

Python

N. Kaelin

24. März 2019

Inhaltsverzeichnis

1	Datentypen	4
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>	6
1.2	Sequentielle Datentypen	6
1.3	Assoziative Datentypen	6
1.4	Mengen	7
2	Verzweigungen	8
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
3	Schleifen	9
3.1	<code>while</code>	9
3.1.1	Durchlauf beenden und zurück nach oben	9
3.1.2	<code>while</code> -Schleife abbrechen	9
3.1.3	<code>else</code> -Teil	9
3.2	<code>for</code>	10
3.2.1	<code>else</code> -Teil	10
4	Funktionen	10
4.1	Funktionsdefinition	10
4.2	Aufruf	11
4.3	Weiteres	11
4.3.1	Standardwert für Parameter	11
4.3.2	Mehrere Rückgabewerte	11
4.3.3	Variable Anzahl von Argumenten	11
4.3.4	Argumente entpacken	12
4.3.5	Beliebige Schlüsselwort-Parameter	12
4.3.6	Schlüsselwortparameter entpacken	12
4.3.7	Globale Variablen	12
4.3.8	Docstring - Funktion dokumentieren	13
4.3.9	Call-by-object-reference	13
5	Exceptions	14
5.1	Unspezifische Exceptions abfangen	14
5.2	Abfangen mehrerer Exceptions	14
5.3	<code>else</code> -Teil	15
5.4	<code>finally</code> -Teil	15
5.5	Exceptions generieren	15

6	Dateien	15
6.1	Datei öffnen	15
6.2	Dateien lesen und schreiben	15
6.2.1	Datei lesen	16
6.2.2	Datei schreiben	16
6.2.3	glob	16
6.2.4	os.path	17
7	Strings	17
7.1	Stringformatierung	17
7.1.1	im C-Stil (à la printf)	17
7.1.2	mit format()	18
7.1.3	mit Stringlitterale	18
7.1.4	mit string-Methoden	19
7.2	Alles über Strings <small>Kap. 19</small>	19
7.2.1	Strings aufspalten	19
7.2.2	Strings kombinieren	20
7.2.3	Suchen von Teilstrings	20
7.2.4	Ersetzen von Teilstrings	20
7.2.5	Strings bereinigen	20
7.2.6	Klein- und Grossbuchstaben	21
7.2.7	Strings testen	21
8	Listen-Abstraktion/List-Comprehension	22
8.1	Neue Liste aus einer bestehenden Liste ableiten	22
8.1.1	Beispiel 1	22
8.1.2	Beispiel 2	22
8.2	Bestehende Liste filtern	22
8.3	Liste von Zahlen => formatierter String	23
8.4	Liste der Schachbrettfelder	23
8.5	Mengen-Abstraktion/Set Comprehension	24
8.5.1	Produkte zweier Zahlen	24
9	Iteratoren und Generatoren	24
9.1	Iteratoren	24
9.2	Generatoren	25
9.2.1	Generator-Expression	26
9.2.2	send()-Methode, Generator als Coroutine	26
10	Listen und Tupel im Detail	26
10.1	Tupel	26
10.2	Listen	27
10.2.1	Element hinzufügen	27
10.2.2	Mehrere Elemente hinzufügen	27
10.2.3	Elemente ersetzen	28
10.2.4	Element entfernen	28
10.3	Sortieren	28
10.3.1	Umgekehrte Reihenfolge	29
10.3.2	Mit spezieller Funktion	29
10.3.3	collections.deque	29
11	lambda, map, filter und reduce	30
11.1	lambda	30
11.2	map	30
11.3	filter	31
11.4	reduce	31

12 Zeichen-Klassen	32
13 Wiederholungen (Quantoren)	32
14 Übereinstimmungen	33
14.1 match-Objekt	33
14.2 Übereinstimmungen finden	33
15 Modifizierungen	35
16 Gruppierung	36
16.1 Weitere Metazeichen	37
16.2 Look-around Assertions	38

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!

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

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

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

²Nicht definiert für den Datentyp `complex`

1.1.3 Bitweise Operatoren für den Datentypen `int`

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`

Methode	Beschreibung
<code>x.real</code>	Realteil von <code>x</code> als Gleitkommazahl
<code>x.imag</code>	Imaginärteil von <code>x</code> als Gleitkommazahl
<code>x.conjugate()</code>	Liefert die zu <code>x</code> konjugiert komplexe Zahl

1.2 Sequentielle Datentypen Kap. 5

- `str`
- `list`
- `tuple`
- `bytes`
- `bytearray`

Die folgenden Operatoren sind für **alle** sequentiellen Datentypen definiert:

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`

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.

Operator	Beschreibung
<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.

Operator	Beschreibung
<code>s.add(el)</code>	Fügt ein neues unveränderliches Element (<code>el</code>) 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 <code>s</code> wird von <code>y</code> 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 <code>s</code> gespeichert.
<code>s.discard(el)</code>	Das Element <code>el</code> wird aus der Menge <code>s</code> entfernt.
<code>s.remove(el)</code>	Gleich wie <code>s.discard(el)</code> nur gibt es hier einen Fehler falls <code>el</code> nicht in <code>s</code> .
<code>s.intersection(y)</code>	Liefert die Schnittmenge <code>s</code> und <code>y</code> .
<code>s.isdisjoint(y)</code>	Liefert True falls Schnittmenge von <code>s</code> und <code>y</code> 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

2.1 if

listings/v2_if1.py

```
if Bedingung:
    Anweisung1
    Anweisung2
```

Anweisungen 1 & 2 nur ausführen, wenn die Bedingung **wahr** ist.

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
    Anweisung2
else:
    Anweisung3
    Anweisung4
```

- Anweisungen 1 & 2, falls Bedingung **wahr**
- Anweisungen 3 & 4, falls Bedingung **unwahr**

Für jeden Datentyp gibt es einen Wert, der als **unwahr** gilt:

Datentyp	False-Wert
NoneType	None
int	0
float	0.0
bool	False
complex	0 + 0j
str	" oder ""(leerer String)
list	[]
tuple	()
bytes	b"
bytearray	bytearray(b")
dict	{}
set	set()
frozenset	frozenset()

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
- Einrücken des Codeblocks

3.1.1 Durchlauf beenden und zurück nach oben

Achtung: Python kennt keine do-while-Schleife.

listings/v2_while2.py

```
while Bedingung:
    Anweisung1
    if Ausnahme:
        continue
    Anweisung2
```

continue beendet den aktuellen Durchlauf und springt nach oben.

3.1.2 while-Schleife abbrechen

listings/v2_while3.py

```
while Bedingung:
    Anweisung1
    if Fehler:
        break
```

break bricht die while-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

3.2.1 else-Teil

listings/v2_for2.py

```
for Variable in Sequenz:
    Anweisung1
else:
    Anweisung2
```

else-Teil wie bei der 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.

listings/v2_func4.py

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

return-Anweisung beendet den Funktionsaufruf mit Rückgabewert:

listings/v2_func5.py

```
def summe(a, b):
    return a + b
```

- leere return-Anweisung liefert None zurück
- 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 namenlosen
```

4.3 Weiteres

4.3.1 Standardwert für Parameter

listings/v2_func7.py

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

```
rosen() # Aufruf 1
rosen('gelb') # Aufruf 2
```

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

- 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 Abfangen mehrerer Exceptions

listings/v3_exception4.py

```
eingabe = '10Fr.'
try:
    x = int(eingabe)
    y = 1/x
except ValueError as e:
    print('Oops!' + str(e))
except ZeroDivisionError as e:
    print('Oops!' + str(e))
```

mehrfache Ausnahmen gruppieren:

listings/v3_exception5.py

```
eingabe = '10Fr.'
try:
    x = int(eingabe)
    y = 1/x
except (ValueError, ZeroDivisionError):
    print('Oops! Bitte wiederholen.')
```

5.3 else-Teil

listings/v3_exception6.py

```
try:
    f = open('datei.txt')
except IOError:
    print('Kann_Datei_nicht_oeffnen.')
else:
    print('Datei_schliessen.')
    f.close()
print('Ende')
```

5.4 finally-Teil

listings/v3_exception7.py

```
try:
    welt_rennen()
finally:
    print('Dinge,_die_so_oder_so_gemacht_werden_muessen.')
```

5.5 Exceptions generieren

listings/v3_exception8.py

```
raise ValueError('Falscher_Wert.')
```

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

- Datei lesen:

listings/v3_datei3.py

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

- Datei schreiben:

listings/v3_datei4.py

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

- Datei schliessen:

listings/v3_datei5.py

```
f.close()
```

6.2.1 Datei lesen

- mit read()

listings/v3_datei6.py

```
f = open('mailaenderli.txt')
text = f.read()
f.close()
print(text)
```

- besser mit der with-Anweisung

listings/v3_datei7.py

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

- Variante mit readlines()

listings/v3_datei8.py

```
with open('mailaenderli.txt') as f:
    zeilen = f.readlines()
print(zeilen)
for zeile in zeilen:
    print(zeile.strip())
```

6.2.2 Datei schreiben

listings/v3_datei9.py

```
personen = ['Alice', 'Bob', 'Charlie']
with open('rangliste.txt', 'w') as f:
    for n, person in enumerate(personen, start=1):
        f.write(str(n) + '. ' + person + '\n')

# Ueberpruefen
with open('rangliste.txt') as f:
    print(f.read())

# Ausgabe: 1. Alice
# Ausgabe: 2. Bob
# Ausgabe: 3. Charlie
```

6.2.3 glob

listings/v3_datei10.py

```
import glob
glob.glob('*.*ipynb')
```

6.2.4 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)
# Ausgabe: True

os.path.isdir(full_path)
# Ausgabe: False

os.path.getsize(full_path)
os.path.split(full_path)
os.path.splitext(full_path)
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=_{}, _I=_{}'.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}, _U=_{:0.f}'.format(spannung)
# Ausgabe: 'U = 12.56, U = 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 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

- 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\nnes
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: 'password'

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

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 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  
  
# (((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 (?:...)
- (?:ab) passt auf ab, Gruppe wird nicht hinterlegt

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+)\s*([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+)\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
```

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']
```