
Skriptsprache Python

Teil 1: Arbeiten mit Python 3

Überblick zu Python

- Python ist ...
 - eine objektorientierte, interaktive Programmiersprache mit dynamischer Typisierung
 - Einfach zu erlernen
 - Zunehmend populärer
- Die Programmiersprache Python wurde von Guido van Rossum (GvR) entwickelt, einem niederländischen Mathematiker und Informatiker
- GvR soll sich angeblich über Weihnachten 1989 gelangweilt und deshalb Python auf Basis einer älteren Programmiersprache namens ABC entwickelt haben

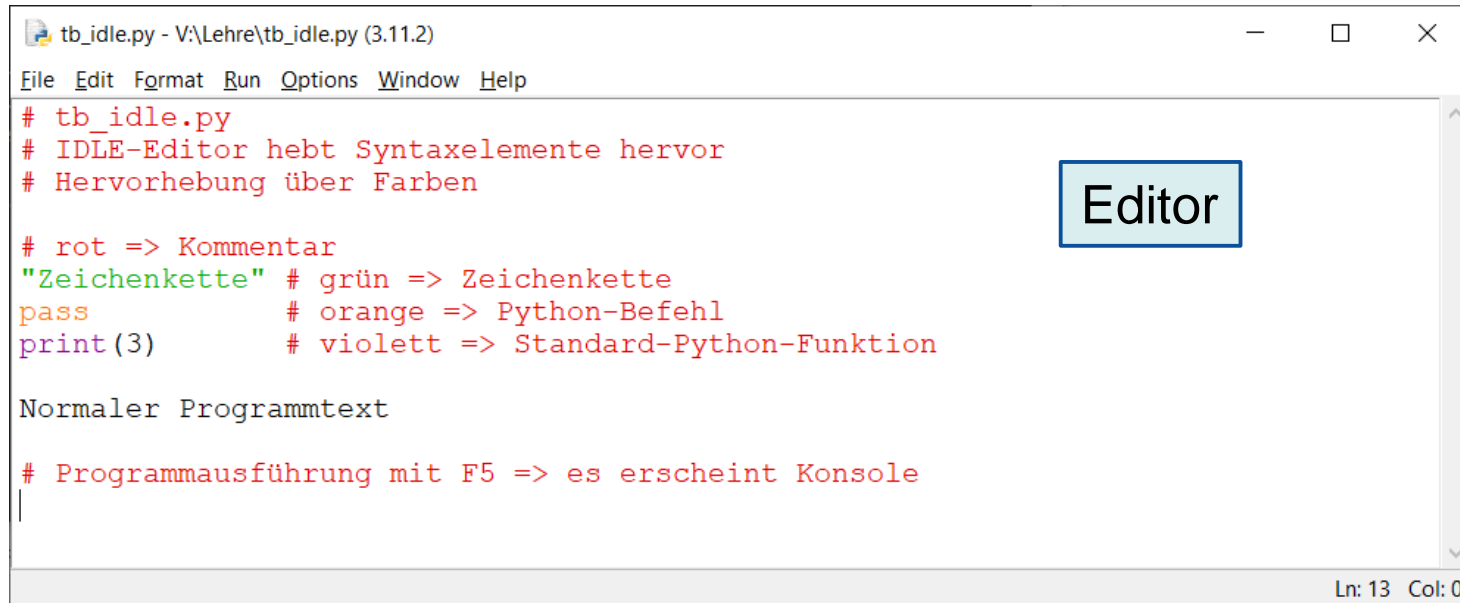
Python-Sprachstandard und Implementierungen

- Den Sprachumfang von Python legt die **Python Software Foundation** fest (PSF → Organisation in Delaware, USA)
- Anders als z.B. C/C++, C# oder JavaScript wird die Sprache also nicht von einer industriellen Standardisierungsorganisation wie ISO oder ECMA gepflegt
- Von der PSF stammt **CPython**, die am häufigsten verwendete Implementierung von Python
- CPython ist Open-Source-Software mit einer nur geringfügig einschränkenden Lizenz, die mit der *GNU Public License* kompatibel ist
- Neben CPython gibt es auch andere Implementierungen, die z.B. einen Just-In-Time-Compiler (pypy) enthalten, in der Dotnet-Umgebung laufen (Ironpython) oder in der Java-VM (jython), usw ...

Popularität von Python

- Diverse Benchmarks (TIOBE, Redmonk, ...) sehen Python unter den fünf populärsten Programmiersprachen neben Java, C/C++, usw.
- Programmpakete wie Blender, Inkscape, Gimp, oder Cinema4D nutzen Python für Skripterweiterungen
- Komplexe Softwarepakete wie die E-Book-Software Calibre oder das Buildtool SCons sind gleich ganz in Python geschrieben
- Auf Github wird Python sehr häufig für Projekte genutzt
- Auch in der serverseitigen Webentwicklung (Django) und bei KI-Frameworks (z.B. TensorFlow von Google) ist Python sehr populär

IDLE-Editor und Shell



tb_idle.py - V:\Lehre\tb_idle.py (3.11.2)

File Edit Format Run Options Window Help

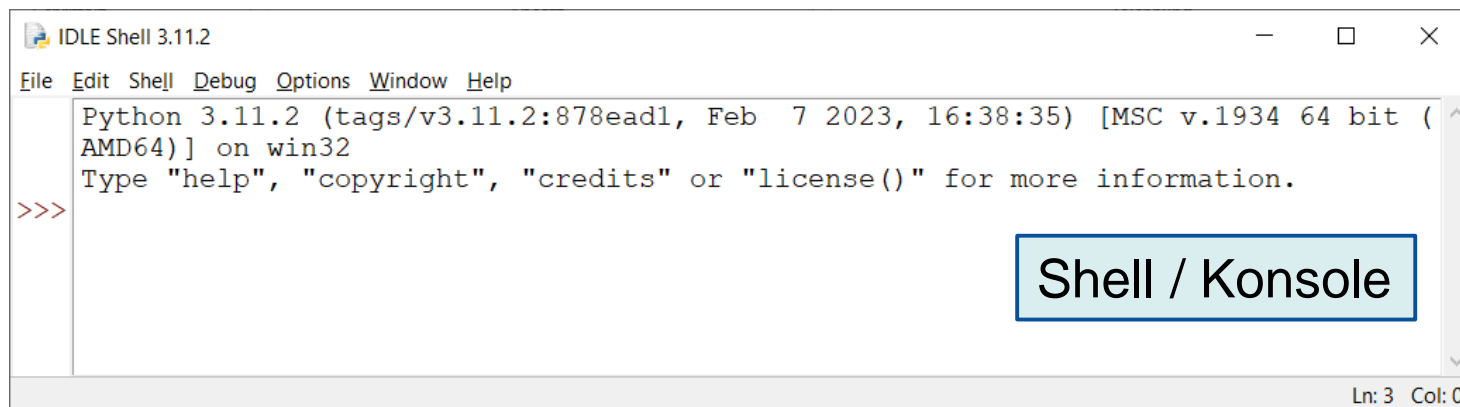
```
# tb_idle.py
# IDLE-Editor hebt Syntaxelemente hervor
# Hervorhebung über Farben

# rot => Kommentar
"Zeichenkette" # grün => Zeichenkette
pass           # orange => Python-Befehl
print(3)        # violett => Standard-Python-Funktion

Normaler Programmtext

# Programmausführung mit F5 => es erscheint Konsole
|
```

Ln: 13 Col: 0



IDLE Shell 3.11.2

File Edit Shell Debug Options Window Help

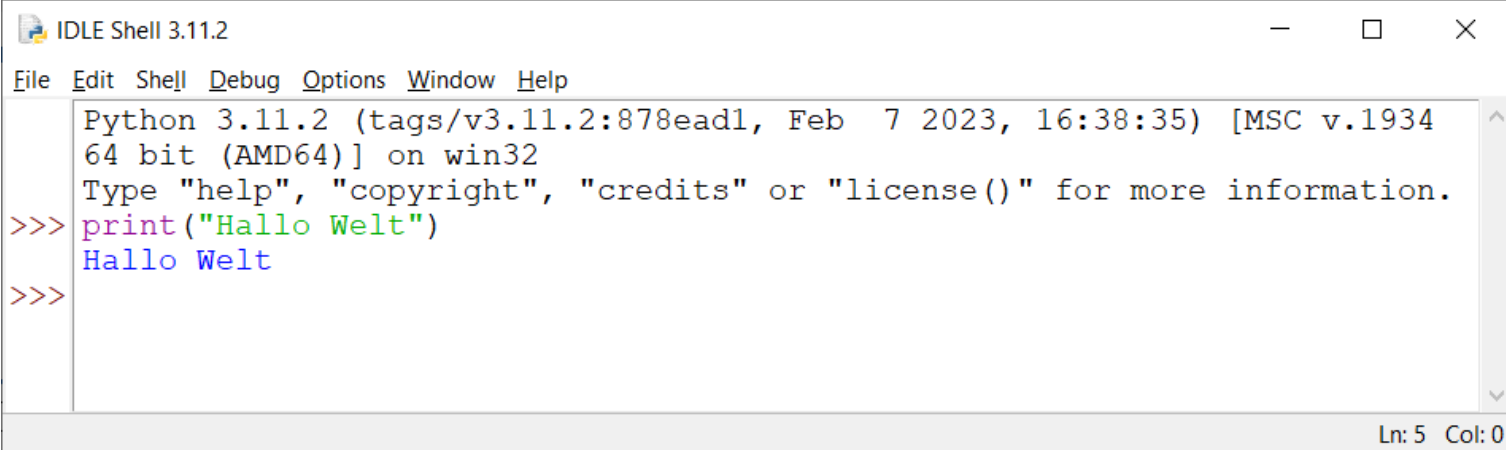
```
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Ln: 3 Col: 0

Erste Schritte mit der interaktiven Python-Shell

Arbeiten in der interaktiven **Python-Shell**

→ in IDLE z.B. Menü Run → Python Shell



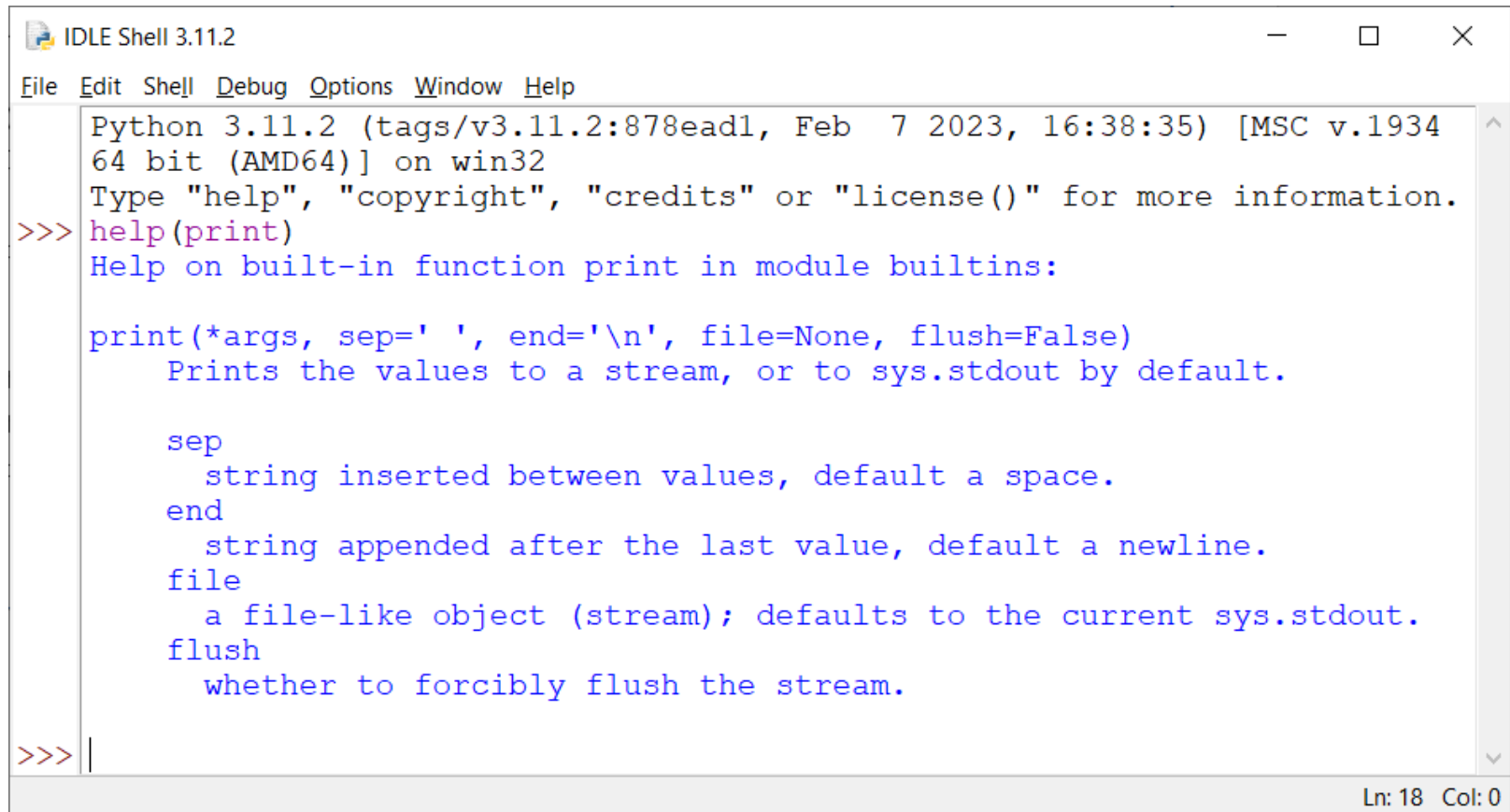
```
Python 3.11.2 (tags/v3.11.2:878ead1, Feb  7 2023, 16:38:35) [MSC v.1934
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hallo Welt")
Hallo Welt
>>>
```

Ln: 5 Col: 0

→ Beenden der Shell mit STRG-D oder „`exit()`“

→ Python unter Windows im DOS-Fenster: Beenden mit STRG-Z

Python-Hilfe verwenden mit help-Funktion



```
IDLE Shell 3.11.2
File Edit Shell Debug Options Window Help
Python 3.11.2 (tags/v3.11.2:878ead1, Feb  7 2023, 16:38:35) [MSC v.1934
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> help(print)
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
        string inserted between values, default a space.
    end
        string appended after the last value, default a newline.
    file
        a file-like object (stream); defaults to the current sys.stdout.
    flush
        whether to forcibly flush the stream.

>>> |
```

Ln: 18 Col: 0

C/C++

```
// cprog1.c
#include <stdio.h>

int lshift(int num, int count) {
    if (count < 1) return num;
    return num << count;
}

int main() {
    int v1=2, v2=3;
    int res = lshift(v1, v2);
    // Ausgabe: lshift=16
    printf("lshift=%d\n", res);
    return 0;
}
```

Python

```
# pyprog1.py

def lshift(num, count):
    if count < 1: return num
    return num << count

# Hauptprogramm
v1=2
v2=3
res=lshift(v1, v2)
# Ausgabe: lshift=16
print("lshift=%d" % res)
```

Vergleich C/C++ mit Python

C/C++

Python

Kommentar

```
// cprog1.c
```

```
# pyprog1.py
```

```
#include <stdio.h>
```

Funktion

```
int lshift(int num, int count) {  
    if (count < 1) return num;  
    return num << count;  
}
```

```
def lshift(num, count):  
    if count < 1: return num  
    return num << count
```

Hauptprogramm

```
int main() {  
    int v1=2, v2=3;  
    int res = lshift(v1, v2);  
    // Ausgabe: lshift=16  
    printf("lshift=%d\n", res);  
    return 0;  
}
```

```
# Hauptprogramm  
v1=2  
v2=3  
res=lshift(v1, v2)  
# Ausgabe: lshift=16  
print("lshift=%d" % res)
```

Über das Beispielprogramm ...

- Kommentare von Raute (#) bis Zeilenende
- Variablen existieren ab erster Zuweisung eines Wertes
- Variablenzuweisung über `<Name> = <Ausdruck>`
- Vergleiche über `<Ausdruck-1> <Op> <Ausdruck-2>`
- Viele Python-Operatoren aus C/C++ bekannt
- Eingabe / Ausgabe über Funktionen `input()` und `print()`
- Funktionsdefinition mit `def` und Werterückgabe mit `return`
- Ablaufsteuerung mit `if / else`, `while`, ...
- Blockzugehörigkeit von Anweisungen über Einrücktiefe bestimmt
 - Anweisungen im `def`-Block mit bestimmter Einrücktiefe
 - Anweisungen im `while`-Block mit größerer Einrücktiefe
 - Untergeordnete Anweisungsblöcke werden durch einen Doppelpunkt eingeführt (:)
- Hauptprogramm: Anweisungen ohne Einrückung

Blockbildung durch Doppelpunkt und Einrücktiefe

Kurzer Block: Rest der
Zeile nach Doppelpunkt

```
def lshift(num, count):  
    if count < 1: return num  
    return num<<count
```

Längerer Block: Folgezeilen nach
Doppelpunkt gleich eingerückt
(durch ____ hervorgehoben)

```
def lshift(num, count):  
    ____if count < 1: return num  
    ____return num<<count
```

Hauptprogramm:
Zeilen ohne Einrückung

```
v1=2  
v2=3  
res=lshift(v1, v2)  
print("lshift=%d" % res)
```

Vergleich C mit Python: Eingabe, if-Abfrage

C/C++

```
// cprog2.c
#include <stdio.h>

int main() {
    int a;
    printf("zahl> ");
    scanf("%d", &a);
    if (a > 0)
        printf("positiv\n");
    else if (a < 0)
        printf("negativ\n");
    else
        printf("null\n");

    return 0;
}
```

Python

```
# pyprog2.py

# String mit input() einlesen
# Umwandlung nach Zahl mit int()

a = int(input("zahl> "))

if a > 0:
    print("positiv")
elif a < 0:
    print("negativ")
else:
    print("null")
```

Vergleich C mit Python: while-Schleife

C/C++

```
// cprog3.c
#include <stdio.h>

int main() {
    int summe = 0;
    int anz = 0;

    while (summe < 20) {
        anz++;
        summe += anz;
    }
    printf("anz = %d\n", anz);
    printf("summe = %d\n", summe);
    return 0;
}
```

Python

```
# pyprog3.py

summe = 0
anz = 0

while summe < 20:
    anz += 1
    summe += anz

print("anz = %d" % anz)
print("summe = %d" % summe)
```

Vergleich C/C++ mit Python: Arrays

C/C++

```
// cprog4.c
#include <stdio.h>

int main() {
    int feld[] = {1, 2, 3, 4, 5};
    int sz = sizeof(feld)/sizeof(int);

    int idx;
    for (idx = 0; idx < sz; idx++) {
        printf("elem = %d\n", feld[idx]);
    }
    return 0;
}
```

Python

```
# pyprog4.py

# Py-Liste => ähnlich C-Array
feld = [1,2,3,4,5]

# For-Schleife über feld
for elem in feld:
    print("elem = %d" % elem)
```

Vergleich C++ mit Python: Arrays

C++

```
#include <iostream>

using namespace std;
int main() {
    int feld[] = {1,2,3,4,5};
    for (auto elem: feld)
        cout << "elem = " << elem << endl;
    return 0;
}
```

Python

```
# pyprog4.py

# Py-Liste => ähnlich C-Array
feld = [1,2,3,4,5]

# For-Schleife über feld
for elem in feld:
    print("elem = %d" % elem)
```


Python-Programmierung mit IDE, z.B. IDLE

- IDLE = **I**ntegrated **D**evelopment and **L**earning **E**nvironment
- Einfache Grafikumgebung zur Entwicklung von Python-Programmen
- Besteht aus:
 - **E**ditor mit automatischer Einrückung und Syntaxhervorhebung
 - Konsole (= **S**hell) zur Programmausführung

Variablen und grundlegende Datentypen

Vergleich: Variablen in C/C++ und Python

```
// Variablendekl. in C
int a = 3;
char *s = "Hallo";
double fval = 3.4;
a = s; // in C Fehler
```



```
# Variablendekl. in Python
a = 3
s = "Hallo"
fval = 3.4
a = s # Typwechsel okay
```

```
# in Python auch Zuweisung mit Typangabe möglich
a = int(3.14)           # Ganzzahl => 3
b = float(3)            # Gleitkommazahl => 3.0
c = str(3)              # Zeichenkette "3"
d = int("1101", 2)      # Ganzzahl aus Zeichenkette binär
e = str(d)              # Zeichenkette "13"
f = 3j                  # rein imaginäre Zahl
g = complex(2,3)        # komplexe Zahl (2+3j)
h = 2+3j                # komplexe Zahl (2+3j)
k = bool(0.5)           # True
l = bool(0)             # False
```

Variablen

- Variablen entstehen durch ihre erste Zuweisung
- Zuweisung mit *Name = Ausdruck*
- Der Variablentyp wird vom zugewiesenen Ausdruck festgelegt
- Weitere Zuweisungen mit anderem Datentyp sind auch möglich

```
>>> z=2                                # z erst Zahl,  
>>> z  
2  
>>> z="Hallo Python"                  # dann Zeichenkette  
>> z  
'Hallo Python'
```

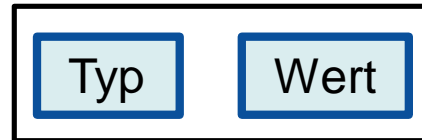
- Fehler bei Auswertung von Variablen vor ihrer ersten Zuweisung

```
>>> var1  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'var1' is not defined
```

- Werte/Konstanten in Python als sogenannte **Objekte** realisiert
- Alle Datentypen in Python werden über Objekte repräsentiert:
Zahlen, Zeichenketten, komplexe Strukturen, ...
- Durch eine Zuweisung zeigt eine Variable auf ein Objekt

Objekt

- Objekt = Datenstruktur mit Typ und Datenwert



- Variablenzuweisung: Variablenname zeigt auf ein Objekt

```
>>> z=2
```

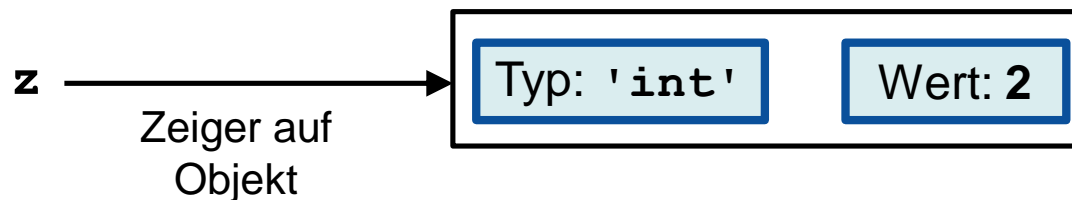
```
>>> z
```

```
2
```

```
>>> type(z)
```

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

← Typ eines Ausdrucks abfragen



Regeln für Variablennamen

- Namen enthalten Buchstaben, Unterstriche oder Ziffern
- Ziffern dürfen nicht am Anfang des Namens stehen
- Unterscheidung zwischen Groß- und Kleinbuchstaben
- Beispiele:

Ölstand, abc, Abc, Die_Straßennummer,
geschw_100, __interne_werte

- Python-Befehle als Variablennamen verboten, z.B.:

and, as, assert, break, class, continue, def, del,
elif, else, except, finally, for, from, global, if,
import, in, is, lambda, nonlocal, not, or, pass,
raise, return, try, yield, while, with

- Ganze Zahlen (Datentyp **int** in Python):
 - Dezimal: **-10, 75, +3**
 - Binär: **0b1100, 0B11**
 - Oktal: **0o0012, 0O17** (nicht erlaubt: 017)
 - Hexadezimal: **0xcafe, 0XAbba**
 - Keine Beschränkung des Wertebereichs, z.B. 2^{5000} als ganze Zahl möglich
- Gleitkommazahlen (Datentyp **float** in Python):
 - **1., 1.2, -3.4, 1e308**
 - Python-**float** entspricht 64-Bit-Gleitkommazahlen nach IEEE 754 (Datentyp **double** in C/C++)

Grundrechenarten in Python

```
>>> 5+2  
7
```

```
>>> 5*2  
10
```

```
>>> 5/2  
2.5
```

```
>>> 4/2  
2.0
```

```
>> 5%2  
1
```

```
>>> 5-2  
3
```

```
>>> 5**2  
25
```

```
>>> 5//2  
2
```

```
>>> 5.0//2  
2.0
```

```
>> 5.5%2  
1.5
```

******: Potenzieren, z.B. $5^2 = 5^{**}2$

//: „Floor“ Division mit Rundung auf Ganzzahl
Mit Integer-Zahlen → **Int**-Ergebnis
Mit Float-Zahl(en) → **Float**-Ergebnis

/: „Float“ Division: → **Float**-Ergebnis

%: Modulo-Operation

Beispiele zu Wertebereich für Ganzzahlen

```
>>> (2**32) * (2**32)
18446744073709551616
```

```
>>> 2**1027-1
143815450788985272618344415263121978689438158315384
525818744064926186140644400770506166781857926028816
896091103897114686127031815051533297994277944511579
299502214314739892388221041775680996875295562466361
668004615070520545873970305179130488432661789730680
408547669038591957796750783773043868285063699379309
7727
```

Wertebereich für Ganzzahlen **uneingeschränkt**
→ Unterschied zu Programmiersprachen wie C/C++

Beispiele zu Wertebereich bei Kommazahlen

```
>>> 2.**1023
8.98846567431158e+307
```

Bei Gleitkommazahlen etwa 16
Stellen Genauigkeit

```
>>> 1.7e308
1.7e308
```

Wertebereich ca. $2,2 \cdot 10^{-308}$ bis $1,8 \cdot 10^{308}$
→ Entspricht double-Zahlen in C
Bei Überlauf entweder **inf** oder **Fehler**

```
>>> 1.8e308
```

inf

```
>>> 2.**1024
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
OverflowError: (34, 'Result too large')
```

Datentyp bool: True / False

```
>>> 3>4
```

```
False
```

```
>>> (1<2) and (2<3)
```

```
True
```

```
>>> (1<2) and (3>4)
```

```
False
```

```
>>> 3*False
```

```
0
```

```
>>> not True
```

```
False
```

```
>>> not -1
```

```
False
```

```
>>> 1<2
```

```
True
```

```
>>> 1 < 2 < 3
```

```
True
```

```
>>> (1<2) or (3>4)
```

```
True
```

```
>>> 4*True
```

```
4
```

```
>>> not False
```

```
True
```

```
>>> not 0
```

```
True
```

Datentyp complex: Imaginäre / Komplexe Zahlen

```
>>> 1+2j
```

```
(1+2j)
```

Imaginäre Zahlen mit **Suffix j** (oder **J**)

```
>>> 2J
```

```
2j
```

```
>>> 2i
```

```
File "<stdin>", line 1
```

```
    2i
```

```
    ^
```

```
SyntaxError: invalid syntax
```

```
>>> 2*j
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'j' is not defined
```

```
>>> (1+0.5j).imag
```

```
0.5
```

```
>>> (1+0.5j).real
```

```
1.0
```

Zugriff auf Real- und Imaginärteil über
.real bzw .imag

Methoden / Attribute komplexer Zahlen

Methode (Z=Zahl)	Aufgabe
Z=complex(re=0, im=0)	Erzeuge komplexe Zahl $re+j \cdot im$
Z.conjugate()	Berechne konjugiert komplexe Zahl zu Z

Attribut (Z=Zahl)	Bedeutung
Z.imag	Imaginärteil von Z
Z.real	Realteil von Z

Weitere Infos über `help(complex)`, `help(complex.real)`, usw.

Der Punkt („.“) arbeitet als Operator für den Attributzugriff von Objekten

Datentyp `NoneType` (= Nichts)

- Der besondere Wert **None** vom Typ `NoneType` steht in Python für das Fehlen eines Wertes (= Nichts)
- Damit kann z.B. signalisiert werden, dass eine Variable noch keinen gültigen Wert hat
- Am Prompt gibt der Python-Interpreter deshalb auch nichts aus, wenn ein Ausdruck `None` als Ergebnis hat

- Beispiel:

```
>>> a=3*3
```

```
>>> a
```

```
9
```

```
>>> a=None
```

```
>>> a
```

```
>>> print(a)
```

```
None
```

Priorität von Operatoren

Operator	
Höchste Priorität ↑	<div>() (Klammerung)</div> <div>a**b (Potenz)</div> <div>+a, -b (Vorzeichen), ~a (bitweise Invert.)</div> <div>a * b, a / b, a % b, a // b</div> <div>a + b, a - b</div> <div>Arithmetische Operatoren</div>
	<div>a << b, a >> b</div> <div>a & b</div> <div>a ^ b</div> <div>a b</div> <div>Bitorientierte Operatoren</div>
	<div>a in b, a not in b, a is b, a is not b, a < b, ..., a == b, a != b</div> <div>not a</div> <div>a and b</div> <div>a or b</div> <div>Logische Operatoren</div>
Niedrigste Priorität ↓	<div>>>> help("+") ← mehr Info zu Operatoren</div>

Erweiterte Zuweisung an Variablen

- Zuweisung mit `<Name> = <Ausdruck>`
- Erweiterte Zuweisung mit `<Name> <Operator> = <Ausdruck>`
- Bedeutung:
 - **Erweiterte / zusammengesetzte** Operation
 - Variable mit **bestehendem** Wert über Operator verknüpfen
 - Ergebnis an Variable zuweisen
- Beispiele:
 - Inkrement: `var += 2`
 - Multiplikation: `var *= 41`

```
>>> z=2
>>> z*=2      # -> z=4, entspricht z=z * 2
>>> z|=1      # -> z=5, entspricht z=z | 1
```

Weitere Infos zu erweiterter Zuweisung
(engl. **augmented assignment**) → `help("+=")`

Zeichenketten (Strings)

Zeichenketten in Python

- **Zeichenkette:** Text zwischen Anführungszeichen `"..."` oder Hochkommas `'...'`
- Formen mit Anführungszeichen / Hochkommas gleichwertig

Anführungszeichen

Hochkomma

`"Hallo Python"` **==** `'Hallo Python'`

Erste Beispiele zu Zeichenkettenverwendung

- für Python sind Zeichenketten mit Hochkommas und Anführungszeichen gleichwertig
- Mischen von Anführungszeichen und Hochkommazeichen interessant
z.B. für skriptgesteuerte Generierung von Quellcode (C, Python, ...)

```
>>> "Hallo Python" == 'Hallo Python'  
True
```

```
>>> print('void show() { printf("hallo welt\\n"); }')  
void show() { printf("hallo welt\n");
```

```
>>> print("putchar('a');");  
putchar('a');
```

Sonderzeichen (sog. Escape-Sequenzen)

Zeichen	Bedeutung
<code>'\n'</code>	Sprung zur nächsten Zeile (n=newline)
<code>'\r'</code>	Sprung an Zeilenanfang (r=return)
<code>'\t'</code>	Tabulator-Zeichen (t=tab)
<code>'\"'</code>	Hochkomma (')
<code>'\"'</code>	Anführungszeichen (")
<code>'\\'</code>	Backslash-Zeichen
<code>'\x30'</code>	Hexcodiertes 8-Bit-Zeichen, z.B. 0x30 (= Ziffer 0)
<code>'\u0041'</code>	16-Bit-Unicode-Zeichen, z.B. 0x41 (= Buchstabe A)
Weitere ...	

```
>>> help("\\") ← mehr Info zu Escape-Sequenzen
```

Arithmetische Operatoren bei Zeichenketten

```
>>> 3 * "Abc"
'AbcAbcAbc'
```

Multiplikation mit Zahl → Wiederholung

```
>>> "Abc" + "123"
'Abc123'
```

Addition → Verkettung

```
>>> "Studying in %s" % "Germany"
'Studying in Germany'
```

Formatierung
→ Modulo-Operator

```
>>> z="123"
>>> z*=3
>>> z+="4"
>>> z
'1231231234'
```

Zusammengesetzte
Zuweisung

Operator zur Zeichenkettenformatierung

- Syntax: `<Formatierungstext> % <Parameter>`
- Angabe eines Formatierungstexts mit speziellen Ersetzungssymbolen, gefolgt von einem einzelnen Parameter oder einem Tupel mehrerer Parameter
- Ersetzungssymbole: %-Zeichen gefolgt von einem Symbolcode, wie bei printf-Funktion aus der C-Standardbibliothek
- Zur Textausgabe angegebene Parameter werden in den Formatierungstext eingesetzt

Formatierungsbeispiele

```
# Hexzahl in Text einfügen => "zahl 2a"
```

```
"zahl %02x" % 42
```

```
# Mehrere Zahlen in Text einfügen; Angabe als Tupel
```

```
# => "werte -21,42"
```

```
"werte %d,%d" % (-21, 42)
```

```
# Zahlenformatierung mit Angabe der gewünschten Ziffernzahl
```

```
# 08 => 8 Ziffern, führende 0
```

```
# "#" => Einleitungszeichen 0x / 0o mit dazu schreiben
```

```
# +12.3f => Zahl mit Vorzeichen und 12 Zeichen (3 Nachkomma)
```

```
# Beispiel ergibt "addr 0x000007ff => +214.455 %"
```

```
"addr %#08x => %+12.3f %" % (2047, 214.454545)
```

```
# Mischen von Datentypen => "len(boo)=3"
```

```
"len(%s)=%d" % ("boo", len("boo"))
```

```
# Verwenden eines Dictionarys => Zeichenkette name und Ganzzahl wert
```

```
"Name=%(name)s => Wert=%(wert)d" % { "name": "Hobbit", "wert": 103 }
```


Formatierungssymbole

Symbol	Bedeutung
%c	Zeichen für den angegebenen Unicode
%s	Zeichenkette für den angegebenen Ausdruck
%d	Ganzzahl mit Vorzeichen (auch %i und %u)
%o	Oktalzahl
%x	Hexzahl (%X für Ziffern als Großbuchstaben)
%e	Gleitkommazahl in Exponentialform (%E für Großbuchstaben)
%f	Gleitkommazahl
%g	%f bzw. %e (je nachdem, was kürzer ist) (%G für E-Form)
%%	Prozentzeichen

Modifikatoren für Formatierungssymbole

Symbol	Bedeutung
*	Weiterer Parameter gibt Anzahl der Zeichen oder Genauigkeit an
-	Einzusetzender Text ist linksjustiert
+	Vorzeichen immer mit ausgeben
<Leer>	Vor positiven Zahlen Leerzeichen schreiben <Leer>=Leerzeichen
#	Präfix für Oktalzahl / Hexzahl ausgeben
0	Von links führende Nullen voranstellen
%	Prozentzeichen darstellen (%%)
m.n	Zahlenausgabe: m=minimale Anzahl von Stellen, n=Anzahl Stellen nach dem Komma
(var)	Einsetzen einer Variable aus einem Dictionary (→ siehe später), das hinter dem Prozentzeichen angegeben wird

Erstellen von formatierten Text mit Z.format()

- Zeichenketten besitzen eine Methode `Z.format()`
- Mit `Z.format(...)` kann Text formatiert werden
 - `Z` ist dabei der Formatierungstext mit den Ersetzungsregeln
 - Die in den Text einzusetzenden Werte werden als Parameter der Methode `Z.format()` angegeben
 - Ersetzungsregeln durch geschweifte Klammern `{}` angegeben
- Beispiele:

```
# Positionsparameter nacheinander verwenden: a=awert, b=11, c=(3, 3)
"a={}, b={}, c={}".format("awert", 11, (3,3))
```

```
# eigene Reihenfolge verwenden: a=(3, 3), b=awert, c=11
"a={2}, b={0}, c={1}".format("awert", 11, (3,3))
```

```
# Parameter formatieren, z.B. Hexzahl einfügen => "zahl 2a"
"zahl {:x}".format(42)
```

```
# benannte Parameter verwenden, auch mit Formatvorgabe
"zahl {val:04x} zur Basis {base:d}".format(val=1023, base=4)
```

Ersetzungssymbole

Symbol	Bedeutung
{}	Nächsten Parameter aus der Parameterliste einsetzen
{<num>}	Parameter Nummer <num> aus der Parameterliste einsetzen
{<name>}	Benannten Parameter <name> aus der Parameterliste einsetzen
{<num>.<attr>}	Attribut <attr> des Parameter Nummer <num> einsetzen → siehe später bei Klassen
{<name>.<attr>}	Attribut <attr> des benannten Parameters <name> einsetzen → siehe später bei Klassen
{<num>[<idx>]}	Element <idx> des Parameter Nummer <num> einsetzen
{<name>[<idx>]}	Element <idx> des benannten Parameters <name> einsetzen
{<elem>:<format>}	Element <elem> mit der Formatierung <format> einsetzen <elem> können vorher genannte

Viele weitere Möglichkeiten, siehe `help("FORMATTING")`

Formatierung von Ersetzungssymbole

Symbol	Bedeutung
{:b}	Binärwert ausgeben
{:c}	Zeichen für den angegebenen Unicode
{:d}	Ganzzahl mit Vorzeichen
{:o}	Oktalzahl
{:x}	Hexzahl ({:X} für Ziffern als Großbuchstaben)
{:e}	Gleitkommazahl in Exponentialform ({:E} für Großbuchstaben)
{:f}	Gleitkommazahl
{:g}	{:f} bzw. {:e} (je nachdem, was kürzer ist) ({:G} für E-Form)

Modifikatoren für Ersetzungssymbole

Symbol	Bedeutung
*	Weiterer Parameter gibt Anzahl der Zeichen oder Genauigkeit an
-	Einzusetzender Text ist linksjustiert
+	Vorzeichen immer mit ausgeben
<Leer>	Vor positiven Zahlen Leerzeichen schreiben <Leer>=Leerzeichen
#	Präfix für Oktalzahl / Hexzahl ausgeben
0	Von links führende Nullen voranstellen
m.n	Zahlenausgabe: m=minimale Anzahl von Stellen, n=Anzahl Stellen nach dem Komma

Zeichenkettenformatierung mit F-Strings

- Textformatierung mit F-Strings (ab Python 3.6)
- Bei F-Strings Formatierung mit geschweiften Klammern {} ähnlich wie bei `z.format`
- Syntax für F-Strings: `f"Formatierungstext"`
- der Formatierungstext enthält die Ersetzungsregeln in geschweiften Klammern {} und darin auch gleich die einzusetzenden Werte / Variablen
- Beispiel:

```
# Variablen formatiert ausgeben
a=42
b=3.2
print(f"a={a:b}, b={b:f}")
# Ausgabe: a=101010, b=3.200000
```

In-/not in-Operator

- Operator **in**: `<Text> in <Zeichenkette>`
- Ergebnis:
 - **True**, wenn `<Text> in <Zeichenkette>` enthalten ist
 - **False**, wenn `<Text> nicht in <Zeichenkette>` enthalten ist
- Operator **not in**: `<Text> not in <Zeichenkette>`
- Ergebnis:
 - **False**, wenn `<Text> in <Zeichenkette>` enthalten ist
 - **True**, wenn `<Text> nicht in <Zeichenkette>` enthalten ist

Beispiel zu in-Operator / not in-Operator

```
>>> ort = "TH Nürnberg - Georg Simon Ohm"
>>> "Georg" in ort
True
>>> "georg" in ort          # Kleinschreibung!
False

>>> "Friedrich" in ort
False
>>> "Friedrich" not in ort
True
>>> not "Friedrich" in ort   # in-Operator negiert
True
```

Mehrzeilige Zeichenketten

- Zeichenkette zwischen dreifachen Anführungszeichen (""") oder dreifachen Hochkommas (''') über mehrere Zeilen erlaubt
- In der Zeichenkette einzelne Anführungszeichen/Hochkommas erlaubt
- Zeilenwechsel innerhalb der Zeichenkette als **\n** enthalten

```
"""
```

```
Dies ist ein "mehrzeiliger"  
Text
```

```
"""
```

```
→ '\nDies ist ein "mehrzeiliger"\nText\n'
```

```
'''Dies ist auch ein "mehrzeiliger"  
Text'''
```

```
→ 'Dies ist auch ein "mehrzeiliger"\nText'
```

Einzelnes Zeichen einer Zeichenkette auslesen

- Länge einer Zeichenkette abfragen mit `len(<Zeichenkette>)`
- Einzelnes Zeichen einer Zeichenkette über Elementzugriff auslesen
 - Syntax: `<Zeichenkette> [<Index>]`
 - `<Index>`:
 - 0 für erstes Zeichen, 1 für zweites Zeichen, usw.
 - Alternativ: -1 für letztes Zeichen, -2 für vorletztes Zeichen, usw.

Beispiel zu Zeichenabfrage in Zeichenketten

```
>>> text= "0123456HALLO WELT"
>>> len(text)    # <- Länge der Zeichenkette ermitteln
17
>>> text[0]      # <- Zeichen an Pos. 0 (= Anfang)
'0'
>>> text[16]     # <- Zeichen an Pos. 16 (= Ende)
'T'
>>> text[-1]     # <- Zeichen an Pos. len(text)+idx = 16 wg.
idx=-1
'T'
>>> text[-2]     # <- vorletztes Zeichen, Pos. -2 bzw. 15
'L'
>>> text[17]     # <- Fehlermeldung für übergroßen Index
... IndexError: string index out of range
```

Teilbereich einer Zeichenkette auslesen (Slicing)

Zeichenkettenausschnitt (engl. [slice](#)) über Bereichszugriff auslesen:

- **Syntax:** `<Zeichenkette> [<Start-Index> : <Stop-Index>]`
- `<Start-Index>`: Beginn des Bereichs
 - kann weggelassen werden, dann Beginn der Zeichenkette
- `<Stop-Index>`: Position des ersten Zeichens nach dem Bereich
 - kann weggelassen werden, dann Pos. nach Ende der Zeichenkette

Slicing mit Schrittweitenangabe (Stride)

- Optional kann auch noch Schrittweite (engl. [stride](#)) für den Teilbereich angegeben werden
- Beispiel: Jedes dritte Zeichen ab der Position 1 bis vor Position 10


```
"Langer-Text"[1:10:3]
```

=> entspricht Elementen 1, 4 und 7 → "aeT"

- Weitere Infos zu Teilbereichen in Python mit `help(":")`

Beispiel zu Slicing in Zeichenketten

```
>>> text= "0123456HALLO WELT"
>>> text[:4]      # <- Bereich von Anfang bis Pos. 3
'0123'
>>> text[7:12]    # <- Bereich von Pos. 7 bis 11
'HALLO'
>>> text[7:]      # <- Bereich von Pos. 7 bis Ende
'HALLO WELT'
>>> text[7]       # <- Zeichen an Pos. 7
'H'
>>> text[12]      # <- Zeichen an Pos. 12 (= Leerzeichen)
' '
>>> text[::2]     # <- jedes zweite Zeichen über ganzen Text
'0246AL ET'
```



Auswahl an Methoden einer Zeichenkette

Methode (Z=Zeichenkette)	Aufgabe
<code>Z.count(text)</code>	Zähle, wie oft <code>text</code> in Zeichenkette vorkommt
<code>Z.endswith(text)</code>	Prüfe, ob Zeichenkette mit <code>text</code> endet (True/False)
<code>Z.find(text, start=0, stop=-1)</code>	Suche Position von <code>text</code> in Zeichenkette ab <code>start</code> bis <code>stop</code> , Defaultwerte <code>start</code> =Anfang, <code>stop</code> =Ende Rückgabe der Position, -1 für Text nicht gefunden
<u><code>Z.join(liste)</code></u>	Erzeuge Zeichenkette aus den Elementen der Liste <code>liste</code> , die durch Zeichenkette <code>Z</code> getrennt sind
<code>Z.replace(alt, neu)</code>	Kopie von <code>Z</code> , bei der alle Texte <code>alt</code> durch <code>neu</code> ersetzt sind
<u><code>Z.split(text)</code></u>	Erzeuge eine Liste aus den durch <code>text</code> getrennte Elementen der Zeichenkette <code>Z</code>
<code>Z.startswith(text)</code>	Prüfe, ob Zeichenkette mit <code>text</code> beginnt (True/False)
<code>Z.strip()</code>	Kopie der Zeichenkette ohne Leerzeichen am Anfang bzw. Ende

Viele weitere Methoden, siehe z.B. in Python: `help(str)`

Genauere Info zu einzelnen Methoden z.B. für `Z.strip()` über `help(str.strip)`

`Z.join()`, `Z.split()` => bei Listen nochmal genauer beschrieben

Beispiele für Zeichenketten-Methoden

```
>>> ort = "TH Nürnberg - Georg Simon Ohm"
```

```
>>> ort.find("Georg")
```

```
14
```

```
>>> ort.find("georg")           # Kleinschreibung!
```

```
-1
```

```
>>> "110011101".count("11")
```

```
2
```

```
>>> "ich weiß: der neue ist der Hit".replace("der", "die")
```

```
'ich weiß: die neue ist die Hit'    # alle Vorkommen ersetzt!
```

Konstanter Charakter von Zeichenketten

- Zeichenketten sind in Python **konstant**
- Schreibzugriff auf Einzelzeichen führt zu Fehlermeldungen
- Einzelne Zeichen einer Zeichenkette können nur durch Zusammensetzen einer Kopie der Zeichenkette geändert werden

```
>>> text="übung"
>>> text[0]
'ü'
>>> text[0]="Ü"
... TypeError: 'str' object does not support item assignment
>>> "Ü"+text[1:]
'Übung'
>>> text.replace("ü", "Ü")
'Übung'
>>> text          # Originaltext bleibt erhalten
'übung'
```

Zeichenketten-Umwandlung

Methode	Aufgabe
<code>str(ausdruck)</code>	Ausdruck <code>ausdruck</code> in Zeichenkette umwandeln
<code>chr(zahl)</code>	Einzelnes Zeichen aus Unicode-Nummer <code>zahl</code> erzeugen
<code>ord(zeichen)</code>	Unicode-Nummer eines Zeichens <code>zeichen</code> ermitteln
<code>float(text)</code>	Zeichenkette <code>text</code> in Gleitkommazahl umwandeln
<code>int(text)</code>	Zeichenkette <code>text</code> in Integer-Zahl umwandeln

Weitere, wie `bin(zahl)`, `oct(zahl)`, `hex(zahl)` ...

```
>>> str(0x17)
```

```
'23'
```

```
>>> chr(0x31)
```

```
'1'
```

```
>>> ord('1')
```

```
49
```

```
>>> float('Inf')
```

```
inf
```

```
>>> hex(49)
```

```
'0x31'
```

```
>>> bin(0x31)
```

```
'0b110001'
```

```
>>> oct(0x14)
```

```
'0o24'
```

Datentyp Liste

- Liste: beliebig lange, geordnete Sequenz von Objekten
- Darstellung: durch Komma getrennte Liste von Elementen in eckigen Klammern:

[*<Element-1>*, *<Element-2>*, ..., *<Element-n>* **]**

- Ordnung der Elemente: Reihenfolge, in der Elemente in die Liste aufgenommen wurden, bleibt erhalten → geordnete Liste
- Länge einer Liste abfragen mit **len(<Liste>)**
- Auch leere Liste möglich: **[]**
 - Länge der leeren Liste: **len([]) → 0**
- Listenelemente sind Objekte → können wieder Listen sein

Zugriff auf Listenelemente

- Zugriff auf Listenelemente über Indizierung in der Form:

`<Liste> [<Index>]`

- Bedeutung des Index:
 - 0 für erstes Element, 1 für zweites Element, usw.
 - -1 für letztes Element, -2 für vorletztes Element, usw.

Zugriff auf Listenausschnitte

- Listenausschnitt kann über Bereichszugriff ausgelesen werden
- Syntax: `<Liste> [<Start-Index> : <Stop-Index>]`
- `<Start-Index>`: Beginn des Bereichs
 - kann weggelassen werden, dann erstes Listenelement
- `<Stop-Index>`: Position des ersten Elements nach dem Bereich
 - kann weggelassen werden, dann Pos. nach Ende der Liste

Listenausschnitt mit Schrittweitenangabe (engl. Stride)

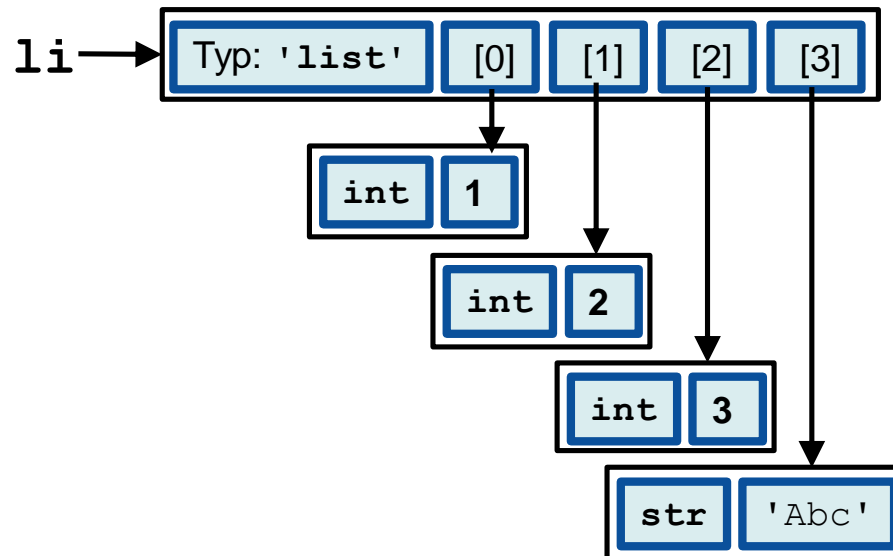
- Optional kann auch noch Schrittweite (engl. `stride`) für den Ausschnitt angegeben werden
- Beispiel: Jedes zweite Element ab der Position 0 bis vor Position 10

`0:10:2` => entspricht Elementen 0, 2, 4, 6, 8

- Weitere Infos zu Ausschnitten in Python mit `help(":")`

Listen-Beispiele (1)

```
>>> li=[1,2,3,"Abc"]
>>> type(li)
<class 'list'>
>>> len(li)
4
>>> li[0]
1
>>> li[3]
'Abc'
>>> li[-1]
'Abc'
>>> li[:2]
[1, 2]
>>> li[2:]
[3, 'Abc']
>>> li[1:3]
[2, 3]
>>> li[1::2]
[2, 'Abc']
```



Listen-Beispiele (2)

```
>>> li2=[1,2,[3,4]]
```

```
>>> type(li2)
```

```
<class 'list'>
```

```
>>> len(li2)
```

```
3
```

```
>>> li2[0]
```

```
1
```

```
>>> li2[-1]
```

```
[3, 4]
```

```
>>> leer=[]
```

```
>>> len(leer)
```

```
0
```

Operatoren für Listen

```
>>> 3 * [1,2]
[1, 2, 1, 2, 1, 2]
```

Multiplikation mit Zahl → Wiederholung

```
>>> [1,2] + ["ABC"]
[1, 2, 'ABC']
```

Addition → Verkettung

```
>>> [1,2] == [1, 2]
True
```

Prüfung == / !=

```
>>> 2 in [1, 2, 3]
True
```

Enthalten-Prüfung

```
>>> 8 not in [1, 2, 3]
True
```

Nicht-Enthalten-Prüfung

Methoden einer Liste

Methoden (L=Liste)	Aufgabe
<code>L.append(val)</code>	Element an Liste anhängen (vgl. +-Operator)
<code>L.clear()</code>	Alle Listenelemente löschen
<code>L.copy()</code>	Elementweise Kopie der Liste zurückgeben
<code>L.count(val)</code>	Zähle, wie oft <code>val</code> in Liste vorkommt
<code>L.index(val, start=0, stop=-1)</code>	Suche Position von <code>val</code> in Liste ab <code>start</code> bis <code>stop</code> , Defaultwerte <code>start=Anfang</code> , <code>stop=Ende</code>
<code>L.insert(idx, val)</code>	Element an Position <code>idx</code> in Liste einfügen
<code>L.pop(idx=-1)</code>	Element an Position <code>idx</code> (Default Ende) aus Liste entfernen und zurückgeben
<code>L.remove(val)</code>	Erstes Element mit Wert <code>val</code> aus Liste löschen
<code>L.reverse()</code>	Reihenfolge der Listenelemente umdrehen
<code>L.sort()</code>	Liste aufsteigend sortieren

Beispiele für Listenmethoden

```
>>> z=[1,2]
>>> z.append(3)      # Element an Liste anhängen -> z=[1,2,3]
>>> z.clear()        # Listenelemente löschen -> z=[]
>>> z=[2,3,-1,1]
>>> z.pop()          # letztes Element zurückgeben -> z=[2,3,-1]
1
>>> z.reverse()      # Reihenfolge umdrehen -> z=[-1,3,2]
>>> z.sort()          # Liste sortieren -> z=[-1, 2, 3]
>>> z.append(2)       # -> z=[-1,2,3,2]
>>> z.count(2)        # zähle, wie oft Element 2 vorkommt
2
>>> z.index(2)        # finde erstes Auftreten von Element 2
1
>>> z.pop(1)         # -> z=[-1,3,2]
2
```

Referenzen auf Listen / Veränderung von Listen

- Listen werden als Objekte (= Listenobjekt) realisiert
- Wenn eine Variable eine Liste „enthält“, zeigt sie auf das Listenobjekt („referenziert das Listenobjekt“)
- Eine Anweisung wie `a=[0, 1, 2, 3]` führt dazu, dass die Variable `a` eine Liste mit vier Objekten für die Zahlen 0 bis 3 referenziert
- Einzelne Listenelemente können verändert / gelöscht werden
 - Veränderung z.B. mit: `a[1] = -1`
 - Löschen z.B. mit: `a.pop(1)`, `del a[1]`
 - Einfügen z.B. mit: `a.insert(1, 1)`

Liste mit mehreren Variablen referenzieren

- Wird eine Variable `a` mit einer Liste als Inhalt an eine andere Variable `b` zugewiesen, referenzieren `a` und `b` das gleiche **Listenobjekt**
- Eine Veränderung der Liste von Variable `a` betrifft dann auch `b`
- Sollen die Listen unabhängig voneinander sein, muss die Liste explizit kopiert werden, z.B. mit `c=a.copy()` oder `c=list(a)`

```
>>> a=[0,1,2,3]
>>> b=a          # b zeigt auf gleiches Listenobjekt wie a
>>> c=a.copy()   # c erhält eigene Kopie der Liste von a
>>> a[1]=-1      # zweites Element von a verändern
>>> b            # davon auch b betroffen, da gleiche Liste
[0, -1, 2, 3]
>>> c            # c nicht betroffen, da eigene Liste
[0, 1, 2, 3]
```

Erweiterte Zuweisung bei gemeinsamen Listenreferenzen

- Auch bei erweiterter Zuweisung `*=` bzw. `+=` alle Variablen betroffen, die eine gemeinsame Liste referenzieren
- Bei erweiterter Zuweisung wird die bestehende Liste modifiziert anstatt eine neue Liste zu erzeugen

```
>>> a=[0,1,2,3]
>>> b=a          # b referenziert gleiches Listenobjekt wie a
>>> b*=2         # erweiterte Zuweisung => Liste verdoppeln

>>> b
[0, 1, 2, 3, 0, 1, 2, 3]
>>> a          # a referenziert gleiche Liste
[0, 1, 2, 3, 0, 1, 2, 3]
```


Zeichenketten-Methoden mit Listen

Methode (Z=Zeichenkette)	Aufgabe
<code>z.join(liste)</code>	Erzeuge Zeichenkette aus den Elementen der Liste <code>liste</code> , die durch Zeichenkette <code>z</code> getrennt sind
<code>z.split(text)</code>	Erzeuge eine Liste aus den durch <code>text</code> getrennten Elementen der Zeichenkette <code>z</code>

Detailliertere Infos in Python => `help(str.join)`, `help(str.split)`

```
>>> z="_".join(["10", "110", "001"])
```

```
>>> z
```

```
'10_110_001'
```

```
>>> z.split("_")
```

```
['10', '110', '001']
```

```
>>> z
```

```
# z durch split() unverändert
```

```
'10_110_001'
```

Datentyp Tupel

Tupel in Python

- Tupel (engl. **tuple**): wie Liste geordnete Sequenz von Objekten
- Darstellung: durch Komma getrennte Liste von Elementen in runden Klammern:

(*<Element-1>*, *<Element-2>*, ..., *<Element-n>* **)**

- Klammern können auch **entfallen**:

<Element-1>, *<Element-2>*, ..., *<Element-n>*

- Länge eines Tupels abfragen mit **len(<Tupel>)**
- Tupel-Elemente sind Objekte → können wieder Tupel sein
- Elementzugriff/Slicing wie bei Listen mit *<Tupel>* **[<Index>]**
- Tupel im Unterschied zu Listen **read-only**: es können keine Elemente hinzugefügt, gelöscht oder verändert werden

Beispiele für Verwendung von Tupel

```
>>> t=(1,2)
```

```
>>> t[1]                # Elementzugriff wie bei Liste mit []
```

```
2
```

```
>>> len(t)
```

```
2
```

```
>>> type(t)
```

```
<class 'tuple'>
```

```
>>> t[1]=0              # Veränderung nicht möglich
```

```
... TypeError: 'tuple' object does not support item assignment
```

```
>>> t=3,4
```

```
>>> t
```

```
(3, 4)
```

Beispiele für Verwendung von Tupel (2)

leeres Tupel

```
>>> type(())  
<class 'tuple'>
```

Besonderheit:

Ein Element in runden Klammern

=> normale Kammersetzung für Ausdruck

```
>>> type(1)  
<class 'int'>
```

Für Tupel mit einem Element => Komma anhängen

```
>>> type((1,))  
<class 'tuple'>
```

Beispiele für Verwendung von Tupel (3)

Addition von Tupeln

```
>>> (1,2)+(3,4)
(1, 2, 3, 4)
```

Multiplikation von Tupeln

```
>>> 3*(1,2)
(1, 2, 1, 2, 1, 2)
```

Tupel verändern => nur über Kopie

```
>>> werte=(0, 1, 2)
>>> werte=(-1,) + werte[1:]
>>> werte
(-1, 1, 2)
```

Vergleich zwischen Tupel und Liste

- Tupel und Liste beliebig lange, geordnete Sequenzen von Objekten
- Tupel über runde Klammern **()** definiert, Liste über eckige Klammern **[]**
- Elementzugriff bei Tupel und Liste gleich über **[<Index/Slice>]**
- Tupel kann nachträglich nicht verändert werden, Liste schon
- Einsatzbereiche
 - Tupel: Datensatz aus potenziell unterschiedlichen Daten mit fester Elementanzahl (z.B. 2D-Vektor hat immer zwei Elemente x und y)
 - Liste: unbekannte Anzahl von ähnlichen Objekten soll zusammen gespeichert werden (z.B. Liste von 2D-Vektoren, die zusammen ein Polygon bilden → einzelne Vektoren wären Tupel, Polygon wäre Liste von Tupeln)

Mehrfachzuweisung (1)

- Python erlaubt es, in einer Anweisung Werte eines Tupels oder einer Liste auf mehrere Variablen verteilt zuzuweisen
- Dabei müssen links vom Zuweisungszeichen (=) durch Komma getrennt genau so viele Variablen stehen, wie in dem Tupel bzw. der Liste vorhanden sind → jedes Element wird einer Variable zugewiesen
- Die Variablenliste kann auch als Tupel oder Liste geschrieben werden
- Beispiel:

```
t=("readme.txt", "2022-02-28", 2346)
```

```
name,date,size=t
```

```
(name,date,size)=t    # geht auch [n,d,s]=t
```

```
l=list(t)             # l ist t in Listenform
```

```
name,date,size=l      # geht auch (n,d,s)=l oder [n,d,s]=l
```

```
name,date=l           # Fehler, wenn eine Variable zu wenig
```

```
... ValueError: too many values to unpack (expected 2)
```

```
pt=(10, 15)           # Punkt: x=10, y=15
```

```
x,y=pt                # Zuweisung an Variable
```


Mehrfachzuweisung (2)

- Mehrfachzuweisung auch ganz ohne Klammern [], () möglich
- Beispiel:
 `a, b = 3, 4 # => a=3; b=4`
 `a, b = b, a # Variableninhalte tauschen`

Datentyp Dictionary

- Ein Python-Dictionary (engl. für Wörterbuch) speichert Objekte als Schlüssel-Wert-Paare
 - Schlüssel: Konstante, die einen Wert eindeutig referenziert → Zeichenkette, Zahl oder Tupel
 - Wert: beliebiges Objekt
- Darstellung:
 - durch Komma getrennte Paare in geschweiften Klammern
 - Jedes Paar in der Kombination Schlüssel + Doppelpunkt + Wert
 - Syntax:
`{ <Schlüssel-1> : <Wert-1>, <Schlüssel-2> : <Wert-2>, ... }`
- Reihenfolge der Schlüssel-Wert-Paare ohne Bedeutung
- Zugriff auf Wert zu einem Schlüssel über:
`<Dictionary> [<Schlüssel>]`

Ausgewählte Methoden eines Dictionary

Methode (D=Dictionary)	Aufgabe
<code>D.clear()</code>	Alle Einträge löschen
<code>D.copy()</code>	Elementweise Kopie des Dictionary zurückgeben
<code>D.get(k, d=None)</code>	Zu Schlüssel <code>k</code> gehörenden Wert zurückgeben; wenn <code>k</code> nicht enthalten ist, <code>d</code> zurückgeben
<code>D.items()</code>	Iterator für Schlüssel-Wert-Paare zurückgeben
<code>D.keys()</code>	Iterator für Schlüssel zurückgeben
<code>D.values()</code>	Iterator für Werte zurückgeben

Iteratoren können z.B. in for-Schleife verwendet werden

Beispiele zu Dictionary

```
>>> d={"be":"Belgien", "us":"USA"}
```

```
>>> d["tv"]="Tuvalu"
```

```
>>> d["be"]
```

```
'Belgien'
```

```
>>> "de" in d
```

```
False
```

```
>>> d["de"]
```

```
... KeyError: 'de'
```

```
>>> d.get("de", "unbekannt")
```

```
'unbekannt'
```

```
>>> d[0]=48          # als Schlüssel auch Zahlen zulässig
```

```
>>> d
```

```
{'be': 'Belgien', 'us': 'USA', 'tv': 'Tuvalu', 0: 48}
```

```
>>> d[0]             # Wert zu Schlüssel 0, nicht erstes Element
```

```
48
```

Weitere Beispiele zu Dictionary

```
>>> d2=dict(be="Belgien", us="USA")
>>> key="be"
>>> del d2[key]          # => d2 enthält nur noch us
>>> key="us"
>>> print(d2.pop(key))   # => Ausgabe USA
```

```
>>> d3=dict(1="Belgien", 2="USA")
SyntaxError: keyword can't be an expression
```

```
>>> d3={ 1: "Belgien", 2: "USA", 10: "Schweden" }
>>> del d3[2]
>>> del d3[0]
KeyError: 0
```

```
>>> d3[1,2,3]=None      # => Bedeutung?
>>> d3[[1,2,3]]=1
TypeError: unhashable type: 'list'
```

Dictionary mit mehreren Variablen referenzieren

- Wird eine Variable `a` mit einem Dictionary als Inhalt an eine andere Variable `b` zugewiesen, referenzieren `a` und `b` das gleiche **Dictionary**
- Eine Veränderung des Dictionarys von Variable `a` betrifft dann auch `b`
- Sollen die Dictionarys unabhängig voneinander sein, muss explizit kopiert werden, z.B. mit `c=a.copy()` oder `c=dict(a)`

```
>>> a={'be': 'Belgien'}
>>> b=a                # b zeigt auf gleiches Dictionary wie a
>>> c=dict(a)          # c erhält eigene Kopie des Dictionary
>>> a["by"]="Bayern"    # Eintrag bei Dictionary a hinzufügen

>>> b                  # davon auch b betroffen
{'be': 'Belgien', 'by': 'Bayern'}

>>> c                  # c nicht betroffen, da eigene Kopie
{'be': 'Belgien'}
```

Speicherverwaltung und Referenzen

Automatische Speicherverwaltung (Garbage Collection)

- Objekte werden in Python per **Referenzierung** verwendet
- Beispiel: zwei Variablen referenzieren die gleiche Liste → Referenz zeigt auf gleiche Liste
- Jedes Objekt hat einen **Referenzzähler** → darin steht, wie viele Referenzen auf das Objekt existieren
- Wenn der Referenzzähler auf Null fällt, wird das Objekt automatisch von Python gelöscht
- Python verwaltet also selbst, welche Objekte und wann gelöscht werden
- Vorgang der automatischen Löschung wird als **Garbage Collection** bzw. automatische Speicherverwaltung bezeichnet
- Folge: Es ist in Python im Normalfall nicht notwendig, Variablen / Objekte explizit zu löschen

Explizites Löschen von Variablen / Objekten mit del

- Explizites Löschen von Objekte / Variablen mit dem Befehl **del**
- Syntax:
del *<Ausdruck>*
- Ausdruck kann z.B. sein:
 - Variablenname → Variable wird entfernt / existiert nicht mehr
 - Index einer Liste → Eintrag wird aus der Liste entfernt
 - Key eines Dictionary → Eintrag wird aus dem Dictionary entfernt
- Mit einem einzelnen del-Befehl können auch mehrere Objekte / Variablen gleichzeitig gelöscht werden:
del *<Ausdruck-1>*, *<Ausdruck-2>*, ...

Beispiel zum Löschen von Listenelementen

```
>>> a=[0,1,2,3]
>>> b=a          # b zeigt auf gleiches Listenobjekt wie a
>>> del a[1]      # zweites Element aus a löschen
>>> b            # davon auch b betroffen, da gleiche Liste
[0, 2, 3]
>>> del a        # Variable a löschen
>>> a            # Fehler, da a nicht mehr existiert
... NameError: name 'a' is not defined
>>> b            # b existiert aber weiter
[0, 2, 3]
```

Ablaufsteuerung mit if-Verzweigung

Bedingte Ausführung mit if-Anweisung

- bedingte Ausführung von Anweisungen in einer if-Abfrage, wenn eine Bedingung logisch **True** ist
- Anweisungen folgen nach Doppelpunkt (:)
 - Entweder durch Semikolon (;) getrennt auf gleicher Zeile
 - Oder **besser**: Mehrere Anweisungen auf nachfolgenden Zeilen
 - Zugehörigkeit zur if-Anweisung durch gleiche Einrücktiefe

if *<Bedingung>*: *<Anweisung-1>* ...

if *<Bedingung>*:
 <Anweisung-1>
 ...
 <Anweisung-m>

<Anweisung(en)> ausführen, wenn
<Bedingung> **True**

Bedeutung von Leerzeichen

- Leerzeichen am Zeilenanfang und der Zeilenumbruch haben Bedeutung
- Ein Zeilenumbruch schließt einen Befehl ab
- Die Einrücktiefe bei benachbarten Zeilen drückt Blockzugehörigkeit aus
 - **Weniger** eingerückte Zeile gehört zu einem **übergeordneten** Block
 - **Weiter** eingerückte Zeile gehört zu einem **untergeordneten** Block
 - Gleich eingerückte benachbarte Zeilen sind im gleichen Block
 - Ein untergeordneter Block wird mit dem Doppelpunkt (:) eingeführt

Beispiel zur if-Anweisung

Beispiel:

```
if x > 0: print("positive Zahl"); print("weiter");
```

besser: pro Zeile eine Anweisung mit Einrückung

```
if x < 0:  
    print("negative Zahl")  
    print("weiter")
```

Mehrfachverzweigung mit if-Anweisung

```
if <Bedingung-1>:  
    <Anweisung-1>  
    ...  
    <Anweisung-m>  
elif <Bedingung-2>:  
    <Anweisung-1>  
    ...  
    <Anweisung-n>  
else:  
    <Anweisung-1>  
    ...  
    <Anweisung-k>
```


Mehrfachverzweigung mit if-Anweisung

```
x = 0
```

```
if x > 0:
    print("positive Zahl")
elif x < 0:
    print("negative Zahl")
elif x < -1e10:
    # wann wird das ausgegeben?
    print("richtig negative Zahl")
else:
    print("Null")
```

If-Ausdruck

- Variante der if-Anweisung kann in **Ausdrücken** verwendet werden, ähnlich dem ?-Operator in C: (z.B. in C: `int a = 3==3 ? 1 : 2`)
- Syntax:
`<True-Ausdruck> if <Bedingung> else <False-Ausdruck>`
- Beim If-Ausdruck ist der **else**-Teil zwingend anzugeben
- Ergebnis des If-Ausdrucks ist ...
 - der `<True-Ausdruck>`, wenn die `<Bedingung>` erfüllt ist bzw.
 - der `<False-Ausdruck>`, wenn die `<Bedingung>` nicht erfüllt ist (**False** / 0 / **None** ergibt)
- Beispiel:
`art = "ungerade" if 3%2 else "gerade"`

If-Anweisung ⇔ If-Ausdruck

- Bei einer If-Anweisung erscheint nach der Bedingung ein Doppelpunkt, bei dem If-Ausdruck nicht
- Der Doppelpunkt (:) eröffnet einen neuen Ausführungsblock
- Beispiele:

```
# if-Anweisung mit Doppelpunkt
# kann nicht rechte Seite einer Zuweisung sein, da Anweisung
if 2==1:
    msg = "unterschiedlich"
else:
    msg = "gleich"


# if-Ausdruck ohne Doppelpunkt
# kann rechte Seite einer Zuweisung sein, da Ausdruck
msg = "gleich" if 2==1 else "unterschiedlich"
```

Ablaufsteuerung mit while-Schleife

While-Schleife

- *Anweisungen* ausführen, solange eine *Bedingung* logisch **True**
- *Anweisungen* folgen nach Doppelpunkt (:)
- Vorzeitiger Schleifenabbruch mit Befehl **break**
- Nächste Schleifenwiederholung mit Befehl **continue**

```
while <Bedingung>:  
    <Anweisung-1>  
    ...  
    <Anweisung-m>
```



Anweisung(en) ausführen,
solange *Bedingung* **True**

Beispiel zu while

```
# Beispiel zu while-Schleife
summe = 0
anz_positiv = 0

while True:
    zahl = int(input("Zahl eingeben: "))
    if zahl < 0:
        break
    elif zahl == 0:
        continue

    summe += zahl
    anz_positiv += 1

print("Summe =", summe)
print("Anzahl positiv =", anz_positiv)
```

While-Schleife mit Else-Teil

- Die While-Schleife kann einen optionalen Else-Teil enthalten
- Der Else-Teil wird nach letzter Iteration der Schleife ausgeführt, wenn die Schleife nicht durch den **break**-Befehl beendet wurde

```
# Beispiel zu while-Schleife mit else-Teil
anzahl = 0


while anzahl < 10:
    zahl = int(input("Zahl eingeben: "))
    if zahl <= 0: break
    anzahl += 1
else:
    print("Es wurden zehn Zahlen eingegeben.")
    print("Alle waren positiv.")
```

Ablaufsteuerung mit for-Schleife

For-Schleife

- *Anweisungen* ausführen für eine *Sequenz* von Werten
- Sequenz z.B. Liste, Tupel, Zeichenkette
- *Anweisungen* folgen nach Doppelpunkt (:)
- Vorzeitiger Schleifenabbruch mit Befehl **break**
- Nächste Schleifenwiederholung mit Befehl **continue**

```
for <Var> in <Sequenz>:  
    <Anweisung-1>  
    ...  
    <Anweisung-m>
```



Anweisung(en) für alle
Elemente in *Liste*
ausführen

Einfaches Beispiel mit for-Schleife

```
# Python-Programm mit Schleifendurchlauf
```

```
werte=[1, 1.0, "zahl", 3]
```

```
for wert in werte:
```

```
    if type(wert)==int: print("Int ", wert)
```

Programmausgabe:

Int 1

Int 3

For-Schleife über Tupel, Liste oder Zeichenkette

Python-Programm mit Schleifendurchläufen

```
t=(1, 2, 3)
```

```
l=[4, 5, 6]
```

```
z="abc"
```

```
for elem in t: print(elem) # nacheinander 1, 2, 3 ausgeben
```

```
for elem in l: print(elem) # nacheinander 4, 5, 6 ausgeben
```

```
for elem in z: print(elem) # nacheinander a, b, c ausgeben
```

Ausgabe in umgedrehter Reihenfolge

```
for elem in reversed(l): print(elem) # --> 6, 5, 4
```

Ausgabe mit Index und Wert

```
for i,e in enumerate(l): print(i,e) # --> 0 4, 1 5, 2 6
```

For-Schleife über Dictionary

Python-Programm mit Schleifendurchlauf für Dict.

```
d={"b":1, "a":2 }
```

```
# key hier nacheinander alle Schlüssel von d  
for key in d: print(key, d[key])
```

```
# key hier sortiert alle Schlüssel von d  
for key in sorted(d): print(key, d[key])
```

```
# Mehrfachzuweisung key und val über d.items()  
# val ist zu jedem key entsprechendes d[key]  
for key, val in d.items(): print(key, val)
```

For-Schleife mit Else-Teil

- For-Schleife kann optionalen Else-Teil enthalten
- Else-Teil wird nach letzter Iteration der Schleife ausgeführt, wenn Schleife nicht durch **break**-Befehl beendet wurde

```
# For-Schleife mit Else-Teil
# summe von Werten ausgeben, wenn nur Int-Zahlen

werte=[1, 1.0, "zahl", 3]

summe = 0
for wert in werte:
    if type(wert) != int: break
    summe += wert
else:
    # reguläres Schleifenende => Werte ausgeben
    print("Summe =", summe)
```

For-Schleife über Zahlenbereich (range)

- Mittels `range(<start>, <stop>, <schrittweite>)` kann ein Zahlenbereich angegeben werden, über den eine for-Schleife laufen soll

- Beispiele:

```
for idx in range(10):  
    print(idx)          # Ausgabe 0, 1, ... 9
```

```
for idx in range(1, 11):  
    print(idx)          # Ausgabe 1, 2, ... 10
```

```
for idx in range(1, 11, 2):  
    print(idx)          # Ausgabe 1, 3, 5, 7, 9
```

- Anmerkung: `range()` definiert eigentlich einen sogenannten [Iterator](#)
- Iterator → Werte werden dynamisch in jedem Schleifendurchlauf nacheinander generiert und nicht alle auf einmal als große Liste

Iteratoren für for-Schleife

- Iterator: Python-Element zur Erzeugung von Datensequenzen
- die einzelnen Werte der Sequenz werden dynamisch in jedem Schleifendurchlauf nacheinander generiert
- Vorteil: geringerer Speicherbedarf, weil bei langen Datensequenzen keine langen Listen entstehen
- Beispiele für Iteratoren:

Iterator	Aufgabe
<code>range(...)</code>	Zahlenbereich mit bestimmter Schrittweite generieren
<code>sorted(seq)</code>	Sortierte Version einer Sequenz <code>seq</code> erzeugen
<code>reversed(seq)</code>	Umgekehrte Version einer Sequenz <code>seq</code> erzeugen
<code>enumerate(seq)</code>	Sequenz elementweise als Tupel aus Nummer des aktuellen Elements und Wert des aktuellen Elements zurückgeben.

Leere Anweisung bei if / for / while

- Der Python-Befehl `pass` besitzt keine besondere Funktionalität
- Die `pass`-Anweisung kann überall als Platzhalter eingesetzt werden, wo Python eine Anweisung verlangt , z.B. bei `if` / `for` / `while`, aber im Programm nichts getan werden soll

```
if x != 42:  
    # else darf nicht sofort folgen, deshalb pass  
    pass  
else:  
    print("falscher Wert")
```


```
for x in range(10):  
    pass  
else:  
    print("Ende erreicht")
```

Ablaufsteuerung mit match-Vergleich

Match-Case-Ablauf

- Ab **Python 3.10** Match-Anweisungen ähnlich zu Switch-Case in C
- Ein Ausdruck wird mit mehreren Alternativen verglichen
- Syntax in Python:

```
match <expr>:  
    case <muster-1>:  
        <Anweisung(en) -1>  
    case <muster-2>:  
        <Anweisung(en) -2>  
    ...
```



Erster Treffer für
<expr> wird ausführen

Match-Case-Beispiel → Zeichenkettenvergleich

- Einfacher Vergleich z.B. für Zeichenkette → ersetzt if/elif-Kette
- Wenn passendes Muster bzw. Match-Case gefunden wurde, ist match-Anweisung beendet
- Anweisungen wie **break** gehören nicht zu Match-Case → anders als in C/C++ kein Fall-through für ein Pattern

```
while True:
    txt = input("Text eingeben: ")
    match txt:
        case "hallo":
            print("Eingabe ist hallo")
        case "test":
            print("Eingabe ist test")
        case "eins" | "zwei":
            print("Eingabe ist Zahl 1 oder 2")
        case _:
            print("Eingabe unbekannt")
            break
```

← Patternvergleich für txt

← Mehrere Patterns zusammen

← Default-Pattern → "_"

← **break** gehört zu **while**

Match-Case-Beispiel → komplexe Muster

- Bei Listen und Tupeln auch komplexe Muster möglich
- Z.B. lange Sequenz, bei dem letztes Element bestimmten Wert hat, usw..

```
v_list = [(1,2), (1,2,-3), (1,2,3), (3,3,8), [4], (4,1,2,3,4,5,6), ()]
for val in v_list:
    match val:
        # Treffer, wenn Sequenz mehrere Elemente hat und das letzte Elem. 3 ist
        case (*_, 3):
            print("sequence ending with 3")
        case (1, 2, a):
            # Treffer, wenn Sequenz mit 1, 2 startet => drittes Element in Var. a
            print("sequence with 1,2,a => a=", a)
        case [4, *b]:
            # Treffer, wenn Sequenz mit 4 startet => Rest in Variable b speichern
            # Sequenz kann Liste oder Tupel sein
            print("arbitrary sequence starting with 4, followed by b=", b)
        case (_, _, a):
            # Treffer, wenn Sequenz 3 Elemente hat => letztes Element in a speichern
            # 3-Tupel mit 1 und 2 am Anfang schon im 2. case abgefangen
            print("3-elem sequence with a=", a)
        case _:
            # Treffer für beliebige Pattern => vgl. "default" in C/C++
            # ist nicht verpflichtend, Default-Case kann auch weggelassen werden
            print("something else")
```

Match-Case-Beispiel → Ausgabe für komplexes Muster

Ausgabe zu Codebeispiel auf der Seite vorher:

Ausgaben zu:

# (1,2)	=> case _	=> something else
# (1,2,-3)	=> case (1, 2, a)	=> sequence with 1,2,a ...
# (1,2,3)	=> case (*_, 3)	=> ... ending with 3
# (3,3,8)	=> case (_, _, a)	=> 3-elem sequence ...
# [4]	=> case [4, *b]	=> arbitrary sequence starting ...
# (4,1,2,3,4,5,6)	=> case [4, *b]	=> arbitrary sequence starting ...
# ()	=> case _	=> something else

something else

sequence with 1,2,a => a= -3

sequence ending with 3

3-elem sequence with a= 8

arbitrary sequence starting with 4, followed by b= []

arbitrary sequence starting with 4, followed by b= [1, 2, 3, 4, 5, 6]

something else

Benutzerdefinierte Funktionen

- Befehl **def** definiert Namen und Anweisungen einer Funktion
- Syntax der Funktionsdefinition:

```
def <Name>(<Param-1>, <Param-2>, ..., <Param-m>) :  
    <Anweisungen>
```

- Bei einer Funktionsdefinition erzeugt Python ein Funktionsobjekt mit den Anweisungen der Funktion und weist es an die Variable **<Name>** zu

Aufruf von Funktionen in Python

- Anweisungen in einer Funktion werden erst abgearbeitet, wenn die Funktion aufgerufen wird
- Syntax des Funktionsaufrufs:

<Name>(<Param-1>, <Param-2>, ..., <Param-m>)

- In Python gibt jede Funktion mit dem Funktionsaufruf einen Wert zurück

Beispiel zu Python-Funktion

```
# Potenzieren mit mypow()  
def mypow(x, y):  
    return x**y
```

- Funktionsname: `mypow`
- 2 Parameter: `x`, `y`
- Rückgabewert: `x**y`
- Aufruf z.B. mit

```
>>> print(mypow(-2, 2))
```

Funktionsende / Rückgabewert

Die Bearbeitung einer Funktion endet ...

- mit Abarbeiten der letzten Codezeile in der Funktion.
 - Der Rückgabewert der Funktion ist dann der Wert **None**
- oder bei Auftreten des Befehls **return**.
 - Ein optionaler Wert hinter dem **return**-Befehl ist dann der Rückgabewert der Funktion.
 - Ohne diesen optionalen Wert ist der Rückgabewert ebenfalls **None**
- Funktionen signalisieren mit dem Rückgabewert **None**, dass sie eigentlich keinen Wert zurückgeben

Weiteres Beispiel zu Python-Funktion

```
# Parameter ausgeben, bis Zeichenkette "stopp" dabei
# Funktion gibt keinen Wert zurück
def showtype(arglist):
    for arg in arglist:
        if str(arg) == "stopp": return
        print(arg)

a=showtype([1, 2, "stopp", []])
print(type(a)) # => liefert NoneType
```

- Im Normalfall müssen bei Funktionsaufrufen alle Parameter angegeben werden
 - Entweder in der Reihenfolge der Definition,
z.B. `mypow(-2, 2)`
 - Oder durch Angaben der Form `<var>=<wert>` in beliebiger Reihenfolge (= **als benannte Parameter**),
z.B. `mypow(y=2, x=-2)`
- Parameter können auch Standardwerte (= **Defaultwerte**) haben
 - Parameter mit Defaultwert haben in **def**-Zeile Wertzuweisungen
 - Hat eine Funktion Parameter mit und ohne Default-Werten, folgen in der **def**-Zeile Parameter mit Defaultwerten am Schluss
 - Parameter mit Defaultwerten können beim Funktionsaufruf in der Parameterliste weggelassen werden → es gilt dann der Standardwert

Beispiel zu Python-Funktion mit Default-Parametern

```
# Potenzieren mit mypow(),  
# standardmäßig quadrieren => y=2  
def mypow2(x, y=2):  
    return x**y
```

Aufruf z.B. mit

```
>>> print(mypow2(-2))  
4  
>>> print(mypow2(-2,2))  
4  
>>> print(mypow2(y=2,x=-2))  
4
```

Funktionen mit variabler Anzahl von Parametern (1)

- Soll eine Funktion mehr als die in der Parameterliste angegebenen Parameter akzeptieren, wird ein spezieller Parameter definiert, in den die überzähligen Wertangaben als Tupel kommen.
- In der Parameterliste erscheint dieser Parameter am Ende und ist mit einem Stern (*) vor dem Namen gekennzeichnet

```
# Funktion, die ihre Parameter als Liste zurückgibt
def myargs(*args):
    return args
```

```
>>> myargs()           # => Rückgabe ()
>>> myargs(1,2,3)      # => Rückgabe (1,2,3)
```

Funktionen mit variabler Anzahl von Parametern (2)

- Normal in der Parameterliste angegebene Parameter bekommen ihre Werte entsprechend ihrer Position zugewiesen
 - erster Wert an ersten Parameter
 - zweiter Wert an zweiten Parameter, usw.
- Alle überzähligen Parameter werden dann als Tupel an den Stern-Parameter übergeben

Funktion: Funktionsparameter zurückgeben

Übergabe: Erster Parameterwert an a

Übergabe: Weitere Parameter an args

Rückgabe: Inhalt von args

```
def myargs2(a, *args):  
    return args
```

```
>>> myargs2(1)                # => Rückgabe ()  
>>> myargs2(1, 2, 3)          # => Rückgabe (2, 3)  
>>> myargs2(1, 2, "str")      # => Rückgabe (2, "str")
```

Funktion mit beliebigen benannten Parametern

- Soll eine Funktion beliebige benannte Parameter akzeptieren, wird ein spezieller Parameter definiert, der die benannten Parameter als **Dictionary** aufnimmt.
- In der Parameterliste erscheint dieser Parameter zuletzt und ist mit zwei Sternen (**) vor dem Namen gekennzeichnet
- Vor dem **-Parameter können auch normale Parameter und der *-Parameter angegeben werden

```
# Funktion, die ihre Parameter als Dictionary zurückgibt
```

```
def mykwargs(**kwargs):  
    return kwargs
```

```
>>> mykwargs() # => {}
```

```
>>> mykwargs(a=2, b=3, c=42) # => {'a': 2, 'b': 3, 'c': 42}
```


Anonyme Funktionen (Lambda-Ausdrücke)

- Eine Funktionsdefinition mit **def** erzeugt ein Funktionsobjekt, das an eine Variable zugewiesen wird
- Ein **Lambda-Ausdruck** erzeugt dagegen ein Funktionsobjekt, das ...
 - das Ergebnis eines einzelnen Python-Ausdrucks zurückgibt
 - nicht automatisch an eine Variable zugewiesen wird
- Syntax:
lambda *<Parameterliste>* : *<Ausdruck>*
- Der Lambda-Ausdruck **entspricht** folgender Funktionsdefinition:
def (*<Parameterliste>*) :
 return *<Ausdruck>*
- aber: eine Funktionsdefinition ohne Namen ist nicht erlaubt!

Beispiel mit Lambda-Ausdruck

```
# tb_lambda.py

def foreach(f, *args):
    res=[]
    for arg in args: res.append(f(arg))
    return res

vec=foreach(lambda x: 2*x*x, 1, 2, 3)
print(vec)

# Ausgabe:
# [2, 8, 18]

print(type(lambda x: 2*x*x))
# Ausgabe: <class 'function'>
```

Dokumentation von Funktionen (Docstrings)

- Zur Dokumentation von Funktionen kann bei jeder Funktionsdefinition ein Text (engl. `doc string`) angegeben werden, der die Aufgabe der Funktion näher erläutert
- Der Text muss die erste Anweisung nach dem Doppelpunkt des `def`-Befehl sein und wird auch bei Aufruf der Funktion `help()` ausgegeben

Beispiel zu Docstrings

```
def myfunc(x):  
    "Funktion zum Testen von Docstrings. Gibt nur x zurück."  
    return x  
  
def myfunc_longdoc(x):  
    """Über eine mehrzeilige Zeichenkette  
    kann auch sehr langer Dokumentationstext eingegeben werden.  
    """  
    return x
```

```
>>> help(myfunc)
```

```
Help on function myfunc in module __main__:
```

```
myfunc(x)
```

```
    Funktion zum Testen von Docstrings. Gibt nur x zurück.
```

```
>>> help(myfunc_longdoc)
```

```
Help on function myfunc_longdoc in module __main__:
```

```
myfunc_longdoc(x)
```

```
    Über eine mehrzeilige Zeichenkette
```

```
    kann auch sehr langer Dokumentationstext eingegeben werden.
```

Variablenzugehörigkeit in Funktionen (1)

- Python-Variablen haben bestimmten **Gültigkeitsbereich** (engl. scope):
 - Global in allen Funktionen
 - Lokal innerhalb einer Funktion
- Globale Variablen
 - Gelten überall, wo sie nicht durch lokale Variablen überdeckt sind
- Lokale Variablen
 - **Variablenzuweisungen** in einer Funktion machen Variablen im Normalfall zu **lokalen Variablen** innerhalb der Funktion
 - Lokale Variablen überdecken globale Variablen gleichen Namens
 - Mit der Anweisung **global** *<Name>* wird eine Variable in einer Funktion als global gültig definiert → Wertänderungen sind auch außerhalb des Gültigkeitsbereichs der Funktion sichtbar
 - Parameter einer Funktion sind lokale Variablen innerhalb der Funktion

Variablenzugehörigkeit in Funktionen (2)

- Intern verwaltet Python die Variablen eines Gültigkeitsbereichs in einem Variablen-Dictionary
- Den Inhalt des Variablen-Dictionaries im aktuellen Gültigkeitsbereich zeigt die Funktion `vars()` an
- Globale Variablen werden im globalen Variablen-Dictionary gespeichert
- Beim Aufruf einer Funktion ...
 - wird ein neues Variablen-Dictionary für die lokalen Variablen der Funktion erzeugt,
 - macht Python dieses Variablen-Dictionary zum aktuell gültigen
- Ist die Funktion abgeschlossen, wird wieder das Variablen-Dictionary des vorherigen Kontextes (global bzw. andere Funktion) aktiv

Variablenzugehörigkeit in Funktionen (3)

- Wertzuweisung an Variablen:
 - aktuell gültiges Variablen-Dictionary wird verwendet
 - Ausnahme: mit dem **global**-Befehl als global markierte Variablen werden im globalen Variablen-Dictionary bearbeitet
 - Weitere Ausnahme: mit dem **nonlocal**-Befehl als nicht-lokal markierte Variablen werden in einem der übergeordneten Variablen-Dictionary bearbeitet (nur relevant, wenn Funktionen verschachtelt definiert werden)
- Abfragen von Variablen:
 - Variable zuerst im aktuell gültigen Variablen-Dictionary suchen,
 - Wenn da nicht vorhanden, wird der erste Fund in den Variablen-Dictionaries des übergeordneten Kontexts verwendet (gleich global oder übergeordnete Funktion bei verschachtelten Funktionen)

Beispiel zum Gültigkeitsbereich von Variablen

```
def addiere(val):      # val als Param lokale Variable (=Lokvar)
    global summe        # summe ist globale Variable
    alt=summe           # Lokvar alt, da Zuweisung
    summe += val
    return summe

# Definiere globale Variablen (Globvars)
summe=0                # Globvar summe
alt=0                  # Globvar alt
val=42                 # Globvar val

print(addiere(1))      # Ausgabe: 1
print(addiere(2))      # Ausgabe: 3
print(addiere(3))      # Ausgabe: 6

# addiere() verändert nur Globvar summe, Globvars alt
# und val in addiere() durch Lokvars überdeckt
print(alt, summe, val) # Ausgabe: 0 6 42
```


Funktionsobjekte

- Mit Funktionsdefinition über **def** *<Name>* ... wird ein Objekt vom Typ Funktion (= Funktionsobjekt) erzeugt
- Das Funktionsobjekt
 - enthält Beschreibung der Parameter und Anweisungen
 - wird von der Variable *<Name>* referenziert
 - kann auch von anderen Variablen referenziert werden
 - Wird ausgeführt mit der Anweisung *<Name>* (...)

```
# Objekt vom Typ Funktion erzeugen
```

```
# und an Variable mypow zuweisen
```

```
def mypow(x, y):  
    return x**y
```

```
>>> mypot=mypow      # Variable mypot zeigt auch auf Pot-Funktion  
>>> a=mypot(2,2)     # Pot-Funktion ausführen und Erg. ausgeben  
>>> b=mypow(2, 2)    # Pot-Funktion ausführen und Erg. ausgeben  
>>> mypow = None     # mypow referenziert jetzt keine Funktion mehr  
>>> b=mypow(2, 2)    # Funktionsaufruf über mypow damit Fehler  
>>> a=mypot(2, 2)    # Funktionsaufruf über mypot geht aber noch
```

Funktionen als Parameter

- Da Funktionen in Python auch Objekte sind, können sie an Variablen zugewiesen, in Listen / Tupeln / Dictionarys gespeichert oder als Parameter an Funktionen übergeben werden
- Beispiel für eine Funktion als Parameter:

```
# tb_funcobj.py
```

```
def foreach(f, *args):  
    res=[]  
    for arg in args: res.append(f(arg))  
    return res
```

```
def square(x): return x*x  
vec=foreach(square, 1, 2, 3)  
print(vec)
```

```
# Ausgabe:  
# [1, 4, 9]
```

- Bei Variablen, Funktionen und Funktionsparametern auch Typangaben möglich
- Syntax für Variablen:
`<Varname> : <Typ>`
`<Varname> : <Typ> = <Zuweisungs-Ausdruck>`
- Syntax für Funktionsparameter und Funktions-Rückgabewert
`def <funktionsname> (<param1> : <Typ> , ...) : ...`
`def <funktionsname> (<param1> : <Typ> , ...) -> <Typ>:`
- Mit Typangaben (= type annotations oder type hints) wird dokumentiert, für welchen Typ das entsprechende Python-Objekt vorgesehen ist
- Die Typzuordnung wird vom Interpreter nicht beachtet → Zuweisungen anderer Typen funktionieren weiterhin ohne Fehlermeldung / Warnung

Beispiele für Typ-Annotationen

```
# Rückgabewert "->" int oder float, Parameter vom Typ int
def annotated_sum(a: int, b: int) -> int | float:
    return a + b
```

```
# Typ-Annotation bei einer Variablen
sum1 : int = annotated_sum(3, 4)
```

```
# Typ-Annotation keine bindende Typ-Zuweisung
# Interpreter akzeptiert auch andere Typen für Parameter / Werte
sum2 : int = annotated_sum("hal", "lo")
```

```
# Typ-Annotation auch ohne Zuweisung möglich
# aber: dann existiert nur Annotation, nicht Variable selbst
a : int
print(a)          # hier jetzt Fehler => a unbekannt
```

List Comprehensions

List Comprehensions

- **List Comprehension:** Python-Ausdruck, der aus einer oder mehreren bestehenden Listen eine neue Liste erzeugen
- Die Elemente der bestehenden Liste(n) werden über einen Ausdruck transformiert in eine neue Ergebnisliste geschrieben
- Syntax:

```
[ <Ausdruck> for <Var> in <Liste> ]
```
- Vorgehen: für jedes Element von *<Liste>* nacheinander:
 - Dieses Element an *<Var>* zuweisen
 - Auswertung von *<Ausdruck>*
 - Auswertungsergebnis an Ergebnisliste anhängen
- *<Var>* ist eine lokale Variable innerhalb der List Comprehension

Beispiel zu List Comprehensions

```
>>> # vgl. tb_lambda.py => nur mit List Comprehension
>>> vec=[ 2*x*x for x in [1, 2, 3] ]
[2, 8, 18]

>>> # x nur lokale Variable innerhalb der List Comprehension
>>> x
... NameError: name 'x' is not defined
```

Zum Vergleich tb_lambda.py:

```
# tb_lambda.py

def foreach(f, *args):
    res=[]
    for arg in args: res.append(f(arg))
    return res

vec=foreach(lambda x: 2*x*x, 1, 2, 3)
print(vec)
```

Daten filtern mit List Comprehensions

- List Comprehensions auch zur Filterung bestehender Listen geeignet
- Durch eine zusätzliche If-Bedingung nach der For-Schleife werden Listenelemente nur an die Ergebnisliste weitergereicht, wenn die If-Bedingung erfüllt ist.
- Syntax:
[<Ausdruck> **for** <Var> **in** <Liste> **if** <Bedingung>]
- Beispiel

```
>>> werte=[ 23, 4.5, "text", 17, 48 ]
```

```
>>> # nur Elemente übernehmen, die ganze Zahlen sind
```

```
>>> [ x for x in werte if type(x)==int ]  
[23, 17, 48]
```


List Comprehension ⇔ For-Anweisung

- Bei der List Comprehension erscheint nach dem for-Ausdruck und auch nach der optionalen If-Bedingung **kein** Doppelpunkt
- Eine For-Schleife als Anweisung eröffnet neuen Ausführungsblock, der über einen Doppelpunkt (:) eingeleitet wird
- Beispiele:

```
# List Comprehension => kein Doppelpunkt
# kann rechte Seite einer Zuweisung sein, da Ausdruck
werte = [ x**2 for x in [0, 1, 2, 3, 4] ]
```

```
# List Comprehension mit Filter (ohne Doppelpunkt)
werte = [ x**2 for x in range(5) if x%2 ]
```

```
# for-Schleife als Anweisung => Doppelpunkt
# kann nicht rechte Seite einer Zuweisung sein, da Anweisung
for x in range(5):
    print(x)
```

Eingabe / Ausgabe

Eingabe mit `input()`

- Die Eingabe von Werten erfolgt in Python mit der Funktion `input()`
- Diese Funktion liefert immer eine Zeichenkette zurück
- Optional kann eine Zeichenkette als Eingabeaufforderung angegeben werden
- Die Eingaben können mit Umwandlungsmethoden ausdrücklich in andere Datentypen umgewandelt werden oder automatisch mit der `eval`-Funktion, die den passenden Typ für die übergebene Zeichenkette ermittelt (auch Listen, Tupel, ...)

Ziel-Datentyp	Umwandlung von <code>Z=input()</code>
Ganzzahl	<code>int(Z)</code>
Gleitkommazahl	<code>float(Z)</code>
Boolean-Wert	<code>bool(Z)</code>
Complex-Wert	<code>complex(Z)</code>
Automatisch	<code>eval(Z)</code>

Ausgabe mit print()

- Die Ausgabe von Werten erfolgt in Python mit der Funktion `print()`
- Allgemeine Syntax der print-Funktion:

```
print(<Wert1>, ..., sep=' ', end='\n')
```

- Werden mehrere Werte in einer Print-Anweisung ausgegeben, sind sie durch einen Separator getrennt, der mit dem optionalen benannten Parameter `sep` angegeben wird (Defaultwert ist Leerzeichen)
- Am Schluss wird noch das Zeichen ausgegeben, das über den optionalen benannten Parameter `end` angegeben wird (Defaultwert ist Zeilenwechsel)
- Es besteht auch die Möglichkeit, die Print-Ausgabe in eine Datei zu leiten. Dazu wird ein File-Objekt (siehe später) an den benannten Parameter `file` übergeben

Textdatei schreiben mit open()

- Dateien lassen sich mit der Funktion open() zum Schreiben öffnen
- Syntax:
`open(<Dateiname>, mode="w")`
- Die Funktion liefert ein File-Objekt zurück, das Methoden zur Bearbeitung der Datei besitzt:

Methoden (F=File-Objekt)	Aufgabe
<code>F.write(<Text>)</code>	Text in Datei schreiben
<code>F.close()</code>	Datei schließen

Textdatei lesen mit open()

- Dateien lassen sich mit der Funktion open() auch zum Lesen öffnen
- Syntax:
open (<Dateiname>, mode="r")
- Die Funktion liefert ein File-Objekt zurück, das Methoden zur Bearbeitung der Datei besitzt:

Methode (F=File-Objekt)	Aufgabe
F.read([<Anzahl>])	<Anzahl> Zeichen aus der Datei lesen. Wenn <Anzahl> fehlt, alle verbleibenden Zeichen lesen
F.readline()	Ein Textzeile aus der Datei lesen
F.readlines()	Alle Zeilen der Datei auslesen und als Liste zurückgeben
F.tell()	Aktuelle Position in der Datei ausgeben
F.seek(<Pos> [, <Basis>])	An die Position <Pos> in der Datei springen. <Basis>=0: relativ zum Dateianfang (Standardwert) <Basis>=1: relativ zur aktuellen Position <Basis>=2: relativ zum Dateende
F.close()	Datei schließen

Beispiele zum Schreiben und Lesen von Dateien

```
# tb_write.py
```

```
f=open("tabelle.txt", "w")
```

```
for idx in range(32, 128):  
    f.write("%3d %c\n" % (idx, chr(idx)))  
f.close()
```

```
# tb_read.py
```

```
f=open("tabelle.txt", "r")
```

```
for line in f:  
    li=line.strip()  
    print(li, end=" => ")  
    dat=li.split()  
    print(dat)
```

```
f.close()
```

Dateien mit with-Befehl öffnen

- Mit dem Befehl `with` lässt sich das Öffnen und Schließen einer Datei komfortabel automatisch ausführen.
- Zusätzlich öffnet der `with`-Befehl einen neuen Anweisungsblock, in dem die Anweisungen für das Lesen / Schreiben von Daten platziert werden können
- Wenn der Anweisungsblock abgeschlossen ist, wird die Datei automatisch wieder geschlossen → auch, falls ein Laufzeitfehler im Python-Code auftreten sollte
- Python-technisch erzeugt `open` einen sog. **Kontextmanager** → (mehr dazu mit `help("with")`)
- Beispiel:

```
with open("txt.txt", "r") as f: txt=f.read()
print(txt)
```

Module und Pakete

- Python unterstützt Technik der **modularen Programmierung**
- Dabei wird eine komplexe Aufgabenstellung in kleinere eigenständige Komponenten (= **Module**) aufgeteilt
- Die einzelnen Module der Aufgabenstellung können unabhängig entwickelt werden und sind oft auch für andere Projekte einsetzbar
- In Python ist ein Modul eine Datei (mit der Endung `.py`), die Funktionen / Klassen und Variablen enthält
- Programme laden Module mit dem Befehl **import** und nutzen damit die in den Modulen definierten Funktionen / Klassen / Variablen

- Syntax für Modul-Import:

```
import <Modulname> [ as <Neuname> ]
```

- Ohne as-Zweig im Import-Befehl wird ein Namensraum <Modulname> für die Elemente des Moduls erzeugt
- Bei vorhandenem as-Zweig kommen die Elemente des Moduls in den Namensraum <Neuname>

- Selektiver Import von Elementen eines Moduls in den aktuellen Namensraum:

```
from <Modulname> import <Name> [ as <Neuname> ]
```

- Import aller Elemente eines Moduls in den aktuellen Namensraum:

```
from <Modulname> import *
```

Zugriff auf Modulnamensraum

- Typischerweise entsteht beim Import eines Moduls ein neuer Namensraum
- Die Elemente des Moduls kommen dann in diesen Namensraum
- Der Zugriff auf die Modulelemente erfolgt über den Punkt-Operator (bzw. Operator zur Attributreferenz):

<Modulname>.<Elementname>

Vorteile durch Modulnamensraum

- Mit Verwendung des Modulnamensraums kann klar separiert werden, welche Funktion aus welchem Modul bzw. Paket stammt
- Dadurch wird verhindert, dass Komponenten aus importierten Modulen bereits existierende Namen überdecken
- Deshalb selten empfehlenswert, mittels `from xyz import *` alle Komponenten eines Moduls in den aktuellen Namensraum zu laden
- Zwei Beispiele für Probleme aus so einem Vorgehen:
 - Funktion `sin()` im Modul `math` und im Paket `numpy` definiert
 - Funktion `open()` ist eine Standard-Python-Funktion
 - durch Import von `os` über „`from os import *`“ wird diese Funktion durch `os.open()` überschrieben
 - `os.open()` verhält sich von der Programmierlogik ganz anders

Beispiele zu Import-Befehl

```
>>> sqrt(4)
... NameError: name 'sqrt' is not defined

>>> import math
>>> # Import erzeugt neuen Namensraum
>>> math.sqrt(4)
2.0

>>> import math as m
>>> m.sqrt(4)
2.0

>>> # selektiver Import => kein neuer Namensraum
>>> from math import sqrt
>>> sqrt(4)
2.0

>>> # selektiver Import mit Umbenennung
>>> from math import sqrt as wurzel
>>> wurzel(4)
2.0
```

Von Benutzern definierte Module

- Beim Import werden Module in bestimmten Verzeichnissen gesucht
- Die Python-Variable `sys.path` enthält eine Liste der Suchorte
- Der Eintrag `sys.path[0]` referenziert den Pfad des aktuellen Skripts
- Benutzereigene Module im gleichen Verzeichnis wie das aktuelle Skript (= lokale Module) können damit ohne spezielle Installation über den Import-Befehl geladen werden
- Dokumentation in benutzereigenen Modulen:
 - Eine Zeichenkette am Anfang der Python-Datei für das Modul dokumentiert die Bedeutung / Aufgabe eines Moduls
 - Einzelne Funktionen verwenden zugeordnete Doc-Strings
 - Hilfestellung zu einem geladenen Modul gibt es mit dem Befehl `help(<Modulname>)`

Beispiel für ein benutzereigenes Modul

```
# temperature.py (gekürzte Version zu ZIP-Archiv)
"""Modul zum Umwandeln von Temperaturen zwischen Celsius, Kelvin und
Fahrenheit"""

def celsius2kelvin(val):
    "Rechne Celsius-Angaben in Kelvin-Angaben um"
    return val + 273

def kelvin2celsius(val):
    "Rechne Kelvin-Angaben in Celsius-Angaben um"
    return val - 273

def celsius2fahrenheit(val):
    "Rechne Celsius-Angaben in Fahrenheit-Angaben um"
    raise NotImplementedError("celsius --> fahrenheit")

def fahrenheit2celsius(val):
    "Rechne Fahrenheit-Angaben in Celsius-Angaben um"
    raise NotImplementedError("fahrenheit --> celsius")
```

```
# tb_temperature.py

from temperature import celsius2kelvin as c2k
import temperature as t

print(c2k(35), t.celsius2kelvin(35))
print(t.celsius2fahrenheit(-10))
```


- Mehrere Module können zu einem **Paket** (engl. Package) zusammengefasst werden
- Ein Paket ist ein Dateisystem-Verzeichnis mit folgendem Inhalt:
 - Die Python-Dateien der beinhalteten Module
 - (seit Python 3.3 optional) eine Initialisierungsdatei `__init__.py`
- Die optionale Initialisierungsdatei ...
 - wird beim Import des Pakets geladen und ausgeführt
 - kann z.B. weitere Module nachladen

Standard-Bibliotheken in Python

- Das Modul os dient zur (weitgehend) plattform-unabhängigen Verwendung von Betriebssystem-Funktionen
- Zu den Funktionen gehören z.B.:
 - Arbeiten mit Dateien: `os.stat()`, `os.chdir()`, ...
 - Arbeiten mit Dateipfaden: `os.path.*`
 - Aufruf externer Programme z.B. mit `os.system()`

```
# tb_script_path.py
```

```
import os
```

```
# Skriptpfad und -namen ausgeben
```

```
print(__file__)
```

```
print(os.path.basename(__file__))
```

```
# Gerätemanager aufrufen, wenn unter Windows
```

```
if os.name == "nt": os.system("devmgmt.msc")
```

- Das Modul shutil stellt Funktionen zum Kopieren von Dateien und Verzeichnisstrukturen bereit
- Zu den Funktionen gehören z.B.:
 - Kopieren von Dateien: `shutil.copyfile()`
 - Kopieren ganzer Verzeichnisse: `shutil.copytree()`
 - Verschieben von Dateien: `shutil.move()`

- Modul glob
 - Funktion `glob.glob()`: Dateien in einem Verzeichnis als Liste zurückgeben. Es kann auch optional ein Namensmuster angegeben werden. Dann werden nur Dateien zurückgegeben, die diesem Muster entsprechen.
- Modul pathlib
 - Eignet sich zum Erzeugen von Verzeichnissen, Auslesen von Verzeichnislisten
 - Überschneidungen zu den Modulen `os` und `glob`
 - Pathlib besitzt aber ein objektorientiertes Interface

- Das Modul math stellt Funktionen bereit, um mathematische Berechnungen mit Zahlen durchzuführen
- Zu den Funktionen gehören z.B.:
 - `math.exp()`, `math.log()`, `math.log2()`, `math.log10()`, ...
 - `math.sin()`, `math.tan()`, `math.cos()`, ...
- Außerdem werden die Konstanten `math.pi` und `math.e` bereitgestellt

- Das Modul sys stellt Funktionen und Variablen bereit, die für die Arbeit mit dem Python-Interpreter wichtig sind
- Dazu gehören Variablen und Funktionen, die Informationen zum Zustand des Python-Interpreters geben oder sein Verhalten beeinflussen
- Darüber hinaus enthält das Modul die Variable `sys.argv`, in der die Kommandozeilen-Parameter als Liste aufgeführt sind, mit denen der Python-Interpreter gestartet wurde.
 - Damit können Parameter an ein Python-Skript übergeben werden

Beispiel zu sys.argv

Python-Skript tb_argv.py:

```
# tb_argv.py

import sys

if __name__ == "__main__": print(sys.argv)
```

Ausgabe der Windows-Konsole cmd.exe:

```
# Ausgabe Windows-CMD

V:\>tb_argv.py
['V:\\tb_argv.py']

V:\>tb_argv.py dies ist ein Test
['V:\\tb_argv.py', 'dies', 'ist', 'ein', 'Test']
```


Python bietet standardmäßig viele weitere Module, z.B. für

- Arbeit mit Internetprotokollen
- Verwendung von Kompressionsalgorithmen (zip, lzma, ...)
- usw ...

Behandlung von Laufzeitfehlern (Exceptions)

Fehlerbehandlung zur Laufzeit

- Bei Fehlern zur Laufzeit eines Programms beendet Python im Normalfall die Programmausführung mit einer Fehlermeldung
- Beispiel: es wird eine Zahl durch 0 geteilt

```
# tb_error.py
```

```
def make_frac(txt):  
    inval=float(txt)  
    return 1/inval
```

```
txt=input("Nenner für 1/Zahl:")  
num=make_frac(txt)  
print("Bruch 1/%s=%g" % (txt, num))  
print("Ende der Berechnung")
```

```
# gekürzte Ausgabe => Eingabe 2
```

```
Nenner für 1/Zahl:2
```

```
Bruch 1/2=0.5
```

```
Ende der Berechnung
```

```
# gekürzte Ausgabe => Eingabe 0
```

```
Nenner für 1/Zahl:0
```

```
Traceback (most recent ...
```

```
ZeroDivisionError: float div ...
```

Laufzeitfehler zurückgeben mit raise

- Befehl **raise** dient zum Zurückmelden eines Laufzeitfehlers
- Argument des Befehls ist Art des Fehlers
 - z.B. `NotImplementedError`, `DivisionByZeroError`, ...

```
# tb_raise.py
```

```
def make_frac(txt):  
    if txt.lower() == "inf":  
        raise NotImplementedError("inf")  
    if txt.lower() == "nan":  
        raise NotImplementedError("nan")  
    inval=float(txt)  
    return 1/inval  
  
txt=input("Nenner für 1/Zahl eingeben:")  
num=make_frac(txt)  
print("Bruch 1/%s=%g" % (txt, num))  
print("Ende der Berechnung")
```

```
# gekürzte Ausgabe  
Nenner für 1/Zahl eingeben:inf  
... NotImplementedError: inf  
  
# gekürzte Ausgabe Neuaufruf  
Nenner für 1/Zahl eingeben:0  
ZeroDivisionError: float div ...  
  
# gekürzte Ausgabe Neuaufruf  
Nenner für 1/Zahl eingeben:1  
Bruch 1/1=1  
Ende der Berechnung
```

Laufzeitfehler mit try/except abfangen

- Laufzeitfehler können über **try-except**-Block abgefangen werden
- Der **try**-Befehl markiert Codeblock, in dem Fehler abgefangen werden
- Codeblock nach Befehl **except** wird zur Fehlerbehandlung aufgerufen

```
# tb_except_1.py
```

```
def make_frac(txt):  
    if txt.lower() == "inf":  
        raise NotImplementedError("inf")  
    if txt.lower() == "nan":  
        raise NotImplementedError("nan")  
    inval=float(txt)  
    return 1/inval  
  
try:  
    txt=input("Nenner für 1/Zahl eingeben:")  
    num=make_frac(txt)  
    print("Bruch 1/%s=%g" % (txt, num))  
except Exception:  
    print("Fehler")
```

Syntax von try-Blöcken

```
try:
    # Codeblock, in dem Fehler abgefangen werden
    <Anweisungen>
except <Fehlerart>:
    # Codeblock, der bei Auftreten von <Fehlerart> ausgeführt wird
    # weitere Blöcke mit except <Fehlerart> können folgen
    <Anweisungen>
except <Fehlerart> as <Name>:
    # Codeblock, der bei Auftreten von <Fehlerart> ausgeführt wird
    # Variable <name> enthält Objekt zur Beschreibung der Fehlerart
    <Anweisungen>
except:
    # Codeblock, der bei Auftreten beliebiger Fehler ausgeführt wird
    <Anweisungen>
else:
    # optionaler Codeblock, wird ausgeführt, wenn es keine Fehler gab
    <Anweisungen>
finally:
    # optionaler Codeblock, wird am Schluss immer ausgeführt
    # sogenannter "Aufräum-Handler"
    # entweder 1+ except-Blöcke oder ein finally-Block notwendig
    <Anweisungen>
```

Beispiel zur Fehlermeldung / Fehlerabfrage

```
# tb_except_2.py

def make_frac(txt):
    if txt.lower() == "inf": raise NotImplementedError("inf")
    if txt.lower() == "nan": raise NotImplementedError("nan")
    inval=float(txt)
    return 1/inval

try:
    txt=input("Nenner für 1/Zahl eingeben:")
    num=make_frac(txt)
except ZeroDivisionError:
    print("1/0 geht nicht")
except ValueError:
    print("Sie haben keine Zahl eingegeben")
except Exception as e:
    # Fehlerobjekt an Variable e zuweisen
    print("Fehler:", e.__class__, e.args)
else:
    print("Bruch 1/%s=%g" % (txt, num))
finally:
    print("Ende der Berechnung")
```

Weiteres Beispiel zur Fehlerbehandlung

```
# tb_except_3.py
```

```
def func():  
    try:  
        return "one"  
    finally:  
        return "two"
```

```
# Ausgabe ist immer two  
print(func())
```

```
# Minimaler try-except-Aufruf
```

```
try:  
    raise Exception("test it again")  
except:  
    print("** exception occurred **")
```

```
# Minimaler try-finally-Aufruf
```

```
try:  
    raise Exception("test it")  
finally:  
    print("** we will always be called **")
```

```
# gekürzte Ausgabe
```

```
two  
** exception occurred **  
** we will always be called **  
Traceback (most recent ...  
Exception: test it
```

Funktionale Programmierung

- Häufig verwendete Programmierstile sind imperative und objektorientierte Programmierung
 - Imperativ → Programmieren mit Variablen, Befehlen, Funktionen
 - Objektorientiert → Programmieren mit Klassen und Instanzen (siehe nächstes Kapitel)
- Daneben existiert noch der Programmierstil der funktionalen Programmierung mit folgenden Eigenschaften:
 - Funktionen sind keine Folge von Befehlen, sondern verketteten andere Funktionen, die ineinander verschachtelt sind
 - Funktionen werden wie andere Datenobjekte behandelt, können also auch Parameter oder Rückgabewerte von Funktionen sein
 - Anonyme Funktionen ohne Namen können anstelle von Funktionsnamen verwendet werden → sogenannte Lambda-Funktionen
 - Möglichst Verzicht auf Speicherelemente (= Variablen)

Funktionale Programmierung in Python

- Python unterstützt Konzepte der Funktionalen Programmierung

- Funktionen als Verkettung von Funktionen → Beispiel:

hier kein Befehl enthalten, nur Funktionen

```
print("Summe ist ", sum([3*x+1 for x in range(10)]))
```

- Funktionen als Datenobjekte :

```
li=[7,-3,1]
```

Funktion abs als Übergabeparamter

```
li.sort(key=abs)
```

- Verwendung anonymer Funktionen:

anonyme Funktion als Übergabeparamter

```
li.sort(key=lambda x: 3*x+1)
```

Objektorientierte Programmierung

Beschreibung von Datenstrukturen in Python

- Bisher: Beschreibung komplexer Datensätze mit Tupeln oder Listen
- Beispiel aus dem Kapitel zu Tupeln:

```
# generische Beschreibung einer Datei als Tupel
t=("readme.txt", "2023-03-06", 2346)
# Zuweisung der Datenfelder an Variablen
name,date,size=t
```

- Mit dem Befehl `class` können eigene Datentypen definiert werden
- Vorteil: eigene Datentypen stellen Felder ihrer Datensätze meist besser lesbar dar

Eigene Datenstrukturen / Datentypen in Python

- Befehl **class** definiert einen neuen benutzereigenen Datentyp

- Syntax zur Definition eines neuen Datentyps:

```
class <Typname>:  
    <optionaler Docstring als Beschreibung>  
    <Anweisungen>
```

- Syntax zur Instanziierung eines Datensatzes:

```
<Instanzname> = <Typname> ( )
```

- Syntax zum Zugriff auf Felder des Typs / Datensatzes:

```
<Instanzname>.<Feldname>
```

```
<Typname>.<Feldname>
```

Der Punkt arbeitet hier als **Operator zur Attribut-Referenz**

Beispiel für Verwendung einer Datenstruktur

```
# Beschreibung einer Datei als Tupel:
```

```
t=("readme.txt", "2023-03-06", 2346)
```

```
# Zuweisung der Datenfelder an Variablen
```

```
name,date,size=t
```

```
# Oder: eigenen Datentyp File erzeugen
```

```
# nach ":" muss Anweisung kommen, deswegen hier pass
```

```
class File: pass
```


```
# Datensatz erzeugen und Felder schreiben
```

```
datei = File()
```

```
datei.name = "readme.txt"
```

```
datei.date = "2023-03-06"
```

```
datei.size = 2346
```



Zur Laufzeit können beliebig
Felder angefügt werden

```
# und Felder wieder auslesen
```

```
print("Der Name der Datei ist", datei.name)
```

Funktionen an Datentypen binden

- Ein benutzereigener Datentyp kann auch Funktionen besitzen
- Diese Funktionen arbeiten typischerweise mit den Feldern eines Datensatzes
- Mit einer speziellen **Initialisierungsfunktion** kann z.B. ein Datensatz automatisch bei seiner Erzeugung initialisiert werden
- Diese **Initialisierungsfunktion** trägt den Namen `__init__` und wird im Anweisungsblock zur Beschreibung des Typs definiert
- Erster Parameter der Funktion `__init__` ist das zu initialisierende Objekt, dann folgen die Parameter aus dem Aufruf zur Erzeugung des Datensatzes

Datentyp mit Initialisierungsfunktion

```
# Beschreibung einer Datei mit dem Datentyp File
class File:
    # Felder des Datensatzes initialisieren
    def __init__(self, name, date, size):
        self.name = name
        self.date = date
        self.size = size

# Datensatz erzeugen
# die Funktion __init__ wird automatisch ausgeführt
# der erste Parameter zeigt auf den Datensatz (=self)
# die weiteren Parameter kommen aus der Klammer () bei
# der Erzeugung des Datensatzes
datei = File("readme.txt", "2023-03-06", 2346)

print("Der Name der Datei ist", datei.name)
```

Eigene datensatzbezogene Funktionen

- Neben der Initialisierungsfunktion sind auch beliebig weitere Funktionen möglich
- Beispiel: eine datensatzbezogene Funktion berechnet Dateigröße in KB

```
import math
```

```
class File:
```

```
    def __init__(self, name, date, size):
```

```
        self.name = name
```

```
        self.date = date
```

```
        self.size = size
```

```
    def getKBSize(self):
```

```
        return math.ceil(self.size/1024)
```

```
datei = File("readme.txt", "2023-03-06", 2346)
```

```
kbsize = datei.getKBSize()
```

```
print(f"{datei.name} ist {kbsize} KB groß")
```

Objektorientierte Programmierung in Python

- Die Erzeugung und Verwendung eigener Datentypen und Datensätze in Python orientiert sich an der Vorgehensweise der Objektorientierten Programmierung (OOP)
- Folgende OOP-Begrifflichkeiten existieren

Neuer Datentyp	→ Klasse
Datensatz	→ Instanz einer Klasse bzw. Objekt
Datentyp-Funktion	→ Methode einer Klasse

- Klassen, Instanzen und Methoden einer Klasse werden alle als Python-Objekte dargestellt und können zur Laufzeit eines Programms verändert werden

OOP-Begrifflichkeiten zum Beispiel

- Im vorherigen Beispiel beschreibt die **Klasse** File einen neuen Datentyp zur Darstellung von Dateien
- Ein konkreter Datensatz heißt **Instanz** der Klasse bzw. **Objekt**

Klasse	→ Beschreibung einer Datensatz-Struktur
Instanz / Objekt	→ konkreter Datensatz / konkrete Werte

- Vorheriges Beispiel:

File	→ Klasse
datei	→ Instanz / konkreter Datensatz
datei.name, ...	→ Felder der Instanz

- Über Klassen unterstützt Python die objektorientierte Programmierung (OOP)
- Denkweise bei der OOP:
 - Ein Programmablauf ist die Kommunikation von **Objekten**
 - **Methoden** der Objekte dienen als Schnittstellen zur Kommunikation
 - jedes Objekt ist Instanz einer oder mehrerer **Klassen**
 - die Klassen in einem Programm haben ein hierarchisches Verhältnis zueinander, das über Vererbungs- bzw. Ableitungsbeziehungen entsteht
- Ein Objekt besitzt dabei
 - Interne Variablen (= **Attribute**), die seinen Zustand beschreiben
 - Schnittstellen (= **Methoden**), um Operationen des Objektes auszuführen

Schritte bei der Klassendefinition

- Ein Objekt zur Beschreibung der Klasse wird erzeugt
- Dieses Objekt besitzt ein Dictionary für die Felder in der Klasse
- Anweisungen in der Klassendefinition werden abgearbeitet. Funktions- und Variablendefinitionen werden im Dictionary für die Felder gespeichert
- Das Objekt zur Klassenbeschreibung wird im aktuell gültigen Variablen-Dictionary durch Variablenzuweisung an den Klassennamen gebunden

Schritte beim Instanziieren einer Klasse

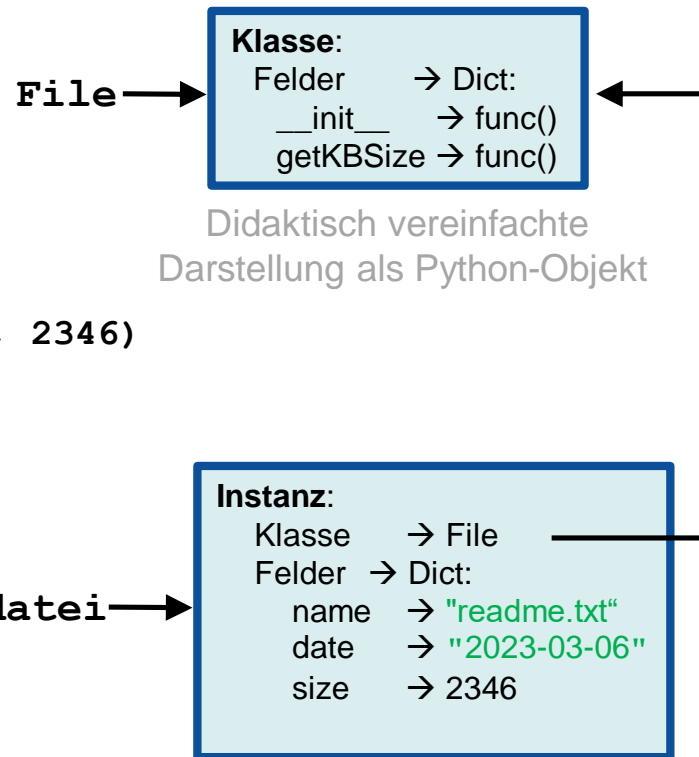
- Ein Objekt zur Beschreibung der Instanz wird erzeugt
- Dieses Objekt hat ein Dictionary für Felder in der Instanz
- Im Objekt für die Instanz wird eine Referenz auf die verwendete Klasse gespeichert
- Die Instanz wird initialisiert

Implementierung von Klassen und Instanzen in Python

```
class File:
    def __init__(self, name, date, size):
        self.name = name
        self.date = date
        self.size = size

    def getKBSize(self):
        return math.ceil(self.size/1024)

datei = File("readme.txt", "2023-03-06", 2346)
kbsize = datei.getKBSize()
```



Zugriff auf Felder einer Instanz

- Der Zugriff auf Felder einer Instanz erfolgt mit dem Operator zur Attribut-Referenz in der Form

<Instanz>.<Feldname>

- Wertzuweisungen mittels Operator zur Attribut-Referenz:
 - Zuweisungen gehen in das Dictionary der Instanz
 - Felder können auch nach der Instanziierung eines Objekts in das Dictionary der Instanz geschrieben werden
- Abfrage von Werten der Felder mittels Operator zur Attribut-Referenz:
 - Feldname wird zuerst im Dictionary der Instanz gesucht,
 - Wenn da nicht vorhanden, wird auch im Dictionary der zugehörigen Klasse gesucht

Zugriff auf Felder einer Klasse außerhalb der Klassendefinition

- Außerhalb einer Klassendefinition findet der Zugriff auf Felder der Klasse über den Operator zur Attribut-Referenz statt in der Form

<Klassenname>.<Feldname>

- Wertzuweisungen:
 - Zuweisungen gehen in das Dictionary der Klasse
- Abfrage von Feldern:
 - Feldnamen zuerst im Dictionary der Instanz suchen,
 - Wenn da nicht vorhanden, wird die Klassenhierarchie abgesucht (→ siehe später)

Wertzuweisung an bereits erzeugte Klassen / Instanzen

```
# tb_class2.py

# leere Klasse erzeugen => keine Felder enthalten
class K: pass

# Klasse instanziiieren
c=K()

# nachträglich Feld an Instanz anfügen geht
# dann aber nicht in der Klasse K enthalten
c.var1 = 43
# print(K.var1) würde Fehler ergeben

# nachträglich Feld an Klasse anfügen geht auch
K.kvar = "kvar"

# die Instanz bekommt das mit
print(c.kvar)
```

Klassenspezifische Funktionen / Methoden

- Neben einfachen Wertzuweisungen kann eine Klassendefinition z.B. auch Funktionsdefinitionen enthalten, die Operationen für die Arbeit mit Instanzen der Klasse beschreiben
- Solche Funktionen werden als **Methoden** der Klasse bezeichnet
- Gezeigtes Beispiel: die Methode `getKBSize()` in der Klasse `File` berechnet Dateigröße einer Instanz in KB:

```
class File:
    def __init__(self, name, date, size):
        ...
    def getKBSize(self):
        return math.ceil(self.size/1024)
```

Methoden mit Instanzreferenz / self-Parameter

- Methoden beziehen sich gewöhnlich auf Instanzen einer Klasse
- Deshalb benötigen sie eine Referenz auf die zu verwendende Instanz
- diese Referenz wird als erster Parameter an die Methode übergeben
- Der Referenz-Parameter wird üblicherweise **self** genannt
- Aufrufbeispiel mit der Klasse File (`datei` ist Instanzreferenz):
 - Ausführliche Form: `File.getKBSize(datei)`
 - Kurzschreibweise: `datei.getKBSize()`
- die Kurzschreibweise wird häufig verwendet und automatisch in die ausführliche Form übersetzt

Variablenzugehörigkeit beim Ausführen einer Methode

- Methoden greifen nach den gleichen Mechanismen auf Variablen zu wie normale Funktionen
- Zugriffe auf Felder der Instanz erfolgen über den Methoden-Parameter `self` gefolgt vom Operator für die Attribut-Referenz
- Zugriffe auf Felder der Klasse erfolgen über den Namen der Klasse gefolgt vom Operator für die Attribut-Referenz

```
class File:
```

```
...
```

```
def getKBSize(self):  
    return math.ceil(self.size/1024)
```

```
datei = File("readme.txt", "2023-03-06", 2346)  
kbsize = datei.getKBSize()
```

→ Im Beispiel: `self = datei`

Methoden für spezielle Operationen

- Einige Methodennamen sind für spezielle Aufgaben reserviert
- Erscheinen diese Methoden in der Klassendefinition, werden sie in bestimmten Fällen automatisch ausgeführt
- besondere Methodennamen:

Methoden	Aufgabe / Operation
<code>__init__(self)</code>	Initialisierer → wird bei Erzeugen einer Instanz automatisch zur Initialisierung der Daten ausgeführt → kann noch weitere Parameter besitzen
<code>__del__(self)</code>	Finalisierer → wird beim Zerstören einer Instanz automatisch ausgeführt
<code>__str__(self)</code>	Methode, um Instanz als Zeichenkette darzustellen

Weitere, z.B. zum Überladen von Operatoren, usw. ...

Verwendung von Initialisierern / Finalisierern

- Ein **Initialisierer** wird verwendet, um die Daten einer Instanz zu initialisieren.
 - Der Initialisierer wird bei der **Erzeugung** des Objekts (= der Klasseninstanz) automatisch aufgerufen
- Ein **Finalisierer** wird vor dem **Löschen** einer Instanz aus dem Speicher automatisch aufgerufen
 - Es kann nicht direkt beeinflusst werden, wann ein Finalisierer aufgerufen wird
 - Ein Finalisierer wird möglicherweise nie aufgerufen, weil z.B. der Python-Interpreter unerwartet beendet wurde
 - Deshalb sollten kritische Ressourcen nicht erst im Finalisierer freigegeben werden

Beispiel zu Initialisierer / Finalisierer

```
# tb_del.py

class C:
    def __init__(self, name=""):
        self.name = name
        print("init of %s: id=%08X" \
              % (self.name, id(self)))

    def __del__(self):
        print("fini of %s, id=%08X" \
              % (self.name, id(self)))

clist=[C("c%d" % idx) for idx in range(5)]
print("** remove items")
clist[2]=None
del clist[3]
print("** clist")
for c in clist: print(c)
print("** end of prog")
```

Hier wird Garbage
Collection aktiv

```
# Ausgabe Windows-CMD
V:\>tb_del.py
init of c0: id=01D92210
init of c1: id=01DF96B0
init of c2: id=01DF9530
init of c3: id=01DF9A90
init of c4: id=01DF9D30
** remove items
fini of c2, id=01DF9530
fini of c3, id=01DF9A90
** clist
<__main__.C object at
0x01D92210>
<__main__.C object at
0x01DF96B0>
None
<__main__.C object at
0x01DF9D30>
** end of prog
fini of c0, id=01D92210
fini of c1, id=01DF96B0
fini of c4, id=01DF9D30
```

Überladung von Operatoren mit „Magic methods“

Methode (K=Klasse)	Aufgabe (inst=Instanz)
K.__add__(val)	Operation inst + val
K.__iadd__(val)	Operation inst += val
K.__radd__(val)	Operation val + inst
K.__sub__(val), ...	Operatoren sub / div / mul
K.__eq__(val), K.__ne__(val), K.__le__(val), ...	Vergleichsoperatoren
K.__setattr__(name, val) K.__getattr__(val)	Punkt-Operator für Attributs-Zuweisung /-Abfrage
K.__len__(), K.__abs__(), ...	Weitere Operationen für len(), abs(), usw.
K.__init__(), K.__del__()	Initialisierer / Finalisierer

→ **Magisch:** wie von „Zauberhand“ in bestimmter Situation automatisch ausgeführt

Beispiel zur Operatorüberladung

```
# tb_new_op.py
from math import sqrt

class MyComplex:
    def __init__(self, rval, ival):
        self.rval = rval
        self.ival = ival

    def __add__(self, obj):
        rval = self.rval
        ival = self.ival
        if isinstance(obj, MyComplex):
            rval += obj.rval
            ival += obj.ival
        else:
            rval += obj
        return MyComplex(rval, ival)

    def __iadd__(self, obj):
        if isinstance(obj, MyComplex):
            self.rval += obj.rval
            self.ival += obj.ival
        else:
            self.rval += obj

        return self
```

```
def __abs__(self):
    x, y = self.rval, self.ival
    rv = sqrt(x*x+y*y)
    return rv

def __str__(self):
    return "MyComplex(%g,%g) "
        %(self.rval, self.ival)
```

```
num1 = MyComplex(2, 3)
num2 = MyComplex(4, 1)
num3 = num1 + num2
num2 += 4

print("num1=", num1)
print("num2=", num2)
print("num3=", num3)
print("abs(num3)=", abs(num3))
```

```
# Ausgabe
num1= MyComplex(2, 3)
num2= MyComplex(8, 1)
num3= MyComplex(6, 4)
abs(num3)= 7.211102550927978
```

- Neue Klassen können auch bestehende Klassen erweitern
- Syntax zur Erweiterung bestehender Klassen:

```
class <Erweiterte-Klasse> ( <Elternklasse-1>, ... ) :  
    <optionaler Docstring als Beschreibung>  
    <Anweisungen>
```

- Die neue Klasse wird dann von einer oder mehreren bestehenden „Elternklassen“ abgeleitet
- Damit besitzt die neue Klasse alle Felder der Elternklasse(n)
- Felder, die in der erweiterten Klasse definiert werden, überdecken gleichnamige Felder in der Elternklasse / den Elternklassen

Einfaches Beispiel zur Klassenhierarchie

```
# tb_fahrzeug.py
```

```
class Fahrzeug:
    def __init__(self, max_kmh):
        self.max_kmh = max_kmh

    def bewege(self):
        print("bewege bewege")

    def berechne_fahrzeit(self, km_strecke):
        return km_strecke / self.max_kmh

    def gib_radzahl(self):
        raise NotImplementedError

class Auto(Fahrzeug):
    def bewege(self):
        print("brum brum")

    def gib_radzahl(self): return 4

class Fahrrad(Fahrzeug):
    def bewege(self):
        print("strampel strampel")

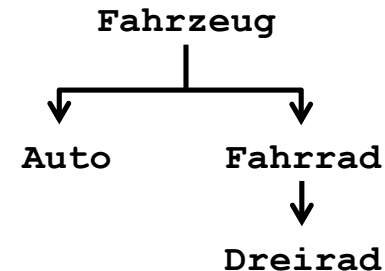
    def gib_radzahl(self): return 2
```

```
class Dreirad(Fahrrad):
    def gib_radzahl(self): return 3

a=Fahrrad(30)
b=Auto(200)
c=Dreirad(25)

transports=(a,b,c)
for tran in transports: tran.bewege()
```

Diagramm zur Klassenhierarchie



Zugriff auf Felder von Elternklassen

```
# tb_class_super.py
```

```
class A:  
    def get_ver(self):  
        return 0
```

```
class B(A):  
    def get_ver(self):  
        return 1
```

```
class C(B):  
    def get_ver(self):  
        return 2
```

```
    def get_b_ver(self):  
        return super().get_ver()
```

```
    def get_a_ver(self):  
        return super(B, self).get_ver()
```

```
c=C()  
print(c.get_ver())  
print(c.get_b_ver(), B.get_ver(c))  
print(c.get_a_ver(), A.get_ver(c))
```

```
# Ausgabe bei Ausführung
```

```
2  
1 1  
0 0
```

generischer Zugriff auf
Elternmethode mit super()

Direkter Zugriff auf
Elternmethode

Python-Darstellung von Klassen und Instanzen

```
class A:
    def __init__(self):
        self.someVar = 42
        self.ver = self.get_ver()
```

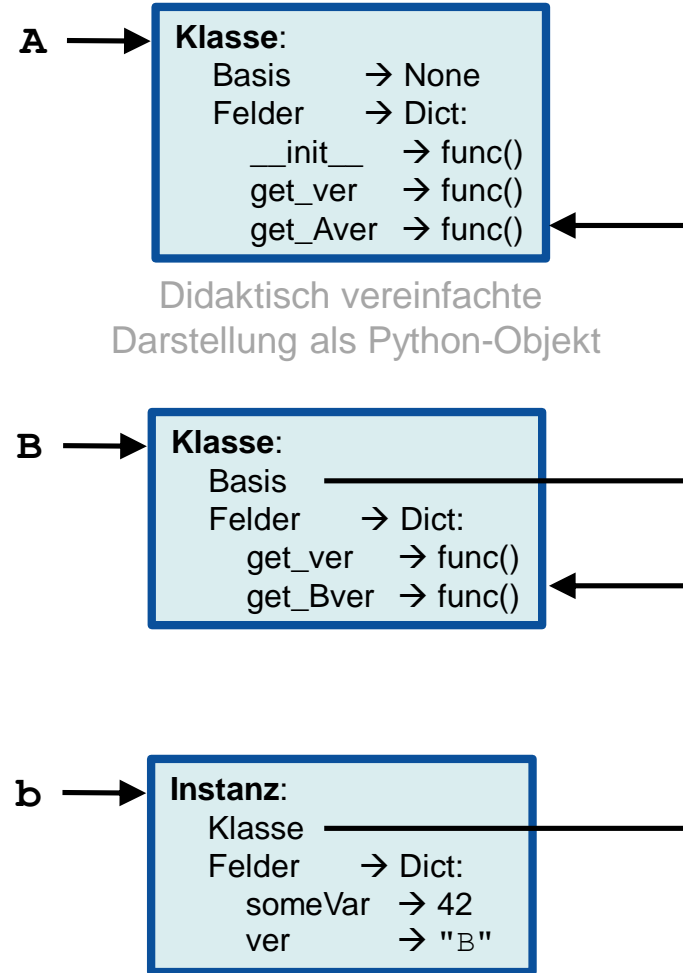
```
    def get_ver(self):
        return "A"
```

```
    def get_Aver(self):
        return "A"
```

```
class B(A):
    # get_ver polymorph
    def get_ver(self):
        return "B"
```

```
    def get_Bver(self):
        return "B"
```

```
b=B()
print(b.get_ver())
print(b.get_Aver())
```



Didaktisch vereinfachte
Darstellung als Python-Objekt

Punkt-Operator sucht Hierarchie bis ersten Treffer ab

Einschränkung des Feldzugriffs bei Klassen / Instanzen

- In Python wird der Zugriff auf Felder einer Klasse / Instanz über den Feldnamen eingeschränkt
- Felder mit mindestens zwei führenden Unterstrichen im Namen werden als **private Felder** einer Klasse / Instanz betrachten und **können** von außen nicht direkt angesprochen werden
 - Ausnahme: Im Namen folgen am Schluss mindestens zwei Unterstriche, wie z.B. bei `__init__`
- Felder mit einem führenden Unterstrich im Namen **sollen** als **geschützte Felder** betrachtet werden.
 - Lesen / Schreiben von außerhalb der Klasse zwar möglich
 - Aber: Empfehlung / Konvention, dies nicht zu tun

Beispiel zum Zugriffsschutz

```
# tb_protect.py

class K:
    __geheim = 1
    _nicht_weitersagen = 2
    def __init__(self): self.__geheim = 1
    def get_var(self):
        return self.__geheim * self._nicht_weitersagen

# Fehler, wenn if 1: print ...: __geheim wird in K nicht gefunden
if 0: print(K.__geheim)

# Der Trick: __geheim wurde umbenannt in _K__geheim
# Schema: Unterstrich + Klassennamen voranstellen
print(K._K__geheim)

# kein spezieller Schutz => _nicht_weitersagen wurde nicht umbenannt
# dieser Zugriff gilt aber als schlechte Praxis
print(K._nicht_weitersagen)

k=K()
print(k._K__geheim)      # gleiches Verhalten wie oben
if 0: print(k.__geheim)  # Fehler in print wg. Umbenennung!
```

Klassendefinition im Vergleich zwischen C++ und Python

C++

```
class K {  
    public: K(int val) {  
        m_val = val;  
    }  
  
    public: int getval() {  
        return m_val;  
    }  
  
    private: int m_val;  
};
```

Python

```
class K:  
    def __init__(self, val):  
        self.__m_val = val  
  
    def getval(self):  
        return self.__m_val
```

- In C++
 - Der Konstruktor (= die Initialisierungsroutine) heißt wie die Klasse
 - Instanzattribute müssen explizit deklariert werden
 - Zugriffsrechte auf Methoden und Attribute über public / ... angegeben
- In Python:
 - Die Initialisierungsroutine heißt `__init__`
 - Der this-Pointer ist explizit in Parameterdeklaration anzugeben, auch bei anderen Methoden → Name üblicherweise `self`
 - In der Initialisierungsroutine werden Attribute einer Instanz durch Zuweisung angelegt → Zugriffsberechtigung über vorangestellte Unterstriche

- **Autor**

Prof. Dr.-Ing. Jürgen Krumm

- **Impressum**

Prof. Dr.-Ing. J. Krumm, TH Nürnberg Georg Simon Ohm,
Fakultät Elektrotechnik Feinwerktechnik Informationstechnik,
Postfach 210320, 90121 Nürnberg, Germany,
Tel:+49-911-5880-1111,
E-mail: juergen.Krumm@th-nuernberg.de

Dieses Skriptum ist nur für den eigenen Gebrauch im Studium gedacht. Eine Weitergabe ist nur mit Zustimmung des Autors gestattet.