

# Python

N. Kaelin, S. Walker

7. April 2019

## Inhaltsverzeichnis

<b>1</b>	<b>Datentypen</b>	<b>4</b>
1.1	Numerische Datentypen . . . . .	4
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
<b>2</b>	<b>Verzweigungen</b>	<b>8</b>
2.1	<code>if</code> . . . . .	8
2.1.1	<code>if</code> -Anweisung mit <code>else</code> -Zweig . . . . .	8
2.1.2	<code>elif</code> -Zweige . . . . .	8
<b>3</b>	<b>Schleifen</b>	<b>8</b>
3.1	<code>while</code> . . . . .	8
3.1.1	<code>continue</code> . . . . .	9
3.1.2	<code>break</code> . . . . .	9
3.1.3	<code>else</code> -Teil . . . . .	9
3.2	<code>for</code> . . . . .	9
<b>4</b>	<b>Funktionen</b>	<b>9</b>
4.1	Funktionsdefinition . . . . .	10
4.2	Aufruf . . . . .	10
4.3	Weiteres . . . . .	10
4.3.1	Standardwert für Parameter . . . . .	10
4.3.2	Mehrere Rückgabewerte . . . . .	10
4.3.3	Variable Anzahl von Argumenten . . . . .	11
4.3.4	Argumente entpacken . . . . .	11
4.3.5	Beliebige Schlüsselwort-Parameter . . . . .	11
4.3.6	Schlüsselwortparameter entpacken . . . . .	11
4.3.7	Globale Variablen . . . . .	11
4.3.8	Docstring - Funktion dokumentieren . . . . .	12
4.3.9	Call-by-object-reference . . . . .	12
<b>5</b>	<b>Exceptions</b>	<b>13</b>
5.1	Unspezifische Exceptions abfangen . . . . .	13
5.2	Master Beispiel . . . . .	13

<b>6</b>	<b>Dateien</b>	<b>14</b>
6.1	Datei öffnen . . . . .	14
6.2	Dateien lesen und schreiben . . . . .	14
6.2.1	with-Anweisung . . . . .	15
6.2.2	glob . . . . .	15
6.2.3	os.path . . . . .	15
<b>7</b>	<b>Strings</b>	<b>15</b>
7.1	Stringformatierung . . . . .	15
7.1.1	im C-Stil (à la printf) . . . . .	16
7.1.2	mit format() . . . . .	16
7.1.3	mit Stringlitterale . . . . .	17
7.1.4	mit string-Methoden . . . . .	17
7.2	Alles über Strings <small>Kap. 19</small> . . . . .	17
7.2.1	Strings aufspalten . . . . .	17
7.2.2	Strings kombinieren . . . . .	18
7.2.3	Suchen von Teilstrings . . . . .	18
7.2.4	Ersetzen von Teilstrings . . . . .	18
7.2.5	Strings bereinigen . . . . .	19
7.2.6	Klein- und Grossbuchstaben . . . . .	19
7.2.7	Strings testen . . . . .	19
<b>8</b>	<b>Listen-Abstraktion/List-Comprehension</b>	<b>20</b>
8.1	Neue Liste aus einer bestehenden Liste ableiten . . . . .	20
8.1.1	Beispiel 1 . . . . .	20
8.1.2	Beispiel 2 . . . . .	20
8.2	Bestehende Liste filtern . . . . .	20
8.3	Liste von Zahlen => formatierter String . . . . .	21
8.4	Liste der Schachbrettfelder . . . . .	21
8.5	Mengen-Abstraktion/Set Comprehension . . . . .	22
8.5.1	Produkte zweier Zahlen . . . . .	22
<b>9</b>	<b>Iteratoren und Generatoren</b>	<b>22</b>
9.1	Iteratoren . . . . .	22
9.2	Generatoren . . . . .	23
9.2.1	Generator-Expression . . . . .	24
9.2.2	send()-Methode, Generator als Coroutine . . . . .	24
<b>10</b>	<b>Listen und Tupel im Detail</b>	<b>24</b>
10.1	Tupel . . . . .	24
10.2	Listen . . . . .	25
10.2.1	Element hinzufügen . . . . .	25
10.2.2	Mehrere Elemente hinzufügen . . . . .	25
10.2.3	Elemente ersetzen . . . . .	26
10.2.4	Element entfernen . . . . .	26
10.3	Sortieren . . . . .	26
10.3.1	Umgekehrte Reihenfolge . . . . .	27
10.3.2	Mit spezieller Funktion . . . . .	27
10.3.3	collections.deque . . . . .	27
<b>11</b>	<b>lambda, map, filter und reduce</b>	<b>28</b>
11.1	lambda . . . . .	28
11.2	map . . . . .	28
11.3	filter . . . . .	29
11.4	reduce . . . . .	29
<b>12</b>	<b>Zeichen-Klassen</b>	<b>30</b>

<b>13 Wiederholungen (Quantoren)</b>	<b>30</b>
<b>14 Übereinstimmungen</b>	<b>31</b>
14.1 match-Objekt . . . . .	31
14.2 Übereinstimmungen finden . . . . .	31
<b>15 Modifizierungen</b>	<b>33</b>
<b>16 Gruppierung</b>	<b>34</b>
16.1 Weitere Metazeichen . . . . .	35
16.2 Look-around Assertions . . . . .	36
<b>17 Klassen</b>	<b>37</b>
17.1 Einfache Klasse definieren . . . . .	37
17.2 Klasse instanzieren . . . . .	37
17.3 Klassen- und Instanz-Variablen . . . . .	37
17.4 Methoden . . . . .	38
17.4.1 <code>__init__()</code> -Methode . . . . .	39
17.4.2 <code>__del__()</code> -Methode . . . . .	39
17.4.3 Methoden aufrufen . . . . .	40
17.4.4 Statische Methoden . . . . .	40
17.4.5 Klassen-Methoden . . . . .	40
17.5 Datenabstraktion . . . . .	41
17.5.1 <code>Public</code> . . . . .	41
17.5.2 <code>Protected</code> . . . . .	41
17.5.3 <code>Private</code> . . . . .	41
17.5.4 Setter- und Getter-Methoden . . . . .	42
17.6 Magische Methoden . . . . .	43
17.6.1 Grundmethoden . . . . .	44
17.6.2 Numerische Datentypen emulieren . . . . .	44
17.7 Klassen testen . . . . .	45
17.8 Eigenes Modul importieren . . . . .	46
17.8.1 Aus dem gleichen Verzeichnis . . . . .	46
17.8.2 Aus einem andere Verzeichnis . . . . .	46
<b>18 Vererbung</b>	<b>47</b>
18.1 Beispiel . . . . .	48
18.2 <code>public</code> , <code>protected</code> und <code>private</code> . . . . .	49
<b>19 Mehrfachvererbung</b>	<b>49</b>
19.0.1 MRO . . . . .	51

# Lektion 1: Variablen und Datentypen

## 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
<b>Numerische Datentypen</b>		
int	Ganze Zahlen	0
float	Gleitkommazahlen	0.0
bool	Boolesche Werte	False
complex	Komplexe Zahlen	0 + 0j
<b>Sequenzielle Datentypen</b>		
str	Zeichenketten oder Strings	''
list	Listen (veränderlich)	[]
tuple	Tupel (unveränderlich)	()
bytes	Sequenz von Bytes (unveränderlich)	b''
bytearray	Sequenz von Bytes (veränderlich)	bytearray(b'')
<b>Assoziative Datentypen</b>		
dict	Dictionary (Schlüssel-Wert-Paare)	{}
<b>Mengen</b>		
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 <sup>1</sup> von $x$ und $y$
$x \% y$	Rest der Division <sup>1</sup> von $x$ durch $y$
$+x$	Positives Vorzeichen
$-x$	Negatives Vorzeichen
$\text{abs}(x)$	Betrag von $x$
$x ** y$	Potenzieren, $x^y$

<sup>1</sup>Nicht 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
<code>==</code>	wahr, wenn $x$ und $y$ gleich sind
<code>!=</code>	wahr, wenn $x$ und $y$ verschieden sind
<code>&lt;</code>	wahr, wenn $x$ kleiner als $y$ ist <sup>2</sup>
<code>&lt;=</code>	wahr, wenn $x$ kleiner oder gleich $y$ ist <sup>2</sup>
<code>&gt;</code>	wahr, wenn $x$ grösser als $y$ ist <sup>2</sup>
<code>&gt;=</code>	wahr, wenn $x$ grösser oder gleich $y$ ist <sup>2</sup>

<sup>2</sup>Nicht definiert für den Datentyp `complex`

### 1.1.3 Bitweise Operatoren für den Datentypen `int`

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

### 1.1.4 Methoden nur für den Datentyp `complex`

Tabelle 5: Methoden für `complex`

Methode	Beschreibung
<code>x.real</code>	Realteil von $x$ als Gleitkommazahl
<code>x.imag</code>	Imaginärteil von $x$ als Gleitkommazahl
<code>x.conjugate()</code>	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

# Lektion 2: Verzweigungen, Schleifen und Funktionen

## 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
```

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

#### 2.1.2 elif-Zweige

listings/v2\_if3.py

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

elif = else if

---

**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:
    Anweisung2
```

else-Teil: wenn die Schleife **nicht** durch break abgebrochen wurde

## 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

Python besitzt eine grosse Standard-Bibliothek, z.B.:

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
```

<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)
```

- 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++

listings/v2\_func5.py

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

## 4.2 Aufruf

listings/v2\_func6.py

```
resultat1 = summe(2, 3)  
resultat2 = summe(a=10, b=2) # Schluessselwortparameter  
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):  
    return (a + b, a - b) # Tupel  
  
summe, differenz = summe_und_differenz(5, 3) # Tupel entpacken
```

### 4.3.3 Variable Anzahl von Argumenten

listings/v2\_func9.py

```
def mittelwert(a, *args):    # a ist zwingend
    print('a_=', 1)
    print('args_=', args)    # die restlichen Argumente sind 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)
    print('kwargs_=', kwargs)    # die restlichen Argumente sind im Dictionary
    kwargs

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

### 4.3.6 Schlüsselwortparameter entpacken

listings/v2\_func12.py

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

### 4.3.7 Globale Variablen

listings/v2\_func13.py

```
modul = 'Python'    # globale Variable

def anmeldung():
    print(modul)    # Variable existiert bereits ausserhalb der Funktion

anmeldung()    # Ausgabe: Python

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

```

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

def wirklich_wechseln():
    global modul #referenzieren auf die globale Variable
    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):
    a.append(4) # Objekt veraendern
    b = [10, 11, 12] # lokale Variable b referenziert neues Objekt

foo(x, y)
print('x_=', x)
print('y_=', y)

```

mit unveränderlichen Objekten:

listings/v2\_func16.py

```

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

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

foo(x, y)
print('x_=', x)
print('y_=', y)

```

# Lektion 3: Exceptions, Dateien und Strings

## 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=, 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.

Beispiel:

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

# Gibt 'true' zurueck wenn full_path eine Datei ist
os.path.isfile(full_path)

# Gibt 'true' zurueck wenn full_path eine Ordner ist
os.path.isdir(full_path)

# Groesse der Datei/Ordner
os.path.getsize(full_path)

# Teilt den Pfad in einen Tupel (Pfad, Dateiname)
os.path.split(full_path)

# Teilt den Pfad in einen Tupel (Pfad, Dateiname, Endung)
os.path.splitext(full_path)

# Macht einen gueltigen Pfad (System abhaengig)
os.path.join('ordner', 'datei.txt')
```

## 7 Strings

### 7.1 Stringformatierung Kap. 12

- Stringformatierung benötigt man um Daten hübsch auszugeben

listings/v3\_strings1.py

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

5	R4	47.00k
---	----	--------

- oder systematisch abzuspeichern

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=%0x%04X, Y=%0x%04X' % (7, 15)) # hex
# Ausgabe: X = 0x0007, Y = 0x000F
```

### 7.1.2 mit format()

listings/v3\_strings4.py

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

- mit Index:

listings/v3\_strings5.py

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

- mit Index und Format:

listings/v3\_strings6.py

```
'U=%{0:.2f}, I=%{0:.f}'.format(spannung)
# Ausgabe: 'U = 12.56, I = 12.560000'
```

- links-/rechtsbündig oder zentriert:

listings/v3\_strings7.py

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

- mit Schlüsselwortparameter:



listings/v3\_strings8.py

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

- mit Dictionary:

listings/v3\_strings9.py

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

### 7.1.3 mit Stringlitterale

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

- Unicode-Nummer => Zeichen

listings/v3\_strings12.py

```
chr(65)
# Ausgabe: ('A')
```

- Zeichen => Unicode-Nummer

listings/v3\_strings13.py

```
ord('A')
# Ausgabe: (65)
```

- String => bytes

listings/v3\_strings14.py

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

### 7.2.1 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']

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

csv.rsplit('; ', maxsplit=2) # max. zwei Trennungen von rechts her
# 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.']
```

### 7.2.2 Strings kombinieren

listings/v3\_strings17.py

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

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

### 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.'

'Ein 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
```

# Lektion 4: Listen-Abstraktion, Generatoren und Ähnliches

## 8 Listen-Abstraktion/List-Comprehension

- 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

#### 8.1.1 Beispiel 1

konventionell:

listings/v4\_list1.py

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

print(quadratzahlen)
```

mit Listen-Abstraktion:

listings/v4\_list2.py

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

print(quadratzahlen)
```

#### 8.1.2 Beispiel 2

konventionell:

listings/v4\_list3.py

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

print(meilen)
```

mit Listen-Abstraktion:

listings/v4\_list4.py

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

print(meilen)
```

### 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']

# konventionell:
fruechte_abc = []
for frucht in fruechte:
    if frucht[0] in 'ABC':
        fruechte_abc.append(frucht)

print(fruechte_abc)

# mit Listen-Abstraktion:
```

```
fruechte_abc = [frucht for frucht in fruechte if frucht[0] in 'ABC']

print(fruechte_abc)
```

### 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))

print(felder)
# Ausgabe: ['a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'a7', 'a8', 'b1', 'b2', 'b3', 'b4',
    'b5', 'b6', 'b7', 'b8', 'c1', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'd1',
    'd2', 'd3', 'd4', 'd5', 'd6', 'd7', 'd8', 'e1', 'e2', 'e3', 'e4', 'e5', 'e6',
    'e7', 'e8', 'f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7', 'f8', 'g1', 'g2', 'g3',
    'g4', 'g5', 'g6', 'g7', 'g8', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'h8']
```

mit Listen-Abstraktion:

listings/v4\_list9.py

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

print(felder)
# Ausgabe: ['a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'a7', 'a8', 'b1', 'b2', 'b3', 'b4',
    'b5', 'b6', 'b7', 'b8', 'c1', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'd1',
    'd2', 'd3', 'd4', 'd5', 'd6', 'd7', 'd8', 'e1', 'e2', 'e3', 'e4', 'e5', 'e6',
    'e7', 'e8', 'f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7', 'f8', 'g1', 'g2', 'g3',
    'g4', 'g5', 'g6', 'g7', 'g8', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'h8']
```

## 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)

print(menge)
# Ausgabe: set([0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 20, 25])
```

mit Mengen-Abstraktion:

listings/v4\_list11.py

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

print(menge)
# Ausgabe: set([0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 20, 25])
```

## 9 Iteratoren und Generatoren

- 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
```

Das Container-Objekt muss die `__iter__()`-Funktion implementieren:

listings/v4\_iter2.py

```
print('__iter__():', hasattr(liste, '__iter__'))
# Ausgabe: ('__iter__():', True)
```

Iterator aus Liste erzeugen:

listings/v4\_iter3.py

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

Ein Iterator muss auch die `__next__()`-Funktion implementieren:

listings/v4\_iter4.py

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

Das nächste Element kann mit `next()` extrahiert werden:

listings/v4\_iter5.py

```
next(iterator)
# Ausgabe: 1
next(iterator)
# Ausgabe: 2
next(iterator)
# Ausgabe: 3
```

... bis kein Element drin ist => `StopIteration`-Exception

listings/v4\_iter6.py

```
next(iterator)
# Ausgabe:
# -----
# StopIteration                                Traceback (most recent call last)
# <ipython-input-8-4ce711c44abc> in <module>()
# ----> 1 next(iterator)
#
# StopIteration:
```

## 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:

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'>
```

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\_iter8.py

```
next(f)
# Ausgabe: 1
for n in range(10):
    print(next(f))
# Ausgabe:
```

```
# 1
# 2
# 3
# 5
# 8
# 13
# 21
# 34
# 55
# 89
```

### 9.2.1 Generator-Expression

Ein Generator kann auch mit einem Ausdruck definiert werden:

listings/v4\_iter9.py

```
gen = (i*i for i in range(1, 10)) # wie List Comprehension, aber mit runden
    Klammern
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:
        wert = yield n # next() liefert None zurueck, send(x) liefert x zurueck
        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

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

### 10.1 Tupel

Leeres Tupel:



listings/v4\_tupel1.py

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

Tupel mit einem Element:

listings/v4\_tupel2.py

```
t = (5,)
print(type(t))
# Ausgabe: <type 'tuple'>
```

Mehrfachzuweisung:

listings/v4\_tupel3.py

```
x, y, z = 1, 2, 3
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 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]

```

### 10.2.3 Elemente ersetzen

listings/v4\_tupel10.py

```

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

```

Einzelnes Element:

listings/v4\_tupel11.py

```

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

```

Einen ganzen Bereich:

listings/v4\_tupel12.py

```

liste[3:] = ['D', 'E']
liste
# Ausgabe: ['a', 'B', 'c', 'D', 'E']

```

### 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']

```

## 10.3 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)
sortiert           # Ausgabe: [3, 4, 5]

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

```

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

listings/v4\_tupel15.py

```

liste.sort()
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('Liste:', liste)           # Ausgabe: ('Liste:', [2, 5, 3, 4, 1])
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)
# Ausgabe: ['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))
# Ausgabe: [('c', 1), ('b', 1), ('a', 2), ('a', 3)]
sorted(liste, key=lambda x: x[1])
# Ausgabe: [('c', 1), ('b', 1), ('a', 2), ('a', 3)]
sorted(liste) # zuerst nach dem ersten Unterelement sortieren, dann nach dem
              zweiten, ...
# Ausgabe: [('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=5)
for n in range(10):
    endlich_lang.append(n)
    print(list(endlich_lang))
# Ausgabe:
# [0]
# [0, 1]
# [0, 1, 2]
# [0, 1, 2, 3]
# [0, 1, 2, 3, 4]
# [1, 2, 3, 4, 5]
# [2, 3, 4, 5, 6]
# [3, 4, 5, 6, 7]
# [4, 5, 6, 7, 8]
# [5, 6, 7, 8, 9]
```

## 11 lambda, map, filter und reduce

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

### 11.1 lambda

Mit lambda können anonyme Funktionen definiert werden.

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, 3], [10, 20, 30]))
# Ausgabe: [11, 22, 33]
```

### 11.3 filter

sequenz = filter(funktion, sequenz)

Wendet die Funktion auf alle Elemente der Sequenz an und gibt nur diejenigen 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  
  
# (((10 + 20)/2 + 30)/2 + 40)/2  
reduce(lambda x, y: (x + y)/2, [10, 20, 30, 40]) # Ausgabe: 31
```

# Lektion 5: Reguläre Ausdrücke

- 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 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:
  - `[A-Fa-f\d]` passt auf eine Hexadezimalziffer
  - `[\s,.]` passt auf ein Leerzeichen, Komma oder Punkt

## 13 Wiederholungen (Quantoren)

- 0 oder mehrmals
  - \* `ca*t` passt auf ct, cat, caat, ...
  - `a[0-9]*b` passt auf ab, a538b, a0b, ...
- 1 oder mehrmals
  - + `ca+t` passt nicht auf ct, aber cat, caat, ...
- 0 oder 1-mal
  - ? `10k?m` passt auf 10m oder 10km
- m bis n-mal
  - {m,n} `ab2,3c` passt auf abbc oder abbbc
  - {3} → genau 3-mal
  - {3,} → mindestens 3-mal

- Gierigkeit deaktivieren  
? macht die obigen Wiederholungen nicht-gierig, z.B. <.+?>

## 14 Übereinstimmungen

- Funktionen, die Übereinstimmungen liefern:
  - `re.match()` Prüft, ob die RA am Stringanfang passt.  
Gibt entweder None oder eine `match`-Objekt zurück.
  - `re.search()` Sucht erstes Auftreten vom RA im String.  
Gibt entweder None oder ein `match`-Objekt zurück.
  - `re.findall()` Findet alle Teilstrings, die mit dem RA passen.  
Gibt eine Liste mit allen Teilstrings zurück.
  - `re.finditer()` Findet alle Teilstrings, die mit dem RA passen.  
Gibt einen Iterator zurück, der `match`-Objekte liefert.

### 14.1 match-Objekt

- Memberfunktionen eines `match`-Objekts:
  - `group()` Gibt den Teilstring zurück, der mit dem RA passt.
  - `start()` Gibt die Startposition des Teilstrings zurück.
  - `end()` Gibt die Endposition des Teilstrings zurück.
  - `span()` Gibt ein Tupel mit (`start`, `end`) zurück

### 14.2 Übereinstimmungen finden

re-Modul importieren:

listings/v5\_ra1.py

```
import re
```

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

listings/v5\_ra2.py

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

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

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

listings/v5\_ra3.py

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

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



## 15 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-Modul importieren:

listings/v5\_ra1.py

```
import re
```

`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,Beispiel.')
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)
```

## 16 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 (?:...)

re-Modul importieren

listings/v5\_ra1.py

```
import re
```

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+)\_([a-z]+)', '123_hallo_welt!')
if m is not None:
    print('groups():', m.groups())
    print('group(0):', m.group(0))
    print('group(1):', m.group(1))
    print('group(2):', m.group(2))
else:
    print('keine_Uebereinstimmung')
# Ausgabe:
# groups(): ('123', 'hallo')
# group(0): 123 hallo
# group(1): 123
# group(2): hallo
```

Mit benannten Gruppen:

listings/v5\_ra12.py

```
m = re.search(r'(?P<zahl>\d+)\_(?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 und Ort=Rappi')
print(liste) # Ausgabe: [('Jahrgang', '1930'), ('Name', 'Hans'), ('Ort', 'Rappi')]
liste = re.findall(r'Ort= (\w+)', 'Jahrgang=1930, Name=Hans und 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
```

## 16.1 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

re-Modul importieren

listings/v5\_ra1.py

```
import re
```

Entweder...oder...

listings/v5\_ra17.py

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

Anfang des Strings

listings/v5\_ra18.py

```
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

listings/v5\_ra19.py

```
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

listings/v5\_ra20.py

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

## 16.2 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

Positive, vorausschauende Annahme

Nach dem Wort muss ".doc" folgen:

listings/v5\_ra21.py

```
re.findall(r'\w+(?=\.doc)', 'bericht.doc_dokument.doc')
# Ausgabe: ['bericht', 'dokument']
```

Negative, vorausschauende Annahme

Nach den Buchstaben dürfen nicht Ziffern folgen:

listings/v5\_ra22.py

```
re.findall(r'[A-Za-z]+(?!\d+)\b', 'abc123_cde')
# Ausgabe: ['cde']
```

Positive, nach hinten schauende Annahme

Vor den Ziffern muss ein #-Zeichen vorausgehen:

listings/v5\_ra23.py

```
re.findall(r'(?<=#)\d+', '#10, #25, 66')
# Ausgabe: ['10', '25']
```

Negative, nach hinten schauende Annahme

Vor den Ziffern darf kein #-Zeichen vorausgehen:

listings/v5\_ra24.py

```
re.findall(r'\b(?<!\#)\d+', '#10, #25, 66')
# Ausgabe: ['66']
```

# Lektion 6: Klassen

## 17 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)
```

### 17.1 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)
```

### 17.2 Klasse instanzieren

listings/v6\_klassen4.py

```
objekt = MeineKlasse()
```

### 17.3 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
```

## 17.4 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.

#### 17.4.1 \_\_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.

#### 17.4.2 \_\_del\_\_()-Methode

Sie wird aufgerufen, bevor die Instanz zerstört wird.

listings/v6\_klassen11.py

```
del s # löscht 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.

### 17.4.3 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 0x0000010B581D24A8>>
```

Grundsätzlich entspricht dies dem folgenden Aufruf:

listings/v6\_klassen14.py

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

### 17.4.4 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
```

### 17.4.5 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
```



## 17.5 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()
```

### 17.5.1 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.
```

### 17.5.2 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.
```

### 17.5.3 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__', '__getattr__', '__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.

```

#### 17.5.4 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: 1000000
```

**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 = 1000000 # Ausgabe: Das Guthaben wurde auf 1000000 geändert.
print(k.guthaben) # Ausgabe: Das Guthaben wurde abgefragt.
# Ausgabe: 1000000
```

**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 = 1000000 # Ausgabe: Das Guthaben wurde auf 1000000 geändert.
print(k.guthaben) # Ausgabe: Das Guthaben wurde abgefragt.
# Ausgabe: 1000000
```

## 17.6 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.

### 17.6.1 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
```

### 17.6.2 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
```

## 17.7 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 geaendert.
# Das Guthaben wurde abgefragt.
# 1000

```

## 17.8 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>

### 17.8.1 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

```

### 17.8.2 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*

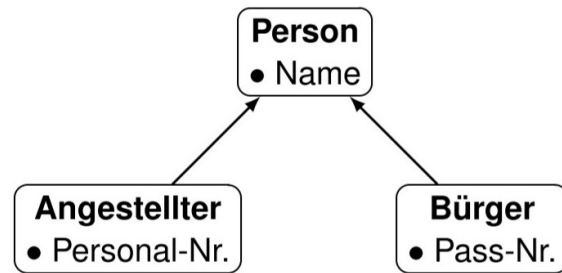
from my_other_module import Bank # Ausgabe:
# Dies ist scripts\my_other_module.py:
# __name__ = my_other_module
b = Bank()
b.guthaben = 500.0 # Ausgabe: Das Guthaben wurde auf 500.0 geaendert.

```

# Lektion 7: Vererbungen und Mehrfachvererbungen

## 18 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
```

## 18.1 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
```



## 18.2 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'
```

## 19 Mehrfachvererbung

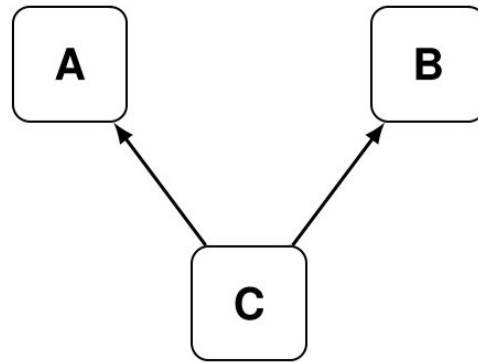
- Eine Subklasse kann von mehreren Superklassen erben:

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

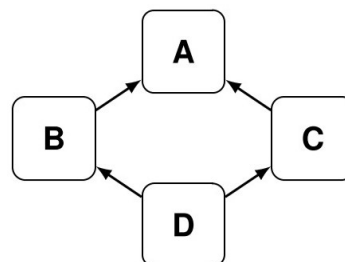
```
class Fahrzeug:
    def __init__(self, antrieb, **kwargs):
        print('Fahrzeug.__init__()', 'kwargs=', kwargs)
        super().__init__(**kwargs)
        self.antrieb = antrieb

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](https://en.wikipedia.org/wiki/C3_linearization))
- Diamond-Problem ist kein Problem mit `super()`



### 19.0.1 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]
```