```
menge_a = set('Python')
menge_b = set('ist super')

# einzigartige Zeichen in a
print("Menge a:", menge_a)

# Zeichen in a, aber nicht in b
print("Menge a - b:", menge_a - menge_b)

# Zeichen in a oder b
```

```
print("Menge a | b:", menge_a | menge_b)

# Zeichen in a und b
print("Menge a & b:", menge_a & menge_b)

# Zeichen in a oder b, aber nicht in beiden (XOR)
print("Menge a ^ b:", menge_a ^ menge_b)
```

```
Menge a: {'P', 't', 'o', 'n', 'y', 'h'}
Menge a - b: {'P', 'o', 'n', 'y', 'h'}
Menge a | b: {'r', 's', 'P', 't', 'i', 'o', 'u', 'n', ' ', 'y', 'h', 'e', 'p'}
Menge a & b: {'t'}
Menge a ^ b: {'r', 's', 'i', 'o', 'u', 'n', 'y', ' ', 'P', 'h', 'e', 'p'}
```

4.3.1 Mengen kopieren

Mengen verhalten sich wie Tupel mit dem Unterschied, dass die Methode `.copy()` für Mengen definiert ist. Allerdings kann der Zugriffsoperator `[]` nicht auf Mengen angewendet werden.

```
# Kopieren durch Zuweisung
set1 = {1, 2, 3}
set2 = set1
print(set1 is set2)

## Neuzuweisen von set1
set1 = {4, 5, 6}
print(f"Die in set2 gespeicherten Werte sind unverändert:\n{set1} {set2}")

# Kopieren durch Methode .copy()
set1 = {1, 2, 3}
set2 = set1.copy()
print(set1 is set2)

## Neuzuweisen von set1
set1 = {4, 5, 6}
print(f"Die in set2 gespeicherten Werte sind unverändert:\n{set1} {set2}")
```

```
True
Die in set2 gespeicherten Werte sind unverändert:
{4, 5, 6} {1, 2, 3}
```

```
False
Die in set2 gespeicherten Werte sind unverändert:
{4, 5, 6} {1, 2, 3}
```

4.4 Dictionaries

Dictionaries bestehen aus Schlüssel-Wert-Paaren. Die Schlüssel können Zahlen oder Zeichenketten sein, jeder Schlüssel darf nur einmal vorkommen. Dictionaries werden mit geschweiften Klammern `{}` definiert. Die Schlüssel und deren zugehörigen Werte werden mit einem Doppelpunkt `:` getrennt. Der Zugriff auf die Werte erfolgt mit dem Zugriffsoperator `[]`, welcher den oder die Schlüssel beinhaltet. Ein Zugriff über die Indexposition der Schlüssel ist nicht möglich, da Zahlen als Schlüssel interpretiert werden.

```
dictionary1 = {1: 'abc', 'b': [1, 2, 3], 'c': ('tupel', 5, 6)}
print(dictionary1, "\n")

print("Werte des Schlüssels 1:", dictionary1[1])
print("Werte des Schlüssels 'b':",  dictionary1['b'])
```

```
{1: 'abc', 'b': [1, 2, 3], 'c': ('tupel', 5, 6)}

Werte des Schlüssels 1: abc
Werte des Schlüssels 'b': [1, 2, 3]
```

Auf die Schlüssel eines Dictionaries kann über die Methode `dictionary.keys()`, auf die Werte mittels der Methode `dictionary.values()` zugegriffen werden.

```
print("Schlüssel:", dictionary1.keys(), "\n")
print("Werte:", dictionary1.values())
```

```
Schlüssel: dict_keys([1, 'b', 'c'])

Werte: dict_values(['abc', [1, 2, 3], ('tupel', 5, 6)])
```

4.4.1 Dictionaries kopieren

Dictionaries verhalten sich beim Kopieren wie Listen, das heißt beim Kopieren durch Zuweisung teilen sich Dictionaries den Speicherbereich.

```
# Kopieren durch Zuweisung
print("dictionary:", dictionary1, "\n")
dictionary2 = dictionary1

## Ändern eines Elements in dictionary2
dictionary2[1] = 'ABC'
print("Auch dictionary1 hat sich durch die Zuweisung in dictionary2 verändert:\n",
       dictionary1, "\n")

# Verwendung der Methode dictionary.copy()
dictionary1 = {1: 'abc', 'b': [1, 2, 3], 'c': ('tupel', 5, 6)}
dictionary2 = dictionary1.copy()

## Ändern eines Elements in dictionary2
dictionary2[1] = 'ABC'
print("dictionary1 bleibt durch die Zuweisung in dictionary2 unverändert:\n",
       dictionary1, "\n")
```

```
dictionary: {1: 'abc', 'b': [1, 2, 3], 'c': ('tupel', 5, 6)}

Auch dictionary1 hat sich durch die Zuweisung in dictionary2 verändert:
 {1: 'ABC', 'b': [1, 2, 3], 'c': ('tupel', 5, 6)}

dictionary1 bleibt durch die Zuweisung in dictionary2 unverändert:
 {1: 'abc', 'b': [1, 2, 3], 'c': ('tupel', 5, 6)}
```

4.5 Übersicht Sammeltypen

Merkmal	Listen	Tupel	Mengen	Dictionary
Beschreibung	flexible Anzahl von Elementen beliebigen Typs	Elemente beliebigen Typs, unveränderlich	ungeordnete Sammlung, jedes Element nur einmal enthalten	Zuordnungstabelle aus Schlüssel-Wert-Paaren

Merkmal	Listen	Tupel	Mengen	Dictionary
Speicherbereich bei Zuweisung geteilt	ja	ja (aber unveränderlich)	ja (aber Zugriffsoperator nicht anwendbar)	ja
Methode .copy() definiert	ja	nein	ja	ja
Slice-Operator anwendbar	ja	ja	nein	ja (nach Schlüssel)

4.6 Löschen: das Schlüsselwort del

Um Sammeltypen, Elemente oder Slices zu löschen kann das Schlüsselwort `del` verwendet werden.

```
# Löschen einer Liste
del liste1

# Löschen eines Indexbereichs aus einer Liste
print("Liste vor dem Löschen:", liste2)
del liste2[1:3]
print("Liste nach dem Löschen:", liste2)

# Löschen eines Schlüsselworts aus einem Dictionary
print("Dictionary vor dem Löschen", dictionary1)
del dictionary1[1]
print("Dictionary nach dem Löschen", dictionary1)
```

```
Liste vor dem Löschen: [1, 'xy', True, 'abc']
Liste nach dem Löschen: [1, 'abc']
Dictionary vor dem Löschen {1: 'abc', 'b': [1, 2, 3], 'c': ('tupel', 5, 6)}
Dictionary nach dem Löschen {'b': [1, 2, 3], 'c': ('tupel', 5, 6)}
```

4.7 Funktionen

Die Sammeltypen können ineinander umgewandelt werden.

```
dictionary = {1: 'Kater', 2: 'Fähre', 3: 'Ricke'}
print( (liste := list(dictionary)) )
print( (menge := set(liste)) )
print( (tupel := tuple(menge)) )
```

```
[1, 2, 3]
{1, 2, 3}
(1, 2, 3)
```

Einige praktische Funktionen lassen sich auch auf Sammeltypen anwenden:

- `len()` gibt die Anzahl der Elemente in einem Sammeltyp zurück.
- `min()`, `max()`, `sum()` gibt das Minimum, Maximum bzw. die Summe eines Sammeltyps zurück (bei Dictionaries wird die Anzahl der Schlüssel gezählt).

4.8 Operationen: Verwendung von Schleifen

Um arithmetische und logische Operatoren auf die in einem Sammeltyp gespeicherten Elemente anzuwenden, wird eine for-Schleife verwendet. Im folgenden Beispiel wird eine Liste 'zahlen' durchlaufen, die darin gespeicherten Zahlen quadriert und das jeweilige Ergebnis an die Liste 'quadratzahlen' angehängt. Auch wird geprüft, ob die quadrierten Zahlen ganzzahlig durch 3 teilbar sind und das Prüfergebnis in einer Liste 'modulo_3' gespeichert.

4.9 Aufgaben Sammeltypen

1. Modifizieren Sie den Programmcode in Code-Block 4.1 so, dass nur die Quadratzahlen gespeichert werden, die ganzzahlig durch 3 teilbar sind.
2. Umrechnung von Geschwindigkeiten Erstellen Sie ein Skript, welches eine Umrechnungstabelle für Geschwindigkeiten erzeugt. Folgende Randbedingungen sollen beachtet werden:

- Die Umrechnung soll von km/h in m/s erfolgen.
- Der Start- und Endwert soll in km/h frei wählbar sein, wobei beide ganzzahlig sein sollen.

Code-Block 4.1

```
zahlen = list(range(1, 11))

quadratzahlen = [] # die Liste muss vor der Schleife angelegt werden
modulo_3 = [] # leere Liste vor der Schleife anlegen

for zahl in zahlen:
  quadratzahl = zahl ** 2
  quadratzahlen.append(quadratzahl)
  modulo_3.append(quadratzahl % 3 == 0)

print(quadratzahlen)
print(modulo_3)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[False, False, True, False, False, True, False, False, True, False]
```

- Die Anzahl der Umrechnungspunkte soll definiert werden können und die Zwischenschritte (in km/h) immer als ganze Zahlen ausgegeben werden.

Tipp: In Ihrem Skript können Sie die Funktion input() verwenden, um Werte per Eingabe zu erfassen.

3. Sortieren: Gegeben ist die Liste `meine_liste = list(range(9, 0, -1))`. Diese soll mittels for-Schleifen sortiert werden.

Tip 10: Musterlösung Aufgaben Sammeltypen

1. Ganzzahlig durch 3 teilbare Quadratzahlen

```
zahlen = list(range(1, 11))

quadratzahlen = [] # die Liste muss vor der Schleife angelegt werden
modulo_3 = [] # leere Liste vor der Schleife anlegen

for zahl in zahlen:
  quadratzahl = zahl ** 2
  if quadratzahl % 3 == 0:
    quadratzahlen.append(quadratzahl)

print(quadratzahlen)
```

```
[9, 36, 81]
```

2. Umrechnung von Geschwindigkeiten

```
# Freie Eingabe
## start = int(input("Startwert in Kilometer pro Stunde eingeben."))
## ende = int(input("Endwert in Kilometer pro Stunde eingeben."))
## ausgabeschritte =  int(input("Anzahl auszugebener Schritte ein geben."))

# Fixe Werte für die Lösung
start = 5
ende = 107
ausgabeschritte = 8

# Liste für km erstellen
schrittweite = (ende - start) / (ausgabeschritte - 1)
liste_km = []
for i in range(ausgabeschritte):
    liste_km.append(round(start + i * schrittweite))

# Umrechnung
# meter = 1000 * kilometer
# Sekunde = Stunde * 60 * 60
liste_m = []
for wert in liste_km:
    liste_m.append(round((wert * 1000) / (60 * 60), 2))

# Ausgabe
print(f"Schrittweite: {schrittweite:.2f}")
print("Kilometer pro Stunde")
print(liste_km)
print("Meter pro Sekunde")
print(liste_m)
```

```
Schrittweite: 14.57
Kilometer pro Stunde
[5, 20, 34, 49, 63, 78, 92, 107]
Meter pro Sekunde
[1.39, 5.56, 9.44, 13.61, 17.5, 21.67, 25.56, 29.72]
```

3. Sortieren: Bubble Sort Algorithmus

```python
# statische Liste, Textausgabe
meine_liste = list(range(9, 0, -1))

if len(meine_liste) > 1:

    print("Liste zu Beginn\t\t  :", meine_liste)

    # äußere Schleife
    Schritt = 0
    for i in range(len(meine_liste) - 1):

    # innere Schleife
        for j in range(len(meine_liste) - 1):
            if meine_liste[j] > meine_liste[j + 1]:
                meine_liste[j], meine_liste[j + 1] = meine_liste[j + 1], meine_liste[j]

        Schritt += 1
        print("Liste nach Schritt ", Schritt, ":", meine_liste)

    print("\nListe sortiert:", *meine_liste) # * unterdrückt die Kommas zwischen den Liste

else:
    print("Die Liste muss mindenstens zwei Elemente enthalten!")
```

```
Liste zu Beginn       : [9, 8, 7, 6, 5, 4, 3, 2, 1]
Liste nach Schritt  1 : [8, 7, 6, 5, 4, 3, 2, 1, 9]
Liste nach Schritt  2 : [7, 6, 5, 4, 3, 2, 1, 8, 9]
Liste nach Schritt  3 : [6, 5, 4, 3, 2, 1, 7, 8, 9]
Liste nach Schritt  4 : [5, 4, 3, 2, 1, 6, 7, 8, 9]
Liste nach Schritt  5 : [4, 3, 2, 1, 5, 6, 7, 8, 9]
Liste nach Schritt  6 : [3, 2, 1, 4, 5, 6, 7, 8, 9]
Liste nach Schritt  7 : [2, 1, 3, 4, 5, 6, 7, 8, 9]
Liste nach Schritt  8 : [1, 2, 3, 4, 5, 6, 7, 8, 9]

Liste sortiert: 1 2 3 4 5 6 7 8 9
```

(Arnold (2023d))

5 Eigene Funktionen definieren

Das Definieren eigener Funktionen eröffnet vielfältige Möglichkeiten in Python:

- Komplexe Programme können mit einer einzigen Zeile Code aufgerufen und ausgeführt werden.
- Funktionen können praktisch beliebig oft aufgerufen werden und sind durch den Einsatz von Parametern und Methoden der Flusskontrolle gleichzeitig in der Lage, flexibel auf wechselnde Bedingungen zu reagieren.
- Funktionen machen Programmcode kürzer und lesbarer. Außerdem gibt es nur eine Stelle, welche bei Änderungen angepasst werden muss.

5.1 Syntax

Das Schlüsselwort `def` leitet die Funktionsdefinition ein. Es wird gefolgt vom Funktionsnamen und den Funktionsparametern, welche in runden Klammern `()` eingeschlossen sind. Der Funktionskopf wird mit einem Doppelpunkt `:` beendet. Der Anweisungsblock der Funktion ist eingerückt. Jede Funktion liefert einen Rückgabewert, welche durch das Schlüsselwort `return` an die aufrufende Stelle zurückgegeben wird. `return` beendet die Ausführung der Schleife, auch wenn es nicht am Ende des Anweisungsblocks steht.

```
def Funktionsname(Parameter1, Parameter2):
    Anweisungsblock
    return Rückgabewert
```

Damit die Funktion ausgeführt wird, muss der definierte Funktionsname aufgerufen werden. In der Funktion ist nach dem Schlüsselwort `return` eine weitere Anweisung enthalten, die nicht mehr ausgeführt wird.

```
# Beispiel 1: Summe der Quadrate

# Definition einer Funktion zur Berechnung der Summe der Quadrate von zwei Argumenten
def sum_quadrate(a, b):
    print('Argument a:', a)
```

```
    print('Argument b:', b)
    print(18 * '=')
    summe = a**2 + b**2
    return summe
    print("Anweisungen nach dem Schlüsselwort return werden nicht mehr ausgeführt.")

print(sum_quadrate(6, 7))
```

```
Argument a: 6
Argument b: 7
==================
85
```

Der Rückgabewert kann in einer Variablen gespeichert werden.

```
ergebnis = sum_quadrate(6, 7)
print(ergebnis)
```

```
Argument a: 6
Argument b: 7
==================
85
```

5.2 Optionale Parameter

Mit Hilfe von optionalen Parametern kann die Programmausführung gesteuert werden. Optionale Parameter müssen nach verpflichtend zu übergebenen Parametern definiert werden. In diesem Beispiel wird die print-Ausgabe der Funktion mit dem Parameter `ausgabe` gesteuert.

```
# Beispiel 2: optionale Argumente

# Definition einer Funktion zur Berechnung der Summe der Quadrate von zwei Argumenten
def sum_quadrate(a, b, ausgabe = False):
    if ausgabe:
      print('Wert Argument a:', a)
      print('Wert Argument b:', b)
      print(18 * '=')
    summe = a**2 + b**2
    return summe
```

```
print(sum_quadrate(42, 7), "\n")
print(sum_quadrate(42, 7, ausgabe = True))
```

```
1813

Wert Argument a: 42
Wert Argument b: 7
==================
1813
```

Gibt es mehrere optionale Parameter, so erfolgt die Zuweisung von Argumenten positional oder über das Schlüsselwort.

```
# Beispiel 3: mehrere optionale Argumente

# Definition einer Funktion zur Berechnung der Summe der Quadrate von zwei Argumenten
def sum_potenzen(a, b, p = 2, ausgabe = False):
    if ausgabe:
      print('Argument a:', a)
      print('Argument b:', b)
      print('Argument p:', p)
      print(18 * '=')
    summe = a**p + b**p
    return summe

# positionale Übergabe
print(sum_potenzen(42, 7, 3, True), "\n")

# Übergabe per Schlüsselwort
print(sum_potenzen(42, 7, ausgabe = True, p = 4))
```

```
Argument a: 42
Argument b: 7
Argument p: 3
==================
74431

Argument a: 42
Argument b: 7
Argument p: 4
==================
3114097
```

5.3 Rückgabewert(e)

Funktionen können in Python nur einen einzigen Rückgabewert haben. Trotzdem können mehrere Rückgabewerte mit einem Komma getrennt werden. Python gibt diese als Tupel zurück.

```
# Beispiel 4: mehrere Rückgabewerte

# Definition einer Funktion zur Berechnung der Summe der Quadrate von zwei Argumenten
def sum_potenzen(a, b, p = 2, ausgabe = False):
    if ausgabe:
      print('Argument a:', a)
      print('Argument b:', b)
      print('Argument p:', p)
      print(18 * '=')
    summe = a**p + b**p
    return a, b, summe

ergebnis = sum_potenzen(2, 7, ausgabe = False, p = 4)
print(ergebnis, type(ergebnis))
```

```
(2, 7, 2417) <class 'tuple'>
```

Mit dem Slice Operator kann ein bestimmter Rückgabewert ausgewählt werden.

```
print(ergebnis[2])

summe_potenzen = sum_potenzen(2, 7, ausgabe = False, p = 4)[2]
print(summe_potenzen, type(summe_potenzen))
```

```
2417
2417 <class 'int'>
```

5.4 Aufgaben Funktionen definieren

1. Palindrom
   Schreiben Sie eine Funktion is_palindrome(), die prüft, ob es sich bei einer übergebenen Zeichenkette um ein Palindrom handelt.

Hinweis: Ein Palindrom ist eine Zeichenkette, die von vorne und von hinten gelesen gleich bleibt, wie beispielsweise 'Anna', 'Otto', 'Lagerregal'. Palindrome müssen nicht aus Buchstaben bestehen, sie können sich auch aus Zahlen oder Buchstaben und Zahlen zusammensetzen wie beispielsweise '345g543'.

2. Fibonacci-Zahlenreihe

Entwickeln Sie eine Funktion fibonacci(n), die die ersten n Zahlen der Fibonacci-Reihe generiert und als Liste zurückgibt. Die Fibonacci-Reihe beginnt mit 0 und 1, jede weitere Zahl ist die Summe der beiden vorhergehenden Zahlen.

3. Verschlüsselung

Bei Geocachen werden oft verschlüsselte Botschaften als Rätsel verwendet. Oft wird folgende Logik zur Verschlüsselung angewendet:

- A -> Z
- B -> Y
- C -> X
- …

Schreiben Sie eine Funktion `verschluesseln(str)`, die einen String als Eingabewert bekommt und einen verschlüsselten String zurückgibt. Wie können Sie einen verschlüsselten String am einfachsten wieder entschlüsseln?

4. Temperaturkonverter

Entwickeln Sie eine Funktion `temperatur_umrechnen(wert, von_einheit, nach_einheit)`, die eine Temperatur von einer Einheit in eine andere umwandelt. Die Funktion soll folgende Parameter verwenden:

- wert: Der Temperaturwert, der umgewandelt werden soll.
- von_einheit / nach_einheit: Die Einheit des Ausgangs- bzw. des Zielwerts als string. Mögliche Werte sind 'C' für Celsius, 'F' für Fahrenheit und 'K' für Kelvin.

Es gelten die folgenden Umrechnungsformeln zwischen den Einheiten:

- Celsius nach Fahrenheit: F = C * 9/5 + 32
- Fahrenheit nach Celsius: C = (F - 32) * 5/9
- Celsius nach Kelvin: K = C + 273.15
- Kelvin nach Celsius: C = K - 273.15
- Fahrenheit nach Kelvin: K = (F - 32) * 5/9 + 273.15

- Kelvin nach Fahrenheit: F = (K - 273.15) * 9/5 + 32

Die Musterlösung kann Marc machen

> Musterlösung Aufgaben Funktionen definieren

(Arnold (2023b))

6 Dateien lesen und schreiben

Maya und Hans haben je sechs Mal einen Würfel geworfen und ihre Wurfergebnisse in einer .txt-Datei protokolliert. Wir wollen mit die Dateien mit Python auswerten, um zu bestimmen, wer von beiden in Summe die höchste Augenzahl erreicht hat.

Daten	Dateiname
Würfelergebnisse Maya	dice-maya.txt
Würfelergebnisse Hans	dice-hans.txt

6.1 Dateiobjekte

Um mit Python auf eine Datei zuzugreifen, muss diese fürs Lesen oder Schreiben geöffnet werden. Dazu wird in Python die Funktion open verwendet. Diese nimmt zwei Argumente, den Pfad der Datei und den Zugriffsmodus, an und liefert ein Dateiobjekt zurück. Aus dem Dateiobjekt werden dann die Inhalte der Datei ausgelesen.

6.1.1 Dateipfad

Der lokale Dateipfad wird ausgehend vom aktuellen Arbeitsverzeichnis angegeben.

```
pfad_maya = "01-daten/dice-maya.txt"
pfad_hans = "01-daten/dice-hans.txt"
```

Tip 11: Arbeitsverzeichnis in Python ermitteln und wechseln

Der Pfad des aktuellen Arbeitsverzeichnisses kann mit dem Modul os mittels `os.getcwd()` ermittelt werden (hier ohne Ausgabe). Mit `os.chdir('neuer_pfad')` kann das Arbeitsverzeichnis ggf. gewechselt werden. Die korrekte Formatierung des Pfads erkennen Sie an der Ausgabe von `os.getcwd()`.

```
import os
print(os.getcwd())
```

Das Importieren von Modulen wird in einem späteren Kapitel behandelt.

6.1.2 Zugriffsmodus

Als Zugriffsmodus stehen unter anderem folgende Optionen zur Verfügung:

Modus	Beschreibung
`r`	lesender Zugriff
`w`	Schreibzugriff, Datei wird überschrieben
`x`	Erzeugt die Datei, Fehlermeldung, wenn die Datei bereits existiert
`a`	Schreibzugriff, Inhalte werden angehängt
`b`	Binärmodus (z. B. für Grafiken)
`t`	Textmodus, default

Die Zugriffsmodi können auch kombiniert werden. Weitere Informationen dazu finden Sie in der Dokumentation. Sofern nicht im Binärmodus auf Dateien zugegriffen wird, liefert die Funktion `open()` den Dateiinhalt als string zurück.

Im Lesemodus wird ein Datenobjekt erzeugt.

```
daten_maya = open(pfad_maya, mode = 'r')
print(daten_maya)
```

```
<_io.TextIOWrapper name='01-daten/dice-maya.txt' mode='r' encoding='UTF-8'>
```

Wenn das Datenobjekt `daten_maya` der Funktion `print()` übergeben wird, gibt Python die Klasse des Objekts zurück, in diesem Fall also _io.TextIOWrapper. Diese Klasse stammt aus dem Modul io und ist für das Lesen und Schreiben von Textdateien zuständig. Ebenfalls werden als Attribute des Dateiobjekts der Dateipfad, der Zugriffsmodus und die Enkodierung der Datei ausgegeben (siehe Beispiel 2). Sollte die Enkodierung nicht automatisch als UTF-8 erkannt werden, kann diese mit dem Argument `encoding = 'UTF-8'` übergeben werden.

```
daten_maya = open(pfad_maya, mode = 'r', encoding = 'UTF-8')
print(daten_maya)
```

```
<_io.TextIOWrapper name='01-daten/dice-maya.txt' mode='r' encoding='UTF-8'>
```

Note 2: Attribute eines Objekts bestimmen

Mit der Funktion `dir(objekt)` können die verfügbaren Attribute eines Objekts ausgegeben werden. Dabei werden jedoch auch die vererbten Attribute und Methoden der Klasse des Objekts ausgegeben, sodass die Ausgabe oft sehr umfangreich ist. Zum Beispiel für die Ganzzahl 1:

```
print(dir(1))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__di
```

Um die Ausgabe auf Attribute einzugrenzen, kann folgende Funktion verwendet werden:

```
objekt = 1

attribute = [attr for attr in dir(objekt) if not callable (getattr(objekt, attr))]
print(attribute)
```

```
['__doc__', 'denominator', 'imag', 'numerator', 'real']
```

Mit doppelten Unterstrichen umschlossene Attribute sind für Python reserviert und nicht für den:die Nutzer:in gedacht. Folgende Funktion entfernt Attribute mit doppelten Unterstrichen aus der Ausgabe:

```
objekt = 1

attribute = [attr for attr in dir(objekt) if not (callable(getattr(objekt, attr)) or attr.s
print(attribute)
```

```
['denominator', 'imag', 'numerator', 'real']
```

Im Fall einer Ganzzahl können Attribute (zur Abgrenzung von Gleitkommazahlen in umschließenden Klammern) wie folgt aufgerufen werden:

```
(1).numerator
```

```
1
```

Wenn wir uns die Attribute des Dateiobjekts ‘daten_maya’ ansehen, fallen Attribute mit einem einzelnen führenden Unterstrich auf.

```
objekt = daten_maya

attribute = [attr for attr in dir(objekt) if not (callable(getattr(objekt, attr)) or attr.s
print(attribute)
```

```
['_CHUNK_SIZE', '_finalizing', 'buffer', 'closed', 'encoding', 'errors', 'line_buffering',
```

Hierbei handelt es sich um Attribute, die nicht durch den:die Nutzer:in aufgerufen werden sollen (weitere Informationen dazu finden Sie hier). Folgender Programmcode gibt alle Attribute ohne führende Unterstriche aus:

```
objekt = daten_maya

attribute = [attr for attr in dir(objekt) if not (callable(getattr(objekt, attr)) or attr.s
print(attribute)
```

```
['buffer', 'closed', 'encoding', 'errors', 'line_buffering', 'mode', 'name', 'newlines', 'w
```

Die Attribute der Datei können mit entsprechenden Befehlen abgerufen werden.

```
print(f"Dateipfad: {daten_maya.name}\n"
      f"Dateiname: {os.path.basename(daten_maya.name)}\n"
      f"Datei ist geschlossen: {daten_maya.closed}\n"
      f"Zugriffsmodus: {daten_maya.mode}"
      f"Enkodierung: : {daten_maya.encoding}")
```

```
Dateipfad: 01-daten/dice-maya.txt
Dateiname: dice-maya.txt
Datei ist geschlossen: False
Zugriffsmodus: rEnkodierung: : UTF-8
```

Tip 12: Rückfalloption

In der Datenanalyse werden in der Regel spezialisierte Pakete wie NumPy oder Pandas verwendet. Diese vereinfachen das Einlesen von Dateien gegenüber der Pythonbasis erheblich. Dennoch ist es sinnvoll, sich mit den Methoden der Pythonbasis zum Einlesen von Dateien vertraut zu machen. Denn das Einlesen mit der Funktion `open()` klappt so gut wie immer - es ist eine gute Rückfalloption.

6.1.3 Dateiinhalt ausgeben

Um den Dateiinhalt auszugeben, kann das Datenobjekt mit einer Schleife zeilenweise durchlaufen und ausgegeben werden. (Die Datei dice-maya hat nur eine Zeile.)

```
i = 0
for zeile in daten_maya:
    print(f"Inhalt Zeile {i}, mit {len(zeile)} Zeichen:")
    print(zeile)
    i += 1
```

```
Inhalt Zeile 0, mit 28 Zeichen:
"5", "6", "2", "1", "4", "5"
```

Dies ist jedoch für größere Dateien nicht sonderlich praktikabel. Die Ausgabe einzelner Zeilen mit der Funktion `print()` kann aber nützlich sein, um die genaue Formatierung der Zeichenkette zu prüfen. In diesem Fall hat Maya ihre Daten in Anführungszeichen gesetzt und mit einem Komma voneinander getrennt.

6.2 Dateien einlesen

Um den gesamten Inhalt einer Datei einzulesen, kann die Methode datenobjekt.read() verwendet werden. Die Methode hat als optionalen Parameter `.read(size)`. size wird als Ganzzahl übergeben und entsprechend viele Zeichen (im Binärmodus entsprechend viele Bytes) werden ggf. bis zum Dateiende ausgelesen. Der Parameter `size` ist nützlich, um die Formatierung des Inhalts einer großen Datei zu prüfen und dabei die Ausgabe auf eine überschaubare Anzahl von Zeichen zu begrenzen.

```
augen_maya = daten_maya.read()

print(f"len(augen_maya): {len(augen_maya)}\n\n"
      f"Inhalt der Datei augen_maya:\n{augen_maya}")
```

```
len(augen_maya): 0

Inhalt der Datei augen_maya:
```

Das hat offensichtlich nicht geklappt, der ausgelesene Dateiinhalt ist leer! Der Grund dafür ist, dass beim Lesen (und beim Schreiben) einer Datei der Dateizeiger die Datei durchläuft. Nachdem die Datei daten_maya in Kapitel 6.1.3 zeilenweise ausgegeben wurde, steht der Dateizeiger am Ende der Datei.

Warning 3: Dateizeiger in Python

Wird eine Datei zeilenweise oder mit der Methode `.read()` ausgelesen, wird der Dateizeiger um die angegebene Zeichenzahl bzw. bis ans Ende der Datei bewegt. Wird beispielsweise ein Datensatz 'daten' geöffnet und mit der Methode `daten.read(3)` die ersten drei Zeichen ausgelesen, bewegt sich der Dateizeiger von der Indexposition 0 zur Indexposition 3 (bzw. steht jeweils davor).

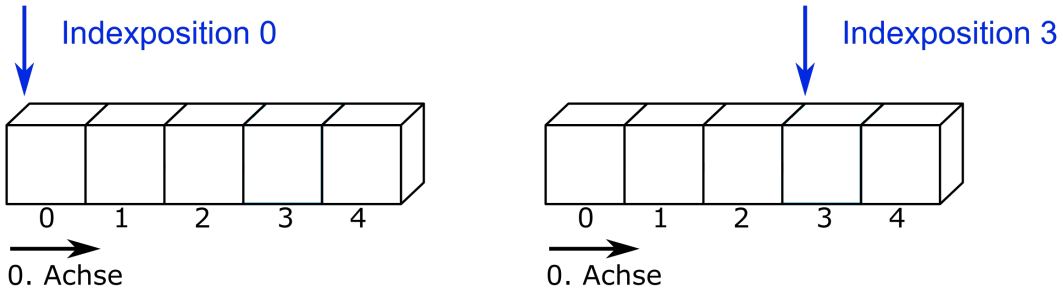


Abbildung 6.1: Bewegung des Dateizeigers beim Auslesen von drei Zeichen

Die Methode `daten.tell()` gibt zurück, an welcher Position sich der Dateizeiger befindet.
Mit der Methode `daten.seek(offset, whence = 0)` wird der Zeiger an eine bestimmte Position gesetzt. Die Methode akzeptiert das Argument offset (Versatz) und das optionale Argument whence (woher), dessen Standardwert 0 (Dateianfang) ist. Für Zugriffe **im Binärmodus** (`open(pfad, mode = 'rb')`) kann das Argument whence außerdem die Werte 1 (aktuelle Position) oder 2 (Dateiende) annehmen.

- `daten.seek(0, 0)` bezeichnet den Dateianfang
- `daten.seek(0, 1)` bezeichnet die aktuelle Position in der Datei
- `daten.seek(0, 2)` bezeichnet das Dateiende
- `daten.seek(-3, 2)` bezeichnet das dritte Zeichen vor dem Dateiende

Wird der Dateizeiger mit der Methode `datenobjekt.seek(0)` an den Dateianfang gestellt, gelingt das Auslesen der Datei.

```
print(f"Position des Dateizeigers vor dem Zurücksetzen auf 0: {daten_maya.tell()}")

daten_maya.seek(0);
print(f"Position des Dateizeigers nach dem Zurücksetzen auf 0: {daten_maya.tell()}")
```

```
augen_maya = daten_maya.read()

print(f"Inhalt des Objekts augen_maya:\n{augen_maya}")
```

```
Position des Dateizeigers vor dem Zurücksetzen auf 0: 28
Position des Dateizeigers nach dem Zurücksetzen auf 0: 0
Inhalt des Objekts augen_maya:
"5", "6", "2", "1", "4", "5"
```

Geben Sie aus dem Datenobjekt daten_maya mit den Methoden .seek() und .read() die Zahlen and zweiter und dritter Stelle, also 6 und 2, aus.

Tip 13: Musterlösung Dateizeiger bewegen

```
daten_maya.seek(6, 0);
print(daten_maya.read(1))

daten_maya.seek(daten_maya.tell() + 4, 0);
print(daten_maya.read(1))
```

```
6
2
```

Um Mayas Würfelergebnisse zu addieren, müssen die Zahlen extrahiert und in Ganzzahlen umgewandelt werden, da im Textmodus stets eine Zeichenfolge zurückgegeben wird.

```
print(type(augen_maya))
```

```
<class 'str'>
```

Dazu werden mit der Methode `str.strip('')` das führende und abschließende Anführungszeichen entfernt sowie anschließend mit der Methode `str.split('", "')` die Zeichenfolge über das Trennzeichen in eine Liste aufgeteilt. Anschließend werden die Listenelemente in Ganzzahlen umgewandelt und summiert. (Methoden der string-Bearbeitung werden im nächsten Abschnitt ausführlich behandelt.)

```
print(f"augen_maya:\n{augen_maya}")

augen_maya = augen_maya.strip('"')
print(f"\naugen_maya.strip('\"'):\n{augen_maya}")
```

```
augen_maya = augen_maya.split('", "')
print(f"\naugen_maya.split('\", \"'):\n{augen_maya}")

augen_maya_int = []
for i in augen_maya:
  augen_maya_int.append(int(i))

print(f"\naugen_maya_int:\n{augen_maya_int}\n\nSumme Augen: {sum(augen_maya_int)}")
```

```
augen_maya:
"5", "6", "2", "1", "4", "5"

augen_maya.strip('"'):
5", "6", "2", "1", "4", "5

augen_maya.split('", "'):
['5', '6', '2', '1', '4', '5']

augen_maya_int:
[5, 6, 2, 1, 4, 5]

Summe Augen: 23
```

Datei schließen

Nach dem Zugriff auf die Datei, muss diese wieder geschlossen werden, um diese für andere Programme freizugeben.

```
daten_maya.close()
```

> **Warning 4: Schreiboperationen mit Python**
>
> Das Schließen einer Datei ist besonders für Schreiboperationen auf Datenobjekten wichtig. Andernfalls kann es passieren, dass Inhalte mit `datenobjekt.write()` nicht vollständig auf den Datenträger geschrieben werden. Siehe dazu die Dokumentation.

6.3 Aufgabe Dateien einlesen

Welche Augenzahl hat Hans erreicht?

Tip 14: Musterlösung Augenzahlvergleich

```
# Erst Einlesen der Datei:
daten_hans = open(pfad_hans, mode = 'r', encoding = 'UTF-8')
augen_hans = daten_hans.read()
print(augen_hans)
# Hier muss man erkennen, dass Hans seinen Namen an den Anfang seiner Liste gesetzt hat. Di

augen_hans = augen_hans.strip('"Hans", ')
augen_hans = augen_hans.strip('"')
augen_hans = augen_hans.split('", "')
print(augen_hans)
# print-Ausgabe zeigt, dass die Liste nun korrekt bereinigt wurde. Sie besteht nur noch aus

# Neue (leere) Liste für die Würfe von Hans anlegen:
augen_hans_int = []
for i in augen_hans:
   augen_hans_int.append(int(i))

print(f"Summe Augenzahl von Hans: {sum(augen_hans_int)}")
```

```
"Hans", "3", "5", "1", "3", "2", "5"
['3', '5', '1', '3', '2', '5']
Summe Augenzahl von Hans: 19
```

Musterlösung von Marc Sönnecken.

6.4 Daten interpretieren

Datensätze liegen typischerweise wenigstens in zweidimensionaler Form vor, d. h. die Daten sind in Zeilen und Spalten organisiert. Außerdem weisen Datensätze in der Regel auch unterschiedliche Datentypen auf. Die Funktion `open(datei)` gibt ein Dateiobjekt zurück, das mit Methoden wie zum Beispiel `dateiobjekt.read()` als Zeichenfolge eingelesen wird. Um die Daten sinnvoll weiterverarbeiten zu können, ist es deshalb notwendig, die Zeichenfolge korrekt zu interpretieren und Daten von Trennzeichen zu unterscheiden.

Für die Bearbeitung von Zeichenfolgen bietet Python eine Reihe von String-Methoden. Einige davon werden in diesem Kapitel exemplarisch verwendet. String-Methoden werden in der Regel mit einem führenden ‘str’ in der Form `str.methode()` genannt.

Beispielsweise soll eine Datei mit den Einwohnerzahlen der europäischen Länder eingelesen werden.

Daten	Dateiname
Einwohner Europas	einwohner_europa_2019.csv

Um einen Überblick über den Aufbau der Datei zu erhalten, werden die ersten drei Zeilen der Datei ausgegeben. Dafür kann die Datei zeilenweise mit einer for-Schleife durchlaufen werden, die mit dem Schlüsselwort `break` abgebrochen wird, wenn die Laufvariable den Wert 3 erreicht hat. Eine andere Möglichkeit ist die Methode `dateiobjekt.readline()`, die eine einzelne Zeile ausliest. Hier wird die Häufigkeit der Schleifenausführung über die Laufvariable mit `for i in range(3):` gesteuert.

6.5 for-Schleife mit break

```
dateipfad = "01-daten/einwohner_europa_2019.csv"
dateiobjekt_einwohner = open(dateipfad, 'r')

# erste 3 Zeilen anschauen
i = 0
for zeile in dateiobjekt_einwohner:

  print(zeile)
  i += 1
  if i == 3:
    break

# Datei schließen
dateiobjekt_einwohner.close()
```

```
GEO,Value

Belgien,11467923

Bulgarien,7000039
```

6.6 Methode dateiobjekt.readline()

Mit der Methode `dateiobjekt.readline()` kann eine einzelne Zeile eingelesen werden.

```
dateipfad = "01-daten/einwohner_europa_2019.csv"
dateiobjekt_einwohner = open(dateipfad, 'r')

for i in range(3):
  print(dateiobjekt_einwohner.readline())

# Datei schließen
dateiobjekt_einwohner.close()
```

```
GEO,Value

Belgien,11467923

Bulgarien,7000039
```

Die Datei hat also zwei Spalten. In der ersten Spalte sind die Ländernamen eingetragen, in der zweiten Spalte die Werte. Als Trennzeichen wird das Komma verwendet. In der ersten Zeile sind die Spaltenbeschriftungen eingetragen.

Im vorherigen Abschnitt haben wir die Methode `dateiobjekt.read()` kennengelernt, mit der eine Datei vollständig als string eingelesen wird. Zunächst wird die Datei mit der Methode `dateiobjekt.read()` in das Objekt einwohner eingelesen und wieder geschlossen.

```
dateipfad = "01-daten/einwohner_europa_2019.csv"
dateiobjekt_einwohner = open(dateipfad, 'r')

einwohner = dateiobjekt_einwohner.read()
print(einwohner)

# Datei schließen
dateiobjekt_einwohner.close();
```

```
GEO,Value
Belgien,11467923
Bulgarien,7000039
Tschechien,10528984
Daenemark,5799763
```

```
Deutschland einschliesslich ehemalige DDR,82940663
Estland,1324820
Irland,4904240
Griechenland,10722287
Spanien,46934632
Frankreich,67028048
Kroatien,4076246
Italien,61068437
Zypern,875898
Lettland,1919968
Litauen,2794184
Luxemburg,612179
Uganda,-1
Ungarn,9772756
Malta,493559
Niederlande,17423013
Oesterreich,8842000
Polen,37972812
Portugal,10276617
Rumaenien,19405156
Slowenien,2080908
Slowakei,5450421
Finnland,5512119
Schweden,10243000
Vereinigtes Koenigreich,66647112
```

Anschließend können die eingelesenen Daten mit der Methode `str.split('\n')` zeilweise aufgeteilt werden. Mit `'\n'` wird als Argument der Zeilenumbruch übergeben. Die Methode liefert eine Liste zurück.

```
liste_einwohner_zeilenweise = einwohner.split("\n")
print(liste_einwohner_zeilenweise[0:3])
```

```
['GEO,Value', 'Belgien,11467923', 'Bulgarien,7000039']
```

Die Liste enthält an der Indexposition die Spaltenbeschriftungen. Diese können mit der Methode `liste.pop(index)` aus der Liste entfernt und zugleich in einem neuen Objekt gespeichert werden.

```
spaltennamen = liste_einwohner_zeilenweise.pop(0)
spaltennamen = spaltennamen.split(',')
print(f"Überschrift Spalte 0: {spaltennamen[0]}\tÜberschrift Spalte 1: {spaltennamen[1]}")
```

```
Überschrift Spalte 0: GEO   Überschrift Spalte 1: Value
```

Anschließend kann die Liste mit der Methode `str.split(',')` nach Ländern und Werten aufgeteilt werden. Der Vorgang bricht allerdings mit einer Fehlermeldung ab. Die Fehlermeldung wird im folgenden Code-Block per Ausnahmebehandlung abgefangen. Neben der Fehlermeldung werden der verursachende Listeneintrag und dessen Indexposition ausgegeben.

```
# Leere Listen vor der Schleife anlegen
geo = []
einwohnerzahl = []

try:
  for zeile in liste_einwohner_zeilenweise:
    eintrag = zeile.split(',')
    geo.append(eintrag[0])
    einwohnerzahl.append(eintrag[1])

  print(spaltennamen[0])
  print(geo, "\n")

  print(spaltennamen[1])
  print(einwohnerzahl)

except Exception as error:
  # print Fehlermeldung
  print(f"Fehlermeldung: {error}")

  # print Eintrag und Index
  print(f"Eintrag: {eintrag}\t Zeilenindex: {liste_einwohner_zeilenweise.index(zeile)}")
```

```
Fehlermeldung: list index out of range
Eintrag: ['']    Zeilenindex: 29
```

Die Fehlermeldung ist so zu deuten, dass eine der Listenoperationen mit dem Slice Operator einen ungültigen Index anspricht. Leicht angepasst, liefert der Code-Block auch die Ursache der Fehlermeldung.

Wird die leere Zeile aus der Liste entfernt, klappt das Aufteilen der Ländernamen und der Werte.

```
# leere Zeile entfernen
liste_einwohner_zeilenweise.remove('')

# Leere Listen vor der Schleife anlegen
geo = []
einwohnerzahl = []

try:
  for zeile in liste_einwohner_zeilenweise:
    eintrag = zeile.split(',')
    geo.append(eintrag[0])
    einwohnerzahl.append(eintrag[1])

  print(spaltennamen[0])
  print(geo, "\n")

  print(spaltennamen[1])
  print(einwohnerzahl)

except IndexError as error:
  print(error)
```

```
GEO
['Belgien', 'Bulgarien', 'Tschechien', 'Daenemark', 'Deutschland einschliesslich ehemalige DI

Value
['11467923', '7000039', '10528984', '5799763', '82940663', '1324820', '4904240', '10722287',
```

6.7 Aufgabe Daten interpretieren

1. Bestimmen Sie das Minimum und das Maximum der Einwohnerzahl und die dazugehörigen Länder.
2. Bereinigen Sie ggf. fehlerhafte Werte.
3. Wie viele Einwohner leben in Europa insgesamt?

- Welchen Datentyp hat die Liste einwohnerzahl?
- Welchen Datentyp haben die Einträge der Liste einwohnerzahl?

Die Musterlösung kann Marc machen

Musterlösung vollständiges Einlesen

6.8 Einlesen als Liste

Ein Dateiobjekt kann auch direkt als Liste eingelesen werden. Die Methode `dateiobjekt.readlines()` gibt eine Liste zurück, in der jede Zeile einen Eintrag darstellt. Ebenso kann die Listenfunktion `list()` auf Dateiobjekte angewendet werden. Beide Vorgehensweisen liefern die gleiche Liste zurück, in der der Zeilenumbruch `\n` mit ausgelesen wird.

```
dateipfad = "01-daten/einwohner_europa_2019.csv"
dateiobjekt_einwohner = open(dateipfad, 'r')

# Methode readlines
einwohner = dateiobjekt_einwohner.readlines()
print(einwohner)

## Dateizeiger zurücksetzen
dateiobjekt_einwohner.seek(0);

# Funktion list
einwohner = list(dateiobjekt_einwohner)
print(einwohner)

# Datei schließen
dateiobjekt_einwohner.close();
```

```
['GEO,Value\n', 'Belgien,11467923\n', 'Bulgarien,7000039\n', 'Tschechien,10528984\n', 'Daener
['GEO,Value\n', 'Belgien,11467923\n', 'Bulgarien,7000039\n', 'Tschechien,10528984\n', 'Daener
```

Um den Zeilenumbruch zu entfernen, könnte mit dem Slice Operator das letzte Zeichen jedes Listeneintrags entfernt werden.

Eine andere Möglichkeit ist die Methode `str.replace(old, new, count=-1)`, mit der Zeichen ersetzt oder gelöscht werden können. Die Parameter `old` und `new` geben die zu ersetzende bzw. die einzusetzende Zeichenfolge an und *müssen positional* übergeben werden. Über den Parameter `count` kann eingestellt werden, wie oft die Zeichenfolge `old` ersetzt werden soll. Standardmäßig wird jedes Vorkommen ersetzt.

```
print('Hund'.replace('Hu', 'Mu'))

zeichenfolge = 'Ein  kurzer Text ohne  doppelte Leerzeichen.'

print(zeichenfolge.replace('  ', ' '))
```

```
Mund
Ein kurzer Text ohne doppelte Leerzeichen.
```

Die Methode `str.replace()` kann auch zum Löschen verwendet werden. Wird für den Parameter `new` eine leere Zeichenfolge übergeben, wird die in `old` übergebene Zeichenfolge gelöscht.

```
print(zeichenfolge.replace(' ', '').replace('doppelte', ''))
```

```
EinkurzerTextohneLeerzeichen.
```

Mit der Methode `str.replace()` kann die eingelesene Liste um den Zeilenumbruch bereinigt werden.

```
dateipfad = "01-daten/einwohner_europa_2019.csv"
dateiobjekt_einwohner = open(dateipfad, 'r')

# Methode readlines
einwohner = dateiobjekt_einwohner.readlines()
einwohner_neu = []

for element in einwohner:
  einwohner_neu.append(element.replace('\n', ''))

einwohner = einwohner_neu
print(einwohner)

# Datei schließen
dateiobjekt_einwohner.close();
```

```
['GEO,Value', 'Belgien,11467923', 'Bulgarien,7000039', 'Tschechien,10528984', 'Daenemark,579
```

6.9 Dateien schreiben

Um Dateien zu schreiben, müssen diese mit der `write`-Methode eines Dateiobjekts verwendet werden. Dieser Methode wird als Argument die zu schreibende Zeichenfolge übergeben.

```
dateipfad = "01-daten/neue_datei.txt"

# Öffne Datei zum Schreiben öffnen
datei = open(dateipfad, mode = 'w')

# Inhalt in die Datei schreiben
datei.write("Prokrastination an Hochschulen\n\n".upper())
datei.write("KAPITEL 1: Aller Anfang ist schwer\nPlatzhalter: Den Rest schreibe ich später.")

# Datei schließen

datei.close()
```

Die Datei kann nun ausgelesen werden.

```
dateiinhalt = open(dateipfad, mode = 'r')
text = dateiinhalt.read()
print(text)

dateiinhalt.close()
```

```
PROKRASTINATION AN HOCHSCHULEN

KAPITEL 1: Aller Anfang ist schwer
Platzhalter: Den Rest schreibe ich später.
```

6.10 Aufgabe Dateien schreiben

1. Erzeugen Sie eine neue Datei mit der Endung `.txt`, die den Namen ihrer Heimatstadt hat. Schreiben Sie in diese Datei 10 Zeilen mit Informationen zur Stadt.

(Arnold (2023b))

7 Module und Pakete importieren

Der Funktionsumfang von Python kann erheblich durch das Importieren von Modulen und Paketen erweitert werden. Module und Pakete sind Bibliotheken, die Funktionsdefinitionen enthalten.

> **!** Important 1: Module und Pakete
>
> **Module** Module sind Dateien, die Funktionsdefinitionen enthalten.
> Module werden durch das Schlüsselwort `import` und ihren Namen importiert, bspw. `import glob`
>
> **Pakete** Pakete sind Sammlungen von Modulen
> In Paketen enthaltene Module werden durch das Schlüsselwort `import` mit der Schreibeweise paket.modul importiert, bspw. `import matplotlib.pyplot`

Module und Pakete werden mit dem Schlüsselwort `import` in Python geladen. Beispielsweise kann das für die Erzeugung (pseudo-)zufälliger Zahlen zuständige Modul random mit dem Befehl `import random` eingebunden werden. Anschließend stehen die Funktionen des Moduls unter dem Modulnamen in der Schreibweise `modul.funktion()` zur Verfügung.

```
import random

print(random.randint(1, 10)) # Zufällige Ganzzahl zwischen 1 und 10
```

```
1
```

Das Paket Matplotlib bringt viele Funktionen zur grafischen Darstellung von Daten mit. Das Modul `matplotlib.pyplot` stellt eine Schnittstelle zu den enthaltenen Funktionen dar.

```
import matplotlib.pyplot

zufallsdaten = [] # leere Liste anlegen
for i in range(10):
    zufallszahl = random.randint(1, 10)
    zufallsdaten.append(zufallszahl)

matplotlib.pyplot.plot(zufallsdaten)
```

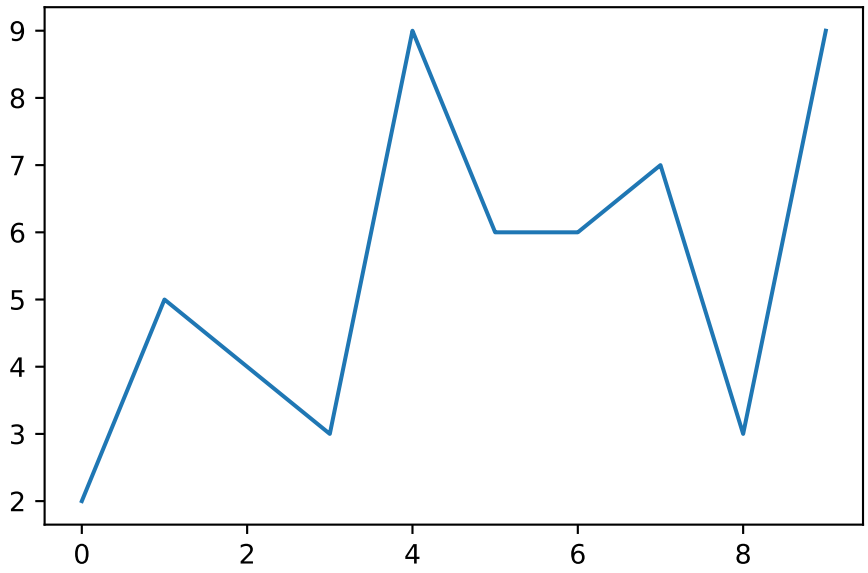


Abbildung 7.1: Grafik mit dem Modul pyplot aus dem Paket matplotlib

Warning 5: Namensraum direkt einbinden

In Python ist es auch möglich, Funktionen direkt in den Namensraum von Python zu importieren, sodass diese ohne die Schreibweise `modul.funktion()` aufgerufen werden können. Dies ist mit dem Schlüsselwort `from` möglich.

```
from random import randint
print(f"Die Funktion randint steht nun direkt zur Verfügung: {randint(1, 100)}")
```

```
Die Funktion randint steht nun direkt zur Verfügung: 72
```

Durch `from modulname import *` ist es sogar möglich, alle Funktionen aus einem Modul in den Namensraum von Python zu importieren. Im Allgemeinen sollte das direkte Importieren von Funktionen oder eines ganzen Moduls in den Namensraum von Python jedoch unterlassen werden. Einerseits wird damit eine Namensraumkollision riskiert, beispielsweise gibt es die Funktion `sum()` in der Pythonbasis, in NumPy und in Pandas. Andererseits wird der Programmcode dadurch weniger nachvollziehbar, da nicht mehr überall ersichtlich ist, aus welchem Modul eine verwendete Funktion stammt.

7.1 import as

Um lange Modulnamen zu vereinfachen, kann beim Importieren das Schlüsselwort `as` verwendet werden, um dem Modul einen neuen Namen zuzuweisen.

```
import matplotlib.pyplot as plt

plt.plot(zufallsdaten)
```

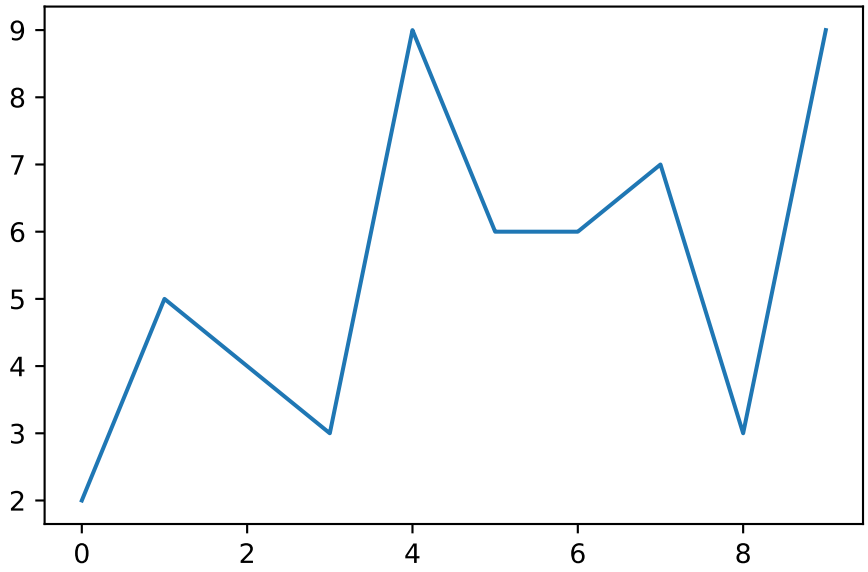


Abbildung 7.2: Grafik mit dem Modul pyplot aus dem Paket matplotlib

Für häufig verwendete Module haben sich bestimmte Kürzel etabliert. In den Bausteinen werden häufig die folgenden Pakete und Kürzel genutzt:

Modul	Kürzel	Befehl
NumPy	np	import numpy as np
Pandas	pd	import pandas as pd
matplotlib.pyplot	plt	import matplotlib.pyplot as plt

7.2 Kleine Modulübersicht

Da es nicht möglich ist, auf alle diese Module einzugehen, werden im folgenden nur einige wenige Module aufgelistet, welche für die Zielgruppe dieses Skripts interessant sein könnten. *Hinweis: Die Eigennamen einiger Module weisen eine Groß- und Kleinschreibung auf, bspw. das Modul NumPy. Beim Importieren der Module werden die Modulnamen jedoch klein geschrieben.* In der folgenden Liste wird auf die Groß- und Kleinschreibung daher verzichtet.

- **math**: mathematische Funktionen und Konstanten
- **scipy**: wissenschaftliche Funktionen
- **sys**: Interaktion mit dem Python-Interpreter
- **os**: Interaktion mit dem Betriebssystem
- **glob**: Durchsuchen von Dateisystempfaden
- **multiprocessing / threading**: Parallelprogramierung mit Prozessen / Threads
- **matplotlib**: Visualisierung von Daten und Erstellen von Abbildungen
- **numpy**: numerische Operationen und Funktionen
- **pandas**: Daten einlesen und auswerten
- **time**: Zeitfunktionen

(Arnold (2023b))

Quellen

Arnold, Simone (2023a). *Datenanalyse mit Python. Datentypen und Grundlagen.*
– (2023b). *Datenanalyse mit Python. Funktionen Module Dateien.*
– (2023c). *Datenanalyse mit Python. Idee des Kurses 'Datenanalyse mit Python'.*
– (2023d). *Datenanalyse mit Python. Schleifen und Abzweigungen.*
Matthes, Eric (2017). *Python Crashkurs: Eine praktische, projektbasierte Programmiereinführung.* 1. Aufl. Heidelberg: dpunkt.verlag. ISBN: 978-3-86490-444-8.