Python

Lehrveranstaltung an der BA Leipzig

Sommersemester 2022 - 5CS21-1

Leipzig

Autor: Dr.-Ing. Mike Müller
E-Mail: mmueller@python-academy.de
Twitter: pyacademy
Version: 7.0

Trainer: Dr.-Ing. Mike Müller
E-Mail: mmueller@python-academy.de

Inhaltsverzeichnis

1 Einleitung

Mit Ihrer Entscheidung Python zu erlernen haben Sie eine gute Wahl getroffen. Python ist wohl eine der produktivsten Programmiersprachen die es gibt. Sie ist leicht zu erlernen, bietet aber auch erfahrenen Programmierern genügend Ausdrucksmöglichkeiten, um komplexe Probleme elegant zu lösen. In der Tat ist sie eine der wenigen Sprachen, die mit dem Attribut elegant belegt ist.

Python bietet eine Menge Vorteile gegenüber anderen, weit bekannteren Sprachen wie C, C++ oder Java. Die Nachteile sind wesentlich geringer und weniger der Sprache als dem Umfeld, wie Marketing anzulasten.

Geschichtliches

Python ist Ende 1989 entstanden als Guido van Rossum eine einfache aber effiziente Scriptsprache für das verteilte Betriebssystem Amoeba benötigte. Er hat für Python Anleihen bei verschiedenen Programmiersprachen wie z.B. Smalltalk, ABC und C genommen. Herausgekommen ist eine erstaunlich konsistente Sprache. Heute hat sich das Attribut *pythonic* herausgebildet. Es bezeichnet eine Vorgehensweise die mit dem Grundgedanken von Python übereinstimmt. Im Jahr 2001 hat die Non-Profit-Organisation Python Software Foundation das Copyright übernommen. Die hauptsächliche Entwicklungsarbeit leistet ein Team von ca. 100 sogenannten Core-Deleopern. Guido van Rossum war bis Anfang 2019 der *Benevolent Dictator For Life* (BDFL). Diese Rolle übernimmt seit 2019 ein fünfköpfiges Steering Council. Dieses Council trifft die endgültige Entscheidung was Bestandteil der Sprache wird. Guido van Rossum ist Mitglied dieses Councils.

1.1 Zielsetzung

Ziel dieses Kurses ist es Sie mit der Programmiersprache Python vertraut zu machen. Unabhängig davon, ob Sie bereits in anderen Sprachen programmieren können, werden Sie am Ende des Kurses in der Lage sein Pythonprogramme zu verstehen und eigene zu entwickeln. Auf Grund der höheren Produktivität und leichteren Erlernbarkeit werden Ihre Fähigkeiten denen entsprechen, die z.B. ein Programmierer in C++ in wochen- oder monatelangen Schulungen und Praxisanwendungen erwirbt. Das ist zumindest meine Meinung, die aber von vielen, ich glaube von fast allen, die sowohl Python als auch C++ beherrschen, geteilt wird. In Zahlen ausgedrückt, wird oft von einer Produktivitätssteigerung um den Faktor fünf bis zehn gesprochen.

Wie beim Lesen und Schreiben von Briefen, Berichten, Büchern und anderen Texten, gibt es auch beim Schreiben und Lesen von Computerprogrammen Unterschiede im Schwierigkeitsgrad. Das Lesen eines guten Romans ist wesentlich leichter als das Schreiben eines solchen. So werden Sie in diesem Kurs lernen einfache Programme zu schreiben aber auch komplexe Programme zu verstehen. Bedingt durch die objektorientierte Programmierung ist es oft nicht nötig alle Details zu kennen, um die Funktionsweise eines Programms zu verstehen. Python bietet eine Unmenge an Bibliotheken, die es auch Programmieren mit weniger Erfahrung erlauben funktionsreiche und vor allem funktionierende Programme zu schreiben.

Neben den Einzelheiten der Sprache kommt es mir vor allem auf ein Gesamtverständnis an. Im Kurs ist es nicht möglich alle Details von Python erschöpfend zu behandeln. Ich möchte Sie aber in die Lage versetzen diese Details selbst zu erkunden, sobald Sie sie benötigen. Ich hoffe am Ende des Kurses haben sie ein Gefühl für die Sprache und der Begriff *pythonic* ruft bei Ihnen Assoziationen hervor, die für elegante, effektive Pythonprogramme stehen, die in der Qualität (nicht Quantität) mehr zum Roman als zum Schulaufsatz tendieren.

Um dieses Ziel zu erreichen ist Ihre aktive Mitarbeit nötig. Nur durch die gemeinsame Arbeit ist das hohe, aber meiner Meinung nach realistische Ziel erreichbar. Ich möchte mit Ihnen in ein fruchtbares Gespräch kommen. Fragen sind ausdrücklich erwünscht. Kritische Anmerkungen sind immer willkommen. Ich werde mich bemühen die Energie der Kritik in verwertbares Wissen zu verwandeln.

1.2 Vorgehensweise

Das Lernen von Neuem kann grundsätzlich auf zwei Arten erfolgen:

(1) durch Ausprobieren, Beobachten des Ergebnisses und Neuprobieren also “Learning by Doing” oder

(2) durch systematisches Vorgehen und Erlernen von einzelnen Informationseinheiten, die später zusammengefügt werden. Eine recht gute Analogie ist das Erlernen der Muttersprache und einer Fremdsprache. Beide Methoden bieten eine Reihe von Vorteilen und Nachteilen.

Lernmethode	Vorteile	Nachteile
ganzheitliche Methode	- praxisorientiertes Wissen	- kaum systematisches Grundwissen
	- Gefühl für die Sprache	- nur Nutzung der in Beispielen
	- Spaß beim Lernen	vorkommenden Sprachelemente
systematische Methode	- solide theoretische Basis	- trockenes Lernen
		- weniger Spaß bei der Sache

In diesem Kurs werde ich beide Methoden miteinander kombinieren. Wobei möglichst immer die ganzheitliche Methode zuerst angewendet wird und die Systematik das Beispiel unterstützt. Das Buch “Objektorientierte Programmierung mit Python” von Michael Weigend ist ein systematisch aufgebautes Buch. Ich habe es deshalb für den Kurs als Lehrbuch ausgewählt. Es bietet eine systematische Einführung in die Materie und ist damit das Gegenstück zu diesem Arbeitsbuch, das die praktische Seite betont. Im Arbeitsbuch sind immer die Seiten des Lehrbuches angegeben, die das gerade behandelte Thema theoretisch untersetzen. Beide zusammen bilden eine Einheit. Das ist das aktuelle Lehrbuch:

Michael Weigend

Python 3: Lernen und professionell anwenden.

Das umfassende Praxisbuch (mitp Professional)

Gebundene Ausgabe: 1056 Seiten

Verlag: mitp

Auflage: 8. überarbeitete Auflage 2019 (31. Juli 2019)

ISBN-10: 374750051X

ISBN-13: 978-3747500514

Zuerst sehen wir uns in einem Tutorial die wichtigsten Merkmale von Python kurz an. Danach beschäftigen wir uns mit einem größeren Pythonprogramm, das die meisten Sprachmerkmale besitzt. Dieses Programm wird zum Download zur Verfügung gestellt. Wir werden in den Übungen immer wieder darauf zurückgreifen. Den größten Teil des Kurses macht eine systematische Sprachbeschreibung aus, in der die im Tutorial angerissenen Themen vertieft, erweitert und in Übungen angewendet werden.

1.3 Python im Vergleich mit anderen Sprachen

Python hat viele Eigenschaften, die es für viele Aufgaben vorteilhaft erscheinen lassen. Insbesondere gegenüber weiter verbreiteten Sprachen wie C++ und Java kann Python mit Arbeitserleichterungen aufwarten, die sich in einfacheren Programmen widerspiegeln. Mark Lutz hat in seinem Buch “Programming Python” einige dieser Eigenschaften zusammengetragen.

Spracheigenschaften von Python (nach Mark Lutz, Python Programming, O’Reilly, 2. Auflage, 2001.)

Eigenschaften	Nutzen
kein Compiler nötig	Schneller Entwicklungszyklus
keine Typdeklarationen	Einfachere, kürzere und flexiblere Programme
Automatische Speicherverwaltung	Garbage Collection macht extra Programmcode unnötig
High-level Datentypen mit Methoden	Schnelle Entwicklung mit eingebauten Typen
Objektorientierte Programmierung	Wiederverwendung von Programmteilen Integration mit C++, Java und COM
Einbetten und Erweitern mit C	Optimierung, Anpassen, Verbindungsprogramme
Klassen, Module, Ausnahmen	Unterstützt modulares Programmieren im Großen
Einfacher, klarer Syntax	Lesbarkeit, Wartbarkeit, Lernbarkeit
Dynamisches Laden von C-Modulen	Einfachere Erweiterungen, kleinere binäre Programme
Dynamisches Laden von Python-Modulen	Programmmodifikation ohne Programmunterbrechung möglich
Alles ist ein Objekt	Weniger Einschränkungen und Ausnahmefälle
Während der Laufzeit programmierbar	Makrofähigkeit, Anpassbar an nichtvorhergesehene Bedürfnisse
Interaktiv, dynamisch	Schrittweise Entwicklung und Testen
Zugang zu Interpreter-Informationen	Metaprogrammierung, Objekte mit Informationen über sich
Interpreter für alle gängigen Betriebssysteme	Betriebssystemübergreifende Programmierung ohne Portierung
Kompilierung zu portablem Bytecode	Ausführungsgeschwindigkeit, Schutz des Quelltextes
portables Standard-GUI	Tkinter läuft auf Windows, Linux, Mac etc.
Internet-Protokoll-Unterstützung	FTP, e-mail, CGI etc. leicht zugänglich
Portable Systemaufrufe	Plattformunabhängige Administrationsprogrammierung
Eingebaute Bibliotheken	Große Auswahl laufender Software
Open Source	Kann frei eingebettet und vertrieben werden

Wie Sie sehen hat Python eine Menge an Vorteilen zu bieten, die nur wenigen Nachteilen gegenüber stehen. Da Python interpretiert ist bzw., vergleichbar mit Java, zu Bytecode kompiliert wird, ist es langsamer als direkt zu Maschinencode übersetzte Programme wie z.B. solche in C. Für viele Anwendungen sind die Ausführungsgeschwindigkeiten aber vollkommen ausreichend. Weiterhin bieten die eingebauten schnellen Datentypen wie Listen und Dictionaries viele Möglichkeiten schnelle Algorithmen ohne großen Aufwand zu entwickeln. Für Fälle wo die Rechengeschwindigkeit eine große Rolle spielt, wie z.B. numerische Berechnungen, gibt es eine Vielzahl von Lösungen wie die Nutzung des Paketes NumPy[1], das Einbinden von Modulen in C und FORTRAN, einen optimierenden Compiler PyPy[2], einfache Möglichkeiten Erweiterungen in C mit Cython[3] zu entwickeln und mehr.

Python ist weitaus weniger bekannt als z.B. C++ oder Java. Dies ist wohl eher dem Umfang des Marketings geschuldet, das eine große Firma wie Sun erbringt. In den letzten Jahren hat Python stark an Popularität gewonnen. Wie Sie unschwer an dem heute stattfinden Kurs erkennen können.

Weniger geeignet ist Python für die systemnahe Programmierung wie die Entwicklung von Gerätetreibern und Betriebssystemen. Obwohl es zu letzterem immer Diskussion in e-mail-Foren gibt.

[1] http://numpy.scipy.org/
[2] http://pypy.org/
[3] http://cython.org/

Hilfesystem

Python bietet eine äußerst umfangreiche Palette an Möglichkeiten Hilfe zu erhalten. Neben der mit der Standardinstallation vorhandenen Hilfe gibt es eine gute Online-Hilfe, Wikis, Mailing-Listen und Diskussionsforen. Mit der Standardinstallation kommen mehrere Hilfedateien die unter *PythonWin* unter dem *Menü Help - Python Manuals* als kompilierte HTML-Datei zum komfortablen Suchen aufgerufen werden können. Unter http://www.python.org kann die Online-Hilfe aufgerufen werden, die einen Zugang zur aktuellen Hilfe über das Internet ermöglicht. In den Wikis sind aktuelle Informationen zu unterschiedlichen Themen aufgelistet. Grundsätzlich kann sich jeder an der Gestaltung dieser Wikis beteiligen, so dass eine breite Palette von Meinungen vertreten sind. Wikis zu einem bestimmten Thema findet am besten per Suchmaschine Es gibt mehre Mailing List und Newsgoups (http://www.python.org/community/lists.html), die intensive über verschiedene Aspekte von Python diskutieren. Auf http://www.python.de gibt es auch deutschsprachige Hinweise mit Links zu weiteren Ressourcen. Im "Kochbuch" von ActiveState (http://code.activestate.com/recipes/langs/python/) gibt es zu sehr vielen Themen Lösungen mit Beispielen. Auch Stackoverflow (http://stackoverflow.com/) enthält viele Antworten zu Programmieraufgaben in Python.

The Zen of Python (by Tim Peters)

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one –obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

2 Syntax

Die Syntax von Python ist recht einfach. Deshalb wurde auch der Ausdruck “executable pseudocode”, also eine ausführbare aber gleichzeitig gut für den Menschen lesbare Beschreibung, geprägt. Dieses Kapitel fasst die wichtigsten Syntax-Element zusammen:

2.1 Kommentare

Kommentare werden mit einem # gekennzeichnet. Jeglicher Text, der nach diesen Zeichen kommt wird vom Python-Interpreter ignoriert und dient nur zur Information des Bearbeiters. Das # kann sowohl am Anfang der Zeile als auch nach ausgeführten Programmteilen stehen:

```
# Kommentar
print('ich komme auf den Bildschirm') # ich nicht
```

```
ich komme auf den Bildschirm
```

2.2 Einrückungen

In vielen Programmiersprachen müssen an vorgeschriebenen Stellen bestimmte Zeichen wie `;` oder `{` und `}` vorkommen, damit das Programm ausgeführt oder übersetzt werden kann.

In Python wird der gleiche Effekte durch obligatorische Einrückungen erreicht. Auch in anderen Sprachen wird von den meisten Programmierern der Quelltext entsprechend des Sinngehaltes eingerückt. In Python ist diese Einrückung aber Pflicht und führt dazu, dass alle die Python-Quelltext schreiben gleich einrücken **müssen**.

Als Faustregel können wir uns merken, dass nach einem Doppelpunkt eingerückt werden muss. Wenn der logische Code-Block beendet ist, wird um den gleichen Betrag wieder ausgerückt:

```
for x in [1, 2, 3]:
    print(x) # hier muss eingerueckt werden
print('wieder ausgerueckt')
```

```
1
2
3
wieder ausgerueckt
```

2.3 Zeilenumbrüche

Zeilen sollten nicht breiter als 79 Zeichen sein. Wenn eine logische Zeile länger sein muss, gibt es zwei Möglichkeiten:

(1) die Weiterführung mit \:

```
'text1'\
'text2'\
'text3'
```

```
'text1text2text3'
```

(2) die Nutzung von Klammern über mehrere Zeilen hinweg:

```
('text1'
 'text2'
 'text3')
```

```
'text1text2text3'
```

Beide Varianten sind äquivalent zu dieser:

```
'text1' 'text2' 'text3'
```

```
'text1text2text3'
```

2.4 Groß- und Kleinschreibung

Python unterscheidet zwischen groß und klein geschriebenen Bezeichnern.

```
a = 4
A = 5
a
```

```
4
```

```
A
```

```
5
```

2.5 Strings

Strings (Zeichenketten) können sowohl mit einfachen als auch mit doppelten Anführungszeichen begrenzt werden:

```
'abc'
```

```
'abc'
```

```
"abc"
```

```
'abc'
```

So lassen sich innerhalb die jeweils anderen Zeichen ohne Fluchtsequenzen verwenden (das Escape-Zeichen ist \.):

```
'"Hallo"'
```

```
'"Hallo"'
```

statt:

```
"\"Hallo\""
```

```
'"Hallo"'
```

und:

```
"don't"
```

```
"don't"
```

statt:

```
'don\'t'
```

```
"don't"
```

Es gibt auch Zeichenketten mit dreifachen Anführungszeichen, jeweils wieder einzeln oder doppelt. Damit können wir leicht mehrzeilige Texte ausdrücken:

```
langer_text = """
Mehrzeiliger Text.
==================
Mehrzeiliger Text kann mehrere Zeilen haben.
Dabei bleiben Leerzeilen erhalten.
"""
langer_text
```

```
'\nMehrzeiliger Text.\n==================\nMehrzeiliger Text kann mehrere Zeilen␣
↪haben.\nDabei bleiben Leerzeilen erhalten.\n'
```

```
print(langer_text)
```

```
Mehrzeiliger Text.
==================
Mehrzeiliger Text kann mehrere Zeilen haben.
Dabei bleiben Leerzeilen erhalten.
```

3 Anweisungen und Ausdrücke

Programme bestehen aus Anweisungen und Ausdrücken (auch Ausdrucksanweisungen genannt). Vereinfacht kann man sich merken, dass Ausdrücke Ergebnisse zurückgeben und Anweisungen nicht.

So bringt der Ausdruck:

```
1 + 1
```

```
2
```

die `2` als Ergebnis. Die Anweisung:

```
z = 1 + 1
```

bringt kein Ergebnis, sondern weist dem Namen `z` den Wert `2` zu. Wir können den Wert wieder abrufen:

```
z
```

```
2
```

Wir können dem Ergebnis eines Ausdrucks auch einen Namen zuweisen:

```
ergebnis = 2 + 2
ergebnis
```

```
4
```

Die `import`-Anweisung importiert ein Modul:

```
import sys
sys
```

```
<module 'sys' (built-in)>
```

Es handelt sich um eine Anweisung. Wir erhalten somit kein Ergebnis zurück, sondern lösen einen sogenannten Seiteneffekt aus. In diesem Fall ist der Seiteneffekt das Importieren eines Moduls und in Folge die Existenz des Namens `sys`.

Das Zuweisen eines Ergebnisses unserer `import`-Anweisung ist nicht möglich:

```
s = import sys
```

```
  File "<ipython-input-7-96e95326a6a6>", line 1
    s = import sys
        ^
SyntaxError: invalid syntax
```

3.1 Übungen

3.1.1 Übung 1

Schreiben Sie einen Ausdruck der die Zahlen `8` und `4` addiert und das Ergebnis durch `3` teilt.

Weisen Sie dem Ergebnis einen Namen zu.

3.1.2 Übung 2

Nutzen Sie die Anweisung zum Import eines Moduls und importieren Sie `os`.

3.1.3 Übung 3

Lösen sie die Aufgaben 1 und 2 indem Sie den Quelltext in ein Modul speichern und diese auf der Kommandozeile ausführen.

4 Entscheidungen

Im Ablauf eines Programmes müssen oft Entscheidungen getroffen werden. In Python kann dies mit Hilfe von `if`, `elif` und `else` erreicht werden.

Nehmen wir zwei Zahlen, die wir mit den Namen `a` und `b` ansprechen:

```
a = 5
b = 6
```

Nun vergleichen wir beide Zahlen:

```
if a < b:
    print('a kleiner b')
```

```
a kleiner b
```

Wir können einen zweiten (und beliebig viele mehr) Vergleich anschließen:

```
if a < b:
    print('a kleiner b')
elif b < a:
    print('b kleiner a')
```

```
a kleiner b
```

Natürlich ist die Ausgabe die gleiche wie oben.

Ändern wir die Werte von `a` und `b` erhalten wir ein anderes Ergebnis:

```
a = 10
b = 20
if a < b:
    print('a kleiner b')
elif b < a:
    print('b kleiner a')
```

```
a kleiner b
```

Abschließend können wir noch mit `else` alle anderen Fälle abfangen:

```
if a < b:
    print('a kleiner b')
elif b < a:
    print('b kleiner a')
else:
    print('a gleich b')
```

```
a kleiner b
```

Wenn wir nun die Werte von `a` und `b` entsprechend ändern, bekommen wir die passende Ausgabe:

```
a = 10
b = 10

if a < b:
    print('a kleiner b')
elif b < a:
    print('b kleiner a')
else:
    print('a gleich b')
```

```
a gleich b
```

Es gibt auch einen, eher weniger genutzten, ternären Ausdruck in einer Zeile:

```
a = 10
b = 20

a if a < b else b
```

```
10
```

Das lässt sich übersetzen in “Gib mir `a`, wenn `a` kleiner als `b` ist, sonst gib mir `b`”.

```
a if a > b else b
```

```
20
```

Dies bedeutet dann “Gib mir `a`, wenn `a` größer als `b` ist, sonst gib mir `b`”.

Wir können zwei Vergleiche in einer Zeile ausführen:

```
2 <= a <= 10
```

```
True
```

```
7 <= a <= 10
```

```
True
```

Im Folgenden sind die Vergleichsoperatoren in Python zusammengestellt:

Operator	Bedeutung
`<`	kleiner als
`>`	größer als
`==`	gleich
`>=`	größer gleich
`<=`	kleiner gleich
`!=`	ungleich
`is`	identisch
`is not`	nicht identisch

Boolsche Operatoren werden oft verwendet, wenn auf bestimmte Bedingungen geprüft wird:

Operator	Bedeutung
`and`	logisches und
`or`	logisches oder
`not`	logisches nicht

Letztendlich gibt es Kontrollen des Enthaltenseins (Mitgliedschaftstest) in Datenstrukturen:

Operator	Bedeutung
`in`	ist in der Datenstruktur enthalten
`not in`	ist nicht in der Datenstruktur enthalten

4.1 Übungen

4.1.1 Übung 1

Schreiben Sie ein Programm, das vom Nutzer die Eingabe einer Zahl verlangt. Vergleichen Sie diese Zahl mit einem Grenzwert und geben Sie aus, ob die Zahl darüber oder darunter liegt bzw. dem Grenzwert entspricht.

Hinweis

Nutzen Sie diesen Code für die Eingabe und die Umwandlung in eine Zahl:

```
eingabe = input('Bitte eine Zahl eingeben')
zahl = float(eingabe)
```

4.1.2 Übung 2

Schreiben Sie ein zweites Programm, das prüft ob der eingegebene Wert innerhalb eines vorgegebenen Bereichs liegt. Nutzen Sie dazu (a) boolsche Operatoren und (b) einen einzeiligen Mehrfachvergleich.

5 Schleifen

Computer sind besonders gut darin gleichartige Operationen sehr oft zu wiederholen. In Python gibt es zwei grundsätzliche Möglichkeiten für Schleifen: die so genannten `for`- und `while`-Schleifen.

5.1 Schleifen mit `for`

Mit der `for`-Schleife können wir die einzelnen Elemente eines Objektes ansprechen. Dieses Objekt muss iterierbar sein. Wir können uns solch ein Objekt z.B. mit `range` erzeugen:

```
list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Das Ergebnis ist eine Liste. Listen sind iterierbar. Wir können also nun alle Elemente der Liste einzeln ausdrucken:

```
for x in range(10):
    print(x)
```

```
0
1
2
3
4
5
6
7
8
9
```

Natürlich können wir statt des Ausdruckens jede beliebige andere Aktion ausführen und die Daten entsprechend verarbeiten.

Wir können eine optionale, untere Grenze für `range` angeben:

```
list(range(10, 15))
```

```
[10, 11, 12, 13, 14]
```

Auch die Angabe der Schrittweite ist möglich:

```
list(range(10, 20, 2))
```

```
[10, 12, 14, 16, 18]
```

Diese darf auch negativ sein:

```
list(range(20, 10, -2))
```

```
[20, 18, 16, 14, 12]
```

5.2 Schleifen mit `while`

Für die `while`-Schleife brauchen wir kein besonderes Objekt, das wir durchlaufen können. Wir müssen allerdings eine Abbruchbedingung vorgeben, da unsere Schleife sonnst nicht beendet wird.

Wir definieren `x` mit einem Wert von Null:

```
x = 0
```

So lange dieses `x` einen Wert kleiner als 10 hat, drucken wir es aus:

```
while x < 10:
    print(x)
    x += 1
```

```
0
1
2
3
4
5
6
7
8
9
```

Wir zählen den Wert von `x` bei jedem Durchlauf um eins hoch (`x += 1` ist gleichbedeutend mit `x = x + 1`). Die Zahl 10 wird nicht ausgedruckt, da die Abbruchbedingung greift und die Schleife beendet.

5.3 Schleifen vorzeitig beenden

Manchmal wollen wir eine Schleife vorzeitig beenden. Dafür können wir `break` nutzen. Wollen wir z.B. unsere Iteration beenden wenn der Wert `5` wird, können wir dies so erreichen:

```
for x in range(10):
    if x == 5:
        break
    print(x)
```

```
0
1
2
3
4
```

Die Schleife wird also vollständig abgebrochen und das Programm läuft mit den Anweisungen, die nach der Schleife kommen weiter.

5.4 Schleifendurchläufe überspringen

Wollen wir nicht die gesamte Schleife abbrechen, sondern nur einen vorzeitig beenden können wir `continue` nutzen:

```
for x in range(10):
    if x == 5:
        continue
    print(x)
```

```
0
1
2
3
4
6
7
8
9
```

Die `5` wird also nicht ausgedruckt. Ab der `6` wird die Schleife wieder vollständig durchlaufen.

5.5 Übungen

5.5.1 Übung 1

Schreiben Sie eine Schleife, die die Quadrate der Zahlen 10 bis 20 ausgibt mit Hilfe von `for`. Hinweise: Das Quadrat lässt sich durch Multiplkation, also `x * x` oder Potenzierung mit 2, also `x ** 2` erzeugen.

5.5.2 Übung 2

Schreiben Sie eine Schleife mit `while`, die das Gleiche tut.

5.5.3 Übung 3

Führen Sie Übungen 1. und 2. am interaktiven Prompt aus. Speichern Sie den Quelltexte in einem Modul und führen Sie Ihr Programm auf der Kommandozeile aus.

5.5.4 Übung 4

Modifizieren Sie Ihre Schleife aus Übung 1. so, dass diese nur jede zweite Zahl ausgibt (beginnend mit der ersten Zahl). Es gibt dafür mehrere Lösungen. Nutzen Sie hier eine `if`-Abfrage und `continue` im Schleifenkörper.

Hinweis: Der Modulo-Operator `%` liefert den Rest einer ganzzahligen Division. Mit der Zahl Zwei als Divisor erhalten wir für alle geraden Zahlen eine Null und für alle ungeraden Zahlen eine Eins:

```
>>> 10 % 2
0
>>> 9 % 2
1
```

5.5.5 Übung 5

Lösen Sie Übung 4. ohne `if`-Abfrage im Schleifenkörper, sondern (a) durch Änderung des Schleifenkopfes für die `for`- und durch (b) eine andere Schrittweite der Zählvariable für die `while`-Schleife.

6 Datentypen

6.1 Statische und dynamische Typisierung

Jede Programmiersprache hat Datentypen. In vielen Sprachen (C, C++, Java) muss man diese Typen fest vereinbaren, d.h. deklarieren (static typing). Eine Änderung des Datentyps bedarf in diesen Sprachen einer besonderen Anweisung (casting). Python hat im Gegensatz dazu eine dynamische Typisierung, d.h. der Typ eines Objektes wird nicht a priori festgelegt sondern wird während der Laufzeit ermittelt. Ein Name kann somit nacheinander beliebige Typen aufnehmen ohne da es dazu einer besonderen Anweisung bedarf. In diesem Sinne ist eine Variable (ein Name) in ihren Typ änderbar.

6.2 Starke und schwache Typisierung

Eine weitere Einteilung der Typisierung lässt sich durch die Kategorien “stark (oder streng)” und “schwach” treffen. Es gibt keine exakte Definition dieser Kategorien. Ein wichtiges Merkmal ist die explizite oder implizite Umwandlung von Datentypen. Schach getypte Sprachen erlauben die implizite Umwandlung. Beispiele für schwach getypte Sprachen sind C, Javascript oder PHP. Beispiele für streng getypte Sprachen sind C++, Java, Haskell und Python. Je nach Anwendung der Kriterien könnten die Sprachen durchaus einer andere Kategorie passen.

Python ist demnach ein dynamisch und streng getypte Sprache.

Für die praktische Arbeit mit Python bedeutet dies, dass flexiblere und elegantere Programme möglich sind. Das trifft insbesondere auf die objektorientierte Programmierung zu. Die Typumwandlung (casting) ist z.B. in Java sehr häufig und kann recht aufwendig und fehleranfällig werden. Für manche Programme, z.B. numerische Berechnungen sind statische Typen von Vorteil, da sich die Typen über die gesamte Berechnungsdauer meist nicht ändern und ohne Testen auf den aktuellen Typ immense Laufzeitverbesserungen möglich sind.Python hat hierfür auch Lösungen, indem Erweiterungen in anderen Sprachen (typischerweise C oder FORTRAN) bzw. das Modul NumPy[4] genutzt werden.

Das oft von Vertretern von statisch typisierten Sprachen angeführte Argument, dass durch die feste Typisierung Fehler vom Compiler gefunden werden, ist nur bedingt richtig. Viele Fehler werden auch bei strickten Typkontrollen nicht gefunden. Typisierung ist keinesfalls ein Ersatz für das Testen. In Python kann es aber passieren, dass durch einen Tippfehler bei einer Zuweisung an einen vermeintlich bestehenden Namen ein neuer Name entsteht:

```
var_name = 10
```

und später:

```
varname = 20
```

Hier ist höchstwahrscheinlich das gleiche Objekt gemeint. Es wurde aber ein neues erstellt. Um Fehler dieser Art und weitere zu finden gibt es die Werkzeuge wie PyLint[5]. Damit wird der Pythonquelltext durchsucht und auf eventuelle Fehlerquellen hingewiesen. Die dynamische Natur von Python bleibt also erhalten, die Vorteile eines Compilers werden aber gleichzeitig genutzt.

[4] http://numpy.scipy.org/
[5] http://pylint.pycqa.org/en/latest/

In Python gibt es verschiedene Datentypen. Diese lassen sich systematisch einfache und komplexe Datentypen einteilen. Die folgenden Abschnitte stellen die wichtigsten vor.

Funktions-Annotionen und Typ-Hinweise

Python erlaubt optionale "type hints" mittels Funktions-Annotionen.
Die Syntax ähnelt der in anderen Sprachen für statische Typdeklaration genutzten:

```
def add(a: int, b: int) -> int:
    return a+ b
```

Python wertet diese zu Programmlaufzeit jedoch nicht aus. Die Funktion `add` funktioniert mit zwei Integer-Argumenten:

```
>>> add(3, 4)
7
```

aber auch wenn ein Argument den Datentyp `float` hat:

```
>>> add(3, 4.5)
7.5
```

Werkzeuge wie Editoren und Integrierte Entwicklungsumgebungen (IDE) können diese nutzen, um dem Programmierer wertvolle Hinweise auf mögliche Fehler zu geben. Das Programm mypy[6] kann die richtige Nutzung von Funktions- und Methodenaufrufen mit Hilfe dieser Typ-Hinweise automatisch überprüfen. Funktions-Annotionen sind durchaus umstritten. Für größere Bibliotheken können sie aber sehr nützlich sein. Für viele kleinere Anwendungen sind sie eher nicht so wichtig.

[6] http://mypy-lang.org/

6.3 Einfache Datentypen

6.3.1 Zahlen

Die Zahlen gehören zu den einfachen Datentypen. Es gibt:

- Ganzzahlen wie `1`
- Gleitkommazahlen wie `12.45`
- komplexe Zahlen wie `1 + 3j`

Wir können die verschiedenen Zahlen ineinander umwandeln. Die Funktionen dazu lauten:

- `int(zahl)`
- `float(zahl)`
- `complex(zahl)`

Integer sind beliebig lange Ganzzahlen, deren Größe nur der verfügbaren Arbeitsspeicher begrenzt. Es gibt keine weiteren Unterarten von Ganzzahlen also keine `short` oder `signed`.

Es gibt einen Gleitkommazahlentyp `float` der auf auch 32-Bit-Computern typischer Weise 64 bit groß ist. Gleitkommazahlen werden mit dem `.` als Dezimalzeichen geschrieben: `1.234`. Weiterhin gibt es den Typ `complex`, mit dem sich komplexe Zahlen mit einem realen und einem imaginären Anteil darstellen lassen: `1 + 3j`.

6.3.2 Der Datentyp None

Mit dem Datentyp `None` kann ausgedrückt werden, dass an dieser Stelle "nichts" ausgedrückt werden soll.

6.3.3 Boolscher Datentyp

In Python gibt es einen Boolschen Datentyp, der die Werte `True` oder `False` annehmen kann. Viele Objekte in Python können in einen solchen Datentyp umgewandelt werden, indem die eingebaute Funktion `bool` genutzt wird. Die Zahl Null ergibt `False`:

```
bool(0)
```

```
False
```

```
bool(0.0)
```

```
False
```

```
bool(0j)
```

```
False
```

Alle anderen Zahlen ergeben `True`:

```
bool(1)
```

```
True
```

Leere Objekte wertet `bool` ebenfalls als `False`:

```
bool([])
```

```
False
```

```
bool('')
```

```
False
```

Objekte, die etwas enthalten ergeben, also deren Länger größer Null ist, `True`:

```
bool([2])
```

```
True
```

Das Objekt `None` hat ebenfalls den Boolschen Wert `False`:

```
bool(None)
```

```
False
```

6.4 Kollektionen

Kollektionen können Objekte sammeln. In Python gibt es drei grundlegende Kollektionen:

- Sequenzen
- Abbildungen (Mappings) und
- Mengen.

Sequenzen können veränderbar oder unveränderbar sein. Die veränderbare Sequenz in Python ist die Liste (`list`). Listen werden durch eckige Klammern `[1, 2, 3]` gekennzeichnet. Nicht veränderbare Sequenzen sind `tuple` und `string`. Tupel können andere Objekte beliebigen Typs aufnehmen; können selbst aber nicht verändert werden. Strings (Zeichenketten) sind Sequenzen von Zeichen wie z.B. `'Hallo'`. Sie können ebenfalls nicht geändert werden. Um Tupel und Strings zu verändern, müssen neue Sequenzen aufgebaut werden, indem Teile der alten verwendet: `'T' + 'text'[1:]` ergibt den neuen String `'Text'`. Die Abbildung in Python ist ein so genanntes Dictionary `dict`. Ein einzigartiger Schlüssel kann genutzt werden, um auf Einträge zuzugreifen `{'Monday': 0, 'Tuesday':1}`. Mengen sind ungeordnete Kollektionen ohne Duplikate, die mit Operationen aus der mathematischen Mengenlehre verarbeitet werden können.

Die verschiedenen Arten von Kollektionen werden in den folgenden Abschnitten näher dargestellt.

6.4.1 Sequenzen

Sequenzen sind Datentypen, die mehrere Elemente haben, die in einer festen Reihenfolge angeordnet sind. Es gibt:

- Strings wie `'Hallo'`
- Listen wie `[1, 2, 3]`
- Tupel wie `('a', 2, 30.6)`

Auf Sequenzen kann man mit einen Index zugreifen:

```
liste = [1, 2, 3]
liste = [1, 2, 3]
```

```
liste[1]
```

```
2
```

Es gibt noch viele andere Möglichkeiten mit Listen zu arbeiten, die wir uns später ansehen.

6.4.2 Dictionarys

In Python gibt es den Datentyp Dictionary, der es erlaubt mit Schlüssel-Wert-Paaren zu arbeiten.

Wir definieren ein Dictionary mir geschweiften Klammern:

```
d = {'a': 50, 'b': 100}
```

Und können dann mit dem Schlüssel den jeweiligen Wert abfragen:

```
d['a']
```

```
50
```

```
d['b']
```

```
100
```

6.4.3 Sets

Sets sind Datentypen, die sich wie mathematische Mengen verhalten. Wir werden später auf diesen Datentyp zurückkommen.

6.4.4 Eigene Datentypen

Zusätzlich dazu kann man noch eigene Datentypen mit dem Schlüssel `class` definieren.

6.4.5 Mehr

Es gibt weitere Datentypen in Python, die wir hier nicht weiter untersuchen.

Diese eingebauten Datentypen können Bytes repräsentieren:

- `bytes` (unveränderlich)
- `bytearray`(veränderlich)

Dies sind Beispiele für Datentypen, die Module der Standardbibliothek zur Verfügung stellen:

- `array.array`
- `collections.defaultdict`
- `collections.deque`
- `collections.namedtuple`
- `collections.Counter`
- `dataclasses.dataclass`

6.5 Übungen

6.5.1 Übung 1

Wandeln Sie `10` in eine Gleitkommazahl und eine komplexe Zahl um.

6.5.2 Übung 2

Wandeln Sie `123.95` in eine Ganzzahl um. Erläutern Sie das Ergebnis.

6.5.3 Übung 3

Addieren Sie zu der Zahl eine Billion (eine Eins mit 12 Nullen, in wissenschaftlicher Notation `1e12`) ein Millionstel (in wissenschaftlicher Notation `1e-6`). Erläutern Sie Ihr Ergebnis. Verringern Sie den Größenunterschied der beiden Summanden schrittweise, bis Sie eine Änderung im Ergebnis sehen.

6.5.4 Übung 4

Legen sie eine Liste mit 5 Elementen an. Greifen Sie auf einzelne Elemente zu.

6.5.5 Übung 5

Legen Sie ein Dictionary mit drei Schlüssel-Wert-Paaren an. Greifen Sie auf einzelne Werte mit Hilfe des Schlüssels zu.

7 Sequenzen im Detail

Sequenzen sind wichtige eingebaute Typen in Python. Sie machen einen guten Teil der Nutzbarkeit von Python aus. Zusammen mit den Dictionarys, die im nächsten Abschnitt erläutert werden, werden Sie sehr oft eingesetzt und sind für eine Vielzahl von Aufgabenstellungen geeignet.

Wichtige Sequenzen in Python sind Strings (Zeichenketten), Tupel und Listen. Ein wichtiges Unterscheidungsmerkmal ist die Veränderbarkeit. Listen sind veränderbar (mutable), Tupel und Strings sind es nicht. Ein Element einer Liste kann deshalb ausgetauscht oder verändert werden wohingegen Strings und Tupel nur gelesen werden können. Für Änderungen muss eine neue Sequenz aufgebaut werden.

7.1 Auf Daten in Sequenzen zugreifen

Sequenzen haben eine Reihe von gemeinsamen Operationen, die in der folgenden Tabelle zusammengefasst sind.

Operation	Ergebnis
`x in s`	`True` wenn ein Element von `s` gleich `x` ist, sonst `False`
`x not in s`	`False` wenn ein Element von `s` gleich `x` ist, sonst True
`s + t`	Zusammenfügen von `s` und `t`
`s * n , n * s`	`n`-maliges Zusammenfügen (flach)
`s[i]`	`i`-tes Element von `s`, Beginn bei `0`
`s[i:j]`	Ausschnitt von `s` von `i` bis `j`
`s[i:j:k]`	Ausschnitt von `s` von `i` bis `j` mit Schritt `k`
`len(s)`	Länge von `s`
`min(s)`	kleinstes Element von `s`
`max(s)`	größtes Element von `s`
`s.count(x)`	ermittelt die Anzahl für die `s[i] == x` gilt
`s.index(x[, i[, j]])`	kleinstes `k`, so dass `s[k] == x` und `i <= k < j`

Wir definieren einen String:

```
text = 'abcdefghijabc'
```

Alle Beispiele würden auch mit einer Liste oder einem Tupel funktionieren.

Wir können testen, ob bestimmte Unter-Sequenzen in unserer Sequenz sind:

```
'a' in text
```

```
True
```

`x` ist nicht enthalten:

```
'x' in text
```

```
False
```

Mit `not` können wir eine negative Frage stellen:

```
'x' not in text
```

```
True
```

Sequenzen lassen sich leicht verketten (konkatenieren):

```
text + text
```

```
'abcdefghijabcabcdefghijabc'
```

Noch schneller geht es mit der Multiplikation:

```
text * 3
```

```
'abcdefghijabcabcdefghijabcabcdefghijabc'
```

Wir können auf beliebige Elemente zugreifen:

```
text[0]
```

```
'a'
```

```
text[1]
```

```
'b'
```

Die Zählung von hinten beginnt mit `-1`:

```
text[-1]
```

```
'c'
```

Bereiche lassen sich leicht ansprechen:

```
text[2:4]
```

```
'cd'
```

Dabei gilt immer: untere Grenze inklusive, obere Grenze exklusive.

Das Ganze geht auch mit einer optionalen Schrittweite:

```
text[2:10:2]
```

```
'cegi'
```

Das Weglassen von Index-Werten weißt Python an ganz am Anfang zu beginnen bzw. bis zum Ende zu gehen:

```
text[::-1]
```

```
'cbajihgfedcba'
```

Wir können auch Länge, Maximal- und Minimal-Werte abfragen:

```
len(text)
```

```
13
```

```
min(text)
```

```
'a'
```

```
max(text)
```

```
'j'
```

Die Anzahl bestimmter Elemente bekommen wir mit der Methode `count`:

```
text.count('a')
```

```
2
```

```
text.count('e')
```

```
1
```

```
text.count('x')
```

```
0
```

Das erste Auftreten eines bestimmten Wertes verrät uns die Methode `index`:

```
text.index('a')
```

```
0
```

Optional können wir den Start-Index für die Suche:

```
text.index('a', 3)
```

```
10
```

und den End-Index angeben:

```
text.index('a', 3, 12)
```

```
10
```

Wenn der Wert nicht im Suchbereich enthalten ist bekommen wir eine Ausnahme:

```
text.index('a', 3, 8)
```

```
ValueError: substring not found
```

7.2 Übungen

7.2.1 Übung 1

Erstellen sie ein Tupel mit ca. 8 bis 15 Ganzzahlen, Gleitkommazahlen und Strings.

Greifen auf das zweite Element dieses Tupels zu.

Erzeugen Sie ein Teil-Tupel vom zweiten bis zum vorletzten Element.

7.2.2 Übung 2

Erstellen Sie ein Tupel, das dreimal so lang ist wie das aus Übung 1 und die gleichen Elemente dreimal enthält. Verwenden Sie dazu mindestens zwei verschiedene Methoden.

7.2.3 Übung 3

Erzeugen Sie ein neues Tupel mit allen Elemente des Tupels von Übung 2 aber in umgekehrter Reihenfolge mit Hilfe von Slicing.

7.2.4 Übung 4

Erstellen Sie ein neues Tupel, das nur jedes zweite Element des Tupels aus Übung 2 enthält.

7.2.5 Übung 5

Überprüfen Sie, ob ein bestimmtes Element im Tupel enthalten ist.

8 Listen im Detail

Listen haben bedingt durch die Möglichkeit der Änderung ihrer Elemente noch zusätzliche Methoden, die in der folgenden Tabelle dargestellt sind.

Operation	Ergebnis
`L[i] = x`	Element an Index `i` in `L` wird durch `x` ersetzt
`L[i:j] = t`	Ausschnitt von `L` von `i` bis `j` wird durch `t` ersetzt
`L[i:j:k] = t`	die Elemente von `L[i:j:k]` werden durch die von `t` ersetzt
`del L[i]`	entfernt Element an Index `i` aus `L`
`del L[i:j]`	entfernt die Elemente von `L` von `i` bis `j`, das Gleiche wie `L[i:j] = []`
`del L[i:j:k]`	entfernt die Elemente von `L[i:j:k]` aus der Liste
`L.append(x)`	`x` an das Ende der Liste anhängen, das Gleiche wie `L[len(L):len(L)] = [x]`
`L.extend(x)`	die Liste um die Element von `x` erweitern, das Gleiche wie `L[len(L):len(L)] = x`
`L.insert(i, x)`	`x` vor Index in die Liste einfügen, das Gleiche wie `L[i:i] = [x]`
`L.clear()`	alle Elemente aus der List entfernen, das Gleich wie `L[:] = []`
`L.pop(index=-1)`	das Element an Index `i` aus der Liste entfernen und zurückgeben, das Gleiche wie `x = L[i]; del L[i];` gibt x zurück, `i` ist letzter Index, wenn nicht übergeben
`L.remove(x)`	das erste Element mit dem Wert `x` aus der Liste entfernen, das Gleiche wie `del L[L.index(x)]`
`L.reverse()`	kehrt die Reihenfolge der Elemente um
`L.sort( key=None, reverse=False)`	sortiert die Elemente aufsteigend

Wir legen eine neue Liste an:

```
L = list(range(10, 81, 10))
L
```

```
[10, 20, 30, 40, 50, 60, 70, 80]
```

Wir können beliebige Elemente ändern:

```
L[3] = 1000
L
```

```
[10, 20, 30, 1000, 50, 60, 70, 80]
```

Auch ganze Bereiche lassen sich austauschen:

```
L[2:4] = [100, 200, 300, 400, 500]
L
```

```
[10, 20, 100, 200, 300, 400, 500, 50, 60, 70, 80]
```

Wir haben zwei alte gegen fünf neue Elemente ausgetauscht.

Wir können sowohl Einzelelemente:

```
del L[4]
L
```

```
[10, 20, 100, 200, 400, 500, 50, 60, 70, 80]
```

als auch ganze Bereiche löschen:

```
del L[2:4]
L
```

```
[10, 20, 400, 500, 50, 60, 70, 80]
```

Die Methode `append` hängt ein Element an unsere Liste an:

```
L.append(7)
L
```

```
[10, 20, 400, 500, 50, 60, 70, 80, 7]
```

Dagegen erweitert die Methode `extend` unsere Liste:

```
L.extend([9, 8, 7])
L
```

```
[10, 20, 400, 500, 50, 60, 70, 80, 7, 9, 8, 7]
```

Der Effekt mit `append` ist anders:

```
L.append([9, 8, 7])
L
```

```
[10, 20, 400, 500, 50, 60, 70, 80, 7, 9, 8, 7, [9, 8, 7]]
```

```
L
```

```
[10, 20, 400, 500, 50, 60, 70, 80, 7, 9, 8, 7, [9, 8, 7]]
```

Wir können auf das zweite Element der verschachtelten Liste so zugreifen:

```
L[-1][1]
```

```
8
```

Zum Einfügen eines Elements an beliebiger Stelle können wir die Methode `insert` nutzen:

```
L.insert(2, 57)
L
```

```
[10, 20, 57, 400, 500, 50, 60, 70, 80, 7, 9, 8, 7, [9, 8, 7]]
```

Das letzte Element unserer Liste können wir mit der Methode `pop` entfernen und erhalten es gleichzeitig als Rückgabewerte:

```
L.pop()
```

```
[9, 8, 7]
```

```
L
```

```
[10, 20, 57, 400, 500, 50, 60, 70, 80, 7, 9, 8, 7]
```

Wenn wir den Zielindex angeben, entfernt `pop` das entsprechende Element:

```
L.pop(5)
```

```
50
```

```
L
```

```
[10, 20, 57, 400, 500, 60, 70, 80, 7, 9, 8, 7]
```

Die Methode `remove` entfernt das angegebene Element an der ersten gefundenen Stelle und wirft eine Ausnahme, wenn das Element nicht in der Liste enthalten ist:

```
L
```

```
[10, 20, 57, 400, 500, 60, 70, 80, 7, 9, 8, 7]
```

```
L.remove(7)
```

Die erste `7` fehlt:

```
L
```

```
[10, 20, 57, 400, 500, 60, 70, 80, 9, 8, 7]
```

Nach dem zweiten Durchgang:

```
L.remove(7)
```

Fehlt auch die zweite `7`:

```
L
```

```
[10, 20, 57, 400, 500, 60, 70, 80, 9, 8]
```

Da nun keine `7` enthalten fehlt der nächste Versuch fehl:

```
L.remove(7)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-b33b8c7e72f4> in <module>
----> 1 L.remove(7)

ValueError: list.remove(x): x not in list
```

Wir können die Anordnung der Elemente mit `reverse` umkehren:

```
L.reverse()
L
```

```
[8, 9, 80, 70, 60, 500, 400, 57, 20, 10]
```

Das Sortieren der Liste übernimmt die Methode `sort`:

```
L.sort()
L
```

```
[8, 9, 10, 20, 57, 60, 70, 80, 400, 500]
```

8.1 Verändern oder neu?

Für Listen gibt es die oben dargestellten Methoden, die die Liste selbst verändern. Weiterhin gibt es noch einige eingebaute Funktionen mit sehr ähnlichen Namen, die die Liste unverändert lassen und eine neue Liste als Ergebnis zurück geben.

So gibt `reversed` eine neue Liste in umgekehrter Reihenfolge zurück:

```
L
```

```
[8, 9, 10, 20, 57, 60, 70, 80, 400, 500]
```

```
list(reversed(L))
```

```
[500, 400, 80, 70, 60, 57, 20, 10, 9, 8]
```

Die Originalliste hat sich dabei nicht verändert:

```
L
```

```
[8, 9, 10, 20, 57, 60, 70, 80, 400, 500]
```

Im Gegensatz dazu gibt die Methode `reverse` kein Ergebnis zurück verändert dafür aber die Originalliste:

```
L.reverse()
```

```
L
```

```
[500, 400, 80, 70, 60, 57, 20, 10, 9, 8]
```

Analog gilt dies für `sorted`. Das sortierte Ergebnis ist eine neue Liste:

```
L = [5, 2, 4, 1, 3]
L
```

```
[5, 2, 4, 1, 3]
```

```
sorted(L)
```

```
[1, 2, 3, 4, 5]
```

```
L
```

```
[5, 2, 4, 1, 3]
```

Die Methode `sort` verändert dagegen die Originalliste:

```
L.sort()
```

```
L
```

```
[1, 2, 3, 4, 5]
```

8.2 Sortieren im Detail

Wir haben eine Liste mit Klein- und Großbuchstaben:

```
unsorted = ['a', 'c', 'A', 'x', 'Y', 'y']
```

Die Großbuchstaben werden vor den Kleinbuchstaben eingeordnet:

```
sorted(unsorted)
```

```
['A', 'Y', 'a', 'c', 'x', 'y']
```

Das Umkehren der Sortierreihenfolge ist einfach:

```
sorted(unsorted, reverse=True)
```

```
['y', 'x', 'c', 'a', 'Y', 'A']
```

Wir können auch eine Funktion angeben, die auf alle Element vor dem Sortieren angewendet wird:

```
sorted(unsorted, key=str.casefold)
```

```
['a', 'A', 'c', 'x', 'Y', 'y']
```

Damit erfolgt das Sortieren mit den Kleinbuchstaben-Versionen aller Strings, da die Funktion `str.casefold` auf alle Elemente unserer Liste angewendet wird. Das Ergebnis von `str.casefold` einen String, der einen Vergleich ohne Beachtung der Groß- und Kleinschreibung erlaubt. Für die meisten Zeichen ist das die kleingeschriebene Version.

Als Bespiel nehmen wir eien Liste mit Tupeln:

```
data = [(1, 2), (5, 1), (3, 4), (6, 2)]
```

Die Sortierung nutzt den ersten Werte des jeweiligen Tupels:

```
sorted(data)
```

```
[(1, 2), (3, 4), (5, 1), (6, 2)]
```

Wir können unsere eigene Funktion definieren, um sie dann mit `key=` zu nutzen:

```
def second(value):
    return value[1]
```

Diese Funktion gibt den zweiten Eintrag des Objektes `value`, das eine Liste oder ein Tupel sein könnte. Eine Liste von Tupeln sortiert `sorted` nun nach dem zweiten Element jeden Tupels:

```
sorted(data, key=second)
```

```
[(5, 1), (1, 2), (6, 2), (3, 4)]
```

8.3 Die eingebauten Funktionen `zip` und `enumerate`

Die eingebaute Funktion `zip` nimmt zwei oder mehr iterierbare Objekte und fügt diese im "Reißverschlussverfahren" zusammen, so dass eine Liste mit Tupeln entsteht. Dabei enthält jedes Tupel ein Elemente von jedem Partner:

```
list(zip([1, 2, 3], [4, 5, 6]))
```

```
[(1, 4), (2, 5), (3, 6)]
```

Dieser Effekt kommt häufig bei der Iteration durch mehrere Datenstrukturen zum Einsatz. Zum Beispiel der der Iteration über zwei Listen:

```
for x, y in zip([1, 2, 3], [4, 5, 6]):
    print(x, y)
```

```
1 4
2 5
3 6
```

Die kürzeste Sequenz bestimmt hierbei wie lang das Ergebnis wird. Die eingebaute Funktion `enumerate` macht es leicht auf den Index in einem Schleifendurchlauf zuzugreifen:

```
for index, value in enumerate('abcde'):
    print(index, value)
```

```
0 a
1 b
2 c
3 d
4 e
```

8.4 List-Comprehensions

Eine sehr elegante Methode eine Liste zu erzeugen sind List-Comprehension. Eine neue Liste wird erzeugt indem für jedes Element einer bestehenden Liste eine Operation durchgeführt wird, die optional noch ein oder mehreren Bedingungen gehorchen kann:

```
L = list(range(2, 11))
L
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
[x * 2 for x in L]
```

```
[4, 6, 8, 10, 12, 14, 16, 18, 20]
```

mit Bedingung:

```
[x * 2 for x in L if x > 5]
```

```
[12, 14, 16, 18, 20]
```

Mit dieser Methode lassen sich auch komplizierte Anweisungen kurz und verständlich formulieren.

8.5 Fortgeschrittene List-Comprehensions

Es gibt keine Begrenzung wie tief List-Comprehensions sein dürfen. Unser Sortier-Beispiel:

```
unsorted
```

```
['a', 'c', 'A', 'x', 'Y', 'y']
```

lässt sich auch ohne Nutzung von `key` mit verschachtelten List-Comprehensions lösen:

```
[outer[2] for outer in sorted(
    [(elem.lower(), n, elem)
     for n, elem in enumerate(unsorted)])]
```

```
['a', 'A', 'c', 'x', 'Y', 'y']
```

Diese Prinzip hat den Namen “Dekorieren-Sortieren-Entdekorieren” (“decorate-sort-undecorate”). Wir dekorieren unsere Listenelemente mit `(elem.lower(), n, elem)`. Da beim Sortieren immer das erste Element zum Vergleich zweier Elemente herangezogen wird, erfolgt die Sortierung nach Kleinbuchstaben. Danach nutzen wir `n`, den Index, den wir mit `enumerate` erhalten, als zweites Sortierkriterium. Damit bleibt die ursprüngliche Reihenfolge zweier gleichwertiger Elemente erhalten und wir haben eine stabile Sortierung. Im Entdekorierungs-Schritt holen wir mit `outer[2]` wieder unser Original-Element, nun sortiert, heraus.

8.6 Übungen

8.6.1 Übung 1

Erstellen sie eine kurze Liste mit mindestens 12 Ganzzahlen.

Ändern Sie den Wert des ersten Elements.

8.6.2 Übung 2

Löschen Sie die Elemente von Index 2 bis 6.

Fügen Sie dann 8 neue Zahlen an Stelle der Elemente von Index 3 bis 5 der neuen Liste in.

8.6.3 Übung 3

Hängen Sie 5 neue Zahlen an diese Liste an. Tun Sie dies:

(a) indem Sie jede Zahl einzeln anhängen und

(b) indem Sie alle 5 Zahlen in einer Liste zusammenfassen und mit einer Operation die einzelnen Zahlen anhängen.

8.6.4 Übung 4

Entfernen Sie das jeweils letzte Element der Liste. Nutzen Sie dazu drei verschiedene Methoden.

8.6.5 Übung 5

Kehren Sie die Reihenfolge der Liste selbst um (in place).

8.6.6 Übung 6

Erzeugen Sie eine Liste mit Tupeln mit jeweils zwei Elementen aus zwei vorher bestehenden Listen.

8.6.7 Übung 7

Erzeugen Sie eine neue Liste aus einer bestehenden Liste von Zahlen. Jedes Element der neuen Liste soll das Zehnfache des jeweiligen Element der alten Liste betragen. Nutzen Sie sowohl die Methode `append`, um die neue Liste aus einer leeren Liste in einer Schleife aufzubauen als auch eine List-Comprehension.

8.6.8 Übung 8

Fortgeschritten: Sortieren sie Ihre Liste. Definieren Sie dazu eine Funktion, die mit dem Schlüsselwortargument `key` genutzt werden kann und die Sortierung so modifiziert, dass alle geraden Zahlen vor den ungeraden Zahlen stehen. Hinweis: der Modulo-Operator `%` ergibt den Rest beim ganzzahligen Teilen und kann mit der Zahl `2` bei der Unterscheidung gerader und ungerader Zahlen helfen:

```
10 % 2
```

```
0
```

```
11 % 2
```

```
1
```

8.6.9 Übung 9

Diskutieren sie die unterschiedlichen Anwendungsfelder für Listen und Tupel.

9 Dictionarys im Detail

Dictionarys, die in anderen Programmiersprachen auch hash table oder assoziativer Array genannt werden, spielen eine zentrale Bedeutung in der Pythonprogrammierung und sind auch in der Implementierung von Python selbst immer wieder zu finden.

Dictionarys bestehen aus Schlüssel (key) - Wert (value) Paaren, die ohne Berücksichtigung der Reihenfolge gespeichert werden. Der Zugriff erfolgt über den Schlüssel. Auch aus sehr großen Dictionarys kann ein Schlüssel und sein zugehöriger Wert sehr schnell heraus gesucht werden.

Folgende Tabelle zeigt wichtige Methoden, die auf Dictionarys unterstützen.

Operation	Ergebnis
`len(d)`	die Anzahl der Einträge in `d`
`d[k]`	der Wert `d` für den Schlüssel `k`
`d[k] = v`	`d[k]` auf `v` setzen
`del d[k]`	`d[k]` aus `d` entfernen
`k in d`	`k` ist in `d.keys()`
`k not in d`	`k` ist nicht in `d.keys()`
`d.keys()`	Ansicht aller Schlüssel von `d`
`d.values()`	Ansicht aller Werte von `d`
`d.items()`	Ansicht aller Schlüssel-Wert-Paare von `d`
`d.update(b)`	`for k in d2.keys(): d[k] = d2[k]`
`d.clear()`	alle Einträge aus `d` entfernen
`d.copy()`	eine (flache) Kopie von `d`
`d.fromkeys(seq[, value])`	Erstellt ein neues Dictionary mit den Elementen von `seq` als Schlüssel, wobei alle Werte auf `value` gesetzt werden.
`d.get(k[, x])`	`d[k]` if `k` in `d`, else `x`
`d.setdefault(k[, x])`	`d[k]` if `k` in `d`, else `x` (zusätzlich `d[k] = x` bei `else`)
`d.pop(k[, x])`	`d[k]` if `k` in `d`, else `x` (und `k` löschen)
`d.popitem()`	irgendein Schlüssel-Wert-Paar entfernen und zurückgeben

Wir legen ein Dictionary an:

```
d = {'a': 100, 'b': 200, 'c': 300}
d
```

```
{'a': 100, 'b': 200, 'c': 300}
```

Die Länge gibt die eingebaute Funktion `len` preis:

```
len(d)
```

```
3
```

Wir können über die Schlüssel auf die Werte zugreifen:

```
d['a']
```

```
100
```

Das Zuweisen von neuen Werte für existierende Schlüssel:

```
d['a'] = 1000
d
```

```
{'a': 1000, 'b': 200, 'c': 300}
```

oder das Anlegen von neuen Schlüsseln:

```
d['x'] = 2000
d
```

```
{'a': 1000, 'b': 200, 'c': 300, 'x': 2000}
```

ist einfach.

Wir können einzelne Elemente mit `del` löschen:

```
del d['a']
d
```

```
{'b': 200, 'c': 300, 'x': 2000}
```

Wir fügen nun `a` wieder als Schlüssel hinzu:

```
d['a'] = 100
d
```

```
{'b': 200, 'c': 300, 'x': 2000, 'a': 100}
```

Jetzt erscheint der Schlüssel `a` am Ende des Dictionarys, denn Dictionarys erhalten die Einfügereihenfolge (dies gilt für Python >= 3.6).

Bisher war unser Dictionary homogen, d.h. all Schlüssel und alle Werte hatten jeweils den gleichen Datentyp. Alle Schlüssel sind Strings und alle Werte sind Ganzzahlen. In der Praxis sind Dictionarys oft homogen. Es ist aber möglich auch heterogene Dictionarys zu erstellen. Es gibt keinerlei Einschränkungen für die Werte. Alle Arten von Objekten könne Werte sein. Die Schlüssel allerdings "hashable" sein. Da alle eingebauten, unveränderliche Datentypen wie die Zahlen (`int`, `float` und `complex`), Strings und Tupel hashable sind kommen diese als Schlüssel in Frage. Wir fügen eine Liste Wert hinzu:

```
d['x'] = [1, 2, 3]
d
```

```
{'b': 200, 'c': 300, 'x': [1, 2, 3], 'a': 100}
```

Die Methode `copy` erstellt eine flache Kopie:

```
d2 = d.copy()
d2
```

```
{'b': 200, 'c': 300, 'x': [1, 2, 3], 'a': 100}
```

Flach bedeutet, dass nur das Dictionary selbst aber nicht die Werte kopiert werden. Dies hat für unveränderlich Objekten keine praktische Bedeutung. Bei unveränderlichen Objekten ist der Effekt der **flachen** Kopie gut zu sehen. Wir modifizieren die Liste in `d2`:

```
d2['x'][1] = 100
d2
```

```
{'b': 200, 'c': 300, 'x': [1, 100, 3], 'a': 100}
```

Schauen wir uns nun `d` an:

```
d
```

```
{'b': 200, 'c': 300, 'x': [1, 100, 3], 'a': 100}
```

Die Liste zum Schlüssel `x` in `d` hat sich auch geändert, da sich `d` und `d2` ihre Werte teilen. Wenn eine flache Kopie für den Anwendungsfall nicht passt, können wir auch eine tiefe Kopie machen:

```
import copy

d3 = copy.deepcopy(d)
```

Jetzt können wir die Liste in `d3` modifizieren:

```
d3
```

```
{'b': 200, 'c': 300, 'x': [1, 100, 3], 'a': 100}
```

```
d3['x'][1] = 200
d3
```

```
{'b': 200, 'c': 300, 'x': [1, 200, 3], 'a': 100}
```

ohne `d` zu beeinflussen:

```
d
```

```
{'b': 200, 'c': 300, 'x': [1, 100, 3], 'a': 100}
```

Wir können den gesamten Inhalt mit der Methode `clear` entfernen:

```
d.clear()
d
```

```
{}
```

Wir haben aber noch unser `d2`:

```
d2
```

```
{'b': 200, 'c': 300, 'x': [1, 100, 3], 'a': 100}
```

Wir können einfach `d2` den alternativen Namen `d` geben, so dass wir weiter mit `d` arbeiten können:

```
d = d2
d
```

```
{'b': 200, 'c': 300, 'x': [1, 100, 3], 'a': 100}
```

Beide Namen `d` und `d2` zeigen auf das selbe Objekt. Wir können das mit dem Schlüsselwort `is` testen:

```
d is d2
```

```
True
```

Der Mitgliedschaftstest mit dem `in`-Operator lässt bezieht sich auf die Schlüssel:

```
'a' in d
```

```
True
```

```
'h' in d
```

```
False
```

```
'h' not in d
```

```
True
```

Wir können alle Schlüssel:

```
d.keys()
```

```
dict_keys(['b', 'c', 'x', 'a'])
```

alle Werte:

```
d.values()
```

```
dict_values([200, 300, [1, 100, 3], 100])
```

und alle Schlüssel-Wert-Paare auflisten:

```
d.items()
```

```
dict_items([('b', 200), ('c', 300), ('x', [1, 100, 3]), ('a', 100)])
```

Die von `d.keys()`, `d.values()` und `d.items()` erzeugten Objekte sind Ansichten (View-Objekte), die fest mit dem Dictionary `d` verbunden sind und deshalb Änderungen in `d` reflektieren. Wir erzeugen die Schlüssel-Ansicht:

```
keys = d.keys()
keys
```

```
dict_keys(['b', 'c', 'x', 'a'])
```

ändern unser Dictionary:

```
del d['x']
d
```

```
{'b': 200, 'c': 300, 'a': 100}
```

und schauen uns `keys` wieder an:

```
keys
```

```
dict_keys(['b', 'c', 'a'])
```

Diese zeigen die aktuellen Schlüssel in `d`.

Mit der Methode `update` können wir leicht zwei Dictionarys zusammen fügen, wobei das zweite Dictionary, das in der Methode als Argument vorkommt, die Werte des ersten überschreibt:

```
new = {'a': 500, 'h': 600}
new
```

```
{'a': 500, 'h': 600}
```

```
d
```

```
{'b': 200, 'c': 300, 'a': 100}
```

```
d.update(new)
d
```

```
{'b': 200, 'c': 300, 'a': 500, 'h': 600}
```

Der Wert des Schlüssels `a` hat sich geändert und das Schlüssel-Wert-Paar `h-600` ist jetzt neu in `d`, da `h` vorher kein Schlüssel in `d` war.

Die Methode `fromkeys` gibt ein neues Dictionary mit den als erstes Argument übergebenen Werten als Schlüssel und dem zweiten Argument als Wert für alle Elemente zurück:

```
dict.fromkeys([1, 2, 3], 'a')
```

```
{1: 'a', 2: 'a', 3: 'a'}
```

Wir nutzen hier `dict`, die Dictionary-Klasse. Ohne zweites Argument sind alle Werte `None`:

```
dict.fromkeys([1, 2, 3])
```

```
{1: None, 2: None, 3: None}
```

Alle Schlüssel bekommen den selben Wert. Das ist bei unveränderlichen Werten wie `a` oder `None` unerheblich. Bei veränderlichen Werten ist dies aber zu beachten. Nehmen wir zum Beispiel eine Liste als Wert:

```
d_magic = dict.fromkeys('abc', [])
d_magic
```

```
{'a': [], 'b': [], 'c': []}
```

Nun modifizieren wir die Liste hinter Schlüssel `a`:

```
d_magic['a'].append(700)
d_magic
```

```
{'a': [700], 'b': [700], 'c': [700]}
```

Obwohl wir nur die Liste hinter `a` geändert haben, haben sich wie durch Magie alle drei Werte des Dictionarys geändert. Das lässt sich mit diesem Beispiel-Code erklären, der äquivalent zu `dict.fromkeys()` funktioniert:

```
d_magic2 = {}
value = []
for key in 'abc':
    d_magic2[key] = value
d_magic2
```

```
{'a': [], 'b': [], 'c': []}
```

Wir modifizieren jetzt wieder den Wert hinter `a`:

```
d_magic2['a'].append(700)
d_magic2
```

```
{'a': [700], 'b': [700], 'c': [700]}
```

Das Problem ist die wiederholte Nutzung von `value`. Wenn wir diese implizite Änderung aller Werte nicht wollen, müssen wir die Liste in der Schleife bei jedem Durchlauf mit `[]` neu erzeugen:

```
d_unmagic = {}
for key in 'abc':
    d_unmagic[key] = []
d_unmagic
```

```
{'a': [], 'b': [], 'c': []}
```

Jetzt modifizieren wir wirklich nur die Liste hinter `a`:

```
d_unmagic['a'].append(700)
d_unmagic
```

```
{'a': [700], 'b': [], 'c': []}
```

Die Methode `get` verhält sich im Prinzip wie der Zugriff mit den eckigen Klammern:

```
d['a']
```

```
500
```

```
d.get('a')
```

```
500
```

Wenn der Schlüssel nicht im Dictionary erhalten wir hier eine Fehlermeldung:

```
d['xxx']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-44-634b17981662> in <module>
----> 1 d['xxx']

KeyError: 'xxx'
```

Mit `get` erhalten wir einen Rückgabewert:

```
res = d.get('xxx')
print(res)
```

```
None
```

Wir können aber auch einen Vorgabewert mitgeben:

```
d.get('xxx', -999)
```

```
-999
```

Wenn der Schlüssel im Dictionary ist hat der Vorgabewert keine Wirkung:

```
d.get('a', -999)
```

```
500
```

Die Methode `get` hat unser Dictionary nicht verändert:

```
d
```

```
{'b': 200, 'c': 300, 'a': 500, 'h': 600}
```

Die Methode `setdefault` verhält sich genau wie `get`, fügt aber den Vorgabewert zusätzlich in das Dictionary ein, wenn der Schlüssel nicht enthalten ist:

```
d.setdefault('xxx', -999)
```

```
-999
```

```
d
```

```
{'b': 200, 'c': 300, 'a': 500, 'h': 600, 'xxx': -999}
```

Das Entfernen und gleichzeitige Zurückgeben eines Wertes erreichen wir mit der Methode `pop`:

```
d.pop('xxx')
```

```
-999
```

```
d
```

```
{'b': 200, 'c': 300, 'a': 500, 'h': 600}
```

```
d.popitem?
```

```
Signature: d.popitem()
Docstring:
Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order.
Raises KeyError if the dict is empty.
Type:      builtin_function_or_method
```

Ohne eine Angabe eines Schlüssels und bei Verwendung von `popitem` trifft das "letzte" Schlüssel-Wert-Paar (Nutzung des Prinzips LIFO (last-in, first-out)):

```
d
```

```
{'b': 200, 'c': 300, 'a': 500, 'h': 600}
```

```
d.popitem()
```

```
('h', 600)
```

```
d
```

```
{'b': 200, 'c': 300, 'a': 500}
```

9.1 Alternative Erzeugung eines Dictionarys

Die eingebaute Funktion `dict` erzeugt aus einer Liste mit Zweier-Tupeln ein Dictionary mit dem jeweils ersten Element als Schlüssel und dem zweiten als Wert:

```
dict([('a', 100), ('b', 200), ('c', 300)])
```

```
{'a': 100, 'b': 200, 'c': 300}
```

Aus zwei iterierbaren Objekten wird dann mit Hilfe von `zip` schnell ein Dictionary:

```
keys = 'abc'
values = [100, 200, 300]
dict(zip(keys, values))
```

```
{'a': 100, 'b': 200, 'c': 300}
```

9.2 Übungen

9.2.1 Übung 1

Erzeugen Sie ein Dictionary mit Namen und Telefonnummern.

Greifen Sie auf die Einträge über den Namen zu.

Testen Sie ob bestimmte Namen eingetragen sind.

Listen Sie alle Telefonnummern auf.

Tun Sie das Gleiche für die Namen.

9.2.2 Übung 2

Greifen Sie auf nicht existierende Namen in Ihrem Dictionary aus Übung 1 zu ohne einen Fehler auszulösen. Geben Sie dabei (a) `None` und (b) die Zahl Null zurück.

Modifizieren Sie die Lösung und fügen Sie diesen Vorgabewert nun auch in das Dictionary ein. Hinweis: Diese Aufgaben lassen sich jeweils mit einem Methodenaufruf des Dictionarys lösen.

9.2.3 Übung 3

Aktualisieren Sie Ihre Telefonnummern mit einem zweiten Dictionary, das für eine Person im bestehenden "Telefonbuch" eine neue Telefonnummer und einen ganz neuen Eintrag von Name und Nummer enthält. Beispiel:

```
neue_nummern = {
    'bestehender Name': '353646',
    'neuer Name': '64467'
}
```

Nutzen Sie dazu zwei Methoden:

- (a) eine Schleife und ändern Sie einzelne Elemente und
- (b) eine dafür nützliche Methode des Dictionarys.

Hinweis: Um den Effekt der Änderungen zu zeigen sollten Sie Ihr Telefonnummern-Dictionary vorher kopieren und die Änderungen an der jeweiligen Kopie vornehmen. Es gibt eine Methode zum Kopieren eines Dictionarys.

9.2.4 Übung 4

Entfernen Sie Einträge aus Ihrem Telefon-Dictionary für von Ihnen vorgegebene Namen (a) ohne sich die entfernte Telefonnummer anzeigen zu lassen und (b) mit Anzeige der entfernten Telefonnummer.

Entfernen Sie eine Telefonnummer ohne den Namen oder die Telefonnummer vorzugeben. Lassen Sie sich dabei den entfernten Namen und die dazu gehörige Telefonnummer anzeigen.

9.2.5 Übung 5

Vergleichen sie die Geschwindigkeit der Suche in einem Dictionary und in einer Liste (siehe Tipp unten zur Zeitmessung). Bauen Sie dazu eine lange Liste und ein großes Dictionary (mit je 100, 1000, 10 000 oder mehr Elementen). Platzieren sie das zu suchende Element in die Mitte der Liste. Im Dictionary wird das zu suchende Element als Schlüssel abgelegt. Prüfen Sie ob das Element in der Liste bzw. dem Dictionary enthalten ist. Nutzen Sie dafür die für die jeweilige Datenstruktur geeignete Methode. Nehmen sie an, dass Sie die Listenposition des zu prüfenden Elements nicht kennen.

Zeitmessung

In Jupyter Notebooks oder in IPython können Sie die magische Funktion `%timeit` nutzen. Zum Beispiel lässt sich so die Zeit für die Berechnung `1 + 1` messen:

```
%timeit 1 + 1
10.6 ns ± 0.071 ns per loop
(mean ± std. dev. of 7 runs, 100000000 loops each)
```

Als Ergebnis erhalten Sie den Mittelwert und die Standardabweichung für sieben Läufe mit einer großen Zahl von Schleifen (hier: 100.000.000). Die Anzahl der Schleifen bestimmt `%timeit` dynamisch so, dass ein Durchlauf ca. eine Sekunde dauert. Sie warten also ca. 7 Sekunden bis das Ergebnis erscheint. Hilfe dazu erhalten Sie mit `%timeit?`.
In einer Python-Quelltext-Datei können Sie die Funktion `time.perf_counter_ns()` (wenn nicht verfügbar `time.perf_counter()` für Werte in Sekunden) nutzen , um einen aktuellen Zeitstempel in Nanosekunden zu ermitteln. Durch Differenzbildung zwischen zwei Zeitstempeln lässt sich die dazwischen vergangene Zeit ermitteln. Hier sind zum Beispiel zwischen dem Berechnen von `start` und `end` ca. 6,5 Sekunden vergangen:

```
import time
start = time.perf_counter_ns()
end = time.perf_counter_ns()
print((end - start) / 1e9)
6.5461903677702056
```

Bei Nutzung von `time.perf_counter()` entfällt die Teilung durch `1e9`, da die Einheit des Zeitstempels Sekunden und nicht Nanosekunden, also 1e9 Sekunden, ist.

10 Mengen im Detail

Bei Mengen handelt es sich um ungeordnete Zusammenstellungen, eindeutiger Objekte, d.h. es gibt keine Mehrfacheinträge. Mit `set([1, 2, 3])` kann eine neue Menge erzeugt werden. Ein `set` bietet insbesondere Operationen, die aus der Mengenlehre stammen. So können Untermengen bestimmt, sowie Vereinigungen, Differenzen und Schnittmengen gebildet werden. Neben den veränderbaren Typ `set` gibt es den nicht veränderbaren Typ `frozenset`. Die Elemente von Mengen dürfen nicht veränderbar sein. Listen können also keine Elemente von Mengen werden.

Mengen unterstützen eine Reihe von gemeinsamen Operationen, die die folgende Tabelle zusammenfasst.

Operation	Äquivalent	Ergebnis
`len(s)`		Kardinalität der Menge `s`
`x in s`		Test ob `x` in `s` enthalten ist
`x not in s`		Test ob `x` nicht in `s` enthalten ist
`s.issubset(t)`	`s <= t`	Test ob jedes Element von `s` in `t` ist
`s.issuperset(t)`	`s >= t`	Test ob jedes Element von `t` in `s` ist
`s.isdisjoint(t)`		`True` wenn die Schnittmenge (intersection) von `s` und `t` leer ist
`s.union(t)`	`s | t`	neue Menge mit allen Elementen von `s` und `t`
`s.intersection(t)`	`s & t`	neue Menge mit den Elementen, die sowohl in `s` als auch in `t` sind
`s.difference(t)`	`s - t`	neue Menge mit den Elementen, die in `s` aber nicht in `t` sind
`s.symmetric_difference(t)`	`s ^ t`	neue Menge mit den Elementen, die entweder in `s` oder in `t` aber nicht in beiden sind
`s.symmetric_difference_upd`		die Elemente von `s.symmetric_difference(t)` ersetzen alle Element von `s`
`s.copy()`		neue Menge mit einer flachen Kopie von `s`
`s.add(t)`		neues Element `t` zur Menge `s` hinzufügen
`s.clear()`		alle Elemente aus der Menge entfernen
`s.remove(x)`		Element `x` aus der Menge entfernen, wirft `KeyError` wenn `x` nicht in `s`
`s.discard(x)`		Element `x` aus der Menge entfernen, tut nichts `x` nicht in `s`
`s.update(t)`		die Elemente der Vereinigungsmenge (union) von `s` und `t` ersetzen alle Element von `s`

Wir definieren eine neue Menge mit:

```
{1, 2, 3, 4, 5, 1, 2, 2}
```

```
{1, 2, 3, 4, 5}
```

oder:

```
s1 = set([1, 2, 3, 4, 5, 1, 2, 2])
s1
```

```
{1, 2, 3, 4, 5}
```

Die Elemente müssen den gleichen Kriterien wie die Schlüssel eines Dictionarys entsprechen, also unveränderlich sein. Mehrfachwerte gibt es nicht.

Der Mitgliedschaftstest funktioniert:

```
1 in s1
```

```
True
```

Wir definieren eine zweite Menge:

```
s2 = set([4, 5, 6, 7, 8])
s2
```

```
{4, 5, 6, 7, 8}
```

Die Schnittmenge (oder Durchschnittsmenge) lässt sich leicht ermitteln:

```
s1.intersection(s2)
```

```
{4, 5}
```

Alternativ:

```
s1 & s2
```

```
{4, 5}
```

Auch die Vereinigungsmenge ist nur einen Methodenaufruf entfernt:

```
s1.union(s2)
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Alternativ:

```
s1 | s2
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Die Differenz in beide Richtungen ist ebenfalls verfügbar:

```
s1 - s2
```

```
{1, 2, 3}
```

```
s2 - s1
```

```
{6, 7, 8}
```

Die symmetrische Differenz, also die Menge die entweder in der ersten oder zweiten Menge, aber nicht in beiden ist ermittelt die Methode `symmetric_difference`:

```
s1.symmetric_difference(s2)
```

```
{1, 2, 3, 6, 7, 8}
```

Alternativ:

```
s1 ^ s2
```

```
{1, 2, 3, 6, 7, 8}
```

Die Differenz einer Menge mit sich selbst ist eine leere Menge:

```
s1 - s1
```

```
set()
```

Die Methode `issuperset` ermittelt , ob eine Menge eine Obermenge einer anderen Menge ist:

```
s1
```

```
{1, 2, 3, 4, 5}
```

```
s1.issuperset(set([1, 2, 3]))
```

```
True
```

```
s1.issuperset(set([1, 2, 30]))
```

```
False
```

Mit `issubset` lässt sich ermitteln ob eine Menge eine Untermenge einer anderen Menge ist:

```
s1.issubset(set(range(10)))
```

```
True
```

```
s1.issubset(set(range(2, 10)))
```

```
False
```

Normale Mengen erlauben das Hinzufügen neuer Elemente:

```
s1.add(10)
s1
```

```
{1, 2, 3, 4, 5, 10}
```

Mit `frozenset` können wir unveränderliche Mengen erzeugen:

```
fs = frozenset([1,2,3])
fs
```

```
frozenset({1, 2, 3})
```

Der Versuch dieser Menge etwas hinzufügen schlägt fehl:

```
fs.add(10)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-23-f22e12847559> in <module>
----> 1 fs.add(10)

AttributeError: 'frozenset' object has no attribute 'add'
```

10.1 Übungen

10.1.1 Übung 1

Definieren Sie zwei Mengen mit Namen von Personen, sodass zwei Namen sowohl in der einen als auch der anderen Menge vorkommen. Zum Beispiel könnten Sie eine Menge von Personen, die mit den ÖPNV (Bus, Bahn etc.) und eine Gruppe, die mit Fahrrad zu Arbeit fahren erstellen. Dabei sollten einige Personen nur mit ÖPNV, einige nur mit dem Fahrrad und einige manchmal mit dem und manchmal mit dem anderen Verkehrsmittel fahren.

Prüfen sie ob bestimmte Namen in den Mengen vorkommen.

Bilden Sie

- Schnittmenge
- Differenz
- Vereinigung

der beiden Mengen.

10.1.2 Übung 2

Prüfen Sie ob eine zweite Menge mit Namen eine Untermenge Ihrer Menge aus Übung 1 ist.

10.1.3 Übung 3

Definieren Sie eine Menge, die ihrerseits wiederum aus Mengen von Namen besteht. Definieren Sie eine zweite Menge, die eine Schnittmenge mit dieser Menge hat.

10.1.4 Übung 4

Erweitern Sie Ihr Programm aus Übung 5 zu Dictionarys und verwenden Sie eine Menge, um zu prüfen, ob das gesuchte Element darin enthalten ist. Vergleichen Sie die Laufzeit mit der für die Liste und das Dictionary (aus Übung 5 zu Dictionarys).

11 Funktionen

Oft wiederholen sich Aufgaben in einem Programm an unterschiedlichen Stellen. Das mehrfache Kopieren gleicher oder sehr ähnlicher Quelltextstücke ist nicht zu empfehlen, da Änderungen dann an unterschiedlichen Stellen ausgeführt werden müssten, was langfristig unweigerlich zu Abweichungen zwischen diesen Stellen führt. Mit Funktionen lassen sich Quelltextabschnitte zusammenfassen, mit einen Namen versehen und von anderer Stelle aus aufrufen.

Eine Funktion kann Argumente entgegen nehmen und ein Ergebnis zurückgeben:

```
def add(x, y):
    """Add two objects."""
    return x + y
```

Unsere Funktion `add` nimmt die Argumente `x` und `y` entgegen und gibt als Ergebnis die Summe beider zurück. Wir können die Funktion nun aufrufen und erhalten unsere Summe:

```
add(10, 20)
```

```
30
```

Wir können uns die Hilfe zu unsere Funktion ansehen:

```
add?
```

```
Signature: add(x, y)
Docstring: Add two objects.
File:      /var/folders/p_/yckgs9wn0xbcmqc92zk3z4240000gn/T/ipykernel_20851/
 ↪1541875889.py
Type:      function
```

Wir sehen, dass der Docstring, als der erste literale String in unserer Funktion erhalten bleibt. Per Konvention nutzen wir ein drei Anführungszeichen für diesen String. Der Doctsring ist auch als Attribut `__doc__` verfügbar:

```
add.__doc__
```

```
'Add two objects.'
```

Wir können unsere Funktion mit beliebigen Objekten aufrufen:

```
add([1, 2, 3], [4, 5, 6])
```

```
[1, 2, 3, 4, 5, 6]
```

```
add('abc', 'xyz')
```

```
'abcxyz'
```

Dies geht solange die beiden Argumente `x` und `y`, die unsere Funktion erhält mit dem Operator `+` zurecht kommen. Wenn dies nicht der Fall ist bekommen wir eine entsprechende Ausnahme:

```
add(10, 'abc')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-3e109cf0fa05> in <module>
----> 1 add(10, 'abc')

<ipython-input-3-9bc94e2bf6d0> in add(x, y)
      1 def add(x, y):
      2     """Add two objects."""
----> 3     return x + y

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Funktionen können aber auch so genannte Seiteneffekte auslösen. Die `return`-Anweisung ist optional.:

```
def show(text):
    print('Ausgabe:', text)
```

Die Funktion `show` nimmt ein Argument (`text`) und gibt es auf dem Bildschirm aus. Wir können das überprüfen:

```
show('Hallo')
```

```
Ausgabe: Hallo
```

Natürlich können Funktionen wesentlich komplexere Algorithmen beinhalten. Sie sind ein gutes Mittel Programme übersichtlich zu strukturieren.

Die Übergabe von Parametern erfolgt nach Position. So wird im ersten Beispiel `add(10, 20)` die `10` dem `x` und die `20` dem `y` zugeordnet. Es sind weitere Formen von Parameterübergabe mit voreingestellten Parametern, Schlüsselwörtern und beliebiger Anzahl von Parametern möglich.

Schlüsselwortparameter müssen in keiner bestimmten Reihenfolge übergeben werden:

```
add(x='abc', y='xyz')
```

```
'abcxyz'
```

Jetzt übergeben wir die Parameter in anderer Reihenfolge, aber wie als Schlüsselwortargumente:

```
add(y='xyz', x='abc')
```

```
'abcxyz'
```

Das Ergebnis ist das gleiche, da die Zuordnung über die Schlüsselworte `x` und `y` und nicht über die Position erfolgt. Das ist insbesondere für lange Parameterlisten nützlich.

Eine Funktion kann Vorgabeparameter haben. So können wir `y` mit einem Wert vorbelegen:

```
def add_optional(x, y=0):
    return x + y
```

Wir können unsere Funktion weiterhin mit zwei Argumenten aufrufen:

```
add_optional(10, 20)
```

```
30
```

Wenn wir `y` bei der Übergabe weglassen, nimmt `y` den Wert `0` an:

```
add_optional(10)
```

```
10
```

Mit unser Original-Funktion funktioniert dies nicht:

```
add(10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-fb866b66e1c7> in <module>
----> 1 add(10)

TypeError: add() missing 1 required positional argument: 'y'
```

Die Anzahl der Argumente und Schlüsselwortargumente muss nicht zum Zeitpunkt der Funktionsdefinition festgelegt werden:

```
def func(*args, **kwargs):
    print(args)
    print(kwargs)
```

```
func(1, 2, 3, g=5, z=10)
```

```
(1, 2, 3)
{'g': 5, 'z': 10}
```

Dabei nimmt `args` alle positionellen und `kwargs` alle Schlüsselwort-Argmuente auf. Beim Funktionsaufruf packt `*` eine Sequenz als positionelle und `**` die Schlüsselwort-Argmuente aus:

```
my_args = (10, 20)
add(*my_args)
```

```
30
```

Dies ist äquivalent zu:

```
add(my_args[0], my_args[1])
```

```
30
```

Das geht auch mit Schlüsselwort-Argmuenten:

```
my_kwargs = {'x': 10, 'y': 20}
```

```
add(**my_kwargs)
```

```
30
```

Dies ist äquivalent zu:

```
add(x=my_kwargs['x'], y=my_kwargs['y'])
```

```
30
```

Ein typische Anwendung wäre der Aufruf von `print` mit mehreren Argumenten. Diese Liste:

```
L = [1, 2, 3]
```

könnten wir so ausgeben:

```
print(L[0], L[1], L[2])
```

```
1 2 3
```

Alternativ könnten wir die *-Methode nutzen:

```
print(*L)
```

```
1 2 3
```

Damit ist die Länge der Liste unerheblich.

Funktionen sind Objekte und können deshalb auch als Werte in Dictionarys gespeichert werden.

Wir definieren zwei einfache Funktionen:

```
def func1():
    return 1

def func2():
    return 2
```

und speichern diese in einem Dictionary:

```
my_funcs = {'case1': func1, 'case2': func2}
```

Nun können wir über dieses Dictionary auf die Funktionen zugreifen und diese auch rufen:

```
my_funcs['case1']
```

```
<function __main__.func1()>
```

```
my_funcs['case1']()
```

```
1
```

```
my_funcs['case2']()
```

```
2
```

Damit lässt sich z.B. das Verhalten einer `case`- oder `switch`-Anweisung nachahmen.

Mit dem so genannten Tupel-Unpacking kann eine Funktion mehrere Rückgabewerte haben:

```
def zwei():
    return 100, 200
```

```
a, b = zwei()
```

```
a
```

```
100
```

```
b
```

```
200
```

Die `return`-Anweisung packt die beiden Zahlen in einen Tupel ein, das beim Aufruf auf der linken Seite wieder ausgepackt wird.

Die Anzahl der `return`-Anweisungen in einer Funktion ist nicht begrenzt:

```
def limit10(value):
    if value > 10:
        return 10
    return value
```

```
limit10(23)
```

```
10
```

```
limit10(3)
```

```
3
```

In Python sind Methoden Funktionen, die einer Klasse zugeordnet sind. Somit trifft alles was für Funktionen gilt auch für Methoden zu. Konsequenterweise werden sowohl Funktionen als auch Methoden mit `def` definiert.

11.1 Übungen

11.1.1 Übung 1

Schreiben Sie eine Funktion, die die Zahlen von 0 bis 9 mit `print()` ausgibt.

11.1.2 Übung 2

Schreiben Sie eine Funktion, die das Doppelte einer Zahl zurück gibt.

11.1.3 Übung 3

Schreiben Sie eine Funktion, die zwei Strings zusammenfügt (konkateniert) und übergeben Sie dieser Funktion Schlüsselwort-Argumente in unterschiedlichen Reihenfolgen.

11.1.4 Übung 4

Schreiben Sie eine Funktion, die das arithmetische Mittel (Summe geteilt durch die Anzahl) von drei ihr als Argumente übergebenen Zahlen berechnet.

11.1.5 Übung 5

Nutzen Sie einen voreingestellten Parameter für die Funktion aus Übung 4, so dass diese auch mit zwei Argumenten aufrufbar ist. Achtung: Die Summe der Argumente muss je nach Fall durch zwei oder drei geteilt werden. Ein sinnvoller Vorgabewert könnte `None` sein, da dieser sich eindeutig mit `var_name is None` von Zahlen unterscheiden lässt.

11.1.6 Übung 6

Fortgeschritten: Schreiben Sie eine Funktion, die eine beliebige andere Funktion aufrufen und deren Laufzeit messen kann. Die Mess-Funktion soll die aufzurufende Funktion als erstes Argument und deren Parameter als weitere Argumente haben. (Hinweis: `*` und `**` erlauben eine unbekannte Anzahl von positionellen und Schlüsselwort-Argumenten zu verarbeiten.) Für die Zeitmessung können Sie `timeit.default_timer` nutzen.

12 Iteratoren und Generatoren

Iteratoren und Generator erzeugen eine Folge von Objekten. Im Gegensatz zu einer Liste, in der alle Objekte gleichzeitig im Arbeitsspeicher existieren, erzeugt Python diese Objekte eins nach dem anderen, wenn sie nötig sind.

12.1 Iteratoren

Python nutzt Iteratoren an vielen Stellen. Beispiele sind `open`, `zip`, `enumerate` und `reversed`. Alle geben nach dem Aufruf einen Iterator zurück.

Beginnen wir mit einer Liste, die 10 Zahlen enthält:

```
L = list(range(2, 12))
L
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

`reversed()` erzeugt den Iterator `r`:

```
r = reversed(L)
```

Das Ergebnis nach einer Umwandlung in eine Liste ist eine Liste in umgekehrter Reihenfolge:

```
list(r)
```

```
[11, 10, 9, 8, 7, 6, 5, 4, 3, 2]
```

Nun ist `r` erschöpft und der erneute Versuch eine Liste daraus zu erzeugen resultiert in einer leeren Liste:

```
list(r)
```

```
[]
```

Nach dem Erstellen eines neuen Iterators:

```
r = reversed(L)

for x in r:
    print(x, end=' ')
```

```
11 10 9 8 7 6 5 4 3 2
```

Hinweis: Das Schlüsselwortargument `end='  '` ändert das Verhalten von `print()`. Anstatt `\n` am Ende, also einen Zeilenumbruch hinzuzufügen (Vorgabe) fügt der Aufruf `print()` ein Leerzeichen ein. Somit erscheinen alle Zahlen auf einer Zeile anstatt eine Zahl pro Zeile.

Der Iterator `r` ist nun erschöpft und der Versuch nochmals darüber zu iterieren resultiert in Null Schritten und es erscheint keine Ausgabe:

```
for x in r:
    print(x, end=' ')
```

Nach dem Erstellen eines weiteren Iterators:

```
r = reversed(L)
```

ist es möglich Einzelschritte mit der eingebauten Funktion `next()` zu machen:

```
next(r)
```

```
11
```

```
next(r)
```

```
10
```

Nach diesen beiden Schritten lässt sich `r` in eine Liste mit den restlichen Elementen umwandeln:

```
list(r)
```

```
[9, 8, 7, 6, 5, 4, 3, 2]
```

danach wirft der Versuch das nächste Element mit `next()` zu holen die Ausnahme `StopIteration`:

```
next(r)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-13-8ebe59a56b1d> in <module>
----> 1 next(r)

StopIteration:
```

Eine `for`-Schleife über den Iterator `r` ist konzeptionell **äquivalent** zu diesen Zeilen:

```
r = reversed(L)
while True:
    try:
        print(next(r), end=' ')
    except StopIteration:
        break
```

```
11 10 9 8 7 6 5 4 3 2
```

Technisch sind Iteratoren Generatoren, die zu einer Kollektionen gehören. Die eingebaute Funktion `iter` konvertiert eine Kollektion in einen Iterator:

```
s = iter(range(3))
```

```
next(s)
```

```
0
```

```
next(s)
```

```
1
```

```
next(s)
```

```
2
```

```
next(s)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-19-61c30b5fe1d5> in <module>
----> 1 next(s)

StopIteration:
```

12.2 Generatoren

Es gibt zwei Methoden einen Generator zu erzeugen: Generatorausdrücke und Generatorfunktionen.

12.2.1 Generatorausdrücke

Generatorausdrücke sehen aus wie *List-Comprehensions* mit runden Klammern:

```
g = (n for n in range(20) if not n % 2)
```

```
list(g)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

12.2.2 Generatorfunktionen

Funktionen, in denen eine `yield`-Anweisung vorkommt, sind Generatorfunktionen.

```
def simple():
    print('Start')
    yield 1
    print('nach 1')
    yield 2
    print('nach 2')
    yield 3
    print('nach 3')
```

Wir erstellen eine Instanz:

```
s = simple()
```

Mit `next()` gehen wir in den Funktionskörper und suchen nach dem nächsten Auftreten von `yield`:

```
next(s)
```

```
Start
```

```
1
```

Wir setzen an der Stelle direkt nach `yield 1` wieder an:

```
next(s)
```

```
nach 1
```

```
2
```

jetzt geht es nach `yield 2` weiter:

```
next(s)
```

```
nach 2
```

```
3
```

Nun gibt es kein `yield` mehr. Deshalb wirft die Nutzung von `next()` die Ausnahme `StopIteration`, den `next()` bedeutet "geh zum nächsten `yield`:

```
next(s)
```

```
nach 3
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-27-61c30b5fe1d5> in <module>
----> 1 next(s)

StopIteration:
```

Generatoren speichern Zustandsinformationen zwischen den Aufrufen von `yield`:

```
def state(start=0):
    value = start
    yield value
    value += 1
    yield value
    value += 1
    yield value
```

```
st = state()
```

```
next(st)
```

```
0
```

```
next(st)
```

```
1
```

```
next(st)
```

```
2
```

Generatoren können auch endlos sein:

```
def endless(start=0):
    value = start
    while True:
            yield value
            value += 1
```

```
e = endless()
```

```
next(e)
```

```
0
```

```
next(e)
```

```
1
```

```
next(e)
```

```
2
```

12.2.3 range

Die eingebaute Funktion `range()` gibt ein Objekt zurück, das einem Iteratoren ähnelt. Der Hauptunterschied zu einem Iterator besteht darin, dass sich das `range`-Objekt nicht erschöpft und deshalb mehrere Durchläufe ermöglicht. Es hat aber keine Methode `next()` und erlaubt deshalb keine Einzelschritte. `range`-Objekt erlauben auch das Slicing (Teil-Bereichs-Zugriff):

```
r20 = range(20)
r20
```

```
range(0, 20)
```

```
r20[5:15]
```

```
range(5, 15)
```

12.3 Übungen

12.3.1 Übung 1

Definieren Sie einen Generator mit Hilfe eines Generatorausdrucks, der das Dreifache der Zahlen von 0 bis 20 ausgibt.

12.3.2 Übung 2

Modifizieren Ihren Generator aus Übung 1 so, dass dieser nur gerade Zahlen ausgibt. Hinweis: Der Modulo-Operator `%` liefert den Rest einer ganzzahligen Division. Mit der Zahl Zwei als Divisor erhalten wir für alle geraden Zahlen eine Null und für alle ungeraden Zahlen eine Eins:

```
>>> 10 % 2
0
>>> 9 % 2
1
```

12.3.3 Übung 3

Schreiben Sie eine Generatorfunktion, die beim ersten Aufruf von `next()` die Zahl `10` und bei allen anderen Aufrufen von `next()` die Zahlen `100`, `200` ... `900` ausgibt.

12.3.4 Übung 4

Vergleichen Sie den Speicherverbrauch zweier Python-Programme, die für die Repräsentation einer Reihe von Zahlen

1. eine Liste und
2. einen Generatorausdruck verwenden.

Die Liste bzw. der Generator soll nacheinander 10.000, 100.000 und 1.000.000 Zahlen repräsentieren.

Es gibt zwei Möglichkeiten den Speicherverbrauch zu messen.

(1) Nutzen Sie das mitgelieferte Script `measure_mem.py`. Importieren dieses in Ihrem Programm:

```
import measure_mem

#  Ihr Programm-Code

measure_mem.show_memory()
```

Sie müssen dafür `psutil` installieren. Wenn Sie einen Import-Fehler erhalten, der sich über das Fehlen `psutil` beschwert, installieren Sie das Modul mit `conda` oder `pip`.

Wenn Sie `conda` nutzen geben Sie dies auf der Kommandozeile ein::

```
conda install psutil
```

andernfalls nutzen Sie `pip`:

```
pip install psutil
```

(2) Wenn Sie `psutils` nicht installieren können, nutzen Sie bitte das für ihr Betriebssystem verfügbare Programm zur Überwachung von Prozessen wie den Taskmanager unter Windows, die Aktivitätsanzeige unter Mac OSX oder `top` unter Linux, um den Speicherverbrauch für beide Programme zu messen.

Hinweise: Mit `input()` können Sie die Ausführung des Programms anhalten. Mit Hilfe der Funktion `getpid()` aus dem Modul `os` können Sie die eigene Prozess-ID bestimmen. Suchen Sie diese ID in dem Prozess-Beobachtungs-Werkzeug Ihres Betriebssystems. Wenn es keine Spalte dafür anzeigt, müssen Sie die Spalte PID in den Anzeigeoptionen des Programms auswählen.

13 Klassen

13.1 Grundlagen

Die Objektorientierung hat sich in den letzten Jahren als vorherrschendes Programmierparadigma etabliert. Ein wichtiger Teil dieser Art der Programmierung sind Klassen.

Mit dem Schlüsselwort `class` definieren wir eine neue Klasse:

```
class Auto:
    """Ein ganz einfaches Auto.
    """

    def __init__(self, marke, farbe):
        self.marke = marke
        self.farbe = farbe
        self.km_stand = 0

    def fahre(self, km):
        """Kilometerstand um `km` erhöhen."""
        self.km_stand += km
```

Unsere Klasse hat die Attribute `marke`, `farbe` und `km_stand`. Die Methode `fahre` erhöht den Kilometerstand um einen gegebenen Betrag. Methoden sind Funktionen, die zu einer Klasse gehören. Die Methode `__init__` hat eine besondere Bedeutung. Sie initialisiert die neue Instanz einer Klasse mit Anfangswerten. Eine Instanz `auto1` erzeugen wir so:

```
auto1 = Auto('VW', 'rot')
```

Unser neues Auto hat einen Kilometerstand von Null:

```
auto1.km_stand
```

```
0
```

Nun fahren wir 15 km:

```
auto1.fahre(15)
```

und schauen wieder auf den Kilometerstand:

```
auto1.km_stand
```

```
15
```

Wenn wir nochmals 20 km fahren:

```
 auto1.fahre(20)
```

steigt der Kilometerstand entsprechend:

```
auto1.km_stand
```

```
35
```

Wir können beliebig viele Instanzen, d.h. Autos, von unsere Klasse `Auto` erstellen:

```
auto2 = Auto('BMW', 'schwarz')
```

Auch dieses neue Auto ist noch nicht gefahren:

```
auto2.km_stand
```

```
0
```

Wir machen eine längere Spritztour:

```
auto2.fahre(100)
```

und der Kilometerstand ist entsprechend angestiegen:

```
auto2.km_stand
```

```
100
```

Attribute lassen sich von außen verändern:

```
auto1.km_stand = 2000
auto1.km_stand
```

```
2000
```

Greifen wir von einer Instanz aus auf eine Methode zu erhalten wir eine gebundene Methode:

```
auto1.fahre
```

```
<bound method Auto.fahre of <__main__.Auto object at 0x1238dd1c0>>
```

Beim Zugriff über die Klasse ist die Methode ungebundenen und eine einfache Funktion:

```
Auto.fahre
```

```
<function __main__.Auto.fahre(self, km)>
```

Die beiden folgenden Wege `fahre` zu rufen haben den gleichen Effekt, nutzen aber eine gebunden bzw. eine ungebundene Methode (Funktion):

```
auto1.km_stand
```

```
2000
```

```
auto1.fahre(100)
auto1.km_stand
```

```
2100
```

```
Auto.fahre(auto1, 100)
auto1.km_stand
```

```
2200
```

Wir können auch Informationen innerhalb einer Klasse ablegen. Wollen wir zum Beispiel Farben für alle Instanzen vorgeben, können wir Klassenattribute nutzen. Wenn der Nutzer beim Anlegen einer neuen Instanz keine Farbe vorgibt, kommt einer der Vorgabewerte zum Einsatz:

```
import random

class EinfachesAuto:
    """Ein einfaches Auto mit Vorgabefarben.
    """

    vorgabe_farben = ['gelb', 'blau', 'orange']

    def __init__(self, marke, farbe=None):
        self.marke = marke
        if not farbe:
            self.farbe = random.choice(self.vorgabe_farben)
        else:
            self.farbe = farbe
        self.km_stand = 0

    def fahre(self, km):
        """Kilometerstand um `km` erhöhen."""
        self.km_stand += km
```

Wenn wir eine Instanz mit Farbvorgabe erstellen, ist alles wie bisher:

```
einfaches_auto1 = EinfachesAuto('VW', 'rot')
einfaches_auto1.farbe
```

```
'rot'
```

Lassen wir die Farbe weg, erhält unser Auto eine der drei Vorgabefarben:

```
einfaches_auto2 = EinfachesAuto('VW')
einfaches_auto2.farbe
```

```
'blau'
```

```
einfaches_auto3 = EinfachesAuto('VW')
einfaches_auto3.farbe
```

```
'orange'
```

13.2 Übungen

13.2.1 Übung 1

Schreiben Sie eine Klasse `Person`, die die Attribute `name` und `standort` hat.

13.2.2 Übung 2

Fügen Sie eine Methode `gehezu` hinzu, die einen neuen `standort` setzt.

13.2.3 Übung 3

Machen Sie mehrere Instanzen dieser Person und lassen Sie diese Personen zu verschiedenen Standorten gehen.

13.3 Vererbung

Von einer so genannten Basisklasse kann eine andere Klasse abgeleitet werden. Dieser Vorgang wird Vererbung genannt.

Wir erweitern unser Auto etwas, sodass auch die Entfernungen der einzelnen Fahrten aufgezeichnet werden:

```
class BetriebsAuto:
    """Ein Betriebs-Auto mit Fahrtenbuch.
    """

    def __init__(self, marke, farbe):
        self.marke = marke
        self.farbe = farbe
        self.km_stand = 0
        self.fahrten = []

    def fahre(self, km):
        """Kilometerstand um `km` erhöhen."""
        self.km_stand += km
        self.fahrten.append(km)
```

```
betriebs_auto1 = BetriebsAuto('VW', 'rot')
betriebs_auto1.fahrten
```

```
[]
```

```
betriebs_auto1.fahre(10)
betriebs_auto1.fahrten
```

```
[10]
```

```
betriebs_auto1.fahre(15)
betriebs_auto1.fahrten
```

```
[10, 15]
```

Wir definieren eine Klasse `LKW`, die von unserem `Auto` erbt:

```
class LKW(BetriebsAuto):
    pass
```

Der `LKW` verhält sich genau so wie das `BetriebsAuto`. Um das Ganze etwas interessanter zu machen, überschreiben wird die Methode `fahre`:

```
class LKW(BetriebsAuto):
    def fahre(self, km):
        """ LKW muessen immer hin und zurueck fahren, deshalb wird
            die km-Anzahl verdoppelt.
```

(continues on next page)

(continued from previous page)

```
        """
        self.km_stand += km * 2
        self.fahrten.append(km * 2)
```

Jetzt erhöht sich der Kilometerstand bei unserem LKW doppelt so schnell wie bei unserem Ausgangsauto:

```
lkw1 = LKW('MAN', 'grün')
lkw1.km_stand
```

```
0
```

```
lkw1.fahrten
```

```
[]
```

```
lkw1.fahre(10)
```

```
lkw1.km_stand
```

```
20
```

```
lkw1.fahrten
```

```
[20]
```

```
lkw1.fahre(15)
```

```
lkw1.km_stand
```

```
50
```

```
lkw1.fahrten
```

```
[20, 30]
```

Die beiden Zeilen in `fahre` in `LKW` sind fast identisch mit den den Zeilen in `fahre` in `BetriebsAuto`. Der einzige Unterschied sind die verdoppelten Kilometer.

In einem solchen in dem die Funktionalität der Eltern- bzw. Super-Klasse erhalten bleiben soll empfiehlt es sich sehr `super` zum Finden der Elternklasse zu nutzen. Besonders bei komplexeren Vererbungshierarchie ist dies meist besser als den Name der Elternklasse direkt zu verwenden.

```
class LKW2(BetriebsAuto):
    def fahre(self, km):
        """ LKW muessen immer hin und zurueck fahren, deshalb wird
            die km-Anzahl verdoppelt.
        """
        super().fahre(km * 2)
```

```
lkw2 = LKW2('Scania', 'blau')
```

```
lkw2.km_stand
```

```
0
```

```
lkw2.fahrten
```

```
[]
```

```
lkw2.fahre(10)
```

```
lkw2.km_stand
```

```
20
```

```
lkw2.fahrten
```

```
[20]
```

Die Suchreihefolge der Methoden (method resolution order) zeigt die Methode `mro()` an:

```
LKW2.mro()
```

```
[__main__.LKW2, __main__.BetriebsAuto, object]
```

13.3.1 Übungen

Übung 1

Schreiben Sie eine Klasse `EingeschraenktePerson`, die von `Person` erbt.

Übung 2

Überschreiben Sie in der Klasse `EingeschraenktePerson` die Methode `gehezu` und verbieten Sie der Person zu bestimmten Ort zu gehen. So können Sie z.B. auf dem Bildschirm `Der Zugang zu Ort xxx ist verboten.` ausgeben und behalten Sie den alten Ort bei, wenn der Zielort in der Liste Ihrer verbotenen Orte ist.

13.4 Operatorüberladung

Operatoren wie `+`, `-`, `in`, `abs`, `len` und viele andere können überladen werden, d.h. der Operator ruft eine vom Nutzer geschriebene Methode auf.

Wir überladen den Operator `+`:

```
class AddierbaresAuto:
    """Ein ganz einfaches Auto.
    """

    def __init__(self, marke, farbe):
        self.marke = marke
        self.farbe = farbe
        self.km_stand = 0

    def fahre(self, km):
        """Kilometerstand um `km` erhöhen."""
        self.km_stand += km

    def __add__(self, other):
        return AddierbaresAuto(self.marke + other.marke,
                               self.farbe + other.farbe)
```

Jetzt können wir zwei Autos addieren und bekommen als Ergebnis ein drittes, das nach der von uns in `__add__` angegebenen Vorschrift zusammengebaut wurde:

```
vw = AddierbaresAuto('VW', 'rot')
bmw = AddierbaresAuto('BMW', 'schwarz')
hybrid = vw + bmw
hybrid.marke
```

```
'VWBMW'
```

```
hybrid.farbe
```

```
'rotschwarz'
```

Die Rückgabe des neuen addierbaren Autos mit `return AddierbaresAuto` ist nicht verberbungssicher, da eine Instanz von `AddierbaresAuto` entsteht.

```
class AddierbaresAutoKind(AddierbaresAuto):
    pass
```

```
vw2 = AddierbaresAutoKind('VW', 'rot')
bmw2 = AddierbaresAutoKind('BMW', 'schwarz')
type(vw + bmw)
```

```
__main__.AddierbaresAuto
```

Es sollte aber eine Instanz der aktuellen Klasse sein. Mit `self.__class__` steht die aktuelle Klassen zur Verfügung. Diese Version funktioniert auch mit Vererbung:

```
class AddierbaresAuto2:
    """Ein ganz einfaches Auto.
    """

    def __init__(self, marke, farbe):
        self.marke = marke
        self.farbe = farbe
        self.km_stand = 0

    def fahre(self, km):
        """Kilometerstand um `km` erhöhen."""
        self.km_stand += km

    def __add__(self, other):
        return self.__class__(self.marke + other.marke,
                              self.farbe + other.farbe)
```

```
class AddierbaresAutoKind2(AddierbaresAuto2):
    pass
```

```
vw3 = AddierbaresAutoKind2('VW', 'rot')
bmw3 = AddierbaresAutoKind2('BMW', 'schwarz')
type(vw3 + bmw3)
```

```
__main__.AddierbaresAutoKind2
```

13.4.1 Spezielle Methoden

Es gibt viele spezielle Methoden, die das Überladen aller Syntax-Elemente von Python erlauben. In der Dokumentation[7] sind alle spezielle Methoden aufgelistet. Die folgende Tabelle fasst einige dieser Methoden beispielhaft zusammen.

Methode	Wirkung
`__add__`	Überladen des Operators `+`.
`__call__`	Instanz kann wie eine Funktion gerufen werden.
`__contains__`	Überladen von `in`. Z.B. `x in liste`.
`__getattr__`	Überladen des Zugriffs auf Attribute. Z.B. `a.b`.
`__init__`	Instanziierung. Überladen der `()` bei `a = A()`.
`__le__`	Überladen des Operators `<=`.
`__len__`	Überladen der eingebauten Funktion `len`.
`__lt__`	Überladen des Operators `<`.
`__str__`	Überladen eingebauten Funktion `str()`.
`__sub__`	Überladen des Operators `-`.

13.4.2 Übungen

Übung 1

Überladen Sie den Operator >> (Hinweis: Nutzen Sie die spezielle Methode `__rshift__`) mit der gleichen Funktionalität wie `gehezu`.

Das Ergebnis sollte so aussehen:

```
person = Person('Fritz', 'zu Hause')
person.standort
```

```
'zu Hause'
```

```
person >> 'hinter den sieben Bergen'
person.standort
```

```
'hinter den sieben Bergen'
```

[7] http://docs.python.org/reference/datamodel.html#special-method-names

14 Ausnahmen und Fehlerbehandlung

Bei der Programmierung treten auch bei sehr konzentrierter Arbeit Fehler auf. Es kommt darauf an mögliche Programmfehler abzufangen und zu behandeln. Python bietet hierfür Ausnahmen an.

Wenn ein Fehler auftritt, wird eine Ausnahme geworfen:

```
1 / 0
```

```
ZeroDivisionError: division by zero
```

In Python ist es möglich eine Aktion zu versuchen und bei einem Fehler entsprechend zu reagieren:

```
try:
    1 / 0
except ZeroDivisionError as err:
    print('Teilung durch Null ist nicht erlaubt.')
    print(f'Fehlermeldung: {err}')
```

```
Teilung durch Null ist nicht erlaubt.
Fehlermeldung: division by zero
```

Damit kann man die Stabilität eines Programm beträchtlich erhöhen.

Das Fehler-Objekt steht nun zur Inspektion zur Verfügung:

```
try:
    1 / 0
except ZeroDivisionError as e:
    err = e
```

```
err
```

```
ZeroDivisionError('division by zero')
```

```
isinstance(err, ZeroDivisionError)
```

```
True
```

```
isinstance(err, ArithmeticError)
```

```
True
```

```
isinstance(err, Exception)
```

```
True
```

```
ZeroDivisionError.mro()
```

```
[ZeroDivisionError, ArithmeticError, Exception, BaseException, object]
```

Wir sehen wie sich die Vererbungshierarchie hier auswirkt:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
```

Wir können einen Fehler auch explizit werfen (auslösen):

```
a = 1
b = 0
if b == 0:
    raise ZeroDivisionError('Teilung durch Null ist nicht erlaubt.')
```

```
ZeroDivisionError: Teilung durch Null ist nicht erlaubt.
```

Dieser kann dann an anderer Stelle im Programm wieder abgefangen werden.

Den letzten aufgetretenen Fehler haben wir noch als `err` und können ihn deshalb wieder werfen:

```
raise err
```

```
ZeroDivisionError: division by zero
```

Soll etwas unabhängig davon, ob eine Aktion erfolgreich war oder nicht ausgeführt werden, können wir dies mit `finally` erreichen.

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print('Teilung durch Null ist nicht erlaubt.')
    finally:
        print('wird immer ausgefuehrt')
```

Sowohl im negativen Fall:

```
divide(1, 0)
```

```
Teilung durch Null ist nicht erlaubt.
wird immer ausgefuehrt
```

als auch im positiven Fall:

```
divide(1, 10)
```

```
wird immer ausgefuehrt
```

```
0.1
```

wird der Code unter `finally` ausgeführt. Obwohl die Funktion nach `return` endet erfolgt die Ausgabe vom Text unter `finally` bevor `return` aktiv wird.

Wir können zwei oder mehr verschiedene Arten von Ausnahmen gleichzeitig fangen:

```
try:
    c = 1 / 0 + xyz
except (ZeroDivisionError, NameError) as err:
    print(err)
```

```
division by zero
```

Beim Abfangen mehrerer Ausnahme nacheinander:

```
try:
    c = 1 / 0
    xyz
except ZeroDivisionError:
    print('Teilung durch Null')
except NameError:
    print('Name nicht definiert')
```

```
Teilung durch Null
```

Dabei ist es wichtig die Vererbungshierarchie (siehe Python-Hilfe) zu beachten. Bei Ausnahmen, die in direkter Linie verwandt sind müssen die Kinder immer vor den Eltern angefangen werden, da im umgekehrten Fall das Abfangen der Eltern die Kinder mit nie erreicht werden können.

Wir können auch unsere eigene Ausnahme definieren:

```
class MyException(Exception):
    pass
```

und diese an einer beliebigen Stelle werfen:

```
value = 5
raise MyException(f'meine Nachricht mit Wert: {value}')
```

```
MyException: meine Nachricht mit Wert: 5
```

14.1 Übungen

14.1.1 Übung 1

Lösen Sie einen `AttributeError` aus, indem Sie ein nicht definiertes Attribute eines Objektes zugreifen. (Hinweis: `myobject.attribute`)

14.1.2 Übung 2

Fangen Sie diesen Fehler ab und geben Sie eine Nachricht aus, dass das Attribut nicht definiert ist.

14.1.3 Übung 3

Gben Sie Sie auf dem Bildschirm `Fertig!` aus, unabhängig davon ob der `AttributeError` auftritt oder nicht.

14.1.4 Übung 4

Legen Sie eine Liste mit fünf Elementen an. Greifen Sie auf das achte Element zu. Fangen Sie diesen Fehler ab und geben Sie eine entsprechende Nachricht auf dem Bildschirm aus.

14.1.5 Übung 5

Schreiben Sie eine Funktion, die als Parameter ein Dictionary, einen Schlüssel und einen Vorgabewert (default) hat. Implementieren Sie mit Hilfe von `try` und `except` mit dieser Funktion das Verhalten der Methode `get` eines Dictionarys.

So soll zum Beispiel dieses Verhalten:

```
my_dict = {'a': 100, 'b' : 200}
my_dict.get('a')
```

```
100
```

```
print(my_dict.get('x'))
```

```
None
```

```
my_dict.get('x', 999)
```

```
999
```

durch die Funktion `my_get` abgebildet werden:

```
my_get(my_dict, 'a')
```

```
100
```

```
print(my_get(my_dict, 'x'))
```

```
None
```

```
my_get(my_dict, 'x', 999)
```

```
999
```

14.1.6 Übung 6

Definieren Sie eine eigene Ausnahme `PositiveOnly`, die Sie nutzen wollen, um negative Zahlen zu verbieten. Testen Sie eine Zahl ob sie negativ ist und werfen Sie Ihre Ausnahme `PositiveOnly`, wenn dies zutrifft. Gegeben Sie dabei auf dem Bildschirm eine entsprechende Nachricht inklusive des Wertes der negativen Zahl aus.

15 Ein- und Ausgabe

Ein Computerprogramm verarbeitet Daten. Dazu benötigt es typischerweise Eingangsdaten und es erzeugt Ausgangsdaten. Diese Daten können über die Tastatur an der Kommandozeile eingeben werden, aus verschiedensten Arten von Dateien oder Datenbank-Systemen oder über eine grafische Oberfläche (GUI graphical user interface) empfangen werden. Auch die Ausgabe kann grundsätzlich über die gleichen Medien - also Bildschirmausgaben auf der Kommandozeile, Dateien, Datenbanken oder GUIs erfolgen. Hier wollen wir uns auf die ersten beiden Optionen beschränken.

15.1 Interaktive Eingabe

Um interaktiv vom Nutzer Daten abzufragen, `input()` iverwenden:

```
eingabe = input('Bitte geben Sie eine Zahl ein: ')
print('Ihre Eingabe war:', eingabe)
```

Um das Programm etwas interessanter zu gestalten, wiederholen wir die Eingabeaufforderung bis mit `q` das Programm beendet wird:

```
while True:
    eingabe = input('Bitte geben Sie eine Zahl ein: ')
    if eingabe == 'q':
        break
    print('Ihre Eingabe war:', eingabe)
```

Da `True` immer wahr ist erzeugt `while True` eine Endlosschleife, aus der man nur mit `break` herauskommt. Wenn die Eingabe `q` ist, wird `break` ausgeführt und das Programm ist damit beendet.

Eine Variation ohne `break`:

```
eingabe = ''
while eingabe != 'q':
    eingabe = input('Bitte geben Sie eine Zahl ein: ')
    print('Ihre Eingabe war:', eingabe)
```

Die `print`-Funktion fügt standardmäßig immer einen Zeilenumbruch am Ende ein. Mit einem Komma am Ende der Anweisungs-Zeile lässt sich das verhindern.

15.2 Kommandozeilenargumente

Eine zweite Möglichkeit Daten vom Nutzer zu bekommen sind Kommandozeilenargumente. Wir schreiben uns ein kleines Programm:

```
# datei argv.py

import sys

print('Kommandozeilenargumente:')
for arg in sys.argv:
    print(arg)
```

Um auf die Kommandozeilenargumente zugreifen zu können, müssen wir das Modul `sys` importieren. Dieses Modul gehört zur Standardbibliothek von Python und wird häufig benötigt. Wir bekommen die Argumente in der Liste `sys.argv` und schreiben mit Hilfe von `for .. in` eine Schleife über diese Liste, in der wir deren Inhalt auf dem Bildschirm ausgeben.

Nun rufen wir unser Programm von der Kommandozeile aus auf:

```
python argv.py 2 Hallo 100.4
```

```
Kommandozeilenargumente:
argv.py
2
Hallo
100.4
```

Das Programm gibt uns alle Argumente, jedes auf einer Zeile, zurück. Beachten Sie das Name des Programms selbst der erste Eintrag in der Liste ist.

Das Modul `argparse` bietet eine sehr leistungsfähige Bibliothek für die Programmierung von Kommando-Zeilen-Werkzeugen. Dieses Beispiel aus den Argparse Tutorial der Python-Dokumentation zeigt ein paar grundlegende Nutzungsmöglichkeiten. Das Programm berechnet die Potenz x^y:

```
#  file use_argparse.py

import argparse

#  input
parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true", help="long output")
group.add_argument("-q", "--quiet", action="store_true", help="short output")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()

#  computing
answer = args.x ** args.y

#  output
if args.quiet:
    print(answer)
elif args.verbose:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
else:
    print("{}^{} == {}".format(args.x, args.y, answer))
```

Die Hilfe ist über Kommandozeile zugänglich:

```
python use_argparse.py -h
```

```
usage: use_argparse.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  long output
  -q, --quiet    short output
```

Wir können jetzt 2^3 berechnen:

```
use_argparse.py 2 3
```

```
2^3 == 8
```

Das geht auch mit kurzer:

```
python use_argparse.py -q 2 3
```

```
8
```

oder langer Ausgabe:

```
python use_argparse.py -v 2 3
```

```
2 to the power 3 equals 8
```

Beide Optionen gleichzeitig sind nicht erlaubt:

```
python use_argparse.py -v -q 2 3
```

```
usage: use_argparse.py [-h] [-v | -q] x y
use_argparse.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

Basis und Exponent müssen Ganzzahlen sein:

```
python use_argparse.py 2.5 3
```

```
usage: use_argparse.py [-h] [-v | -q] x y
use_argparse.py: error: argument x: invalid int value: '2.5'
```

15.3 Dateien schreiben

Eine Datei kann in den Modi `r` für “read”, `w` für “write”, `a` für “append” und mit einem zusätzlichen `+` für das Anhängen an bestehende Dateien geöffnet werden. Binäre Dateien werden mit einem zusätzlichen `b` geöffnet. Dieser binäre Modus ist insbesondere für Windows-Betriebssysteme wichtig, weil Python die Zeilenenden von im Text-Modus geöffneten Dateien zu `\n` normalisiert und damit binäre Daten verändern könnte. Vorgabewert ist `r`, der wirksam wird, wenn nichts angeben wird.

```
# file write.py
```

(continues on next page)

(continued from previous page)

```
fobj = open('data_out.txt', 'w')
fobj.write('first line\n')
fobj.write('second line\nthird line\n')
fobj.close()
```

15.4 Die `with`-Anweisung

Die eingebaute Funktion `open` unterstützt die `with`-Anweisung. Öffnen wir eine Datei mit der Hilfe von `with`, wird diese garantiert geschlossen sobald wir den Kontext verlassen, d.h. wieder ausrücken:

```
with open('abc', 'w') as fobj:
    print(fobj.closed)
    print(fobj.closed)
print(fobj.closed)
```

```
False
False
True
```

Die beiden eingerückten Zeilen sind im Kontext. Im Kontext ist die Datei geöffnet. Außerhalb des Kontextes, also nach der Ausrückung, ist die Datei geschlossen. Das beutet `with open()` öffnet eine Datei mit dem Versprechen diese auch wieder zu schließen. Das Öffnen oben mit `with` ist konzeptionell äquivalent zu:

```
try:
    fobj = open('abc', 'w')
    print(fobj.closed)
    print(fobj.closed)
finally:
    fobj.close()
print(fobj.closed)
```

```
False
False
True
```

15.5 Dateien lesen

Für größere Mengen an Daten sind sowohl interaktiver Input als auch Kommandozeilenargumente nicht geeignet. Wir können Daten aber auch aus einer Datei lesen. Wir haben eine Beispielsdatei mit Daten in Spaltenform:

```
date         value1  value2
01.01.2010      100    50.1
01.01.2011      200    55.1
01.01.2012      300    60.1
01.01.2013      400    65.1
01.01.2014      500    70.1
01.01.2015      600    75.1
01.01.2016      700    80.1
01.01.2017      800    85.1
01.01.2018      900    90.1
01.01.2019     1000    95.1
01.01.2020     1100   100.1
```

Wir können die gesamte Datei als ein Objekt, einen String einlesen:

```
with open('data.txt') as fobj:
    data = fobj.read()

print(data)
```

```
date         value1  value2
01.01.2010      100    50.1
01.01.2011      200    55.1
01.01.2012      300    60.1
01.01.2013      400    65.1
01.01.2014      500    70.1
01.01.2015      600    75.1
01.01.2016      700    80.1
01.01.2017      800    85.1
01.01.2018      900    90.1
01.01.2019     1000    95.1
01.01.2020     1100   100.1
```

Mit `readlines` statt `read` erhalten wir eine Liste:

```
with open('data.txt') as fobj:
    lines = fobj.readlines()

lines
```

```
['date         value1  value2\n',
 '01.01.2010      100    50.1\n',
 '01.01.2011      200    55.1\n',
 '01.01.2012      300    60.1\n',
 '01.01.2013      400    65.1\n',
 '01.01.2014      500    70.1\n',
 '01.01.2015      600    75.1\n',
 '01.01.2016      700    80.1\n',
 '01.01.2017      800    85.1\n',
 '01.01.2018      900    90.1\n',
 '01.01.2019     1000    95.1\n',
 '01.01.2020     1100   100.1']
```

Ein Dateiobjekt ist ein Iterator. Wir können deshalb unser Wissen über Iteratoren anwenden und direkt über Dateien iterieren:

```
with open('daten.txt') as fobj:
    for line in fobj:
        print(line, end='')
```

```
Datum          Wert1   Wert2
01.01.2010      100    50.1
01.01.2011      200    55.1
01.01.2012      300    60.1
01.01.2013      400    65.1
01.01.2014      500    70.1
01.01.2015      600    75.1
01.01.2016      700    80.1
01.01.2017      800    85.1
01.01.2018      900    90.1
01.01.2019     1000    95.1
01.01.2020     1100   100.1
```

Wir können uns auch explizit einzelne Zeilen holen:

```
with open('daten.txt') as fobj:
    line1 = next(fobj)
    next(fobj)
    line3 = next(fobj)
```

```
line1
```

```
'Datum          Wert1   Wert2\n'
```

```
line3
```

```
'01.01.2011      200    55.1\n'
```

Der Vorteil gegenüber `readlines` besteht darin, dass immer nur eine Zeile aus der Datei in den Arbeitsspeicher geladen wird. Damit können beliebig große Dateien bearbeitet werden.

15.6 Methoden zum Lesen und Schreiben von Dateien

Es gibt einige Methoden zum Lesen, Schreiben und sich in der Datei bewegen. Die wichtigsten sind hier aufgelistet:

Attribut oder Methode	Beschreibung
`close()`	Datei schließen und speichern
`closed`	`True` wenn geschlossen, `False` wenn geöffnet
`flush()`	Datei speichern aber nicht schließen
`mode`	Modus, z.B. `'r'` für “read”
`next()`	nächste Zeile einer Textdatei
`read([bytes])`	Dateiinhalt als String einlesen (maximal bytes)
`readline()`	nächste Zeile als String einlesen
`readlines()`	gesamte Datei zeilenweise in Liste lesen
`seek(position)`	Cursor zu Position bewegen
`tell()`	aktuelle Cursor-Position
`write(str)`	String in Datei schreiben

15.7 Datenstrukturen einfach speichern

In Python gibt es ein Modul `pickle` (für sauer einlegen, also haltbar machen), mit dem Pythondatentypen ohne vorherige Umwandlung in Strings, wie bei Schreibe- und Lesevorgängen in Dateien nötig, gespeichert werden können. Mit den Funktionen `dump(object, file)` und `load(file)` können fast alle Pythonobjekte für den späteren Gebrauch in einer Datei gespeichert werden, ohne dass der Programmierer die Einzelheiten dieser Speicherung kennen muss.

```
import pickle
```

```
data = ['a', 'b' ,'c']
```

Datenstruktur in einer Datei ablegen:

```
with open('data.store', 'wb') as fobj:
    pickle.dump(data, fobj)
```

Datenstruktur wieder lesen:

```
with open('data.store', 'rb') as fobj:
    loaded_data = pickle.load(fobj)
```

```
loaded_data
```

```
['a', 'b', 'c']
```

```
# ```{warning}
```

Warnung

`pickle`-Dateien sind ausführbar
Mit `pickle` serialisierte Datenstrukturen sind ausführbar und müssen aus Sicherheitssicht wie Quelltext behandelt werden.

15.8 Übungen

15.8.1 Übung 1

Schreiben Sie ein Programm, dass Sie zur Eingabe Ihres Namens auffordert und diesen am Bildschirm wieder ausgibt.

15.8.2 Übung 2

Modifizieren Sie das Programm so, dass die Abfrage so lange wiederholt wird, bis mit dem Wort `end` die Ausführung des Programms beendet wird.

15.8.3 Übung 3

Erzeugen Sie mit Python eine neue Textdatei und schreiben Sie die Zahlen von 1 bis 10, eine Zahl pro Zeile, hinein.

15.8.4 Übung 4

Lesen Sie die gerade erzeugte Datei

(a) in einen String und

(b) in eine Liste.

15.8.5 Übung 5

Öffnen Sie die Datei so, dass sie nach Verlassen des Kontextes automatisch geschlossen wird.

15.8.6 Übung 6

Verwenden SIe das Modul `pickle` um eine Liste `[1, 2, 3]` in eine Datei zu speichern und lesen Sie diese wieder ein.

15.8.7 Übung 7

Fortgeschrittene Aufgabe: Nutzen Sie das Modul `argparse` aus der Standard-Bibliothek, um Kommandozeilenargumente um optionale Namen für eine Eingabedatei und eine Ausgabedatei zu verarbeiten. Das Programm sollte diese Ausgabe erzeugen:

```
python argparse_example.py --input myinput.txt --output myoutput.txt
Using input file: myinput.txt
Using output file: myoutput.txt
```

16 Die eigene Bibliothek - Beispiel: Rechnen mit Listen

Das einfachste Programm ist eine Python-Datei, also eine Datei mit der Endung `.py`. Programme wachsen und die Datei wird immer größer. Deshalb ist es sinnvoll den Quelltext eines Programms oder einer Bibliothek über mehre Datei zu verteilen.

Python bietet dafür das Konzept eines Paketes (eng. package). Ein Paket ist ein Verzeichnis, das Python-Quelltext-Dateien enthält. Die Anwesenheit einer Datei `__init__.py` macht ein Verzeichnis zu Paket.

Hinweis: In Python 3 funktioniert ein Paket auch ohne `__init__.py`. Es ist aber gut Praxis trotzdem eine leere `__init__.py` hinzuzufügen. Die `__init__.py` kann später Quelltext enthalten, der bestimmte Aufgaben erfüllen kann.

Eine Python-Quelltext-Datei, also eine Datei mit der Erweiterung `.py` ist ein Modul (eng. module). Diese Module sind importierbar. Damit lassen sich die darin enthaltenen Funktionen und Klassen importieren und nutzen.

16.1 Listen-Mathematik

In Python arbeiten mathematisch Operationen wie arithmetische oder trigonometrische Funktionen mit Skalaren. Diese Operationen arbeiten nicht direkt mit Listen. Eine Lösung kann die Anwendung von List-Comprehensions sein:

```
L = [1, 2, 3]
[x + y for x, y in zip(L, L)]
```

```
[2, 4, 6]
```

Die hier entwickelte Bibliothek löst diese Aufgabe. Diese Bibliothek:

- ist eine Beispiel-Implementierung und dient nur zu Demonstration wie Pakete funktionieren
- enthält sehr viel redundanten Code, der stark vereinfacht werden könnte
- bietet Funktionalität, die NumPy viel besser umsetzt
- ist deshalb nicht sonderlich nützlich
- ist inhaltlich wegen der hohen Redundanz schnell zu verstehen

16.2 Verzeichnis-Struktur

Die Verzeichnis-Struktur sieht so aus:

```
listmath/
├── __init__.py
├── cmath
│   ├── __init__.py
│   ├── arithmetics.py
│   └── trigonometrics.py
└── math
    ├── __init__.py
    ├── arithmetics.py
    └── trigonometrics.py
```

Die Funktionalität unserer Bibliothek `listmath` ist also über vier Module verteil, die wiederum in den beiden Unterverzeichnissen `math` und `cmath` liegen.

Der Inhalt der Datei `listmath/math/arithmetics.py` sieht so aus.

```
"""List arithmetics.
"""

__all__ = ['add', 'sub', 'mul', 'div']


def add(seq1, seq2):
    """Add two sequences element-wise.

    Returns a list.
    """
    return [x + y for x, y in zip(seq1, seq2)]


def sub(seq1, seq2):
    """Subtract two sequences element-wise.

    Returns a list.
    """
    return [x - y for x, y in zip(seq1, seq2)]


def mul(seq1, seq2):
    """Multiply two sequences element-wise.

    Returns a list.
    """
    return [x * y for x, y in zip(seq1, seq2)]


def div(seq1, seq2):
    """Divide two sequences element-wise.

    Returns a list.
    """
    return [x / y for x, y in zip(seq1, seq2)]
```

Die vier Funktionen sind alle nach dem gleichen Muster aufgebaut. Nur der Funktionsname und der zugehörige Operator +, −, * und * ändert sich zwischen den Funktionen. Deshalb wäre es relativ einfach und naheliegend nur eine Funktion zu nutzen und diese entsprechend zu parametrisieren. Es geht hier aber darum vier leicht verständliche Funktionen in einem Modul zu haben.

16.3 Import

Wenn das Verzeichnis in dem sich `listmath` befindet im aktuellen Arbeitsverzeichnis liegt, lassen sich unser Funktionen importieren:

```
import listmath.math.arithmetics
```

und mit unserer Liste:

```
L
```

```
[1, 2, 3]
```

nutzen:

```
listmath.math.arithmetics.add(L, L)
```

```
[2, 4, 6]
```

```
listmath.math.arithmetics.mul(L, L)
```

```
[1, 4, 9]
```

Dabei ist `listmath` der Name des Paketes und `math` der Name des Unter-Paketes. `arithmetics` ist der Name des Moduls und `add` der Name der Funktion.

Der Name `listmath.math.arithmetics.mul` ist sehr lang. Es gibt mehrere Möglichkeiten diesen Namen zu verkürzen. Mit `from <full_package_name> import function` lässt sich die Nutzbarkeit deutlich erhöhen:

```
from listmath.math.arithmetics import add
```

```
add(L, L)
```

```
[2, 4, 6]
```

Alternativ, funktioniert auch das Umbenennen mit `as`:

```
import listmath.math.arithmetics as arith
```

```
arith.add(L, L)
```

```
[2, 4, 6]
```

Beide Strategien funktionieren auch zusammen:

```
from listmath.math.arithmetics import add as list_add
```

```
list_add(L, L)
```

```
[2, 4, 6]
```

Das Umbenennen ist insbesondere nützlich wenn mehrere Bibliotheken unterschiedliche Funktionen mit dem gleichen Namen anbieten. Namen wie `read` oder `write` kommen sehr häufig vor.

Die Liste `__all__` enthält alle Namen, die Python mittels`from xxx import *` importiert. Der Name `sub` existiert noch nicht:

```
try:
    sub
except NameError as err:
    print(err)
```

```
name 'sub' is not defined
```

Nachdem `*`-Import:

```
from listmath.math.arithmetics import *
```

ist `sub` definiert:

```
sub
```

```
<function listmath.math.arithmetics.sub(seq1, seq2)>
```

Dieser Ansatz ist sehr problematisch. Wenn eine andere Bibliothek ebenfalls eine Funktion `sub` anbietet, die auch mit dem `*`-Import hinzugekommen ist, überschreibt die später importierte Funktion die gleichnamige vorher importierte. Das kann zu großer Verwirrung führen, da nicht direkt ersichtlich ist, das sich hier zwei Funktionen überschreiben. Deshalb sollten `*`-Importe in einem Programm, das eine Bibliothek nutz normaler Weise nicht vorkommen. Es gibt Anwendungsfälle für den `*`-Import bei der internen Gestaltung einer Bibliothek. Das ist aber ein fortgeschrittenes Thema.

Die Datei `listmath/math/trigonometrics.py` enthält vier trigonometrische Funktionen, die wiederum bis auf die Funktionsnamen und die gerufenen Funktionen `sin`, `cos` und `tan` identisch sind:

```
"""List trigonometrics.
"""

__all__ = ['sin', 'cos', 'tan', 'cot']

import math


def sin(seq):
    """Calculate sine element-wise.

    Returns a list.
    """
    return [math.sin(x) for x in seq]


def cos(seq):
    """Calculate cosine element-wise.

    Returns a list.
    """
    return [math.cos(x) for x in seq]


def tan(seq):
    """Calculate tangent element-wise.

    Returns a list.
    """
    return [math.tan(x) for x in seq]


def cot(seq):
    """Calculate cotangent element-wise.
```

(continues on next page)

(continued from previous page)

```
    Returns a list.
    """
    return [1 / math.tan(x) for x in seq]
```

Der Import ist analog:

```
from listmath.math.trigonometrics import sin
```

```
sin(L)
```

```
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672]
```

Alle diese Funktionen arbeiten mit den Datentypen `float` und `int`. Die trigonometrischen Funktionen funktionieren nicht mit dem Datentyp `complex`. Deshalb gibt es alle acht Funktionen aus dem Unter-Paket nochmals für Komplexe Zahlen. Für die arithmetische Operationen wäre das nicht unbedingt nötig, da die Funktionen aus `listmath.math.arithmetics` auch mit komplexen Zahlen funktionieren:

```
from listmath.math.arithmetics import add as math_add
```

```
math_add(L, L)
```

```
[2, 4, 6]
```

```
list_with_complex = [1, 2, 3 + 0j]
```

```
math_add(list_with_complex, list_with_complex)
```

```
[2, 4, (6+0j)]
```

Da `list_with_complex` Ganzzahlen und komplexe Zahlen enthält ist das Ergebnis auch entsprechend gemischt. Die Datei `listmath/cmath/arithmetics.py` bietet deshalb Funktionen, die immer komplexe Zahlen erzeugen:

```
"""Complex list arithmetics.

Results are always complex numbers.
"""

__all__ = ['add', 'sub', 'mul', 'div']

import listmath.math.arithmetics as arith


def add(seq1, seq2):
    """Add two sequences element-wise.

    Returns a list of complex numbers.
    """
    return [complex(x) for x in arith.add(seq1, seq2)]


def sub(seq1, seq2):
    """Subtract two sequences element-wise.

    Returns a list of complex numbers.
    """
    return [complex(x) for x in arith.sub(seq1, seq2)]
```

(continues on next page)

(continued from previous page)

```
def mul(seq1, seq2):
    """Multiply two sequences element-wise.

    Returns a list of complex numbers.
    """
    return [complex(x) for x in arith.mul(seq1, seq2)]


def div(seq1, seq2):
    """Divide two sequences element-wise.

    Returns a list of complex numbers.
    """
    return [complex(x) for x in arith.div(seq1, seq2)]
```

Alle Funktionen sehen wiederum **sehr** ähnlich aus. Die Funktionen nutzen die mit `import listmath.math.arithmetics as arith` importierten Funktionen aus `listmath/math/arithmetics.py`. Die Funktionen wandeln alle Ergebnisse in komplexe Zahlen um:

```
from listmath.cmath.arithmetics import add as list_complex_add
```

```
list_complex_add(L, L)
```

```
[(2+0j), (4+0j), (6+0j)]
```

```
list_complex_add(list_with_complex, list_with_complex)
```

```
[(2+0j), (4+0j), (6+0j)]
```

Die trigonometrischen Funktionen nutzen das Modul `cmath` aus der Standard-Bibliothek:

```
"""Complex list trigonometrics.
"""

import cmath


def sin(seq):
    """Calculate sine element-wise.

    Returns a list of complex numbers.
    """
    return [cmath.sin(x) for x in seq]


def cos(seq):
    """Calculate cosine element-wise.

    Returns a list of complex numbers.
    """
    return [cmath.cos(x) for x in seq]


def tan(seq):
    """Calculate tangent element-wise.

    Returns a list of complex numbers.
    """
    return [cmath.tan(x) for x in seq]
```

(continues on next page)

(continued from previous page)

```
def cot(seq):
    """Calculate cotangent element-wise.

    Returns a list of complex numbers.
    """
    return [1 / cmath.tan(x) for x in seq]
```

Nach dem Import:

```
from listmath.cmath.trigonometrics import sin as csin
```

```
csin(L)
```

```
[(0.8414709848078965+0j), (0.9092974268256817-0j), (0.1411200080598672-0j)]
```

```
csin(list_with_complex)
```

```
[(0.8414709848078965+0j), (0.9092974268256817-0j), (0.1411200080598672-0j)]
```

16.4 Übungen

16.4.1 Übung 1

Erstellen Sie im Datei-Explorer oder der Kommandozeile ein Verzeichnis und Dateien, die ein Python-Paket mit einem Modul mit einer Funktion enthält. Die Funktion soll `Hello world!` ausgeben. Aus dem Verzeichnis in dem sich Ihr Paket befindet soll dies funktionieren:

```
from mypackage.mymodule import hello

hello()
```

```
Hello world!
```

16.4.2 Übung 2

Erweitern Sie ihr Paket um ein Unter-Paket. Nach der Erweiterung soll dies möglich sein:

```
from mypackage.utils.helpers import show_help
```

```
show_help()
```

```
I would help you, if I could.
```

17 Module und Pakete

Das Beispiel `listmath` ist ein Paket. Dieses Kapitel stellt das Konzept der Module und Pakete nochmals systematisch dar.

17.1 Definition

Dateien mit Python-Quelltext werden Module genannt. Bei größeren Projekten empfiehlt es sich die Module hierarchisch anzuordnen. Hierzu werden einfach Verzeichnisse genutzt, die mit einer Datei namens `__init__.py` kenntlich gemacht werden. `listmath` ist ein solches Paket.

Module und Pakete können mit der `import`-Anweisung zugänglich gemacht werden. Es gibt mehrere Möglichkeiten:

1. Import mit dem vollen Namen: `import time` oder `import listmath.math.arithmetics`
2. Selektiver Import: `from listmath.math.trigonometrics import sin`
3. Umbennen während des Imports: `from listmath.math.arithmetics import add as math_add`
4. Direkter Import in den aktuellen Namensraum (nicht empfohlen): `from listmath.cmath.arithmetics import *`

17.2 Pakete finden

Module und Pakete müssen von Python gefunden werden. Es gibt mehrere Möglichkeiten:

1. Pakete im Verzeichnis `site-packages` finden Python standardmäßig. Alle Pakete die `pip` oder `conda` installieren liegen in diesen Verzeichnis.
2. Verwendung der Umgebungsvariable `PYTHONPATH`. Alle Pfade in dieser Liste werden nach Modulen und Paketen durchsucht.
3. Hinzufügen eines neuen Suchpfades im laufenden Programm.

```
import sys
sys.path.append('mein/pfad/fuer/pakete')
```

4. Nutzung einer Pfadkonfigurationsdatei. Es handelt sich dabei um eine Datei mit der Erweiterung `.pth`, die in PYTHONHOME, also dem Installationsverzeichnis von Python liegt. Eigene Pfaddateien sollten im Verzeichnis `site-packages` abgelegt werden.

Alle diese Pfade sind nach Programmstart in `sys.path` zu finden. Nach dem Import sind alle Module im Dictionary `sys.modules` verfügbar. Beim nächsten Import des selben Moduls nutz Python dieses Modul anstatt den Import-Prozess nochmals zu durchlaufen.

17.3 Übung

17.3.1 Übung 1

Kontrollieren Sie, ob das Paket `listmath` bereits im Suchpfad von Python ist. (Hinweis: Nutzen Sie `sys.path`.)

17.3.2 Übung 2

Importieren Sie ein Modul aus dem Paket `listmath` und zeigen Sie dessen Docstring an.

17.3.3 Übung 3

Nutzen Sie die Kommandos `dir` und `help`, um mehr über dieses Modul zu erfahren.

18 Objekte im Detail

Zwei unterschiedliche Objekte können gleichwertig sein:

```
li1 = [1, 2, 3]
li2 = [1, 2, 3]
li1 == li2
```

```
True
```

sie sind aber nicht identisch:

```
li1 is li2
```

```
False
```

Denn sie haben unterschiedliche IDs:

```
id(li1)
```

```
4903511296
```

```
id(li2)
```

```
4903511232
```

Die Zuweisung mit = bedeutet: “Nutze den Namen links vom = für das Objekt rechts vom =:

```
li3 = li2
li3
```

```
[1, 2, 3]
```

Die Veränderung von `li3`:

```
li3[1] = 100
li3
```

```
[1, 100, 3]
```

verändert auch `li2`:

```
li2
```

```
[1, 100, 3]
```

da es sich um die selbe Liste handelt:

```
li2 is li3
```

```
True
```

Die Nutzung des Namens `li3` für ein anderes Objekt:

```
li3 = 5
li3
```

```
5
```

hat keinen Einfluss auf `li2`:

```
li2
```

```
[1, 100, 3]
```

Literale Strings verhalten sich etwas anders:

```
s1 = 'abc'
s2 = 'abc'
```

```
s1 is s2
```

```
True
```

Das selbe String-Objekt ist über unterschiedlichen Namen, die mit zwei unabhängigen Zuweisungen erzeugt wurden, erreichbar. Diese Optimierung ist möglich weil Strings unveränderlich sind.

Die Ganzzahlen von -5 bis 256 (in CPython, -1000 bis 1000 in IronPython) referenziert Python:

```
i1 = 10
i2 = 10
i1 is i2
```

```
True
```

```
c = 2000
d = 2000
c is d
```

```
False
```

Da Ganzzahlen unveränderlich sind stellt dieses Verhalten bei der praktischen Anwendung kein Problem dar. Bei veränderlichen Objekten ist die Unterscheidung zwischen Gleichheit und Identität allerdings wichtig wie das Beispiel mit den Listen oben gezeigt hat.

In Python ist alles ein Objekt. Selbst Zahlen und Funktionen sind Objekte. Jedes Objekt hat einen Wert und einen Typ. Haben zwei Objekte den gleichen Wert, sind sie *gleich*, müssen aber nicht identisch sein. Die Identität kann mit der Standardfunktion `id()` geprüft werden. Ist das Ergebnis dieser Funktion, die physische Speicheradresse, gleich, sind beide Objekt identisch.

Der Typ eines Objektes lässt sich durch die Standardfunktion `type()` ermitteln:

```
a = 'abc'
type(a)
```

```
str
```

In Python ist alles ein Objekt und deshalb eine Instanz von etwas:

```
isinstance(a, str)
```

```
True
```

Damit testen wir nicht nur auf den aktuellen Typ sondern auch auf den der Eltern:

```
isinstance(True, int)
```

```
True
```

denn `bool` erbt von `int`:

```
bool.mro()
```

```
[bool, int, object]
```

und `True` ist eine spezielle Ganzzahl:

```
True + True
```

```
2
```

Die beiden eingebauten Funktionen `str` für die Umwandlung in eine Zeichekette (String) und `repr` für die Umwandlung in eine Repräsentation erzeugen oft die gleichen Ergebnisse:

```
str([1, 2, 3])
```

```
'[1, 2, 3]'
```

```
repr([1, 2, 3])
```

```
'[1, 2, 3]'
```

Es gibt aber auch manchmal geringfügige Unterschiede:

```
str('Hallo')
```

```
'Hallo'
```

```
repr('Hallo')
```

```
"'Hallo'"
```

Für eingebaute Datentypen gilt oft dass `eval(repr(obj))` das Objekt `obj` wieder erzeugt:

```
eval("'Hallo'")
```

```
'Hallo'
```

```
try:
    eval('Hallo')
except NameError as err:
    print(err)
```

```
name 'Hallo' is not defined
```

19 Namen für Objekte

In Python können Objekte anonym sein oder einen Namen tragen. Auf anonyme Objekte kann nach ihrer Verwendung nicht mehr zugreifen:

```
1 + 1
```

```
2
```

Die beiden Zahlen sind anonyme Objekte, die nicht mehr zugänglich sind, ohne wieder neue anonyme Objekte, z.B. durch Wiederholung der Eingabe zu schaffen. Diese Objekte werden als Literale bezeichnet. Dagegen können Objekte Namen bekommen, die oft auch Variablen genannt werden.

```
eins = 1
eins + eins
```

```
2
```

Namen werden aus Buchstaben und Zahlen gebildet, wobei ein Name nicht mit einer Zahl beginnen darf. Viele Sonderzeichen sind nicht erlaubt. Eine Ausnahme macht hier der Unterstrich _, der an beliebiger Stelle genutzt werden kann. Python unterstützt Unicode. Deshalb dürfen Namen auch Umlaute oder Buchstaben anderer Alphabete wie kyrillische, griechische oder chinesische Buchstaben enthalten. In der Praxis sollte möglichst Englisch als Sprache für die Namen zum Einsatz kommen. Damit kommen dann nur die 26 Buchstaben der englischen Alphabet in Frage.

Beispiele für gültige und ungültige Namen sind:

gültige Namen	ungültige Namen	Grund
`math`	`math$`	Sonderzeichen `$`
`math88`	`88math`	Zahl am Anfang
`Auto`	`Auto+`	Sonderzeichen +
`math_add`	`math add`	Leerzeichen

Bestimmte reservierte Wörter, die Schlüsselwörter (keywords), haben eine bestimmte Bedeutung und dürfen deshalb nicht für eigene Namen verwendet werden. Diese werden in einem Editor, der Pythonsyntax erkennt meist anders-farbig dargestellt. Eine Zuweisung zu einem Schlüsselwort führt zu einer Ausnahme:

```
for = 5
```

```
  File "<ipython-input-5-36df406aa65d>", line 1
    for = 5
        ^
SyntaxError: invalid syntax
```

Die Schlüsselwörter in Python 3.8 sind:

```
False      None     True       and        as         assert
async      await    break      class      continue   def
del        elif     else       except     finally    for
from       global   if         import     in         is
```

(continues on next page)

(continued from previous page)

```
lambda    nonlocal not        or        pass      raise
return    try      while      with      yield
```

Unterstriche oder mixedCase

Namen können auf unterschiedlichste Weise geschrieben werden. In Python werden Groß- und Kleinbuchstaben als unterschiedliche Zeichen interpretiert.Das *Python Enhancement Proposal* Nummer 8, der Style Guide for Python Code[8] enthält Empfehlungen, wie Python-Programme formatiert werden sollen. Dieses Dokument hat einen umfassenden Abschnitt über Namenskonventionen. Dieses Arbeitsbuch versucht diese Konventionen zu beachten. Namen für Funktionen, Methoden und Attribute sollen laut Empfehlung des PEP8 mit Kleinbuchstaben mit Unterstrichen als Trennzeichen zwischen Worten geschrieben werden. Also so: `add_numbers` und `loop_counter`. Diese Schreibweise wird gern als `snake_case` bezeichnet. Manchaml sieht man auch Namen mit gemischter Groß- und Kleinschreibung geschrieben. Also z.B. `addNumbers` und `loopCounter`. Es gibt einige ältere Python-Projekte, die auch diese gemischte Schreibweise nutzen. Abgesehen von diesem kleinen Unterschied in der Benennungspraxis, hat der meiste Open-Source-Quelltext in Python einen äußerst konsistenten Stil. Das macht das Lesen von Quelltext wesentlich einfacher. Wenn Sie sich der Verwendung der beiden Schreibweisen bewusst sind, ist die Namensgebung meist kein Problem.
Werkzeuge wie `pylint`, `pycodestyle` oder `flake8` überprüfen auch Namenskonventionen. Viele IDEs habe solche Werkzeuge eingebaut und machen den Programmierer sofort auf verbesserungswürdige Namen gemäß PEP8 aufmerksam.

[8] http://www.python.org/dev/peps/pep-0008/

19.1 Übungen

19.1.1 Übung 1

Wenden Sie PyLint auf die Module von `listmath` an.

19.1.2 Übung 2

Bauen Sie temporär absichtlich Fehler in eines der Module ein und prüfen Sie wiederholt mit PyLint.

19.1.3 Übung 3

Nutzen Sie PyLint für eigene Programme, zum Beispiel für die Lösungen anderer Übungen.

20 Namensräume und Gültigkeitsbereiche

20.1 Namensräume sauber halten

Die Namen von Objekten müssen eindeutig sein. Weißt man einem Namen eines Objektes noch einmal zu, ist das alte Objekt über diesen Namen nicht mehr zugänglich. Viele Namen wie `write`, `read` oder `run` werden häufig verwendet. Insbesondere wenn man Module importiert, würden so Namen überschrieben und welche Objekte über diese Namen zugänglich sind hängt von der Import-Reihenfolge ab. Um dies zu vermeiden, gibt es in Python Namensräume. Ein anderer Name für dieses Konzept ist “Scoping”. Gleiche Namen können sich auf unterschiedliche Objekte beziehen, wenn sie in unterschiedlichen Namensräumen leben.

Jedes Modul hat seinen eigenen Namensraum. Wenn ein Modul importiert wird, wird dieser Namensraum verfügbar:

```
import sys
```

und Objekte mit der Punktnotation verfügbar:

```
sys.byteorder
```

```
'little'
```

Der Name `byteorder` ist aber nicht ohne den Modul-Namen `sys` verfügbar:

```
try:
    byteorder
except NameError as err:
    print(err)
```

```
name 'byteorder' is not defined
```

Das sind die Namen im aktuellen, dem globalen Namensraum:

```
globals_before = len(globals())
globals_before
```

```
29
```

Wird aber die `*`-Form des Importes genutzt:

```
from sys import *  # So besser nicht!
```

ist der Name nun ohne Modul-Namen verfügbar:

```
byteorder
```

```
'little'
```

und es die Anzahl der der globalen Namen ist stark angestiegen:

```
len(globals()) - globals_before
```

```
77
```

Das stimmt ungefähr mit der Anzahl der “öffentlichen” Namen überein:

```
sum(1 for x in dir(sys) if not x.startswith('_'))
```

```
72
```

Die Abweichung erklärt sich aus dem Namen `globals_before`, der in `globals_before` noch nicht enthalten ist und einem anderem Effekt im Jupyter Notebook, das hier zum Einsatz kommt.

Am interaktiven Python-Prompt passen die Zahlen:

```
>>> import sys
>>> len(globals())
8
>>> from sys import *
>>> len(globals())
79
>>> sum(1 for x in dir(sys) if not x.startswith('_'))
71
>>>
```

Würde im aktuellen Namensraum schon ein Objekt mit dem Namen `byteorder` existieren, würde der Name `byteorder` aus dem Modul `sys` diesen überschreiben. Bei Import von mehreren Modulen kann es auch dazu kommen, dass sich die Namen der Module gegenseitig überschreiben. Welches Objekt dann tatsächlich zugänglich wäre hinge dann von der Reihenfolge der `import`-Anweisung ab. Dieser Effekt wird **Verschmutzung des Namensraumes (name space pollution)** genannt. Um saubere Namensräume zu erhalten sollte also ein qualifizierter Import mit Erhalt des Namensraumes erfolgen. Die `*`-Variante ist nur wenigen Sonderfällen vorbehalten, um z.B. in `__init__.py`-Dateien aus Unter-Pakten zu importieren.

20.2 Die LGB-Regel

Um sich den Namensraum eines Objektes anzusehen, kann man die eingebaute Funktion `dir()` nutzen:

```
dir(sys)[20:30]
```

```
['_home',
 '_xoptions',
 'abiflags',
 'addaudithook',
 'api_version',
 'argv',
 'audit',
 'base_exec_prefix',
 'base_prefix',
 'breakpointhook']
```

Das Modul `__builtins__` enthält alle eingebauten Objekte. Es wird auch oft als eingebauter Namensraum bezeichnet. In ihm befinden sich eine ganze Menge von Objekten, die immer zur Verfügung stehen:

```
dir(__builtins__)[:10]
```

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
```

(continues on next page)

(continued from previous page)

```
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError']
```

```
dir(__builtins__)[-10:]
```

```
['slice',
 'sorted',
 'staticmethod',
 'str',
 'sum',
 'super',
 'tuple',
 'type',
 'vars',
 'zip']
```

Es gibt neben dem eingebauten Namensraum noch den globalen und den lokalen Namensraum. Durch die Befehle `locals()` und `globals()` kann man sich den Inhalt dieser Namensräume ansehen.

Wird `locals()` innerhalb einer Funktion aufgerufen enthält das zurückgegebene Dictionary nur die innerhalb dieser Funktion verfügbaren Namen:

```
def func():
    a = 1
    print(locals())
    print(len(globals()))

func()
```

```
{'a': 1}
117
```

Auf dem Modul-Level stimmen beide Namensräume überein:

```
locals() is globals()
```

```
True
```

Die eingebaute Funktion `vars` gibt den Namensraum eines Objektes in einem Dictionary zurück. Die Schlüssel dieses Dictionarys entsprechen der Ausgabe von `dir`:

```
names_vars = set(vars(sys).keys())
names_dir = set(dir(sys))
```

```
names_dir == names_vars
```

```
True
```

Die Ebenen der Namensräume kann man so beschreiben:

Namensraum	Ebene
eingebaut	überall
global	auf Modulebene
lokal	in Klassen, Methoden und Funktionen

Die Suchreihenfolge eines nicht qualifizierten Namens ist folgender:

```
1. lokaler Namensraum
2. globaler Namensraum
3. eingebauter Namensraum
```

Wird der Namen gefunden, wird er verwendet, andernfalls wird im nächsten Namensraum gesucht. Bringt auch die Suche im eingebauten Namensraum keinen Erfolg wird eine Ausnahme `NameError` erzeugt. Diese Namensraumsuche heißt auch **LGB**-Regel.

20.3 Die LEGB-Regel

Wird innerhhalb einer Funkton oder Methode eine weitere Funktion definiert, so sind alle Variablen in dieser, als genestet bezeichenten Funktion, auch nur in dieser gültig. Dieser genestete Namensraum wird als *nested scope* bezeichnet. In unserm Beispielprogamm ist dieser Fall nicht aufgetreten. Mehr Informationen im dazugehörigen PEP[9].

Zum Beispiel ist die Funktion `inner` nur innerhalb der Funktion `outer` sichtbar:

```
def outer(outer_arg):
    def inner(inner_arg):
        return inner_arg + outer_arg
    return inner(5)
```

```
outer(10)
```

```
15
```

```
outer(20)
```

```
25
```

Die Funktion `inner` kann auf den Namensraum von `outer` in analoger Weise wie eine Funktion auf den globalen Namensraum zugreifen. Wenn eine Funktion innerhalb einer anderen Funktion oder Methode definiert ist, erweitert sich unsere Regel LGB (Local, Global, Built-in) zur **LEGB** (Local, Enclosing, Global, Built-in).

Die Suchreihenfolge eines nicht qualifizierten Namens dann ist folgender:

```
1. lokaler Namensraum (local)
2. umschließender Namensraum (enclosing)
3. globaler Namensraum (global)
4. eingebauter Namensraum (built-in)
```

20.4 Global und nonlocal

Normalerweise sollen Funktionen nur ihre lokalen Variablen verändern, wie in diesem Beispiel:

```
a = 5

def func():
    a = 6

func()

a
```

[9] http://www.python.org/dev/peps/pep-0227/

```
5
```

Der Wert von `a` hat sich durch den Funktionsaufruf nicht verändert, da `a` innerhalb der Funktion eine lokale Variable ist. Wir können aber auch eine globale Variable aus einer Funktion heraus verändern, wenn wir das explizit angeben:

```
def func():
    global a
    a = 6

func()

a
```

```
6
```

Obwohl so etwas möglich ist, sollten wir es nur nutzten wenn es wirklich nötig ist, sonst können schnell sehr schwer wartbare Programme entstehen. Es (fast) immer bessere Wege das Problem ohne `global` zu lösen.

Das Schlüsselwort `nonlocal` funktioniert analog zu `global` aber für eine verschachtelte Funktion und die sie jeweils umschließende Funktion.

```
def func():
    x = 5
    def inner():
        x = 6
    inner()
    return x
```

```
func()
```

```
5
```

Der Wert von `x` in der äußeren Funktion bleibt `5`. Das Setzen auf `6` in der verschachtelten Funktion erzeugt ein neues `x`, das nur innerhalb dieser Funktion gültig ist.

Wir können `nonlocal` analog zu `global` nutzen, um `x` aus der äußeren Funktion zu verändern:

```
def func2():
    x = 5
    def inner():
        nonlocal x
        x = 6
    inner()
    return x
```

```
func2()
```

```
6
```

So etwas kann durchaus nützlich sein und hat nur im begrenzten Bereich der Funktion einen Einfluss und nicht wie bei `global` auf das gesamte Modul.

20.5 Übungen

20.5.1 Übung 1

Bestimmen Sie wie viele Attribute das Modul `os` hat.

20.5.2 Übung 2

Aus welchen Namensräumen kommen `x` und `abs` wenn nur dieser Code einer Python-Quelltext-Datei steht?

```
x = 10
abs
```

21 Strings

21.1 String-Methoden

Python ist sehr gut für die Bearbeitung von Texten geeignet. Ein Stringobjekt hat eine Vielzahl von Methoden, die auf es angewendet werden können. Bedingt durch die Unveränderbarkeit von Strings wird ein neuer String zurückgegeben, der die Wirkung der Methode reflektiert. Wir schauen uns wichtige Methode, gruppiert nach Aufgabenschwerpunkten an.

21.1.1 Position ändern

Strings bieten mehrere Methoden, um einen neuen String zu erzeugen, der an einer bestimmen Stelle in einen (meist) längeren String steht. Wir starten mit einem Test-Strings:

```
s = 'Das ist ein Text!'
```

Wir können den String in der Mitte mit einer Breite von 40 Zeichen anordnen:

```
s.center(40)
```

```
'           Das ist ein Text!            '
```

Optional lässt sich ein Füllzeichen wie zum Beispiel # verwenden:

```
s.center(40, '#')
```

```
'###########Das ist ein Text!############'
```

In gleicher Weise ist eine Ausrichtung nach rechts:

```
s.rjust(40)
```

```
'                       Das ist ein Text!'
```

oder links:

```
s.ljust(40, '#')
```

```
'Das ist ein Text!########################'
```

mit oder ohne Füllzeichen ausrichten.

21.1.2 Groß- und Klein-Schreibung ändern

Wir können den String in Großbuchstaben:

```
s.upper()
```

```
'DAS IST EIN TEXT!'
```

oder in Kleinbuchstaben:

```
s.lower()
```

```
'das ist ein text!'
```

umwandeln. Wir können den String mit einem Großbustaben beginnen lassen und den Rest kleinschreiben:

```
s.capitalize()
```

```
'Das ist ein text!'
```

oder jedes Wort nach dieser Methode umwandeln:

```
s.title()
```

```
'Das Ist Ein Text!'
```

Weiterhin ist es möglich Klein- in Großbustaben und umgekehrt zu ändern:

```
s.swapcase()
```

```
'dAS IST EIN tEXT!'
```

Für den Vergleich von Strings ohne Berücksichtigung der Groß- und Klein-Schreibung mit besonderer Berücksichtigung alle Unicode-Strings gibt es eine besondere Methode, die die meisten Fälle wir `s.lower()` funktioniert:

```
s.casefold()
```

```
'das ist ein text!'
```

21.1.3 Anfang und Ende

Typische Anwendungsfälle ist der Test ob ein String mit einer bestimmen String-Sequenzen (Sub-String) beginnt:

```
s.startswith('Das')
```

```
True
```

oder endet:

```
s.endswith('!')
```

```
True
```

Der gesamte String passt natürlich auch:

```
s.startswith(s)
```

```
True
```

```
s.endswith(s)
```

```
True
```

21.1.4 Test von String-Arten

Es gibt zahlreiche `is_`-Methoden, um zu testen ob ein String nur eine bestimmte Art von Zeichen enthält:

```
s.isalnum()
```

```
False
```

```
'abc123'.isalnum()
```

```
True
```

```
'123'.isdecimal()
```

```
True
```

```
'valid_python_name'.isidentifier()
```

```
True
```

```
'invalid python_name'.isidentifier()
```

```
False
```

Es gibt ein Dutzend Methoden mit diesem Muster:

```
[name for name in dir(s) if name.startswith('is')]
```

```
['isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper']
```

21.1.5 Strings aus Sequenzen

Ein wichtiger Anwendungsfall ist Sequenzen, wie Listen oder Tupel, mit mehreren Strings zu einem String zusammenzufügen:

```
''.join(['A', 'B', 'C'])
```

```
'ABC'
```

Das geht auch mit einem Trenn-String:

```
'#:'.join(['A', 'B', 'C'])
```

```
'A#:B#:C'
```

21.1.6 Umhüllende Leerzeichen aus Strings entfernen

Beim Einlesen von Texten entstehen oft unerwünschte Zeichen wie Leerzeichen, Tabs und Zeilenumbrüche (white spacees) zu Beginn oder am Ende eines Strings. Es gibt spezielle Methoden für deren Entfernung. Mit diesem String:

```
s2 = '\t  Das ist ein Text!\n'
```

können wir die unerwünschten Zeichen von beiden Seiten:

```
s2.strip()
```

```
'Das ist ein Text!'
```

nur von links:

```
s2.lstrip()
```

```
'Das ist ein Text!\n'
```

oder nur von rechts:

```
s2.rstrip()
```

```
'\t  Das ist ein Text!'
```

entfernen.

21.1.7 Strings teilen

Ein weiter häufiger Anwendunsgfall ist das Teilen eines Strings an bestimmten Stellen. Häufig ist das Teilen an allen white-space-Zeichen:

```
s.split()
```

```
['Das', 'ist', 'ein', 'Text!']
```

oder an bestimmten Zeichen:

```
s.split('e')
```

```
['Das ist ', 'in T', 'xt!']
```

21.1.8 Weitere Methoden

Es gibt noch viele andere Methoden. Ersetzen:

```
s.replace('i', 'y')
```

```
'Das yst eyn Text!'
```

Mit Nullen auffüllen:

```
s.zfill(40)
```

```
'0000000000000000000000Das ist ein Text!'
```

Suchen nach Teil-Strings:

```
s.find('ein')
```

```
8
```

Im Gegensatz zu `s.index('nicht enthalten')` gibt es hier keine Ausnahme, sondern eine `-1`:

```
s.find('nicht enthalten')
```

```
-1
```

Reguläre Ausdrücke

Reguläre Ausdrücke sind ein sehr mächtiges Mittel um Text zu durchsuchen und zu bearbeiten. Wer sich näher für die Materie interessiert sei das Buch Reguläre Ausdrücke [FRIEDL2003] empfohlen. Hier wird auf 500 Seiten alles dargestellt, was man über dieses Gebiet wissen muss.
[FRIEDL2003] Friedl, Jeffrey E. F. Reguläre Ausdrücke, O'Reilly 2003.

21.2 Formatierung

21.3 Mit f-Strings

Python bietet sogenannte f-Strings mit vorangestelltem `f`. Sie suchen nach definierten Python-Objekt-Namen wie `hours` oder `name`, die innerhalb von `{}` stehen. Diese Methode gibt es seit Python 3.6.

Beginnen wir mit ein paar Objekten mit unterschiedlichen Typen:

```
name = 'Paul'
hours = 5
minutes = 10.567
print(f'Hello {name}!\nIt took {hours} hours and {minutes:5.2f}.')
```

```
Hello Paul!
It took 5 hours and 10.57.
```

Das ist äquivalent zur Wiederholung der Namen in `.format()`:

```
print('Hello {name}!\nIt took {hours} hours.'.format(name=name, hours=hours))
```

```
Hello Paul!
It took 5 hours.
```

`f`-Strings werten auch Ausdrücke aus:

```
L = list(range(2, 11))
print(f'erstes Element: {L[0]}, Summe: {sum(L)}')
```

```
erstes Element: 2, Summe: 54
```

Die Auswertung passiert zur “Compile”-Zeit, d.h. zur Import-Zeit für Objekte, die auf Modul-Ebene definiert sind

```
template = f'{not_defined=}'
```

```
NameError: name 'not_defined' is not defined
```

Da Python Funktions-Körper erst zum Aufrufzeitpunkt auswertet, lässt sich ein `f`-String ohne Kenntnis der Werte definieren:

```
def format_later(arg):
    return f'Got {arg=}'
```

Jetzt können wir unsere Funktion aufrufen und die Formatierung dynamisch ausführen:

```
format_later(10)
```

```
'Got arg=10'
```

Wir können eine Funktion schreiben, die unser Werte von oben formatiert:

```
def make_letter(name='Jane', hours=0, minutes=0):
    return f'Hello {name}!\nIt took {hours} hours and {minutes:5.2f}.'
```

Jetzt können wir die Funktion nutzen:

```
print(make_letter(name=name))
```

```
Hello Paul!
It took 0 hours and  0.00.
```

Wenn die Daten schon in einem Dictionary vorliegen:

```
data = {'name': name, 'hours': hours, 'minutes': minutes}
```

können wir das Entpacken von Schlüssel-Wort-Argumenten mit `**` nutzen:

```
print(make_letter(**data))
```

```
Hello Paul!
It took 5 hours and 10.57.
```

21.4 Wichtige Formatierungstypen

Typ	Bedeutung
`'d'`	"Decimal Integer." Ausgabe als Zahl mit Basis 10.
`'e'`	Wissenschaftliche Darstellung: z.B. `4.556000e+01`
`'f'`	Fixed-point notation: z.B. `45.56`
`'s'`	String: z.B. `text`
`'g'`	"General format." Schaltet automatisch zwischen `'f'` und `'e'` in Abhängigkeit von der Größe um

Die Python-Dokumentation enthält eine ausführliche Erläuterung zu allen Möglichkeiten dieser Format Specification Mini-Language[10].

21.4.1 Mit `.format()`

Es gibt eine zweite, leistungsfähigere Möglichkeit die "Format Specification Mini-Language". Das Ersetzen innerhalb des Strings funktioniert wie bei `f`-Strings. Die Werte müssen allerdings explizit innerhalb von `.format()` erscheinen. Dies Methode ist seit Python 2.6 verfügbar.

Wir können nach Position mit:

```
print('Hello {}!\nIt took {} hours and {} minutes.'.format('Paul', 5, 10.567))
```

```
Hello Paul!
It took 5 hours and 10.567 minutes.
```

oder mit Nummerierung:

```
print('Hello {0}!\nIt took {1} hours and {2} minutes.'.format('Paul', 5, 10.567))
```

```
Hello Paul!
It took 5 hours and 10.567 minutes.
```

mit Breitenangabe:

```
print('Hello {}!\nIt took {:6} hours and {} minutes.'.format('Paul', 5, 10.567))
```

```
Hello Paul!
It took      5 hours and 10.567 minutes.
```

mit Füllwert und dynamischer Anzahl von Kommastellen:

```
print('Hello {}!\nIt took {:{fill}6} hours and {:5.{deci}f} minutes.'.format(
    'Paul', 5, 10.567, fill=0, deci=1))
```

```
Hello Paul!
It took 000005 hours and  10.6 minutes.
```

nach Name:

```
print('Hello {name}!\nIt took {hours} hours and {minutes} minutes.'.format(
    name='Paul', hours=5, minutes=10.567))
```

```
Hello Paul!
It took 5 hours and 10.567 minutes.
```

oder auch mit einem Dictionary und dessen Auflösung mit `**`:

[10] https://docs.python.org/3/library/string.html#format-specification-mini-language

```
print('Hello {name}!\nIt took {hours} hours and {minutes} minutes.'.format(
        **{'name': 'Paul', 'hours': 5, 'minutes': 10.567}))
```

```
Hello Paul!
It took 5 hours and 10.567 minutes.
```

arbeiten.

Der Vorteil beim Dictionary liegt darin, dass wir die Reihenfolge nicht beachten müssen und Werte die wiederholt im Text auftauchen nur einmal im Dictionary sind.

21.4.2 Traditionell mit `%`

Der Formatierungsoperator `%` wird häufig eingesetzt, um gut lesbar formatierte Ausdrucke auf dem Bildschirm oder in Dateien zu erzeugen. Er kann in zwei Varianten, mit einem Tupel und der Ersetzung nach Position und als Dictionary und der Ersetzung nach Namen genutzt werden. Diese Methode existiert seit Python 1. Da sie so weit verbreitet ist, ist sie nicht abgekündigt (deprecated).

Mit einem Tupel:

```
print('Hello %s!\nIt took %d hours and %5.2f minutes.' %('Paul', 5, 10.567))
```

```
Hello Paul!
It took 5 hours and 10.57 minutes.
```

Oder mit einem Dictionary:

```
print('Hello %(name)s!\nIt took %(hours)d hoursand %(minutes)5.2f minutes.' % {
    'name':'Paul', 'hours': 5, 'minutes': 10.567})
```

```
Hello Paul!
It took 5 hoursand 10.57 minutes.
```

21.5 Übungen

21.5.1 Übung 1

Schreiben sie den Text `Das soll in der Mitte stehen` *ungefähr* in die Mitte einer Zeile des Terminals oder Notebooks und füllen Sie den Rest der Zeile mit Ausrufezeichen aus (`!!!!!!`).

21.5.2 Übung 2

Überprüfen Sie, ob verschiedene Strings nur aus Zahlen oder nur aus Buchstaben bestehen.

21.5.3 Übung 3

Schreiben Sie einen Satz und wandeln Sie ihn in eine Liste um, in der jedes Wort ein Element wird. Setzen Sie aus der Liste einen String zusammen, in dem die Worte durch Unterstriche verbunden sind.

21.5.4 Übung 4

Schreiben sie ein kleines Template-System für Serienbriefe. Fügen sie Empfängernamen und die Summe, die er ihnen schuldet automatisch ein. Die Summe sollte dreimal im Dokument erscheinen. Nutzen Sie `template.format()` mit Dictionary-Methode für die Ersetzung. Hinweise: Für Templates eigne sich ein mehrzeiliger String mit drei Anführungszeichen besonders.

Vorlage erstellen (könnte aus Datei gelesen werden):

```
template = """
Sehr gehrte{endung} {anrede} {name},

Sie schulden mir noch {betrag}.

Bitte überweisen Sie den Betrag von {betrag:5.2f} bis zum {datum:%d.%m.%Y}.

Viele Grüße
Der Eintreiber
"""
```

Daten erzeugen (könnte aus eine Datenbank kommen):

Briefe erzeugen:

```
for entry in data:
    print('#' * 80)
    print(template.format(**entry))
```

```
################################################################################

Sehr gehrte Frau Erika Mustermann,

Sie schulden mir noch 100.

Bitte überweisen Sie den Betrag von 100.00 bis zum 20.02.2022.

Viele Grüße
Der Eintreiber

################################################################################

Sehr gehrter Herr Max Mustermann,

Sie schulden mir noch 150.

Bitte überweisen Sie den Betrag von 150.00 bis zum 08.03.2022.

Viele Grüße
Der Eintreiber
```

22 Systemfunktionen

Python eignet sich gut, um Befehle an das Betriebssystem weiterzuleiten. Dies ist oft nötig, um ein Programm lauffähig zu machen. Pfade für das Lesen und Schreiben von Dateien müssen gefunden werden. Umgebungsvariablen müssen gelesen oder gesetzt werden. Informationen über die Zugriffsrechte auf Dateien werden benötigt. Die Systemzeit ist für viele Aufgaben nützlich.

22.1 Modul - `sys`

Python bietet betriebssystemunabhängigen Zugang zu den Systemfunktionen, so dass Programme ohne Änderung auf unterschiedlichen Betriebssystemen gleichermaßen funktionieren. Das Modul `sys` bietet einen Zugang zum Laufzeitsystem. Es funktioniert auf allen Plattformen gleich. Da nicht alle Betriebssysteme die gleiche Funktionalität bieten, kann es jedoch vorkommen, dass einige Optionen nur unter bestimmten Betriebssystemen möglich sind. Im Hilfesystem wird immer auf die Verfügbarkeit der entsprechenden Methoden oder Funktionen für die verschiedenen Betriebssysteme hingewiesen.

API-Version herausfinden:

```
import sys
```

```
sys.api_version
```

```
1013
```

Kommandozeilenargumente:

```
sys.argv
```

```
['script.py', 'arg1', 'arg2']
```

für den Aufruf:

```
python script.py arg1 arg2
```

Ist das Objekt eingebaut?:

```
sys.builtin_module_names
```

```
('_abc',
 '_ast',
 '_codecs',
 '_collections',
 '_functools',
 '_imp',
 '_io',
 '_locale',
 '_operator',
 '_signal',
```

(continues on next page)

(continued from previous page)

```
 '_sre',
 '_stat',
 '_string',
 '_symtable',
 '_thread',
 '_tracemalloc',
 '_warnings',
 '_weakref',
 'atexit',
 'builtins',
 'errno',
 'faulthandler',
 'gc',
 'itertools',
 'marshal',
 'posix',
 'pwd',
 'sys',
 'time',
 'xxsubtype')
```

Byteorder:

```
sys.byteorder
```

```
'little'
```

Das Attribut `maxsize` zeigt an wie groß der Standarddatentyp für Ganzzahlen des Systems ist:

```
sys.maxsize
```

```
9223372036854775807
```

Python's Suchpfad für Module:

```
sys.path
```

```
['/path/to/something'
 '/path/to/something/site-packages'
 '/path/to/something/else'
]
```

Auf welchem Betriebssystem befinden wir uns?:

```
sys.platform
```

je nach Plattform:

- `win32`
- `linux2`
- `darwin`

Wie sieht der interaktive Prompt aus?

Im Notebook:

```
sys.ps1
```

```
'In : '
```

```
sys.ps2
```

```
'...: '
```

Im interaktiven Modus auf der Kommandozeile:

```
sys.ps1
```

```
'>>> '
```

```
sys.ps2
```

```
'... '
```

Standard-Input:

```
sys.stdin
```

```
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>
```

Python-Version:

```
sys.version
```

```
'3.8.12 | packaged by conda-forge | (default, Oct 12 2021, 21:50:38) \n[Clang 11.1.
↪0 ]'
```

```
sys.version_info
```

```
sys.version_info(major=3, minor=8, micro=12, releaselevel='final', serial=0)
```

Aktuelle Ausnahmen:

```
sys.exc_info()
```

```
(None, None, None)
```

```
try:
    1 / 0
except ZeroDivisionError:
    print(*sys.exc_info(), sep='\n')
```

```
<class 'ZeroDivisionError'>
division by zero
<traceback object at 0x1572df5c0>
```

22.2 Modul - os

Python bietet auch vielfältige Möglichkeiten zur Systemadministration in seiner Standardbibliothek. Das Modul `os` ist eine Schnittstelle zum Betriebssystem. So gibt `os.getcwd()` das aktuelle Arbeitsverzeichnis zurück, mit `os.chmod(mypath, mode)` können die Zugriffsrechte einer Datei oder eines Verzeichnisses `myppath` in den Modus `mode` geändert werden und `os.listdir(path)` zeigt alle Dateien im Verzeichnis `mypath` an.

```
import os
```

In ein neues Verzeichnis wechseln:

```
os.chdir('path/to/new/dir')
```

Aktuelles Arbeitsverzeichnis herausfinden:

```
os.getcwd()
```

```
'path/to/new/dir'
```

Inhalt des gegebenen Verzeichnisses anzeigen:

```
os.listdir('path/to/my/dir')
```

Neues Verzeichnis anlegen:

```
os.mkdir('my/new/dir')
```

Mit relativen Pfaden:

```
os.mkdir('test')
os.chdir('test')
```

Eine Datei löschen:

```
os.remove('remove_me.txt')
```

22.3 Modul - `os.path`

Das Modul `os.path` bietet eine Reihe von Pfadmanipulationen an. So kann man z.B. mit `os.path.join(path1, path2, ...)` Pfade aus Bestandteilen entsprechend den Konventionen des benutzten Betriebssystems zusammensetzen. Für die Zerteilung von Pfaden bietet `os.path.split(path)` eine Betriebssystem übergreifende Möglichkeit.

Absoluten Pfad-Namen finden:

```
os.path.abspath('my_file')
```

```
'/full/path/to/my_file'
```

Datei-Namen herausfinden:

```
 os.path.basename('my/path/to/my/file.txt')
```

```
'file.txt'
```

Letzte Zugriffszeit (in Sekunden seit 1. Januar 1970):

```
os.path.getatime('test.txt')
```

```
1641792261.123451
```

Ist der gegebene Pfad eine Datei oder ein Verzeichnis?:

```
 os.path.isfile('test.txt')
```

```
True
```

```
os.path.isdir('test.txt')
```

```
False
```

Teile zu einem Pfad zusammenfügen.

Auf Unix:

```
os.path.join('path', 'to', 'my', 'file.txt')
```

```
'path/to/my/file.txt'
```

Auf Windows:

```
os.path.join('path', 'to', 'my', 'file.txt')
```

```
'path\\to\\my\\file.txt'
```

Letzten Teil des Pfades abspalten:

```
os.path.split('/path/to/my/file.txt')
```

```
('/path/to/my', 'file.txt')
```

Unter **Windows** können wir Pfade direkt aus einem Datei-Explorer kopieren, wenn wir so genannte "raw strings" mit dem vorgestellten `r` verwenden:

```
 os.path.split(r'c:\path\to\my\file.txt')
```

```
('c:\\path\\to\\my', 'file.txt')
```

Alternativ können wir den \ entwerten:

```
os.path.split('c:\\path\\to\\my\\file.txt')
```

```
('c:\\path\\to\\my', 'file.txt')
```

Wenn wir dies nicht tun, bekommen wir ggf. unerwünschte Ergebnisse wenn die Fluchtsequenzen `\n`, `\t`, oder `\f` im Namen haben:

```
print(os.path.split('c:\path\to\my\file.txt')[0])
```

```
'c:\\path    o'
```

Wenn wir Methoden aus den Modulen `os` und `os.path` kombinieren, können wir die Größe eines Verzeichnisses inklusive aller Unterverzeichnisse:

```
total_size = 0
for root, dirpath, file_names in os.walk('my_top_dir'):
    for name in file_names:
        total_size += os.path.getsize(os.path.join(dirpath, name))
```

22.4 Modul - `shutil`

Das Modul `shutil` bietet verschiedene Methoden um Dateien zu kopieren, zu löschen und zu verschieben.

Kopieren von Daten und Rechten (mode bits)(`cp src dst`):

```
shutil.copy(src, dst)
```

Kopieren von Daten und allen Metadaten (stat info) (`cp -p src dst`):

```
shutil.copy2(src, dst)
```

Kopieren der Daten von `src` nach `dst`:

```
shutil.copyfile(src, dst)
```

Kopieren der Daten von einem dateiartigen Objekt (z.B. geöffnete Datei) `fsrc` in ein zweites dateiartiges Objekt `fdst`:

```
copyfileobj(fsrc, fdst)
```

Verschieben von `src` nach `dst` (rekursiv für Verzeichnisse):

```
shutil.move(src, dst)
```

Rekursives Kopieren eines Verzeichnisbaums mit `copytree()`:

```
shutil.copytree(src, dst)
```

22.5 Mehr Informationen

Eine ausführliche Liste aller Methoden dieser Module ist in der Python-Hilfe File and Directory Access[11] IPython[12] ist ein verbesserter interaktiver Modus für Python, unter anderem auch den Zugang zum Betriebssystem stark vereinfacht.

22.6 Übungen

22.6.1 Übung 1

Bestimmen sie das aktuelle Arbeitsverzeichnis aus Python heraus.

22.6.2 Übung 2

Geben Sie die genutzte Python-Version aus.

[11] https://docs.python.org/3/library/filesys.html
[12] https://ipython.org/

22.6.3 Übung 3

Wechseln Sie in ein anderes Verzeichnis, das Dateien und Unterverzeichnisse enthält.

Listen die den Verzeichnisinhalt mit Hilfe des Moduls `os` auf.

22.6.4 Übung 4

Bauen Sie einen Pfad mit einer vom Betriebssystem unabhängigen Methode zusammen.

22.6.5 Übung 5

Finden Sie heraus, ob ein gegebener Pfad eine Datei oder ein Verzeichnis ist.

```
import os

os.path.isfile('sub/file1.txt')
```

```
True
```

22.6.6 Übung 6

Kopieren Sie eine Datei von einem Verzeichnis in an anderes.

22.6.7 Übung 7

Fortgeschritten: Listen Sie alle Dateien aller Unterverzeichnisse eines gegebenen Pfades auf, auf die innerhalb der letzten 24 Stunden zugegriffen wurde.