

Python

N. Kaelin, S. Walker

24. April 2019

Inhaltsverzeichnis

1	Datentypen	4
1.1	Numerische Datentypen	5
1.1.1	Arithmetische Operationen	5
1.1.2	Vergleichende Operatoren	5
1.1.3	Bitweise Operatoren für den Datentyp <code>int</code>	5
1.1.4	Methoden nur für den Datentyp <code>complex</code>	5
1.2	Sequentielle Datentypen	6
1.3	Assoziative Datentypen	6
1.4	Mengen	7
2	Verzweigungen	7
2.1	<code>if</code>	7
2.1.1	<code>if</code> -Anweisung mit <code>else</code> -Zweig	7
2.1.2	<code>elif</code> -Zweige	7
3	Schleifen	8
3.1	<code>while</code>	8
3.1.1	<code>continue</code>	8
3.1.2	<code>break</code>	8
3.1.3	<code>else</code> -Teil	8
3.2	<code>for</code>	8
4	Funktionen	8
4.1	Funktionsdefinition	8
4.2	Aufruf	9
4.3	Weiteres	9
4.3.1	Standardwert für Parameter	9
4.3.2	Mehrere Rückgabewerte	9
4.3.3	Variable Anzahl von Argumenten	9
4.3.4	Argumente entpacken	9
4.3.5	Beliebige Schlüsselwort-Parameter	9
4.3.6	Schlüsselwortparameter entpacken	9
4.3.7	Globale Variablen	10
4.3.8	Docstring - Funktion dokumentieren	10
4.3.9	Call-by-object-reference	10
5	Exceptions	10
5.1	Unspezifische Exceptions abfangen	11
5.2	Master Beispiel	11

6	Dateien	12
6.1	Datei öffnen	12
6.2	Dateien lesen und schreiben	12
6.2.1	with-Anweisung	12
6.2.2	glob	12
6.2.3	os.path	12
7	Strings	13
7.1	Stringformatierung	13
7.1.1	im C-Stil (à la printf)	13
7.1.2	mit format()	13
7.1.3	mit Stringlitterale	14
7.1.4	mit string-Methoden	14
7.2	Alles über Strings <small>Kap. 19</small>	14
7.2.1	Strings kombinieren	14
7.2.2	Strings aufspalten	14
7.2.3	Suchen von Teilstrings	14
7.2.4	Ersetzen von Teilstrings	14
7.2.5	Strings bereinigen	14
7.2.6	Klein- und Grossbuchstaben	15
7.2.7	Strings testen	15
8	Listen-Abstraktion/List-Comprehension	15
8.1	Neue Liste aus einer bestehenden Liste ableiten	15
8.2	Bestehende Liste filtern	15
8.3	Liste von Zahlen => formatierter String	15
8.4	Liste der Schachbrettfelder	16
8.5	Mengen-Abstraktion/Set Comprehension	16
8.5.1	Produkte zweier Zahlen	16
9	Iteratoren und Generatoren	16
9.1	Iteratoren	16
9.2	Generatoren	17
9.2.1	Generator-Expression	17
9.2.2	send()-Methode, Generator als Coroutine	17
10	Listen und Tupel im Detail	17
10.1	Tupel	17
10.2	Listen	18
10.2.1	Element hinzufügen	18
10.2.2	Elemente ersetzen	18
10.2.3	Mehrere Elemente hinzufügen	18
10.2.4	Element entfernen	18
10.3	Sortieren	19
10.3.1	Umgekehrte Reihenfolge	19
10.3.2	Mit spezieller Funktion	19
10.3.3	collections.deque	19
11	lambda, map, filter und reduce	19
11.1	lambda	19
11.2	map	20
11.3	filter	20
11.4	reduce	20

12 Reguläre Ausdrücke	20
12.1 Zeichen-Klassen	20
12.2 Wiederholungen (Quantoren)	21
12.3 Übereinstimmungen	21
12.3.1 match-Objekt	21
12.3.2 Übereinstimmungen finden	22
12.4 Modifizierungen	22
12.5 Gruppierung	23
12.5.1 Weitere Metazeichen	24
12.5.2 Look-around Assertions	25
13 Klassen	25
13.1 Einfache Klasse definieren	26
13.2 Klasse instanzieren	26
13.3 Klassen- und Instanz-Variablen	26
13.4 Methoden	27
13.4.1 __init__()-Methode	27
13.4.2 __del__()-Methode	28
13.4.3 Methoden aufrufen	28
13.4.4 Statische Methoden	28
13.4.5 Klassen-Methoden	29
13.5 Datenabstraktion	29
13.5.1 Public	29
13.5.2 Protected	29
13.5.3 Private	30
13.5.4 Setter- und Getter-Methoden	30
13.6 Magische Methoden	31
13.6.1 Grundmethoden	32
13.6.2 Numerische Datentypen emulieren	32
13.7 Klassen testen	32
13.8 Eigenes Modul importieren	33
13.8.1 Aus dem gleichen Verzeichnis	33
13.8.2 Aus einem andere Verzeichnis	34
14 Vererbung	34
14.1 Beispiel	35
14.2 public, protected und private	36
15 Mehrfachvererbung	36
15.0.1 MRO	37
16 NumPy	38
16.1 ndarray erzeugen	38
16.1.1 ndarray-Datentypen	39
16.2 Arithmetische Operationen	39
16.3 Indexierung	41
16.3.1 Slicing	41
16.4 Mathematische Funktionen	42
16.4.1 Lineare Algebra	42
16.4.2 Matplotlib	43

1 Datentypen

- Variablen bezeichnen keinen bestimmten Typ.
- Dynamische Typdeklaration
 - **Automatische Zuweisung** des Datentyps bei Deklaration
 - Datentyp ist während dem Programmablauf **veränderbar**
 - Wert- und Typänderung erlaubt!

Tabelle 1: Datentypen

Datentyp	Beschreibung	False-Wert
NoneType	Indikator für nichts, keinen Wert	None
Numerische Datentypen		
int	Ganze Zahlen	0
float	Gleitkommazahlen	0.0
bool	Boolesche Werte	False
complex	Komplexe Zahlen	0 + 0j
Sequenzielle Datentypen		
str	Zeichenketten oder Strings	''
list	Listen (veränderlich)	[]
tuple	Tupel (unveränderlich)	()
bytes	Sequenz von Bytes (unveränderlich)	b''
bytearray	Sequenz von Bytes (veränderlich)	bytearray(b'')
Assoziative Datentypen		
dict	Dictionary (Schlüssel-Wert-Paare)	{}
Mengen		
set	Menge mit einmalig vorkommenden Objekten	set()
frozenset	Wie set jedoch unveränderlich	frozenset()

- Python erkennt den Datentyp automatisch
- Python ordnet jeder Variablen den Datentyp zu
- Datentypen prüfen:
 - `type(object)`
 - `isinstance(object, ct)`
- Python achtet auf Typverletzungen
- Python kennt keine implizite Typumwandlung

1.1 Numerische Datentypen Kap. 4

• bool • int • float • complex

1.1.1 Arithmetische Operationen

Tabelle 2: Arithmetische Operationen

Operator	Beschreibung
$x + y$	Summe von x und y
$x - y$	Differenz von x und y
$x * y$	Produkt von x und y
x / y	Quotient von x und y
$x // y$	Ganzzahliger Quotient ¹ von x und y
$x \% y$	Rest der Division ¹ von x durch y
$+x$	Positives Vorzeichen
$-x$	Negatives Vorzeichen
$\text{abs}(x)$	Betrag von x
$x ** y$	Potenzieren, x^y

¹Nicht definiert für den Datentyp complex

Achtung: $x++$ und $x--$ gibt es **nicht**, aber $x += 1$, $x -= 1$, $x *= 2$, ...

1.1.2 Vergleichende Operatoren

Tabelle 3: Vergleichende Operatoren

Operator	Beschreibung
$==$	wahr, wenn x und y gleich sind
$!=$	wahr, wenn x und y verschieden sind
$<$	wahr, wenn x kleiner als y ist ²
$<=$	wahr, wenn x kleiner oder gleich y ist ²
$>$	wahr, wenn x grösser als y ist ²
$>=$	wahr, wenn x grösser oder gleich y ist ²

²Nicht definiert für den Datentyp complex

1.1.3 Bitweise Operatoren für den Datentypen int

1.1.4 Methoden nur für den Datentyp complex

Tabelle 4: Bitweise Operatoren

Operator	Beschreibung
$x \& y$	bitweises UND von x und y
$x y$	bitweises ODER von x und y
$x \wedge y$	bitweises EXOR von x und y
$\sim x$	bitweises Komplement von x
$x \ll n$	Bit-Verschiebung um n Stellen nach links
$x \gg n$	Bit-Verschiebung um n Stellen nach rechts

Tabelle 5: Methoden für complex

Methode	Beschreibung
$x.\text{real}$	Realteil von x als Gleitkommazahl
$x.\text{imag}$	Imaginärteil von x als Gleitkommazahl
$x.\text{conjugate}()$	Liefert die zu x konjugiert komplexe Zahl

1.2 Sequentielle Datentypen Kap. 5

• str • list • tuple • bytes • bytearray

Tabelle 6: Methoden für sequenzielle Datentypen

Operator	Beschreibung
<code>x in s</code>	Prüft, ob <code>x</code> in <code>s</code> enthalten ist.
<code>x not in s</code>	Prüft, ob <code>x</code> nicht in <code>s</code> enthalten ist.
<code>s + t</code>	Verkettung der beiden Sequenzen <code>s</code> und <code>t</code> .
<code>s * n</code>	Verkettung von <code>n</code> Kopien der Sequenz <code>s</code> .
<code>s[i]</code>	Liefert das <code>i</code> -te Element von <code>s</code> .
<code>s[i:j]</code>	Liefert den Ausschnitt aus <code>s</code> von <code>i</code> bis <code>j</code> .
<code>s[i:j:k]</code>	Liefert jedes <code>k</code> -te Element im Ausschnitt von <code>s</code> zwischen <code>i</code> und <code>j</code> .
<code>len(s)</code>	Liefert die Anzahl Elemente in der Sequenz <code>s</code> .
<code>max(s)</code>	Liefert das grösste Element in <code>s</code> (sofern eine Ordnung definiert ist).
<code>min(s)</code>	Liefert das kleinste Element in <code>s</code> (sofern eine Ordnung definiert ist).
<code>s.index(x)</code>	Liefert den Index des ersten Vorkommens von <code>x</code> in <code>s</code> .
<code>s.count(x)</code>	Zählt, wie oft <code>x</code> in <code>s</code> vorkommt.

1.3 Assoziative Datentypen Kap. 6

• dict

Tabelle 7: Methoden für Assoziative Datentypen

Operator	Beschreibung
<code>len(d)</code>	Liefert die Anzahl Schlüssel-Wert-Paare in <code>d</code>
<code>d[k]</code>	Zugriff auf den Wert mit dem Schlüssel <code>k</code>
<code>k in d</code>	Liefert <code>True</code> , wenn der Schlüssel <code>k</code> in <code>d</code> ist.
<code>k not in d</code>	Liefert <code>True</code> , wenn der Schlüssel <code>k</code> nicht in <code>d</code> ist.
<code>d.clear()</code>	Löscht alle Elemente aus dem Dictionary.
<code>d.copy()</code>	Erstellt eine Kopie des Dictionaries.
<code>d.get([k, [x]])</code>	Gibt den Wert des Schlüssels <code>k</code> zurück, ansonsten den Wert <code>[x]</code> .
<code>d.items()</code>	Gibt eine Liste der Schlüssel-Wert-Paare als Tuple zurück.
<code>d.keys()</code>	Gibt eine Liste aller Schlüsselwerte zurück.
<code>d.update(d2)</code>	Fügt ein Dictionary <code>d2</code> zu <code>d</code> hinzu.
<code>d.pop(k)</code>	Entfernt das Element mit Schlüssel <code>k</code> .
<code>d.popitem()</code>	Entfernt das zuletzt eingefügte Schlüssel-Wert-Paar.
<code>d.setdefault(k, [x])</code>	Setzt den Wert <code>[x]</code> für den Schlüssel <code>k</code> .

1.4 Mengen Kap. 7

- set
- frozenset

Ein set enthält eine ungeordnete Sammlung von einmaligen und unveränderlichen Elementen. In anderen Worten: Ein Element kann in einem set-Objekt nicht mehrmals vorkommen, was bei Listen und Tupel jedoch möglich ist.

Tabelle 8: Methoden für Mengen

Operator	Beschreibung
<code>s.add(el)</code>	Fügt ein neues unveränderliches Element (el) ein
<code>s.clear()</code>	Löscht alle Elemente einer Menge.
<code>s.copy()</code>	Erstellt eine Kopie der Menge.
<code>s.difference(y)</code>	Die Menge s wird von y subtrahiert und in einer neuen Menge gespeichert.
<code>s.difference_update(y)</code>	Gleich wie <code>s.difference(y)</code> nur wird hier das Ergebnis direkt in s gespeichert.
<code>s.discard(el)</code>	Das Element el wird aus der Menge s entfernt.
<code>s.remove(el)</code>	Gleich wie <code>s.discard(el)</code> nur gibt es hier einen Fehler falls el nicht in s.
<code>s.intersection(y)</code>	Liefert die Schnittmenge s und y.
<code>s.isdisjoint(y)</code>	Liefert True falls Schnittmenge von s und y leer ist.
<code>s.pop()</code>	Liefert ein beliebiges Element welches zugleich aus der Menge entfernt wird

2 Verzweigungen Kap. 9

2.1 if

listings/v2_if1.py

```
if Bedingung:
    Anweisung1 # Anweisungen 1 & 2 nur ausfuehren, wenn die Bedingung wahr ist
    Anweisung2
```

Achtung: Alle Anweisungen im gleichen Codeblock müssen gleich eingerückt sein, z.B. mit vier Leerzeichen, sonst wird ein Fehler ausgegeben.

2.1.1 if-Anweisung mit else-Zweig

listings/v2_if2.py

```
if Bedingung:
    Anweisung1 # Anweisung 1 & 2, falls Bedingung wahr
    Anweisung2
else:
    Anweisung3 # Anweisung 3 & 4, falls Bedingung unwahr
    Anweisung4
```

2.1.2 elif-Zweige

listings/v2_if3.py

```
if Bedingung1:
    Anweisung1
elif Bedingung2: # elif => else if
    Anweisung2
elif Bedingung3:
    Anweisung3
else:
    Anweisung4
```

Für jeden Datentyp gibt es einen Wert, der als **unwahr** gilt. Siehe Tabelle 1 auf der Seite 4.

Achtung: Python kennt keine switch-case-Anweisung.

3 Schleifen Kap. 10

3.1 while

listings/v2_while1.py

```
while Bedingung:
    Anweisung1 # Anweisung1 wird wiederholt, solange die Bedingung wahr ist
```

Achtung: Python kennt keine do-while-Schleife.

3.1.1 continue

listings/v2_while2.py

```
while Bedingung:
    Anweisung1
    if Ausnahme:
        continue # beendet den aktuellen
                  # Durchlauf und springt nach oben.
    Anweisung2
```

3.1.2 break

listings/v2_while3.py

```
while Bedingung:
    Anweisung1
    if Fehler:
        break # bricht die Schleife vorzeitig ab
```

3.1.3 else-Teil

listings/v2_while4.py

```
while Bedingung:
    Anweisung1
    if Fehler:
        break
else: # Wenn Schleife nicht durch break
      # abgebrochen wurde
    Anweisung2
```

3.2 for

listings/v2_for1.py

```
for Variable in Sequenz:
    Anweisung1
```

- dient zur Iteration einer Sequenz
- Sequenz muss ein iterierbares Objekt sein:
list, tuple, dict, str, bytes, bytearray, set, frozenset
- Die for-Schleife kennt auch continue und break somit gibt es auch einen else teil analog zur while-schleife.

4 Funktionen Kap. 14

listings/v2_func1.py

```
import time # time.time(), time.sleep()
import math # math.pi, math.cos(), math.log10()
import zipfile # ZIP-Dateien manipulieren
import socket # UDP-/TCP-Kommunikation
```

Standart Bibliotheken: <https://docs.python.org/3/library/>

und eingebaute Datentypen: <https://docs.python.org/3/library/stdtypes.html>

und eingebaute Funktionen: <https://docs.python.org/3/library/functions.html>

4.1 Funktionsdefinition

einfache Funktionsdefinition:

listings/v2_func2.py

```
def Funktionsname(Parameterliste):
    Anweisungen
```


Beispiel:

listings/v2_func3.py

```
def begruessung(vorname, nachname):
    print('Hallo', vorname, nachname)
```

listings/v2_func5.py

```
def summe(a, b):
    return a + b # return beendet die Funktion
                # mit Rueckgabewert a+b
```

- Der Funktionsname kann frei gewählt werden
- Parameternamen durch Kommas trennen
- Codeblock gleichmässig einrücken
- Der Rückgabewert der Funktion ist None, falls nichts angegeben wird.
- return-Anweisung beendet den Funktionsaufruf
- es sind mehrere return-Anweisungen sind erlaubt, wie in C/C++

4.2 Aufruf

listings/v2_func6.py

```
resultat1 = summe(2, 3)
resultat2 = summe(a=10, b=2) # Schlüsselwortparameter
resultat3 = summe(b=2, a=10) # Reihenfolge ist egal
resultat4 = summe(20, b=4)  # zuerst die namelosen
```

4.3 Weiteres

4.3.1 Standardwert für Parameter

listings/v2_func7.py

```
def rosen(farbe='rot'):
    print('Rosen sind' + farbe + '.')

rosen() # Ausgabe: 'Rosen sind rot.'
rosen('gelb') # Ausgabe: 'Rosen sind gelb.'
```

4.3.2 Mehrere Rückgabewerte

listings/v2_func8.py

```
def summe_und_differenz(a, b):
    # Rueckgabe Typ ist Tupel
    return (a + b, a - b)

# Tupel entpacken
summe, differenz = summe_und_differenz(5, 3)
```

4.3.3 Variable Anzahl von Argumenten

listings/v2_func9.py

```
def mittelwert(a, *args): # a ist zwingend
    print('a=', 1)
    print('args=', args) # rest im Tupel args
    a += sum(args)
    return a/len(args) + 1

mittelwert(2, 3, 7)
```

4.3.4 Argumente entpacken

listings/v2_func10.py

```
def distanz(x, y, z):
    print('x=', x)
    print('y=', y)
    print('z=', z)
    return (x**2 + y**2 + z**2)**0.5

position = (2, 3, 6)
distanz(*position) # Tupel entpacken
```

4.3.5 Beliebige Schlüsselwort-Parameter

listings/v2_func11.py

```
def einfache_funktion(x, **kwargs):
    print('x=', x)
    # die restlichen Argumente sind im Dictionary kwargs
    print('kwargs=', kwargs)

einfache_funktion(x='Hey', farbe='rot', durchmesser=10)
```

4.3.6 Schlüsselwortparameter entpacken

listings/v2_func12.py

```
punkt = {'x':1, 'y':2, 'z':2}
# Dictionary entpacken
distanz(**punkt)
```

4.3.7 Globale Variablen

listings/v2_func13.py

```
modul = 'Python'      # globale Variable

def anmeldung():
    # Variable modul ist Global
    print(modul)

anmeldung()           # Ausgabe: Python

def wechseln():
    # erstellt eine neue lokale Variable
    modul = 'C++'
    print('lokal:', modul)

wechseln()            # Ausgabe: lokal: C++
print('global:', modul) # Ausgabe: global: Python

def wirklich_wechseln():
    # referenzieren auf die globale Variable
    global modul
    modul = 'C++'
    print('lokal:', modul)

wirklich_wechseln()   # Ausgabe: lokal: C++
print('global:', modul) # Ausgabe: global: C++
```

4.3.8 Docstring - Funktion dokumentieren

PEP 257 - Docstring Conventions

<https://www.python.org/dev/peps/pep-0257>

listings/v2_func14.py

```
def meine_funktion(a, b):
    '''Gibt die Argumente a und b in umgekehrter
       Reihenfolge als Tupel zurueck.'''
    return(b, a)

meine_funktion.__doc__ # Ausgabe: 'Gibt die
                        Arguemnte ...'
help(meine_funktion)
```

4.3.9 Call-by-object-reference

mit veränderlichen Objekten:

listings/v2_func15.py

```
x = [1, 2, 3]
y = [7, 8, 9]

def foo(a, b):
    # Objekt veraendern
    a.append(4)
    # lokale Variable b referenziert neues Objekt
    b = [10, 11, 12]

foo(x, y)
print('x_=', x) # Ausgabe: x = [1, 2, 3, 4]
print('y_=', y) # Ausgabe: y = [7, 8, 9]
```

mit unveränderlichen Objekten:

listings/v2_func16.py

```
x = (1, 2, 3)
y = (7, 8, 9)

def foo(a, b):
    # Objekt veraendern ist nicht erlaubt
    # a.append(4)

    #lokale Variable b referenziert neues Objekt
    b = (10, 11, 12)

foo(x, y)
print('x_=', x) # Ausgabe: x = (1, 2, 3)
print('y_=', y) # Ausgabe: y = (7, 8, 9)
```

5 Exceptions Kap. 20

- Fehler (<https://docs.python.org/3/tutorial/errors.html>) können auftreten, z.B.:

listings/v3_exception1.py

```
int('bla')  => ValueError
5/0         => ZeroDivisionError
a[1000]     => IndexError
10 + 'Fr.'  => TypeError
```

und führen zu einem Abbruch des Programms

- Fehler können abgefangen werden:

listings/v3_exception2.py

```
try:
    x = int(input('Zahl eingeben: '))
```

```
except:
    print('Falsche_Eingabe!')
```

5.1 Unspezifische Exceptions abfangen

Nicht empfohlen, da auch Exceptions geschluckt werden, die weitergegeben werden sollten, z.B. KeyboardInterrupt.

listings/v3_exception3.py

```
eingabe = '10_Fr.'
try:
    x = int(eingabe)
except:
    print('Oops!_Irgendein_Fehler_ist_aufgetreten.')
```

5.2 Master Beispiel

listings/v3_exception9.py

```
eingabe = 5
try:
    if type(eingabe) is list:
        raise SyntaxError
    x = int(eingabe)
    y = 1/x
    if x > 100:
        raise ValueError('Wert_ist_zu_Gross!') #es wird ein Fehler generiert
    f = open('dat.txt')
except (ValueError, IOError) as e: # Mehrere Exception gleich behandeln
    # die Variable e enthaelt die Fehlermeldung
    print('Err:_ ' + str(e))
except ZeroDivisionError:
    print('Eingabe_darf_nicht_0_sein!')
else: # Wird ausgefuehrt wenn kein Fehler auftrat
    print('Alles_Okey')
    f.close()
finally: # Wird immer ausgefuehrt auch wenn das Programm unterbrochen wird
    print('Auf_wiedersehen')
print('Prog._laeuft_noch')
```

Eingabe	File	Ausgaben
5	existiert	Alles Okey Auf wiedersehen Prog. laeuft noch
0	existiert	Eingabe darf nicht 0 sein! Auf wiedersehen Prog. laeuft noch
200	existiert	Error: Wert ist zu Gross! Auf wiedersehen Prog. laeuft noch
'10.-'	existiert	Err: invalid literal for int() with base 10: '10.-' Auf wiedersehen Prog. laeuft noch
[5, 1]	existiert	Auf wiedersehen Lange Fehlermeldung
5	exist. nicht	Err: [Errno 2] No such file or directory: 'dat.txt' Auf wiedersehen Prog. laeuft noch

'Auf wiedersehen' wird immer ausgegeben, 'Prog. laeuft noch' wird dann ausgegeben wenn kein Fehler auftrat oder dieser abgefangen wurde.

6 Dateien Kap. 11

6.1 Datei öffnen

- Datei mit der `open()`-Funktion öffnen:

listings/v3_datei1.py

```
f = open('dokument.txt')           # lesen
f = open('dokument.txt', 'r')      # lesen
f = open('dokument.txt', 'w')      # schreiben
f = open('dokument.txt', 'a')      # anhaengen
f = open('dokument.txt', 'rb')     # binaer
f = open('dokument.txt', 'wb')     # binaer
```

- Weitere Parameter findet man in der Hilfe (<https://docs.python.org/3/library/functions.html#open>):

listings/v3_datei2.py

```
open(file, mode='r', buffering=None, encoding=None, errors=None,
      newline=None, closefd=True, opener=None)
```

6.2 Dateien lesen und schreiben

listings/v3_datei0.py

```
fr = open('dokument.txt', 'r')      # Datei zum lesen oeffnen
fw = open('dokument.txt', 'w')      # Datei zum schreiben oeffnen

inhalt = fr.read()                  # gesamte Datei lesen
inhalt = fr.read(n)                 # n Zeichen lesen
zeilen = fr.readlines()             # Liste aller Zeilen

fw.write('hello')                   # String schreiben
fw.writelines(['1', '2'])           # Liste von Strings

fr.close()                          # Dateien schliessen
fw.close()
```

6.2.1 with-Anweisung

Dateien sollten besser mit einer `with`-Anweisung geöffnet werden, dadurch wird sie am ende des Blocks automatisch geschlossen.

listings/v3_datei7.py

```
with open('mailaenderli.txt') as f:
    text = f.read()
print(text)
```

6.2.2 glob

listings/v3_datei10.py

```
import glob
# Gibt eine liste mit allen Dateinamen welche mit '.py' enden
print(glob.glob('*.py'))
```

6.2.3 os.path

listings/v3_datei11.py

```
import os
full_path = os.path.abspath('mailaenderli.txt')
print(full_path) # Ausgabe: kompletter Pfad der datei

os.path.isfile(full_path) # Return 'true' wenn full_path eine Datei ist
os.path.isdir(full_path) # Return 'true' wenn full_path eine Ordner ist
os.path.getsize(full_path) # Groesse der Datei/Ordner
os.path.split(full_path) # Teilt den Pfad in einen Tupel (Pfad, Dateiname)
os.path.splitext(full_path) # Pfad in einen Tupel (Pfad, Dateiname, Endung)
os.path.join('ordner', 'datei.txt') # Macht einen gueltigen Pfad (System abhaengig)
```

7 Strings

7.1 Stringformatierung Kap. 12

Stringformatierung benötigt man um Daten hübsch auszugeben oder systematisch abzuspeichern.

listings/v3_strings1.py

Menge	Name	Wert
3	R1	1.50k
7	R2	0.10k
2	R3	22.00k
5	R4	47.00k

listings/v3_strings2.py

Menge	Name	Wert
3	R1	1500
7	R2	100
2	R3	22000
5	R4	47000

7.1.1 im C-Stil (à la printf)

listings/v3_strings3.py

```
spannung = 12.56
strom = 0.5
N = 10
print('N=%d, U=%f, I=%.3f' % (N, spannung, strom))
# Ausgabe: N = 10, U = 12.560000, I = 0.500
print('U=%g' % spannung) # generelles Format
# Ausgabe: U = 12.56
print('X=%04X, Y=%04X' % (7, 15)) # hex
# Ausgabe: X = 0x0007, Y = 0x000F
```

7.1.2 mit format()

listings/v3_strings23.py

```
spannung = 12.56
strom = 0.5
'U={}, I={}'.format(spannung, strom)
# Ausgabe: 'U = 12.56, I = 0.5'

'U={0}, I={1}'.format(spannung, strom) # Mit Index
# Ausgabe: 'U = 12.56, I = 0.5'

'U={0:.2f}, I={0:.f}'.format(spannung) # Mit Index und Format:
# Ausgabe: 'U = 12.56, I = 12.560000'

'{:>8.2f}'.format(spannung) # Rechtsbündig
# Ausgabe: ' 12.56'
'{:<8.2f}'.format(spannung) # Linksbündig
# Ausgabe: '12.56 '
'{:^8.2f}'.format(spannung) # Zentriert
# Ausgabe: ' 12.56 '

'U={u}, I={i}'.format(u=spannung, i=strom) # Mit Schlüsselwortparameter:
# Ausgabe: 'U = 12.56, I = 0.5'

messung = {'spannung': 24, 'strom': 2.5} # Mit Dictionary
'U={spannung}, I={strom}'.format(**messung)
# Ausgabe: 'U = 24, I = 2.5'
```

7.1.3 mit Stringliterals

listings/v3_strings10.py

```
lokale_variable = 13
f'Wert={lokale_variable:.3f}'
# Ausgabe: 'Wert = 13.000'
```

7.1.4 mit string-Methoden

listings/v3_strings11.py

```
s = 'Python'
s.center(20, '=')
# Ausgabe: '====Python===='
s.ljust(20, '-')
# Ausgabe: 'Python-----'
s.rjust(20, '*')
# Ausgabe: '*****Python'
'1234'.zfill(20)
# Ausgabe: '00000000000000001234'
```

7.2 Alles über Strings Kap. 19

listings/v3_strings24.py

```
# Unicode-Nummer => Zeichen
chr(65)
# Ausgabe: ('A')

# Zeichen => Unicode-Nummer
ord('A')
# Ausgabe: (65)

# String => bytes
bin_data = 'A'.encode('utf-8')
print(bin_data)
# Ausgabe: b'A'
bin_data.decode('utf-8')
# Ausgabe: 'A'
```

7.2.1 Strings kombinieren

listings/v3_strings17.py

```
''.join(['a', 'b', 'c'])
# Ausgabe: 'abc'

', '.join(['a', 'b', 'c'])
# Ausgabe: 'a,b,c'
```

7.2.2 Strings aufspalten

• split()

listings/v3_strings15.py

```
'Python ist eine Schlange.'.split()
# Ausgabe: ['Python', 'ist', 'eine', 'Schlange.']

csv = '1;2000;30.3;44;505'
csv.split(';')
# Ausgabe: ['1', '2000', '30.3', '44', '505']

# max. zwei Trennungen von links her
csv.split(';', maxsplit=2)
# Ausgabe: ['1', '2000', '30.3;44;505']

# max. zwei Trennungen von rechts her
csv.rsplit(';', maxsplit=2)
# Ausgabe: ['1;2000;30.3', '44', '505']

'1;2;;;3;4'.split(';')
# Ausgabe: ['1', '2', '', '', '', '3', '4']
```

• splitlines()

listings/v3_strings16.py

```
csv = '''Dies ist
ein mehrzeiliger
Text.'''
csv.splitlines()
# Ausgabe: ['Dies ist', 'ein mehrzeiliger', 'Text.']
```

7.2.3 Suchen von Teilstrings

listings/v3_strings18.py

```
spruch = '''Wir sollten heute das tun, von dem
wir uns morgen wuenschen es gestern getan zu
haben.'''
'morgen' in spruch # Ausgabe: True
spruch.find('heute') # Ausgabe: 12
spruch.count('en') # Ausgabe: 4
```

7.2.4 Ersetzen von Teilstrings

listings/v3_strings19.py

```
spruch.replace('sollten', 'muessten')
# Ausgabe: 'Wir muessten heute das tun,\nvon dem
wir uns morgen wuenschen\nes gestern getan zu
haben.'
```

7.2.5 Strings bereinigen

listings/v3_strings20.py

```
s = '   Dieser String sollte saubere Enden haben. \n'
print(s) # Ausgabe:   Dieser String sollte saubere Enden haben.
s.strip() # Ausgabe: 'Dieser String sollte saubere Enden haben.'
'Eine Satz ohne Satzzeichen am Schluss?'.rstrip('!?')
# Ausgabe: 'Ein Satz ohne Satzzeichen am Schluss'
```

7.2.6 Klein- und Grossbuchstaben

listings/v3_strings21.py

```
'Passwort'.lower()
# Ausgabe: 'passwort'

'Passwort'.upper()
# Ausgabe: 'PASSWORT'
```

7.2.7 Strings testen

listings/v3_strings22.py

```
'255'.isdigit() # Ausgabe: True
'hallo'.isalpha() # Ausgabe: True
'Gleis7'.isalnum() # Ausgabe: True
'klein'.islower() # Ausgabe: True
'GROSS'.isupper() # Ausgabe: True
'Haus'.istitle() # Ausgabe: True
```

8 Listen-Abstraktion/List-Comprehension Kap. 31

- Einfache Methode, um Listen zu erzeugen
 - aus Strings, Dictionaries, Mengen, Bytes, ...
 - bestehende Listen abändern
 - bestehende Listen filtern
- Alles auf einer Zeile
 - übersichtlicher Code

8.1 Neue Liste aus einer bestehenden Liste ableiten

Beispiel 1:

konventionell:

listings/v4_list1.py

```
quadratzahlen = []
for n in range(11):
    quadratzahlen.append(n*n)
```

mit Listen-Abstraktion:

listings/v4_list2.py

```
quadratzahlen = [n*n for n in range(11)]
```

Beispiel 2:

konventionell:

listings/v4_list3.py

```
kilometer = [30, 50, 60, 80, 100, 120]
meilen = []
for km in kilometer:
    meilen.append(km*0.621371)
```

mit Listen-Abstraktion:

listings/v4_list4.py

```
kilometer = [30, 50, 60, 80, 100, 120]
meilen = [km*0.621371 for n in kilometer]
```

8.2 Bestehende Liste filtern

Beispiel: Nur Früchte behalten, deren Name mit A, B oder C beginnen.

listings/v4_list5.py

```
fruechte = ['Apfel', 'Erdbeer', 'Clementine', 'Kokosnuss', 'Birne', 'Himbeere']
fruechte_abc = [] # konventionell:
for frucht in fruechte:
    if frucht[0] in 'ABC':
        fruechte_abc.append(frucht)

fruechte_abc = [frucht for frucht in fruechte if frucht[0] in 'ABC'] # mit Listen-Abstraktion:
```

8.3 Liste von Zahlen => formatierter String

konventionell:

listings/v4_list6.py

```
temp = []
for km, mi in zip(kilometer, meilen):
    temp.append('{:.0f}km={:.0f}mi'.format(km, mi))
s = ', '.join(temp)
print(s) # Ausgabe: 30km=19mi, 50km=31mi, 60km=37mi, 80km=50mi, 100km=62mi, 120km=75mi
```

mit Listen-Abstraktion:

listings/v4_list7.py

```
s = ', '.join(['{:.0f}km={:.0f}mi'.format(km, mi) for km, mi in zip(kilometer, meilen)])
print(s) # Ausgabe: 30km=19mi, 50km=31mi, 60km=37mi, 80km=50mi, 100km=62mi, 120km=75mi
```

8.4 Liste der Schachbrettfelder**konventionell:**

listings/v4_list8.py

```
felder = []
for b in buchstaben:
    for z in zahlen:
        felder.append(b + str(z))
```

mit Listen-Abstraktion:

listings/v4_list9.py

```
felder = [b + str(z) for b in buchstaben for z in zahlen]
```

8.5 Mengen-Abstraktion/Set Comprehension**8.5.1 Produkte zweier Zahlen****konventionell:**

listings/v4_list10.py

```
menge = set()
for x in range(6):
    for y in range(6):
        menge.add(x*y)
```

mit Mengen-Abstraktion:

listings/v4_list11.py

```
menge = {x*y for x in range(6) for y in range(6)}
```

9 Iteratoren und Generatoren Kap. 32

- Iterator
 - greift nacheinander auf die Elemente einer Menge von Objekten zu
 - fundamentaler Bestandteil von Python, z.B. in for-Schleifen
- Generator
 - ist eine besondere Art, um einen Iterator zu implementieren
 - wird mittels einer speziellen Funktion erzeugt

9.1 Iteratoren

Iteratoren werden benutzt, um über einen Container zu iterieren.
Die for-Schleife erzeugt aus dem Listen-Objekt einen Iterator:

listings/v4_iter1.py

```
liste = [1, 2, 3]
for element in liste:
    print(element)
# Ausgabe: 1
#          2
#          3
```

Ein Iterator muss auch die `__next__()`-Funktion implementieren. Das nächste Element kann dann mit `next()` extrahiert werden.

Beispiel:

listings/v4_iter4.py

```
iterator = iter(liste) # Iterator aus Liste erzeugen
print(type(iterator)) # Ausgabe: <type 'listiterator'>

print('__iter__():', hasattr(iterator, '__iter__'))
print('__next__():', hasattr(iterator, '__next__'))
# Ausgabe: ('__iter__():', True)
#          ('__next__():', False)

next(iterator) # Ausgabe: 1
next(iterator) # Ausgabe: 2
next(iterator) # Ausgabe: 3 ... bis kein Element drin ist => StopIteration-Exception
```


9.2 Generatoren

Ein Generator ist auch ein Iterator.

Ein Generator wird erstellt, indem man eine Funktion aufruft, die eine oder mehrere yield-Anweisungen hat.

Bei der yield-Anweisung wird die Funktion (wie mit return) verlassen, aber Python merkt sich den Zustand der lokalen Variable und wo der Generator verlassen wurde.

listings/v4_iter7.py

```
def fibonacci_zahlen():
    a = 0
    b = 1
    while True:
        yield b
        a, b = b, a + b

print(type(fibonacci_zahlen))
# Ausgabe: <type 'function'>
f = fibonacci_zahlen()
print(type(f))
# Ausgabe: <type 'generator'>

for n in range(10):
    print(next(f)) # Ausgabe: die ersten 10 Werte
```

9.2.1 Generator-Expression

Ein Generator kann auch mit einem Ausdruck definiert werden:

listings/v4_iter9.py

```
# wie List Comprehension, aber mit runden Klammern
gen = (i*i for i in range(1, 10))
print(type(gen))
# Ausgabe: <type 'generator'>
```

9.2.2 send()-Methode, Generator als Coroutine

Die send()-Methode verhält sich im Prinzip wie die next()-Methode, aber sendet gleichzeitig noch einen Wert an den Generator:

listings/v4_iter10.py

```
def counter():
    n = 0
    while True: # next() setzt wert auf None, send(x) auf x
        wert = yield n
        if wert is not None:
            n = wert
        else:
            n += 1

c = counter()
next(c) # Ausgabe: 0
c.send(50) # Ausgabe: 50
next(c) # Ausgabe: 51
```

10 Listen und Tupel im Detail Kap. 16

- Tupel
 - Packing
 - Unpacking
- Listen
 - Elemente hinzufügen
 - Sortieren

10.1 Tupel

listings/v4_tupel1.py

```
t = () # Leeres Tupel:
print(type(t)) # Ausgabe: <type 'tuple'>

t = (5,) # Tupel mit einem Element
print(type(t)) # Ausgabe: <type 'tuple'>

x, y, z = 1, 2, 3 # Mehrfachzuweisung
print(x) # Ausgabe: 1
print(y) # Ausgabe: 2
print(z) # Ausgabe: 3
```

Packing:

listings/v4_tupel4.py

```
t = 'Peter', 'Mueller'
t
# Ausgabe: ('Peter', 'Mueller')
```

Unpacking:

listings/v4_tupel5.py

```
vorname, nachname = t
print(vorname) # Ausgabe: Peter
print(nachname) # Ausgabe: Mueller
```

Packing mit Rest:

listings/v4_tupel6.py

```
vorname, nachname, *adresse = ('Peter', 'Mueller', 'Oberseestrasse_10', 8640, 'Rapperswil')
print(vorname) # Ausgabe: Peter
print(nachname) # Ausgabe: Mueller
print(adresse) # Ausgabe: Oberseestrasse 10, 8640, Rapperswil
```

10.2 Listen

10.2.1 Element hinzufügen

listings/v4_tupel7.py

```
liste = ['a', 'b', 'c']
liste.append('X') # rechts
liste
# Ausgabe: ['a', 'b', 'c', 'X']
liste.insert(2, 'Y') # mit Index
liste
# Ausgabe: ['a', 'b', 'Y', 'c', 'X']
```

10.2.2 Elemente ersetzen

listings/v4_tupel10.py

```
liste = ['a', 'b', 'c', 'd', 'e', 'f']

liste[1] = 'B' # Einzelnes Element ersetzen
liste
# Ausgabe: ['a', 'B', 'c', 'd', 'e', 'f']

liste[3:] = ['D', 'E'] # Bereich ersetzen
liste # Ausgabe: ['a', 'B', 'c', 'D', 'E', 'f']
```

10.2.4 Element entfernen

listings/v4_tupel13.py

```
liste = ['a', 'b', 'c', 'd', 'e', 'f']

element = liste.pop() # letztes Element rechts
print(element) # Ausgabe: f
liste # Ausgabe: ['a', 'b', 'c', 'd', 'e']

element = liste.pop(0) # mit Index
print(element) # Ausgabe: a
liste # Ausgabe: ['b', 'c', 'd', 'e']

liste.remove('c') # mit einem bestimmten Wert
liste # Ausgabe: ['b', 'd', 'e']
```

10.2.3 Mehrere Elemente hinzufügen

listings/v4_tupel8.py

```
liste = ['a', 'b', 'c']
liste = liste + [1, 2] # zu vermeiden, sehr langsam
liste
# Ausgabe: ['a', 'b', 'c', 1, 2]
liste += [3, 4] # viel schneller
liste
# Ausgabe: ['a', 'b', 'c', 1, 2, 1, 2, 3, 4]
liste.extend([5, 6]) # noch schneller
liste
# Ausgabe: ['a', 'b', 'c', 1, 2, 1, 2, 3, 4, 3, 4, 5, 6]
```

Mehrere Elemente zwischendrin einfügen:

listings/v4_tupel9.py

```
liste[3:3] = ['#', '$']
liste
# Ausgabe: ['a', 'b', 'c', '#', '$', 1, 2, 1, 2, 3, 4, 3, 4, 5, 6]
```

10.3 Sortieren

`sorted()` liefert eine neue sortierte Liste zurück:

listings/v4_tupel14.py

```
liste = [2, 5, 3, 4, 1]
sortiert = sorted(liste)
print('Liste:', liste)
# Ausgabe: ('Liste:', [2, 5, 3, 4, 1])
print('sortiert:', sortiert)
# Ausgabe: ('sortiert:', [1, 2, 3, 4, 5])

t = (5,4,3)
sortiert = sorted(t)
print(sortiert)
# Ausgabe: [3, 4, 5]

s = 'python'
sortiert = sorted(s)
print(sortiert)
# Ausgabe: ['h', 'n', 'o', 'p', 't', 'y']
```

`sort()` modifiziert die Liste selbst (In-Place-Sortierung):

listings/v4_tupel15.py

```
liste.sort()
print(liste)
# Ausgabe: [1, 2, 3, 4, 5]
```

10.3.1 Umgekehrte Reihenfolge

listings/v4_tupel16.py

```
liste = [2, 5, 3, 4, 1]
sortiert = sorted(liste, reverse=True)
print('sortiert:', sortiert)
# Ausgabe: ('sortiert:', [5, 4, 3, 2, 1])

liste.sort(reverse=True)
liste
# Ausgabe: [5, 4, 3, 2, 1]
```

10.3.2 Mit spezieller Funktion

listings/v4_tupel17.py

```
liste = ['laenger', 'lang', 'am_laengsten']
sorted(liste, key=len)
# ['lang', 'laenger', 'am laengsten']

# nur [1]-tes Element (stabile Sortierung)
liste = [('a', 3), ('a', 2), ('c', 1), ('b', 1)]
from operator import itemgetter
sorted(liste, key=itemgetter(1))
# [('c', 1), ('b', 1), ('a', 2), ('a', 3)]
sorted(liste, key=lambda x: x[1])
# [('c', 1), ('b', 1), ('a', 2), ('a', 3)]

# Nach 1. Element, dann nach dem 2. sortieren
sorted(liste)
# [('a', 2), ('a', 3), ('b', 1), ('c', 1)]
```

10.3.3 collections.deque

Falls ein Stack oder FIFO-Buffer mit folgenden Eigenschaften benötigt wird:

- Thread-sicher
- Speicher-optimiert
- schnell

<https://docs.python.org/3/library/collections.html#collections.deque>

listings/v4_tupel18.py

```
from collections import deque
liste = deque([1, 2, 3])
print(liste)
# Ausgabe: deque([1, 2, 3])
liste.rotate(1)
print(liste)
# Ausgabe: deque([3, 1, 2])
endlich_lang = deque(maxlen=4)
for n in range(5):
    endlich_lang.append(n)
    print(list(endlich_lang))
# Ausgabe:
# [0]
# [0, 1]
# [0, 1, 2]
# [0, 1, 2, 3]
# [1, 2, 3, 4]
```

11 lambda, map, filter und reduce Kap. 30

- lambda
 - anonyme Funktionen bauen
- map, filter und reduce
 - Für die funktionale Programmierung
 - auch mit List Comprehension möglich

11.1 lambda

`lambda` definiert eine anonyme Funktionen.

listings/v4_tupel19.py

```
summe = lambda x,y: x + y
print(type(summe))
# Ausgabe: <type 'function'>
summe(2, 3)
# Ausgabe: 5
```

11.2 map

sequenz = map(funktion, sequenz)

Wendet die Funktion auf alle Elemente der Sequenz an und gibt die Resultate als Sequenz zurück.

listings/v4_tupel20.py

```
list(map(lambda x: x*x, [1, 2, 3]))
# Ausgabe: [1, 4, 9]
```

Funktion mit zwei Parametern benötigt zwei Listen:

listings/v4_tupel21.py

```
list(map(lambda x,y: x + y, [1, 2], [10, 20]))
# Ausgabe: [11, 22]
```

11.3 filter

sequenz = filter(funktion, sequenz)

Wendet die Funktion auf alle Elemente der Sequenz an und gibt nur diejenige Elemente zurück, für die die Funktion True liefert.

listings/v4_tupel22.py

```
list(filter(lambda x: True if x >= 0 else False, [5, -8, 3, -1])) # Ausgabe: [5, 3]
```

11.4 reduce

resultat = reduce(funktion, sequenz)

Wendet die Funktion (mit zwei Parametern) fortlaufen auf die Sequenz an und liefert einen einzelnen Wert zurück.

listings/v4_tupel23.py

```
from functools import reduce
reduce(lambda x, y: (x + y)/2, [10, 20, 30, 40]) # Ausgabe: 31
```

12 Reguläre Ausdrücke Kap. 36

- Regular Expressions (RE, regex, regex pattern)
- Bilden eine kleine Programmiersprache innerhalb von Python
- Sind verfügbar im re-Modul (<https://docs.python.org/3/library/re.html>) - import re
- Definieren Muster, auf die nur gewisse Strings passen, z.B.:
 - Entspricht die angegebene E-Mail-Adresse dem Muster?
 - Welche Wörter im Text beginnen mit "ver-" und enden mit "-en"?
- Die meisten Buchstaben und Zeichen passen auf sich selbst:
 - test passt genau auf sich selbst
- Folgende Metazeichen haben eine spezielle Bedeutung:
 - . ^ \$ * + ? { } [] \ | ()
 - . passt auf alle Zeichen, ausser Newline-Zeichen

12.1 Zeichen-Klassen

- Die Metazeichen [und] definieren eine Zeichen-Klasse
 - abc** passt auf alle Zeichen a, b oder c
 - a-z** passt auf einen Kleinbuchstaben
 - a-zA-Z** passt auf einen Klein- oder Grossbuchstaben
- Andere Metazeichen sind in Zeichen-Klasse nicht aktiv:
 - akm\$** passt auf die Zeichen a, k, m oder \$, wobei \$ sonst ein Metazeichen ist.
- Das ^-Zeichen definiert die komplementäre Menge:
 - ^abc** passt auf alle Zeichen, ausser a, b und c

- Vordefinierte Zeichen-Klassen:

<code>\d</code>	Dezimalziffer	<code>[0-9]</code>
<code>\D</code>	keine Dezimalziffer	<code>[^0-9]</code>
<code>\s</code>	Leer- oder Steuerzeichen	<code>[\t\n\r\f\v]</code>
<code>\S</code>	kein Leer- oder Steuerzeichen	<code>[^\t\n\r\f\v]</code>
<code>\w</code>	Unicode-Wortzeichen (auch Umlaute)	<code>[a-zA-Z0-9_]</code>
<code>\W</code>	kein Wortzeichen	<code>[^a-zA-Z0-9_]</code>
- Verwendung in Zeichen-Klassen:

<code>[A-Fa-f\d]</code>	passt auf eine Hexadezimalziffer
<code>[\s,.]</code>	passt auf ein Leerzeichen, Komma oder Punkt

12.2 Wiederholungen (Quantoren)

0 oder mehr	*	<code>ca*t</code> passt auf <code>ct</code> , <code>cat</code> , <code>caat</code> , ... <code>a[0-9]*b</code> passt auf <code>ab</code> , <code>a538b</code> , <code>a0b</code> , ...
1 oder mehr	+	<code>ca+t</code> passt nicht auf <code>ct</code> , aber <code>cat</code> , <code>caat</code> , ...
0 oder 1-mal	?	<code>10k?m</code> passt auf <code>10m</code> oder <code>10km</code>
m bis n-mal	<code>{m,n}</code>	<code>ab{2,3}c</code> passt auf <code>abbc</code> oder <code>abbbc</code>
	<code>{3}</code>	→ genau 3-mal
	<code>{3,}</code>	→ mindestens 3-mal

12.3 Übereinstimmungen

Funktionen, die Übereinstimmungen liefern:

<code>re.match()</code>	Prüft, ob die RA am Stringanfang passt. Return <code>None</code> oder <code>match</code> -Objekt.
<code>re.search()</code>	Sucht erstes Auftreten vom RA im String. Return <code>None</code> oder <code>match</code> -Objekt.
<code>re.findall()</code>	Findet alle Teilstrings, die mit dem RA passen. Return Liste mit allen Teilstrings.
<code>re.finditer()</code>	Findet alle Teilstrings, die mit dem RA passen. Return Iterator, welcher <code>match</code> -Objekte liefert.

12.3.1 match-Objekt

Memberfunktionen eines `match`-Objekts:

<code>group()</code>	Return Teilstring, welcher mit dem RA passt.
<code>start()</code>	Return Startposition des Teilstrings.
<code>end()</code>	Return Endposition des Teilstrings.
<code>span()</code>	Return Tupel mit (<code>start</code> , <code>end</code>).

12.3.2 Übereinstimmungen finden

`re.match(pattern, string, flags=0)`

listings/v5_ra2.py

```
m = re.match(r'[a-z]+', 'hallo_welt!')
print(m)
# <re.Match object; span=(0, 5), match='hallo'>

if m is not None:
    print('group:', m.group())    # group: hallo
    print('start:', m.start())    # start: 0
    print('end:', m.end())        # end: 5
    print('span:', m.span())      # span: (0, 5)
```

`re.search(pattern, string, flags=0)`

listings/v5_ra3.py

```
m = re.search(r'[a-z]+', '123_hallo_welt!')
print(m)
# <re.Match object; span=(4, 9), match='hallo'>

if m is not None:
    print('group:', m.group())    # group: hallo
    print('start:', m.start())    # start: 4
    print('end:', m.end())        # end: 9
    print('span:', m.span())      # span: (4, 9)
```

`re.findall(pattern, string, flags=0)`

listings/v5_ra4.py

```
liste = re.findall(r'[a-z]+', 'hallo_welt!')
print(liste)
# Ausgabe: ['hallo', 'welt']
```

`re.finditer(pattern, string, flags=0)`

listings/v5_ra5.py

```
for m in re.finditer(r'[a-z]+', 'hallo_welt!'):
    print('---')
    print('group:', m.group())
    print('start:', m.start())
    print('end:', m.end())
    print('span:', m.span())
# Ausgabe:
# ---
# group: hallo
# start: 0
# end: 5
# span: (0, 5)
# ---
# group: welt
# start: 6
# end: 10
# span: (6, 10)
```

12.4 Modifizierungen

Funktionen, die Modifizierungen durchführen:

`re.split()` Trennt den String dort, wo der RA passt.

Gibt eine Liste mit den Teilstrings zurück.

`re.sub()` Ersetzt jeden Teilstring, der mit dem RA passt.

Gibt den neuen String zurück.

`re.subn()` Gleich wie bei `re.sub()`,

gibt aber einen Tupel (Neuer String, Anzahl) zurück.

`re.split(pattern, string, maxsplit=0, flags=0)`

Der String wird überall dort getrennt, wo ein Teilstring auf den RA passt, z.B.: zwischen den Wörtern.

listings/v5_ra6.py

```
liste = re.split(r'\W+', 'Nun_dies_ist_ein_einfaches_Beiispiel.')
print(liste) # Ausgabe: ['Nun', 'dies', 'ist', 'ein', 'einfaches', 'Beispiel', '']
```

`re.sub(pattern, repl, string, count=0, flags=0)`

Jeder Teilstring, der auf den RA passt, wird mit dem repl-String ersetzt:

listings/v5_ra7.py

```
s = re.sub(r'\d+', '<Zahl>', '3_Stuecke_kosten_250_Franken.')
print(s) # Ausgabe: <Zahl> Stuecke kosten <Zahl> Franken.
```

Mit `count` kann die Anzahl Ersetzungen limitiert werden:

listings/v5_ra8.py

```
s = re.sub(r'\d+', '<Zahl>', '3_Stuecke_kosten_250_Franken.', count=1)
print(s) # Ausgabe: <Zahl> Stuecke kosten 250 Franken.
```

Eine Funktion bei `repl` angeben. Das Argument ist ein `match`-Objekt, der Rückgabewert muss ein String sein.

listings/v5_ra9.py

```
def func(m):
    return '(' + m.group() + ')'

s = re.sub(r'\d+', func, '3_Stuecke_kosten_250_Franken.')
print(s) # Ausgabe: (3) Stuecke kosten (250) Franken.
```

`re.subn(pattern, repl, string, count=0, flags=0)`

Gleich wie bei `re.sub()`, aber es wird ein Tupel mit dem neuen String und die Anzahl der Ersetzungen zurückgegeben:

listings/v5_ra10.py

```
resultat = re.subn(r'\d+', '<Zahl>', '3_Stuecke_kosten_250_Franken.')
print(resultat) # Ausgabe: ('<Zahl> Stuecke kosten <Zahl> Franken.', 2)
```

12.5 Gruppierung

- Teile eines Ausdrucks können gruppiert werden
- Normale Gruppierung mit `()`
 - `(ab)+c` passt auf `abc`, `ababc`, ...
 - `(ab)\1` mit Rückwärtsreferenz, passt auf `abab`
- Benannte Gruppierung mit `(?P<...>)`
 - `(?P<zahl>\d+)` passt auf `13`
 - `(?P<zahl>\d+)-(?P=zahl)` mit Referenz, passt auf `13-13`
- Passive Gruppierung (non-capturing group) mit `(?:...)`
 - `(?:ab)` passt auf `ab`, Gruppe wird nicht hinterlegt

`match`-Objekt

Mittels der `groups()`-Memberfunktion eines `match`-Objektes erhält man ein Tupel mit den Übereinstimmungen der einzelnen Gruppen.

Folgende Funktionen liefern ein `match`-Objekt: `re.match()`, `re.search()`, und `re.finditer()`.

listings/v5_ra11.py

```
m = re.search(r'(\d+)\s+([a-z]+)', '123_hallo_welt!')
if m is not None:
    print('groups():', m.groups()) # groups(): ('123', 'hallo')
    print('group(0):', m.group(0)) # group(0): 123 hallo
    print('group(1):', m.group(1)) # group(1): 123
    print('group(2):', m.group(2)) # group(2): hallo

# Mit benannten Gruppen:
m = re.search(r'(?P<zahl>\d+)\s+(?P<wort>\w+)', '123_hallo_welt!')
print(m.group('zahl')) # Ausgabe: 123
print(m.group('wort')) # Ausgabe: hallo
m.groupdict() # als Dictionary
# Ausgabe: {'zahl': '123', 'wort': 'hallo'}
```

`re.findall()`

Falls Gruppen im RA angegeben werden, dann werden nur die Übereinstimmungen der Gruppen als Liste von Tupeln zurückgegeben.

listings/v5_ra13.py

```
liste = re.findall(r'(\w+)=(\w+)', 'Jahrgang=1930,Name=Hans,Ort=Rappi')
print(liste) # Ausgabe: [('Jahrgang', '1930'), ('Name', 'Hans'), ('Ort', 'Rappi')]
liste = re.findall(r'Ort=(\w+)', 'Jahrgang=1930,Name=Hans,Ort=Rappi')
print(liste) # Ausgabe: ['Rappi']
liste = re.findall(r'(dum)\1', 'dumdum') # mit Rueckwaertsreferenz der Gruppe
print(liste) # Ausgabe: ['dum']
```

verschachtelte Gruppen, öffnende Klammern definieren die Reihenfolge

listings/v5_ra14.py

```
liste = re.findall(r'((dum)\2)', 'dumdum') # (dum) ist jetzt die zweite Gruppe
print(liste) # Ausgabe: [('dumdum', 'dum')]
```

`re.split()`

Falls Gruppen im RA angegeben werden, dann werden auch die Übereinstimmungen der Gruppen in der Liste zurückgegeben.

listings/v5_ra15.py

```
liste = re.split(r'(\W+)', 'Nun,dies_ist_ein_(simples)_Beispiel.')
print(liste) # Ausgabe: ['Nun', ' ', ' ', 'dies', ' ', ' ', 'ist', ' ', ' ', 'ein', ' (', 'simples', ') ', ' ', 'Beispiel', ' ', ' ', '']
```

`re.sub()`

listings/v5_ra16.py

```
s = re.sub(r'(\d+)/(\d+)/(\d+)', r'\2.\1.\3', '03/20/2019') # mit Gruppen-Referenzen
print(s) # Ausgabe: 20.03.2019
```

12.5.1 Weitere Metazeichen

Spezielle Prüfzeichen (belegen keinen Platz):

- | x|y passt entweder auf x oder y
- ^ steht für den Anfang des Strings
oder für den Anfang jeder Zeile (bei `flag=re.MULTILINE`)
- \$ steht für das Ende des Strings
oder für das Ende jeder Zeile (bei `flag=re.MULTILINE`)
- \A steht für den Anfang des Strings
- \Z steht für das Ende des Strings
- \b steht für eine Wortgrenze
- \B steht für das Gegenteil von \b

listings/v5_ra17.py

```
# Entweder...oder...
for m in re.finditer(r'\d+(V|A)', '230V_und_10A_bei_230hm'):
    print(m.group())
# Ausgabe:
# 230V
# 10A

# Anfang des Strings
re.findall(r'^\w+', 'Hallo_Welt') # Ausgabe: ['Hallo']
re.findall(r'^\w+', 'Erste_Zeile\nZweite_Zeile', flags=re.MULTILINE) # Ausgabe: ['Erste', 'Zweite']
re.findall(r'\A\d', '123456') # Ausgabe: ['1']

# Ende des Strings
re.findall(r'\w+$', 'Hallo_Welt') # Ausgabe: ['Welt']
re.findall(r'\w+$', 'Punkt_A\nPunkt_B', flags=re.MULTILINE) # Ausgabe: ['A', 'B']
re.findall(r'\d\Z', '123456') # Ausgabe: ['6']

# Wortgrenze
re.sub(r'\bschoen\b', 'herrlich', 'Das_Wetter_ist_schoen_oder_unschoen.')
# Ausgabe: 'Das Wetter ist herrlich oder unschoen.'
```


12.5.2 Look-around Assertions

- positive, vorausschauende Annahme
(?=Ausdruck) Ausdruck muss hier folgen
- negative, vorausschauende Annahme
(?!Ausdruck) Ausdruck darf hier nicht folgen
- positive, nach hinten schauende Annahme
(?<=Ausdruck) Ausdruck muss hier vorangehen
- negative, nach hinten schauende Annahme
(?<!=Ausdruck) Ausdruck darf hier nicht vorausgehen

listings/v5_ra21.py

```
# Positive, vorausschauende Annahme
re.findall(r'\w+(?=\.doc)', 'bericht.doc_dokument.doc') # Nach dem Wort muss ".doc" folgen
# Ausgabe: ['bericht', 'dokument']

# Negative, vorausschauende Annahme
re.findall(r'[A-Za-z]+(?!\d+)\b', 'abc123_cde') # Nach dem Wort darf keine Ziffer folgen
# Ausgabe: ['cde']

# Positive, nach hinten schauende Annahme
re.findall(r'(?<=#)\d+', '#10, #25, #66') # Vor den Ziffern muss ein #-Zeichen sein
# Ausgabe: ['10', '25']

# Negative, nach hinten schauende Annahme
re.findall(r'\b(?<!=#)\d+', '#10, #25, #66') # Vor den Ziffern darf kein #-Zeichen sein:
# Ausgabe: ['66']
```

Lektion 6: Klassen

13 Klassen

- Die Klassendefinition beginnt mit dem Schlüsselwort class

listings/v6_klassen1.py

```
class MeineKlasse:
    pass
```

- Eine Klasse mit Variablen und Methoden:

listings/v6_klassen2.py

```
class MeineKlasse:
    i = 0

    def __init__(self, name):
        self.name = name

    def gruss(self):
        print('Hallo', self.name)
```

13.1 Einfache Klasse definieren

listings/v6_klassen3.py

```
class MeineKlasse:
    '''Diese Klasse hat nicht viel drin.'''
    pass

MeineKlasse.__doc__
```

```
# Ausgabe: 'Diese Klasse hat nicht viel drin.'
help(MeineKlasse)
# Ausgabe:
# Help on class MeineKlasse in module __main__:
#
# class MeineKlasse(builtins.object)
# |   Diese Klasse hat nicht viel drin.
# |
# |   Data descriptors defined here:
# |
# |   __dict__
# |       dictionary for instance variables (if defined)
# |
# |   __weakref__
# |       list of weak references to the object (if defined)
```

13.2 Klasse instanzieren

listings/v6_klassen4.py

```
objekt = MeineKlasse()
```

13.3 Klassen- und Instanz-Variablen

listings/v6_klassen5.py

```
class MeineKlasse:
    speed_of_light = 299792458 # Klassen-Variable

    def __init__(self):
        self.name = 'unbekannt' # Instanz-Variable
```

Die Daten einer **Klassen-Variable** sind für alle Klassen-Objekte gleich.

listings/v6_klassen6.py

```
x = MeineKlasse()
y = MeineKlasse()
print('x:', x.speed_of_light) # Ausgabe: x: 299792458
print('y:', y.speed_of_light) # Ausgabe: y: 299792458
```

Die Daten einer **Instanz-Variable** sind für jedes Klassen-Objekt individuell.

listings/v6_klassen7.py

```
x.name = 'Hans'
y.name = 'Peter'
print(x.name) # Ausgabe: Hans
print(y.name) #Ausgabe: Peter
```

Achtung: bei gleichem Name haben die Instanz-Variablen Vorrang.

listings/v6_klassen8.py

```
x.speed_of_light = 10 # hier wird eine neue Instanz-Variable erzeugt
print('x:', x.speed_of_light) # Ausgabe: x: 10
print('y:', y.speed_of_light) # Ausgabe: y: 299792458
print('MeineKlasse:', MeineKlasse.speed_of_light) # Ausgabe: MeineKlasse: 299792458
```

13.4 Methoden

listings/v6_klassen9.py

```
class MeineKlasse:
    '''Beschreibung der Klasse.'''
    speed_of_light = 299792458
```

```

def __init__(self, name):
    '''Diese Methode initialisiert die Variablen.'''
    self.name = name
    print(self.name, 'wurde_erstellt.')
```

```

def __del__(self):
    '''Diese Methode räumt alles auf bevor es zerstört wird.'''
    print(self.name, 'wurde_zerstört.')
```

```

def hallo(self):
    '''Sagt Hallo.'''
    print('Hallo', self.name)
```

```

help(MeineKlasse)
# Ausgabe:
# Help on class MeineKlasse in module __main__:
#
# class MeineKlasse(builtins.object)
# |   MeineKlasse(name)
# |
# |   Beschreibung der Klasse.
# |
# |   Methods defined here:
# |
# |   __del__(self)
# |       Diese Methode räumt alles auf bevor es zerstört wird.
# |
# |   __init__(self, name)
# |       Diese Methode initialisiert die Variablen.
# |
# |   hallo(self)
# |       Sagt Hallo.
# |
# |   -----
# |   Data descriptors defined here:
# |
# |   __dict__
# |       dictionary for instance variables (if defined)
# |
# |   __weakref__
# |       list of weak references to the object (if defined)
# |
# |   -----
# |   Data and other attributes defined here:
# |
# |   speed_of_light = 299792458
```

Unterschiede zwischen Methoden und einer gewöhnlichen Funktion:

- eine Methode wird innerhalb eines **class**-Blocks definiert.
- der erste Parameter (**self**) einer Methode ist immer eine Referenz auf die Instanz, von der sie aufgerufen wird.

Hinweise:

- Eine Variable, die mit "self." innerhalb einer Methode erstellt wird, ist automatisch eine Instanz-Variable.
- Eine Variable, z.B. `speed_of_light`, die ausserhalb einer Methode erstellt wird, ist automatisch eine Klassen-Variable.

13.4.1 __init__()-Methode

Sie dient zur Initialisierung der Instanz. Sie wird unmittelbar nach dem Konstruktor aufgerufen.

listings/v6_klassen10.py

```

s = MeineKlasse('Wall-E') # name='Wall-E'
# Ausgabe: Wall-E wurde erstellt.
```

Dringend empfohlen: alle Instanz-Variablen in der `__init__()`-Methode initialisieren.

13.4.2 `__del__()`-Methode

Sie wird aufgerufen, bevor die Instanz zerstört wird.

listings/v6_klassen11.py

```
del s # loescht nur die Referenz s auf das Objekt.  
# Ausgabe: Wall-E wurde zerstört.
```

Hinweis: Das Objekt selber wird vom Garbage Collector entfernt, sobald keine Referenzen mehr darauf zeigen.

13.4.3 Methoden aufrufen

Der **self**-Parameter wird beim Aufruf nicht angegeben.

listings/v6_klassen12.py

```
s = MeineKlasse('Wall-E') # name='Wall-E'  
s.hallo()  
# Ausgabe:  
# Wall-E wurde erstellt.  
# Hallo Wall-E
```

Python bindet alle Methoden automatisch an die Instanz.

listings/v6_klassen13.py

```
print(s.hallo) # Ausgabe: <bound method MeineKlasse.hallo of <__main__.MeineKlasse object at 0  
x00000010B581D24A8>>
```

Grundsätzlich entspricht dies dem folgenden Aufruf:

listings/v6_klassen14.py

```
MeineKlasse.hallo(self=s) # Ausgabe: Hallo Wall-E
```

13.4.4 Statische Methoden

Sie sind nicht an eine Instanz gebunden, d.h. sie benötigen keinen **self**-Parameter.

Variante 1:

listings/v6_klassen15.py

```
def quadrieren(x):  
    return x*x  
  
class MeineKlasse:  
    quadrieren = staticmethod(quadrieren)
```

Variante 2 mit Dekorateur:

listings/v6_klassen16.py

```
class MeineKlasse:  
    @staticmethod  
    def quadrieren(x):  
        return x*x  
  
MeineKlasse.quadrieren(3) # Ausgabe: 9
```

13.4.5 Klassen-Methoden

Sie sind an eine Klasse gebunden. Variante 1:

listings/v6_klassen17.py

```
class MeineKlasse:  
    speed_of_light = 299792458  
  
    @classmethod  
    def c0(cls):
```

```
print('Speed_of_light=', cls.speed_of_light)
```

```
MeineKlasse.c0() # Ausgabe: Speed of light = 299792458
```

13.5 Datenabstraktion

- Datenabstraktion = Datenkapselung + Geheimnisprinzip
- Datenkapselung (Zugriff kontrollieren)
 - Setter- und Getter-Methoden
 - set_variable(value), get_variable()
- Geheimnisprinzip (interne Information verstecken)
 - public
 - protected
 - private

Der Zugang zu den Instanz-Attributen (Variablen und Methoden) sind in drei Stufen definiert: **public**, **protected** und **private**.

Hinweis: Das ist alles nur eine Konvention. In Python gibt es keinen Datenschutz.

listings/v6_klassen18.py

```
class MeineKlasse:

    def __init__(self):
        self.pub = 'Ich_bin_oeffentlich.'
        self._prot = 'Ich_bin_protected.'
        self.__priv = 'Ich_bin_privat.'

    def pub_funktion(self):
        print(self.pub)

    def _prot_funktion(self):
        print(self._prot)

    def __priv_funktion(self):
        print(self.__priv)

objekt = MeineKlasse()
```

13.5.1 Public

Attribute ohne führende Unterstriche im Namen sind als **public** zu betrachten. Man kann und darf auch von ausserhalb der Klasse darauf zugreifen.

listings/v6_klassen19.py

```
objekt.pub = 'Hier_macht_jeder_was_er_will.'
objekt.pub_funktion() # Ausgabe: Hier macht jeder was er will.
```

13.5.2 Protected

Attribute mit einem führenden Unterstrich im Namen sind als **protected** zu betrachten, d.h. man könnte theoretisch von aussen darauf zugreifen, man sollte aber nicht, es ist unerwünscht. Sie werden v.a. bei Vererbungen wichtig.

listings/v6_klassen20.py

```
print(objekt._prot) # Ausgabe: Ich bin protected.
objekt._prot_funktion() # Ausgabe: Ich bin protected.
```

13.5.3 Private

Attribute mit zwei führenden Unterstrichen im Namen sind **private**. Sie sind von aussen nicht sichtbar.

listings/v6_klassen21.py

```

objekt.__priv # Ausgabe:
# -----
# AttributeError                                Traceback (most recent call last)
# <ipython-input-5-9bcc05561362> in <module>
# ----> 1 objekt.__priv
#
# AttributeError: 'MeineKlasse' object has no attribute '__priv'

objekt.__priv_funktion() # Ausgabe:
# -----
# AttributeError                                Traceback (most recent call last)
# <ipython-input-16-6e6c693108e6> in <module>
# ----> 1 objekt.__priv_funktion()
#
# AttributeError: 'MeineKlasse' object has no attribute '__priv_funktion'

```

Im Prinzip gibt es einen Umweg um dies zu umgehen. **Achtung:** höchst illegal!

listings/v6_klassen22.py

```

objekt.__dict__ # Ausgabe:
# {'pub': 'Hier macht jeder was er will.',
#  '_prot': 'Ich bin protected.',
#  '_MeineKlasse__priv': 'Ich bin privat.'}

dir(objekt) # Ausgabe:
# ['_MeineKlasse__priv', '_MeineKlasse__priv_funktion', '__class__',
#  '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
#  '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
#  '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
#  '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
#  '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_prot',
#  '_prot_funktion', 'pub', 'pub_funktion']

objekt._MeineKlasse__priv # Ausgabe: 'Ich bin privat.'
objekt._MeineKlasse__priv_funktion() # Ausgabe: Ich bin privat.

```

13.5.4 Setter- und Getter-Methoden

Setter- und Getter-Methoden für private Instanz-Variablen auf phytonsche Art und Weise implementieren.

Konventionell: Set- und Get-Methoden explizit benutzen.

listings/v6_klassen23.py

```

class Bank:
    def __init__(self):
        self.__guthaben = 0

    def get_guthaben(self):
        print('Das Guthaben wurde abgefragt.')
        return self.__guthaben

    def set_guthaben(self, n):
        self.__guthaben = n
        print('Das Guthaben wurde auf {} geändert.'.format(self.__guthaben))

k = Bank()
k.set_guthaben(10000000) # Ausgabe: Das Guthaben wurde auf 10000000 geändert.
print(k.get_guthaben()) # Ausgabe: Das Guthaben wurde abgefragt.
# Ausgabe: 10000000

```

Property: Von aussen wie auf eine öffentliche Variable zugreifen, Set- und Get-Methoden werden implizit aufgerufen.

<https://docs.python.org/3/library/functions.html#property>

listings/v6_klassen24.py

```

class Bank:

```

```

def __init__(self):
    self.__guthaben = 0

def __get_guthaben(self):
    print('Das Guthaben wurde abgefragt.')
    return self.__guthaben

def __set_guthaben(self, n):
    self.__guthaben = n
    print('Das Guthaben wurde auf {} geändert.'.format(self.__guthaben))

guthaben = property(__get_guthaben, __set_guthaben)

k = Bank()
k.guthaben = 10000000 # Ausgabe: Das Guthaben wurde auf 10000000 geändert.
print(k.guthaben) # Ausgabe: Das Guthaben wurde abgefragt.
# Ausgabe: 10000000

```

Property mit Dekorateuren: Auf pythonische Art und Weise.

<https://docs.python.org/3/glossary.html#term-decorator>

listings/v6_klassen25.py

```

class Bank:
    def __init__(self):
        self.__guthaben = 0

    @property
    def guthaben(self):
        print('Das Guthaben wurde abgefragt.')
        return self.__guthaben

    @guthaben.setter
    def guthaben(self, n):
        self.__guthaben = n
        print('Das Guthaben wurde auf {} geändert.'.format(self.__guthaben))

k = Bank()
k.guthaben = 10000000 # Ausgabe: Das Guthaben wurde auf 10000000 geändert.
print(k.guthaben) # Ausgabe: Das Guthaben wurde abgefragt.
# Ausgabe: 10000000

```

13.6 Magische Methoden

- Besondere Fähigkeiten für Klassen
(<https://docs.python.org/3/reference/datamodel.html#special-method-names>)
- Grundfunktionen
 - `__init__()`, `__del__()`, `__str__()`, ...
- Operatoren überladen
 - binäre Operatoren: `+` `-` `*` `%` ...
 - numerische Operatoren: `__int__()`, `__float__()`, `__abs__()`, ...
 - ...
- Containertypen emulieren
 - `__len__()`, `__iter__()`, `__contains__()`, ...
- ...

Sie sind spezielle Methoden, um Klassen besondere Fähigkeiten zu geben. Es werden hier nur einige Beispiele gezeigt.

13.6.1 Grundmethoden

<https://docs.python.org/3/reference/datamodel.html#basic-customization>

Zwei davon haben wir schon kennengelernt:

- `__init__()`
- `__del__()`

Der Rückgabewert von `__str__()` gibt an, was `str(obj)` zurückgeben soll, z.B.:

listings/v6_klassen26.py

```
class Konto:
    def __init__(self, guthaben, iban):
        self.guthaben = guthaben
        self.iban = iban

    def __str__(self):
        return 'IBAN:_{}\nGuthaben:{}'.format(self.iban, self.guthaben)

k = Konto(50, 'CH42_4738_2934_9267_0878_5')
print(k) # Ausgabe:
# IBAN: CH42 4738 2934 9267 0878 5
# Guthaben: 50
```

13.6.2 Numerische Datentypen emulieren

<https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>

Der Rückgabewert von `__float__()` gibt an, was `float(obj)` zurückgeben soll.

Mit der `__add__()`-Methode wird der `+` Operator überladen.

Mit der `__sub__()`-Methode wird der `-` Operator überladen.

listings/v6_klassen27.py

```
class Konto:
    def __init__(self, guthaben, iban):
        self.guthaben = guthaben
        self.iban = iban

    def __float__(self):
        return float(self.guthaben)

    def __add__(self, other):
        return self.guthaben + other.guthaben

    def __sub__(self, other):
        return self.guthaben - other.guthaben

k1 = Konto(50, 'CH42_4738_2934_9267_0878_5')
k2 = Konto(23, 'CH27_1036_5802_2994_9234_3')
print('float(k1)=', float(k1)) # Ausgabe: float(k1) = 50.0
print('float(k2)=', float(k2)) # Ausgabe: float(k2) = 23.0
print('k1+k2=', k1 + k2) # Ausgabe: k1 + k2 = 73
print('k1-k2=', k1 - k2) # Ausgabe: k1 - k2 = 27
```

13.7 Klassen testen

- Klassen werden in separate Pythondateien gespeichert
- Testcode in die gleiche Datei integrieren
- Testcode in eine `if`-Anweisung platzieren:

listings/v6_klassen28.py

```
if __name__ == '__main__':
    Testcode
```

listings/v6_my_module.py

```
# -*- coding: utf-8 -*-
print('Dies ist {}: \n__name__={}'.format(__file__, __name__))

class MeineKlasse:
    def __init__(self, name):
        self.name = name

    def gruss(self):
```



```

    print('Hallo', self.name)

# -----
if __name__ == '__main__':
    k = MeineKlasse('Python')
    k.gruss()

# Konsolen-Ausgabe:
# Dies ist C:/Users/Noah/Documents/GitHub/Python_Zusammenfassung/listings/v6_my_module.py:
# __name__ = __main__
# Hallo Python

```

listings/scripts/my_other_module.py

```

# -*- coding: utf-8 -*-
print('Dies ist {}: \n__name__={}'.format(__file__, __name__))

class Bank:
    def __init__(self):
        self.__guthaben = 0

    @property
    def guthaben(self):
        print('Das Guthaben wurde abgefragt.')
        return self.__guthaben

    @guthaben.setter
    def guthaben(self, n):
        self.__guthaben = n
        print('Das Guthaben wurde auf {} geändert.'.format(self.__guthaben))

# --- Klasse testen -----
if __name__ == '__main__':
    b = Bank()
    b.guthaben = 1000
    print(b.guthaben)

# Konsolen-Ausgabe:
# Dies ist C:/Users/Noah/Documents/GitHub/Python_Zusammenfassung/listings/scripts/my_other_module.py
# :
# __name__ = __main__
# Das Guthaben wurde auf 1000 geändert.
# Das Guthaben wurde abgefragt.
# 1000

```

13.8 Eigenes Modul importieren

- Klasse aus einer separaten Pythondatei importieren
 - aus dem gleichen Verzeichnis
 - aus einem anderen Verzeichnis

<https://docs.python.org/3/tutorial/modules.html>

13.8.1 Aus dem gleichen Verzeichnis

listings/v6_klassen29.py

```

from my_module import MeineKlasse
# Ausgabe: Dies ist C:\Users\Noah\switchdrive\Python\vorlesung\w06\code\my_module.py:
# __name__ = my_module
m = MeineKlasse('Python_User')
m.gruss() # Ausgabe: Hallo Python User

```

13.8.2 Aus einem andere Verzeichnis

listings/v6_klassen30.py

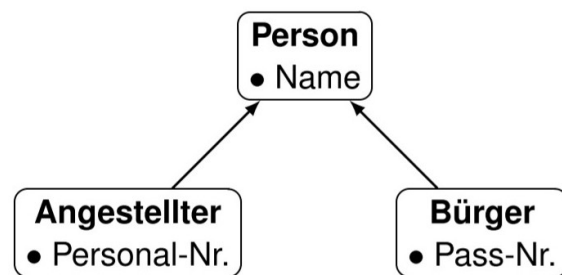
```
import sys
sys.path.append('scripts')
print('\n'.join(sys.path)) # Liste der Suchorte
# Ausgabe: *alle in Frage kommenden Verzeichnisse*

from my_other_module import Bank # Ausgabe:
# Dies ist scripts\my_other_module.py:
# __name__ = my_other_module
b = Bank()
b.guthaben = 500.0 # Ausgabe: Das Guthaben wurde auf 500.0 geaendert.
```

Lektion 7: Vererbungen und Mehrfachvererbungen

14 Vererbung

- eine neue Klasse aus einer bestehenden Klasse ableiten:
- **Person** ist eine: Oberklasse, Basisklasse, Elternklasse oder Superklasse
- **Angestellter** und **Bürger** sind eine: Unterklasse, abgeleitete Klasse, Kindklasse oder Subklasse
- Superklasse/Basisklasse:



listings/v7_vererbung1.py

```
class Person:
    pass
```

- Für die Vererbung: Superklasse in runden Klammern angeben
- Subklasse/abgeleitete Klasse:

listings/v7_vererbung2.py

```
class Angestellter(Person):
    pass
```

- Variablen und Methoden (public und protected) werden direkt übernommen:

listings/v7_vererbung3.py

```
class Person:
    var = 123

    def func(self):
        print('Person')

class Angestellter(Person):
    pass

a = Angestellter()
print(a.var) # Ausgabe: 123
a.func() # Ausgabe: Person
```

- Methoden werden überschrieben, falls sie gleich heissen:

listings/v7_vererbung4.py

```
class Person:
```

```

def func(self):
    print('Person')

class Angestellter(Person):
    def func(self):
        print('Angestellter')

a = Angestellter()
a.func() # Ausgabe: Angestellter

```

- Zugriff auf die Superklasse mit `super()`

listings/v7_vererbung5.py

```

class Person:
    def func(self):
        print('Person')

class Angestellter(Person):
    def func(self):
        super().func()
        print('Angestellter')

a = Angestellter()
a.func() # Ausgabe:
# Person
# Angestellter

```

14.1 Beispiel

listings/v7_vererbung6.py

```

class Person:
    def __init__(self, name):
        self.name = name
        print('__init__() von Person')

```

Die Person-Klasse instanzieren:

listings/v7_vererbung7.py

```

p = Person('Laura') # Ausgabe: __init__() von Person
print(p.name) # Ausgabe: Laura

```

Angestellte-Klasse erbt von der Person-Klasse:

listings/v7_vererbung8.py

```

class Angestellter(Person):
    def __init__(self, name, personalnummer):
        # Initialisierungsmethode der Superklasse aufrufen
        super().__init__(name)
        # oder Person.__init__(self, name)
        self.personalnummer = personalnummer
        print('__init__() von Angestellter')

```

Die Angestellter-Klasse instanzieren:

listings/v7_vererbung9.py

```

a = Angestellter('Max', 123456) # Ausgabe:
# __init__() von Person
# __init__() von Angestellter

```

```
print(a.name) # Ausgabe: Max
print(a.personalnummer) # Ausgabe: 123456
```

14.2 public, protected und private

Die Konvention ist wie folgt:

public: für öffentliche Variablen und Methoden

protected: (1 führender Unterstrich) für nicht-öffentliche Variablen und Methoden

private: (2 führende Unterstriche) für nicht-öffentliche Variablen und Methoden, um Namenskonflikte in Subklassen zu vermeiden

<https://www.python.org/dev/peps/pep-0008/#method-names-and-instance-variables>

listings/v7_vererbung10.py

```
class SuperKlasse:
    def __init__(self):
        self.pub = 'public_Variable'
        self._prot = 'protected_Variable'
        self.__priv = 'private_Variable'

    def pub_func(self):
        print('public_Methode')

    def _prot_func(self):
        print('protected_Methode')

    def __priv_func(self):
        print('private_Methode')

class SubKlasse(SuperKlasse):
    def __init__(self):
        self.pub_func()
        self._prot_func()
        self.__priv_func() # nicht erreichbar, kann in der Subklasse wiederbenutzt werden

sub = SubKlasse() # Ausgabe:
# public Methode
# protected Methode

# -----
# AttributeError                                Traceback (most recent call last)
# <ipython-input-12-479270c9858a> in <module>
# ----> 1 sub = SubKlasse()

# <ipython-input-11-1794f0b16121> in __init__(self)
#      3         self.pub_func()
#      4         self._prot_func()
# ----> 5         self.__priv_func() # nicht erreichbar, kann in der Subklasse wiederbenutzt werden

# AttributeError: 'SubKlasse' object has no attribute '_SubKlasse__priv_func'
```

15 Mehrfachvererbung

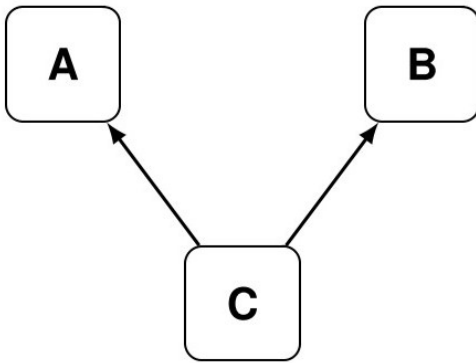
- Eine Subklasse kann von mehreren Superklassen erben:

listings/v7_vererbung11.py

```
class A:
    pass

class B:
    pass
```

```
class C(A, B):
    pass
```



Am besten die `__init__`-Methode der Klassen kooperativ machen, d.h.

- immer `super()` benutzen
- Schlüsselwort-Argumente benutzen
- unbenutzte Schlüsselwort-Argumente weitergeben (`**kwargs`)

listings/v7_vererbung12.py

```

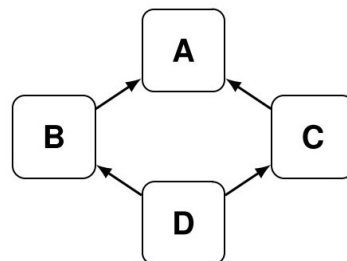
class Fahrzeug:
    def __init__(self, antrieb, **kwargs):
        print('Fahrzeug.__init__()', 'kwargs=', kwargs)
        super().__init__(**kwargs)
        self.antrieb = antrieb

class Computer:
    def __init__(self, display, **kwargs):
        print('Computer.__init__()', 'kwargs=', kwargs)
        super().__init__(**kwargs)
        self.display = display

class Tesla(Fahrzeug, Computer):
    def __init__(self, display, dual_motor, **kwargs):
        print('Tesla.__init__()')
        super().__init__(
            antrieb='elektrisch',
            display=display,
            **kwargs
        )
        self.dual_motor = dual_motor

t = Tesla(display='17_Zoll', dual_motor=True) # Ausgabe:
# Tesla.__init__()
# Fahrzeug.__init__(), kwargs = {'display': '17 Zoll'}
# Computer.__init__(), kwargs = {}
t.__dict__ # Ausgabe: {'display': '17 Zoll', 'antrieb': 'elektrisch', 'dual_motor': True}
  
```

- `super()` ruft automatisch die Methode der nächsten Klasse auf
- Method Resolution Order (MRO) → C4 Superclass Linearization (https://en.wikipedia.org/wiki/C3_linearization)
- Diamond-Problem ist kein Problem mit `super()`



15.0.1 MRO

Mehrfachvererbung in Diamant-Anordnung:

listings/v7_vererbung13.py

```

class A:
    def __init__(self):
  
```

```

    print("A.__init__")
    super().__init__()

class B(A):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()

class D(B, C):
    def __init__(self):
        print("D.__init__")
        super().__init__()

```

super() ruft die Methoden der Reihe nach auf:

listings/v7_vererbung14.py

```

d = D() # Ausgabe:
# D.__init__
# B.__init__
# C.__init__
# A.__init__

```

Die Reihenfolge wird vom MRO-Algorithmus festgelegt:

listings/v7_vererbung15.py

```

D.mro() # Ausgabe: [__main__.D, __main__.B, __main__.C, __main__.A, object]

```

Lektion 8: NumPy und Matplotlib

16 NumPy

- Python-Bibliothek
import numpy as np
- Einfache Handhabung mit Vektoren und Matrizen
 - mehrdimensionale Arrays
- Funktionen für numerische Berechnungen
 - Grundlegende Operationen
 - Mathematische Funktionen (sin, cos, sqrt, exp, ...)
 - Lineare Algebra
 - ...
- Effiziente und schnelle Ausführung
 - kompilierte Funktionen und Algorithmen
 - Array-basierte Operationen → keine for-Schleifen
- Ähnlichkeit zu MATLAB®
<https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.htm>

16.1 ndarray erzeugen

- N-dimensionales Array (<https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html>)
- ndarray erzeugen

listings/v8_numpy1.py

```

arr1 = np.array([1, 2, 3])
print(arr1) # Ausgabe: [1 2 3]

```

```
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2) # Ausgabe:
# [[1 2 3]
#  [4 5 6]]
arr2.ndim # Ausgabe: 2
arr2.shape # Ausgabe: (2, 3)
```

- Weitere Funktionen, um Arrays zu erzeugen

Funktion	Resultat
<code>np.arange(3)</code>	<code>array([0, 1, 2])</code>
<code>np.ones((2,2))</code>	<code>array([[1., 1.], [1., 1.]])</code>
<code>np.ones_like(arr1)</code>	<code>array([1, 1, 1])</code>
<code>np.zeros((2,2))</code>	<code>array([[0., 0.], [0., 0.]])</code>
<code>np.zeros_like(arr1)</code>	<code>array([0, 0, 0])</code>
<code>np.full((2,2), 7.0)</code>	<code>array([[7., 7.], [7., 7.]])</code>
<code>np.full_like(arr1, 7)</code>	<code>array([7, 7, 7])</code>
<code>np.eye(2)</code>	<code>array([[1., 0.], [0., 1.]])</code>
<code>np.identity(2)</code>	<code>array([[1., 0.], [0., 1.]])</code>
<code>np.linspace(0, 1, 5)</code>	<code>array([0., 0.25, 0.5, 0.75, 1.])</code>
<code>np.logspace(0, 1, 4)</code>	<code>array([1., 2.1544, 4.6416, 10.])</code>
<code>np.random.randn(3)</code>	<code>array([0.7576, 0.0135, -0.8934])</code>

16.1.1 ndarray-Datentypen

- Datentyp wird automatisch ermittelt, z.B. `np.int64` oder `np.float64`
- Datentyp erzwingen
`np.array([1, 2, 3], dtype=np.complex)`
- Mögliche Datentypen

<code>np.int8, np.uint8</code>	8-Bit Ganzzahlen
<code>np.int16, np.uint16</code>	16-Bit Ganzzahlen
<code>np.int32, np.uint32</code>	32-Bit Ganzzahlen
<code>np.int64, np.uint64</code>	64-Bit Ganzzahlen
<code>np.float16</code>	Float mit halber Genauigkeit
<code>np.float32</code>	Float mit einfacher Genauigkeit
<code>np.float64</code>	Float mit doppelter Genauigkeit
<code>np.float128</code>	Float mit vierfacher Genauigkeit
<code>np.complex64/128/256</code>	Komplexe Zahl
<code>np.bool</code>	Boolescher Wert, True/False

16.2 Arithmetische Operationen

- Arithmetische Operationen werden elementweise ausgeführt

listings/v8_numpy2.py

```
arr = np.array([1., 2., 3.])
```

Operation	Resultat
<code>arr + arr</code>	<code>array([2., 4., 6.])</code>
<code>arr + 1</code>	<code>array([2., 3., 4.])</code>
<code>arr - arr</code>	<code>array([0., 0., 0.])</code>
<code>arr - 1</code>	<code>array([0., 1., 2.])</code>
<code>arr*arr</code>	<code>array([1., 4., 9.])</code>
<code>arr*2</code>	<code>array([2., 4., 6.])</code>
<code>arr/arr</code>	<code>array([1., 1., 1.])</code>
<code>arr/2</code>	<code>array([0.5, 1., 1.5])</code>
<code>arr**2</code>	<code>array([1., 4., 9.])</code>
<code>arr > 2</code>	<code>array([False, False, True], dtype=bool)</code>

16.3 Indexierung

- Indexierung von 2D-Arrays

listings/v8_numpy3.py

```
arr[axis0, axis1]
```

- Beispiele:

listings/v8_numpy4.py

```
arr[0, 0] # Ausgabe: 1.0
arr[2, 0] # Ausgabe: 7.0
arr[0, 2] # Ausgabe: 3.0
```

		axis=1		
		0	1	2
axis=0	0	1.0	2.0	3.0
	1	4.0	5.0	6.0
	2	7.0	8.0	9.0

16.3.1 Slicing

arr	Ausdruck	Shape	Resultat									
<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	5	6	7	8	9	arr[:2, 1:]	(2,2)	array([[2, 3], [5, 6]])
1	2	3										
4	5	6										
7	8	9										
<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	5	6	7	8	9	arr[2]	(3,)	array([7, 8, 9])
1	2	3										
4	5	6										
7	8	9										
<table><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	4	5	6	7	8	9	arr[2, :]	(3,)	array([7, 8, 9])			
4	5	6										
7	8	9										
<table><tr><td>7</td><td>8</td><td>9</td></tr></table>	7	8	9	arr[2:, :]	(1,3)	array([[7, 8, 9]])						
7	8	9										
<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	5	6	7	8	9	arr[:, :2]	(3,2)	array([[1, 2], [4, 5], [7, 8]])
1	2	3										
4	5	6										
7	8	9										
<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	5	6	7	8	9	arr[1, :2]	(2,)	array([4, 5])
1	2	3										
4	5	6										
7	8	9										
<table><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	4	5	6	7	8	9	arr[1:2, :2]	(1,2)	array([[4, 5]])			
4	5	6										
7	8	9										

→ ndim bleibt erhalten, falls bei jeder axis ein ":" steht.

- Ein Slice ist immer eine Referenz, keine Kopie!

listings/v8_numpy5.py

```
arr = np.arange(8)
print(arr) # Ausgabe: array([0, 1, 2, 3, 4, 5, 6, 7])
s = arr[2:5] # array([2, 3, 4])
s[0] = 13 # modifiziert auch arr
print(arr) # Ausgabe: array([0, 1, 13, 3, 4, 5, 6, 7])
```

- Kopien werden mit .copy() erzeugt:

listings/v8_numpy6.py

```
s = arr[2:5].copy()
```

- Zuweisung eines Skalars zu einem Slice wird ausgebreitet:

listings/v8_numpy7.py

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
arr[2:5] = 9
print(arr) # Ausgabe: array([1, 2, 9, 9, 9, 6, 7, 8])
```

- Bei fehlender Dimension wird das Array automatisch erweitert:

listings/v8_numpy8.py

```
arr1 = np.ones((3, 2)) # shape=(3, 2)
arr2 = np.array([1, 2]) # shape=(2,)
arr1 + arr2 # Ausgabe:
# array([[2., 3.],
#        [2., 3.],
#        [2., 3.]])
```

→ Bedingung: letzte Dimension ist gleich oder nur 1 lang.

16.4 Mathematische Funktionen

- NumPy beinhaltet viele mathematische Funktionen:

<https://docs.scipy.org/doc/numpy/reference/routines.math.html>

- np.sin()
- np.cos()
- np.exp()
- np.cumsum()
- ...

- Diese Funktionen operieren über das gesamte Array

listings/v8_numpy9.py

```
t = linspace(1, 3, 5)
np.log10(t) # Ausgabe: array([0., 0.17609, 0.30103, 0.39794, 0.47712])
np.cumsum(t) # Ausgabe: array([1., 2.5, 4.5, 7., 10.])
np.mean(t) # Ausgabe: 2.0
```

16.4.1 Lineare Algebra

- Liste der Funktionen:

<https://docs.scipy.org/doc/numpy/reference/routines.linalg.html>

- Matrix definieren

listings/v8_numpy14.py

```
M = np.array([[1, 2], [3, 4]])
# array([[1, 2],
#        [3, 4]])
```

- Vektor definieren

listings/v8_numpy15.py

```
v = np.array([5, 6])
# array([5, 6])
```

- Matrix **M** mit Vektor **v** multiplizieren

listings/v8_numpy10.py

```
np.dot(M, v)
M.dot(v)
M @ v # Ausgabe: ab Python 3.5
```

- Matrix transponieren \mathbf{M}^T

listings/v8_numpy11.py

```
np.transpose(M)
M.T
```

- Matrix invertieren \mathbf{M}^{-1}

listings/v8_numpy12.py

```
np.linalg.inv(M)
```

- Shape eines Vektors ändern

listings/v8_numpy16.py

```
v2 = np.array([1, 2, 3, 4, 5, 6])
# array([1, 2, 3, 4, 5, 6])
v2.reshape((2, 3))
# array([[1, 2, 3],
#        [4, 5, 6]])
v2.reshape((6, 1))
# array([[1],
#        [2],
#        [3],
#        [4],
#        [5],
#        [6]])
v2.shape = (3, 2)
# array([[1, 2],
#        [3, 4],
#        [5, 6]])
```

16.4.2 Matplotlib

- Python-Bibliothek
import matplotlib.pyplot as plt
- Erstellen von publizierbaren Diagrammen und Darstellungen
- 100% kompatibel zu NumPy-Arrays
- MATLAB®-ähnliche Funktionen:
https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html
- Kann auch objekt-orientiert verwendet werden, z.B. in GUIs
- Grosse Beispiel-Sammlung:
<https://matplotlib.org/gallery/index.html>
- Einfaches Beispiel:

listings/v8_numpy13.py

```
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(100)/100
s1 = np.sin(2*np.pi*t)
s2 = s1 + np.random.randn(*s1.shape)/4

plt.figure()
plt.plot(t, s1, '-.', label='Simulation')
plt.plot(t, s2, '-.', label='Messung')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig('diagramm.pdf')
```

