



Einführung in das Programmieren mit Python

Vortragender: Thomas Pipek

Grundlagen des Programmierens

»» Erste Schritte mit Python

Was ist Programmieren

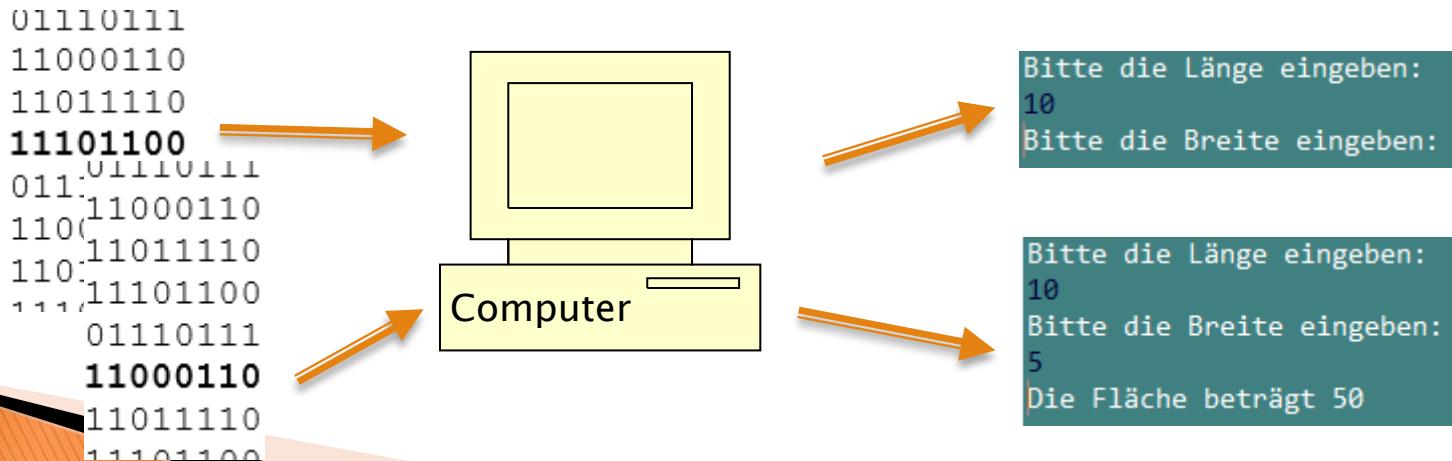
► Ein Computer

- ist ein Objekt das bestimmte Befehle versteht und ausführt

- versteht nur Befehle in "Maschinensprache"

► Ein Computerprogramm

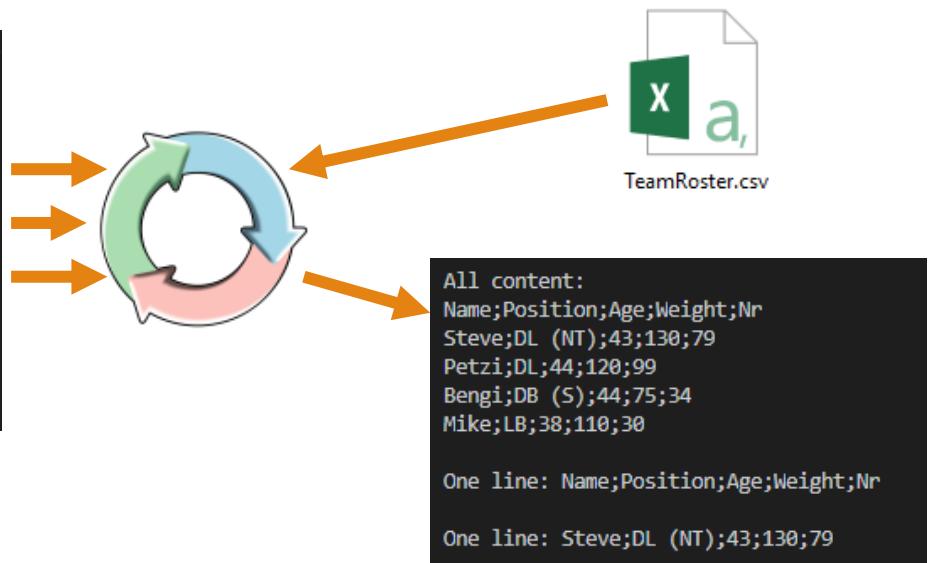
- ist eine Abfolge von Befehlen, welche der Computer der Reihe nach interpretiert und ausführt



Was ist Programmieren

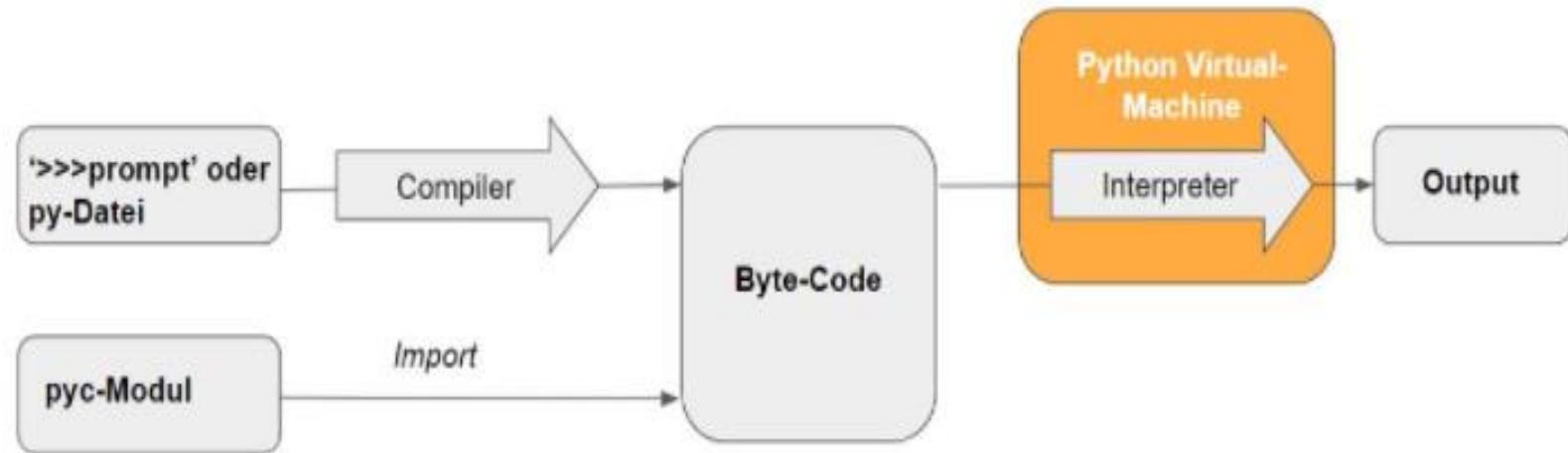
- ▶ Mit einer Programmiersprache
 - lassen sich die Befehle verständlicher darstellen.
- ▶ Ein Interpreter
 - liest die Befehle von der Programmiersprache und führt die Anweisungen am System aus

```
1 # reading a file
2 file = open("TeamRoster.csv", "r")
3
4 # read everything in 1 go
5 allContent = file.read()
6 print("All content:\n" + allContent)
7
8 file.seek(0, 0) # like C: 0 Bytes Offset, 0 = SEEK_SET from the beginning
9
10 # read lines
11 line = ""
12 while line != "":
13     line = file.readline() # contains the \n character!!!!
14     print("One line: " + line)
15
16 file.close()
```



Python Architektur

- ▶ Bei der Ausführung von Python-Programmen kommt eine Mischung aus Compiler und Interpreter zum Einsatz



Datentypen

- ▶ Programme arbeiten mit verschiedenen Arten von Werten, z.B.
 - ganze Zahl: 10
 - Fließkommazahl: 3.14
 - Zeichenfolge: "Hallo Python"
 - Wahrheitswert: true
- ▶ Der genaue Typ wird durch den **Datentyp** bestimmt
 - z.B. int oder bool

Tabelle: Primitive Datentypen

Typ	Inhalt
bool	Wahrheitswert (True / False)
int	Ganzzahl mit Vorzeichen
float	Fließkommazahl
complex	Komplexe Zahl
str	Zeichenketten, Strings

Python kennt kein statisches Typsystem, das genaue Verhalten kann vom verwendeten Interpreter abhängen.

Der **int** hat seit Python 3.x keine Größenbeschränkung, der **float** entspricht meistens dem C double.

Datentypen

Ganzzahl

Binär	Dezimal	Hex
00000000 00000000 00000000 00011011	27	0x0000001B
11111111 11111111 11111111 11100101	-27	0xFFFFFE5
10000000 00000000 00000000 00011011	-2147483621	0x8000001B

Vorzeichen

Fließkommazahl

Binär	Dezimal
00111110 10011001 10011001 10011010	0.3
10111110 10011001 10011001 10011010	-0.3



VZ Exponent Mantisse

Variablen

- ▶ Behälter zur temporären Ablage von Daten
 - Datentyp (-> Wertebereich und Verhalten)
 - zu einem Zeitpunkt genau einen Wert
 - Namen für den Zugriff
 - bestimmten Platz im Hauptspeicher
- ▶ Wert kann im Programmverlauf geändert werden



Konstante

- ▶ Werte, die im Programmablauf nicht verändert werden können
 - Literale
 - Ganzzahlen: 10
 - Fließkommazahlen: 3.14
 - Zeichenfolgen: "Hallo Python", 'Bye Bye Python'
 - Wahrheitswert: True
 - Leer-Wert (=Typ & Wert nicht definiert): None

Anlegen von Variablen

► "normale" Variablen

- Deklaration durch Definition

```
name = "Grumml"
```

```
anzahlTeilnehmer = 7
```

► Konstanten

- werden im Programmverlauf nicht geändert
- Schreibung in Großbuchstaben
- meist in eigenem Modul definiert
- nur per Konvention !!

```
MAX_TEILNEHMER = 10
```

```
FIRMEN_NAME = "Allesköninger AG"
```

Tabelle: Wichtige Operatoren

	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo
**	Potenzieren
=	Zuweisung
+ = - = * = / = % = ** =	Abkürzung für Operation plus Zuweisung

	Bedeutung
==	gleich
!=	ungleich
<	kleiner
>	größer
<=	kleiner oder gleich
>=	größer oder gleich
and	logisches UND
or	logisches ODER
not	logisches NOT

Python Programm

Variablen f. d.
Benutzerinput

Berechnung
(Operation)

Kommentar (gehört nicht zu
den Programm-Befehlen)

```
# Programm zur Berechnung der Fläche eines Rechtecks

print("Bitte die Länge eingeben:")
laenge = int(input())

print("Bitte die Breite eingeben:")
breite = int(input())

flaeche = laenge * breite;
print("Die Fläche beträgt {}".format(flaeche))
```

Ausgabe

- ▶ Als Schnittstelle zum Benutzer steht die Konsole (Textoberfläche) zur Verfügung
 - print() Funktion zur Ausgabe von Werten
 - beliebige Parameter (=Werte), werden mit Leerzeichen getrennt ausgegeben

```
print("Hallo Welt")          => Hallo Welt  
print(37.78)                 => 37.78
```

```
a = 42  
print(a)                     => 42
```

```
print("Wert von a ist", a)    => Wert von a ist 42
```

Formatierte Ausgabe

- ▶ Basis ist die `format`-Methode von `String`
 - Indizierte oder benannte Platzhalter mit {}
 - Zusätzliche Formatangaben möglich

```
text = "Wert von a ist {}".format(42)
print(text)                                => Wert von a ist 42
```

```
hour = 9
minute = 4
print("Zeit ist {:02d}:{:02d}".format(hour, minute))
=> Zeit ist 09:04
```

```
radius = 12.456
print("Ein Kreis mit Radius {:.2f} hat eine Fläche von
{:.2f}".format(r = radius, f = radius*radius*math.pi))
=> Ein Kreis mit Radius 12.46 hat eine
Fläche von 487.42
```

Formatierte Ausgabe

- ▶ Ab Python 3.6 gibt es String Interpolation
 - Shortcut für die `format()` Funktion
 - String wird mit dem Präfix 'f' oder 'F' angegeben

```
a = 42
text = f"Wert von a ist {a}"
print(text)                                => Wert von a ist 42
```

```
hour = 9
minute = 4
print(f"Zeit ist {hour:02d}:{minute:02d}")
=> Zeit ist 09:04
```

```
radius = 12.456
print(F"Ein Kreis mit Radius {radius:.2f} hat eine Fläche von
{radius*radius*math.pi:.2f}")
=> Ein Kreis mit Radius 12.46 hat eine
Fläche von 487.42
```

Eingabe

- ▶ Zum Einlesen von der Tastatur gibt es die Funktion `input()`
 - mit Eingabeaufforderung (optional)
 - liefert immer String zurück
 - Andere Datentypen müssen umgewandelt werden

```
name = input("Gib deinen Namen ein")      <= Chris  
print("Hallo", name)                      => Hallo Chris
```

```
text = input("Gib eine Zahl ein")    => "42"  
num = int(text)                      => 42
```

Algorithmische Grundstrukturen

» Programmablauf steuern

Algorithmus

- ▶ Ist eine Handlungsanweisung, bestehend aus elementaren Befehlen (Anweisungen)
 - Sequenz (nacheinander)
 - eine Anweisung wird nach einer anderen ausgeführt
 - Alternative
 - ein Teil des Algorithmus wird nur ausgeführt, wenn ...
 - Schleife (wiederholt ausführen)
 - ein Teil des Algorithmus wird wiederholt ausgeführt, und zwar solange ...

Sequenz



```
print("Bitte die Länge eingeben:")
laenge = int(input())

print("Bitte die Breite eingeben:")
breite = int(input())

flaeche = laenge * breite;
print("Die Fläche beträgt {}"
      .format(flaeche))
```

Beispiel Fläche berechnen: Aus- und Eingaben müssen in der richtigen Reihenfolge sein!

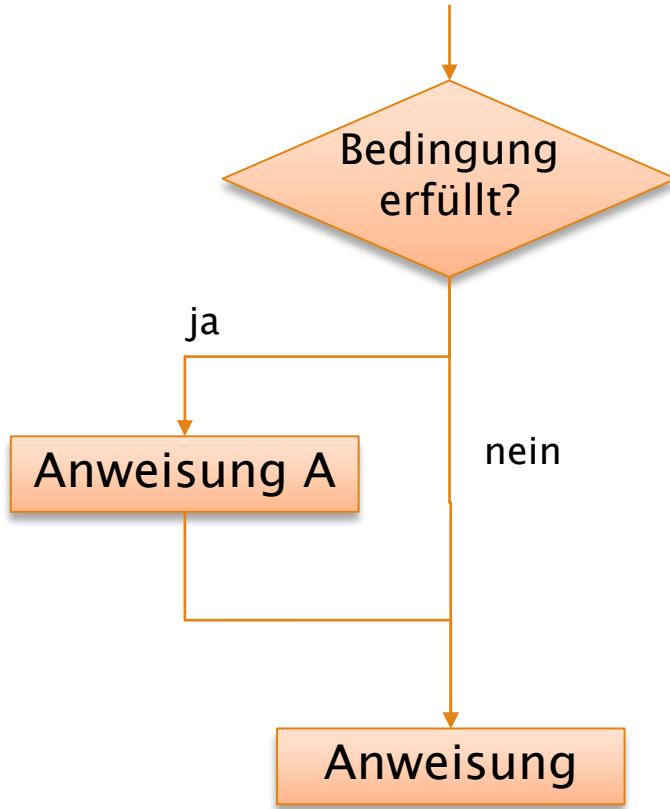
Verzweigung (Alternative)

```
if Ausdruck :  
    Anweisung(en)
```

```
if Ausdruck :  
    Anweisung(en)  
else :  
    Anweisung(en)
```

```
i = ...  
if i != 0 :  
    ..... # Anweisung(en) für den True-Fall  
else :  
    ..... # Anweisung(en) für den False-Fall
```

Verzweigung (Alternative)



```
preis = 300
```

```
...
```

```
jahre = ...
```

```
# für Stammkunden
```

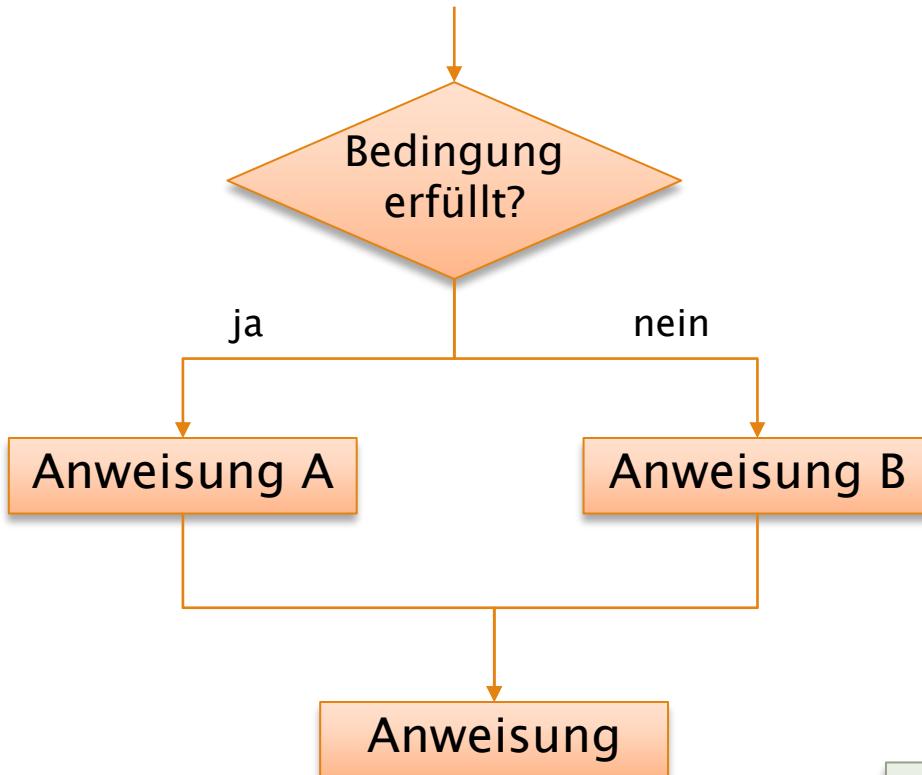
```
if jahre > 3 :
```

```
    # Rabatt abziehen
```

```
    preis -= 50
```

Beispiel Fitnesscenter: Testen ob es
ein Stammkunde ist

Verzweigung (Alternative)

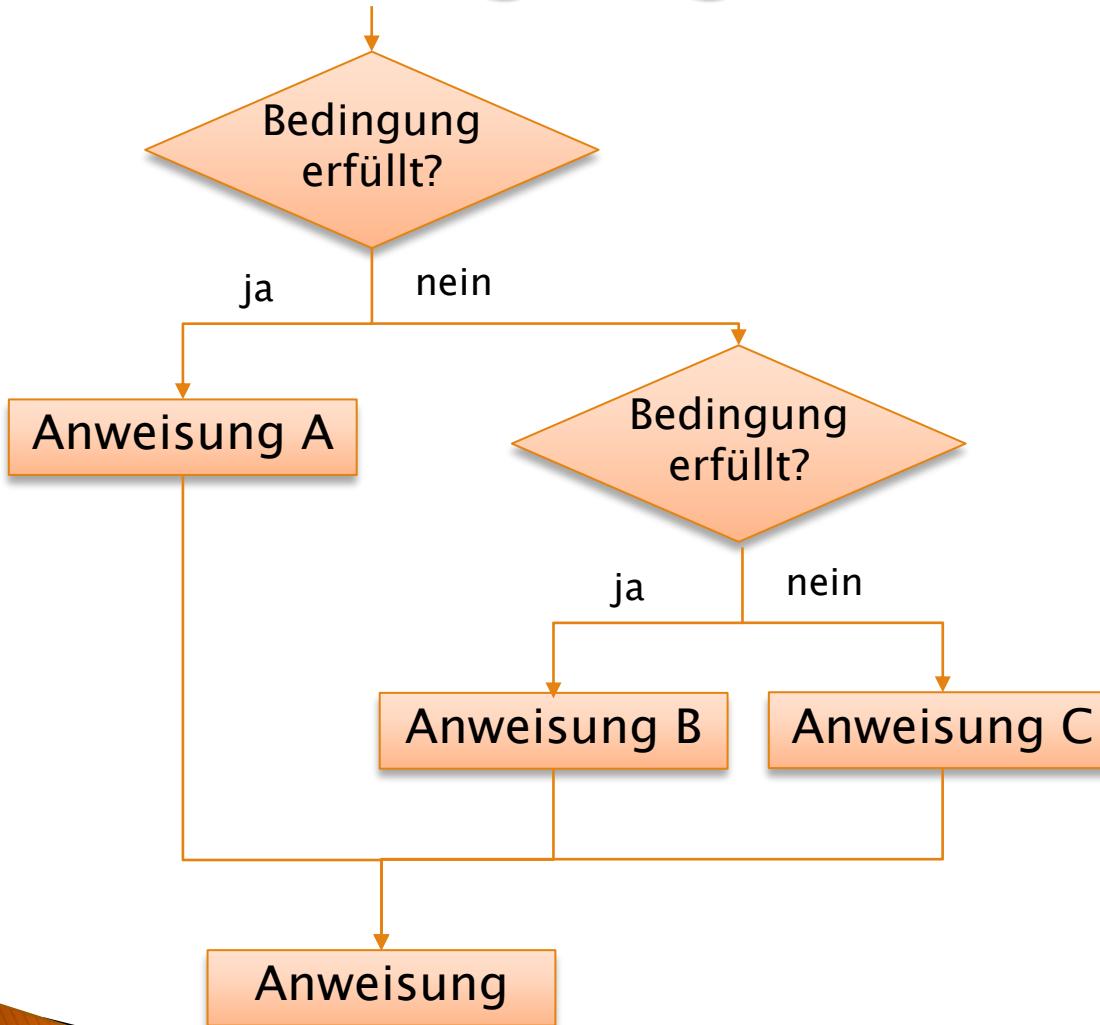


```
preis = 0  
student = ...
```

```
# wenn es ein Student ist  
if student == 'j' :  
    # Studenten-Preis  
    preis = 400  
else :  
    # sonst: Normal-Preis  
    preis = 500
```

Beispiel Fitnesscenter: Testen ob der Kunde Student ist

Verzweigung (Alternative)



```
preis = 0  
jahre = ...  
  
if jahre <= 14 :  
    # Kinder-Tarif  
    preis = 5  
elif jahre < 60 :  
    # Normal-Tarif  
    preis = 10  
else :  
    # Senioren-Tarif  
    preis = 8
```

Beispiel Fahrkarten:
Preisstaffel je
nach Altersgruppe

Boole'sche Ausdrücke

- ▶ Ob ein Ausdruck als Wahr (True) oder Falsch (False) erkannt wird, hängt vom Datentyp ab
- ▶ Als Falsch werden erkannt:
 - bool: False
 - int, float: 0 bzw. 0.0
 - str(ing): ""
- ▶ Alle anderen Werte werden als Wahr erkannt!

Match/Case ab Python 3.10

Match/Case von Python entspricht dem
Switch/Case von Java/C#

```
match operator:  
    case "+":  
        ergebnis = zahl1 + zahl2  
    case "-":  
        ergebnis = zahl1 - zahl2  
    case "*":  
        ergebnis = zahl1 * zahl2  
    case "/":  
        ergebnis = zahl1 / zahl2  
    case _:  
        print("Ungültiger Operator!")
```

Match/Case ab Python 3.10

Zusätzlich können Text-Matches durchgeführt werden

```
match rechnung.split():
    case [zahl1, "+", zahl2]:
        ergebnis = float(zahl1) + float(zahl2)
    case [zahl1, "-", zahl2]:
        ergebnis = float(zahl1) - float(zahl2)
    case [zahl1, "*", zahl2]:
        ergebnis = float(zahl1) * float(zahl2)
    case [zahl1, "/", zahl2]:
        ergebnis = float(zahl1) / float(zahl2)
    case _:
        print("Ungültiger Operator!")
```

Wiederholung (Iteration)

```
while Ausdruck :  
    Anweisung(en)
```

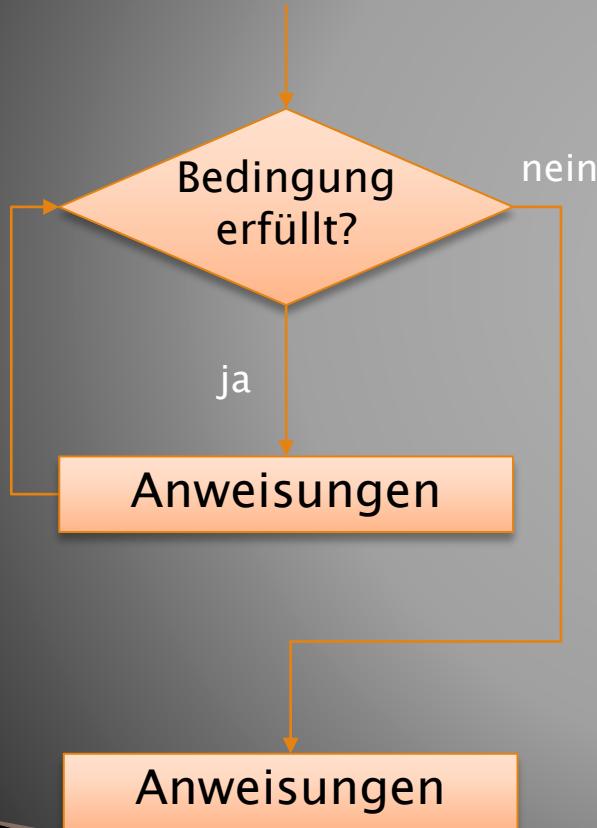
```
while Ausdruck :  
    Anweisung(en)  
else :  
    Anweisung(en)
```

- Anweisung bzw. Block wird 0 bis n Mal ausgeführt

```
i = 1;  
...  
while i < 9 :  
    # Block wird ausgeführt solange i < 9 ist  
    .....  
    i = i + 1  
else :  
    ..... # Block wird ausgeführt wenn Bedingung false liefert
```

Wiederholung (Iteration)

► while-Schleife



Bedingung steht im Schleifenkopf

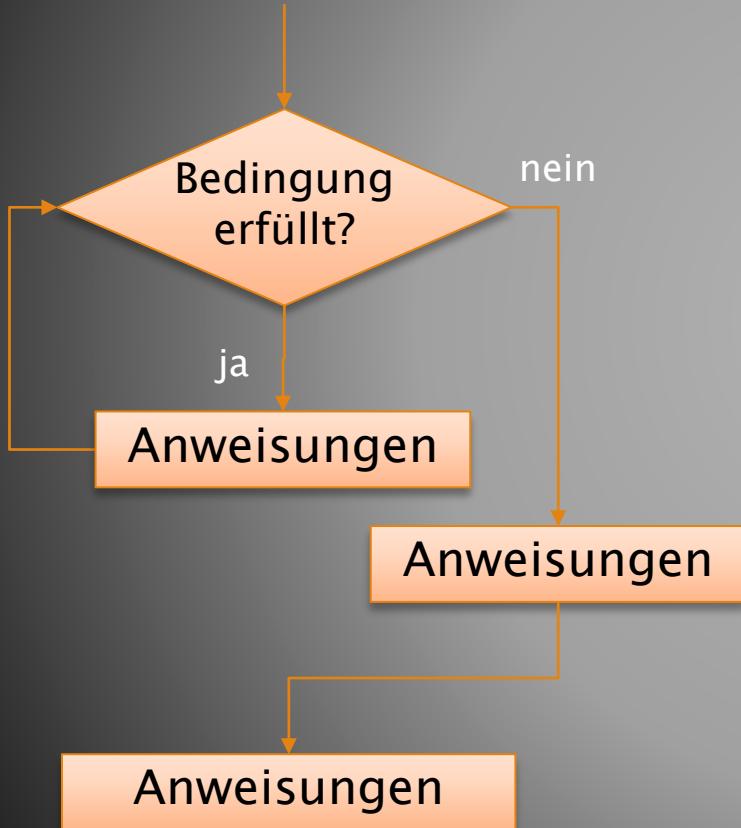
```
zahl = ...
ziffSum = 0
// solange zahl ungleich 0
while zahl != 0 :
    // berechnen der ziffer
    ziffer = zahl % 10
    // ... zur Summe hinzuzählen
    ziffSum += ziffer
    // ziffer entfernen
    zahl /= 10
```

Beispiel Ziffernsumme: letzte Ziffer dazuzählen solange die Zahl ungleich 0 ist

Wiederholung (Iteration)

Bedingung steht
im Schleifenkopf

► while-Schleife



```
myList = ...
# solange etwas in der Liste ist
while len(myList) > 0 :
    # Element holen
    myList.pop()
    # wenn len() 0 liefert
else :
    # Code ausführen
    print(...)
```

else wird immer ausgeführt, auch wenn die Schleife vorher durchlaufen wurde!

Wiederholung (for)

```
for iterator in sequenz :  
    Anweisung(en)
```

- Anweisung bzw. Block wird 0 bis N mal ausgeführt
- die Variable i existiert auch nach der for-Schleife

```
for i in range(5) :  
    # Block wird für jedes Element in range(5) ausgeführt  
    # => 0, 1, 2, 3, 4  
    .....  
    # variable i ist noch vorhanden, enthält den letzten Wert!  
    print(i)
```

Wiederholung (for)

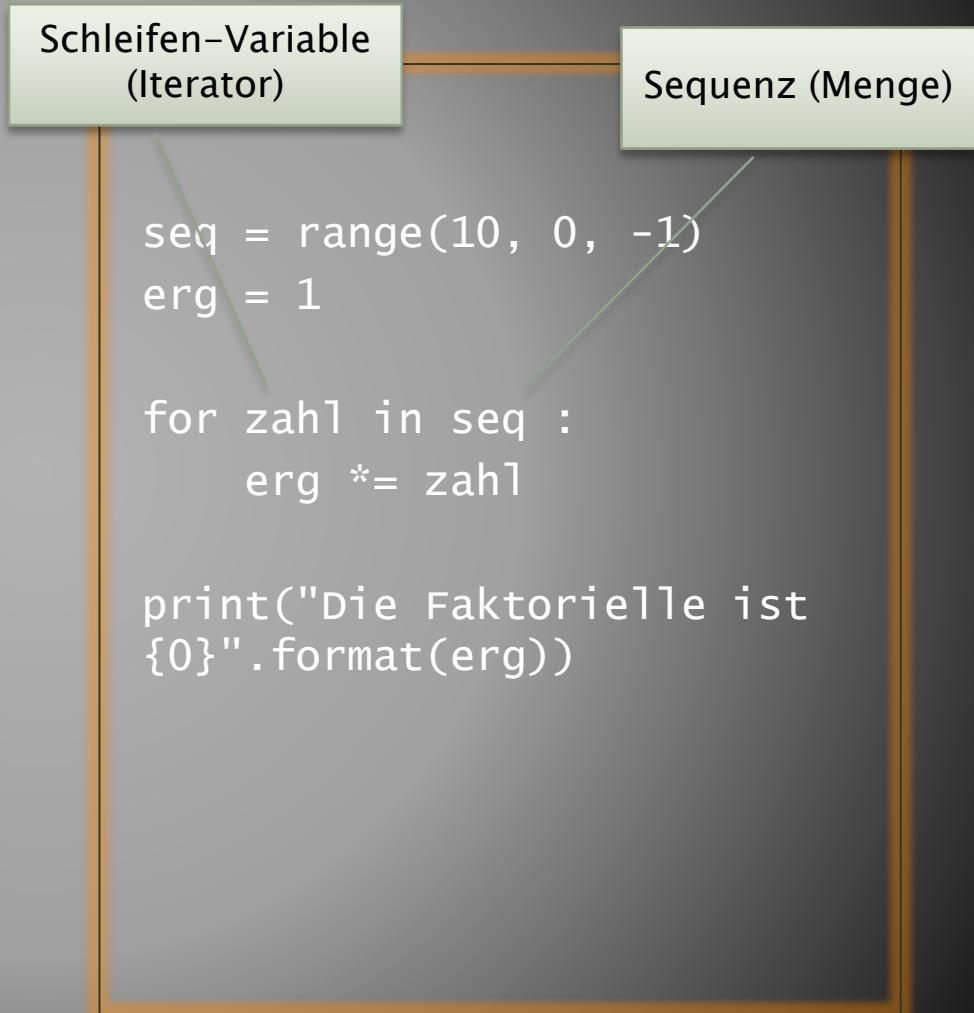
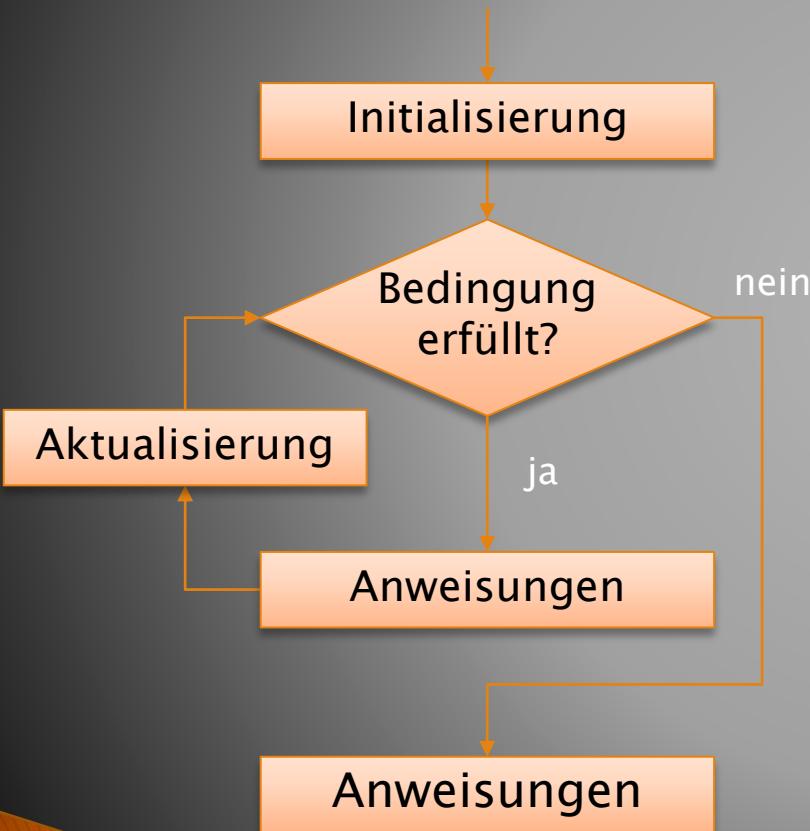
```
for iterator in sequenz :  
    Anweisung(en)  
else :  
    Anweisung(en)
```

- Anweisung bzw. Block wird 0 bis n Mal ausgeführt
- die Variable i existiert nur, wenn Schleife mindestens 1-mal durchlaufen wurde !!!

```
for i in range(5) :  
    # Block wird für jedes Element ausgeführt  
    .....  
else :  
    # wird nach Ende der Sequenz, oder wenn Sequenz leer ist  
    # ausgeführt  
    .....
```

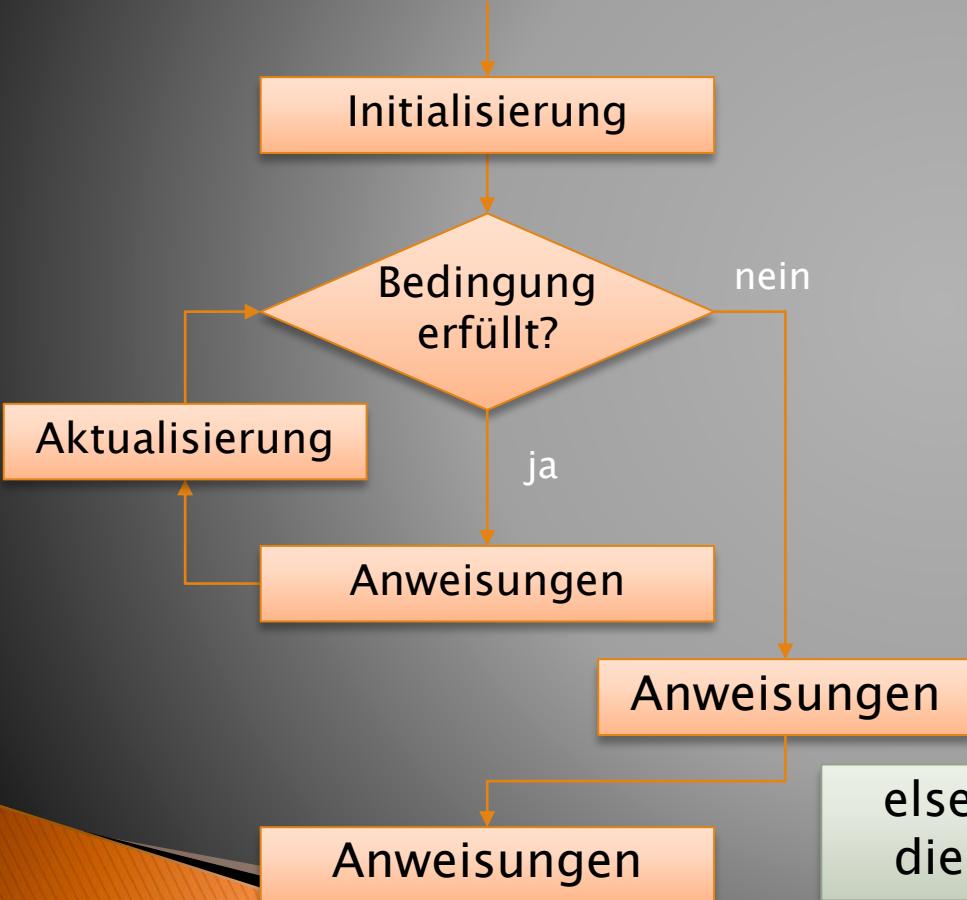
Wiederholung (for-Schleife)

► for-Schleife



Wiederholung (for-Schleife)

► for-Schleife



Schleifen-Variable
(Iterator)

Sequenz (Menge)

seq = range(...)

it enthält der Reihe nach
die Werte aus seq

for it : seq :
 # Aktion mit it

...

else :
 # seq abgearbeitet

...

else wird immer ausgeführt, auch wenn
die Schleife vorher durchlaufen wurde!

Kontrollstrukturen – break

```
zufall = ... # Zufallszahl generieren

versuch = 0
while versuch != zufall :
    versuch = int(input("Errate die Zahl: "))
    if versuch > 0 :
        if versuch > zufall :
            print("Zu groß")
        elif versuch < zufall :
            print("Zu klein")
        else :
            print("Schade, dass du aufgibst!")
            break
    else :
        print("Gratuliere, das war richtig")

print("Ende")
```

erlaubt in allen Schleifen

bei break wird das else der schleife nicht ausgeführt

Kontrollstrukturen – continue

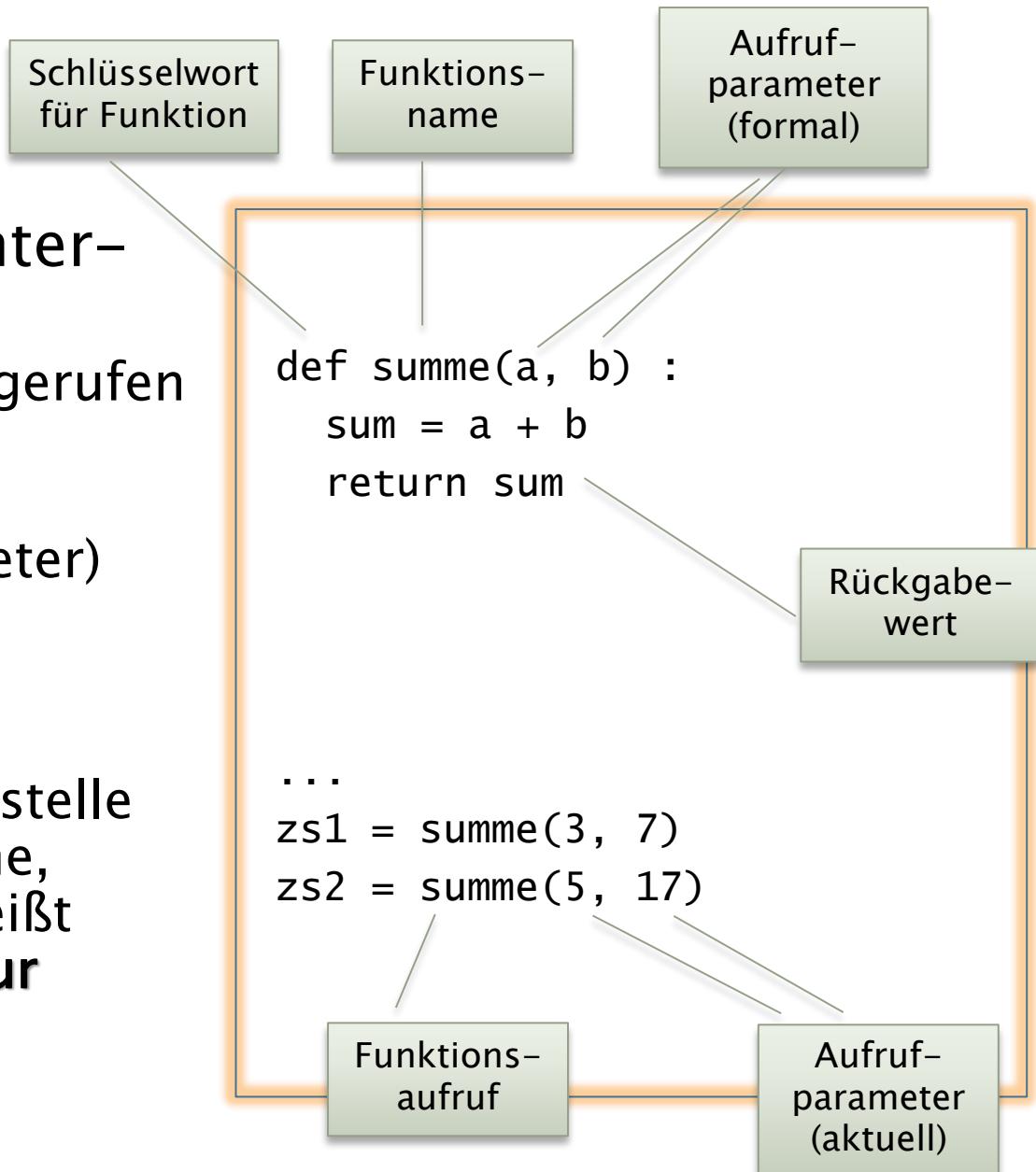
```
x = 0

while x < 10 :
    x += 1
    if x < 10 :
        continue
    print("x ist {}".format(x))
    ...
    ...
```

erlaubt in allen
Schleifen

Funktion

- ▶ Definiert einen Unter-Algorithmus
 - kann mehrfach aufgerufen werden
 - kann beim Aufruf Argumente (Parameter) erhalten
 - kann einen Wert zurückliefern
 - Die formale Schnittstelle (Rückgabetyp, Name, Parametertypen) heißt **(Funktions-)Signatur**



Optionale Parameter

Default-Wert

- ▶ Eine Funktion
 - definiert für Parameter Default-Werte
 - Diese können beim Aufruf entfallen
 - Immer nur am Ende der Parameterliste (von rechts nach links)

```
def summe(a, b, c = 0) :  
    erg = a + b + c  
    return erg
```

```
def produkt(a, b = 1, c = 1) ;  
    erg = a * b * c  
    return erg
```

...

```
s1 = summe(20, 12, 45)  
s2 = summe(4, 77)
```

```
p1 = produkt(13, 17, 25)  
p2 = produkt(12.5)
```

Default-Werte
Interpreter ergänzt die Werte der optionalen Parameter

Parameter-Namen

Default-Wert

- ▶ Eine Funktion kann auch mittels Parameter-Namen aufgerufen werden. Damit ist die Reihenfolge der Parameter egal.

```
def summe(a, b, c = 0) :  
    erg = a + b + c  
    return erg
```

...

```
s1 = summe(a=2, b=3)  
s2 = summe(a=2, c=3, b=3)
```

Ausnahme- behandlung

» Fehler erkennen, behandeln,
melden

Ausnahmen

- Fehler werden mittels Exceptions gemeldet
- Müssen im Code behandelt werden, sonst Programmabbruch

```
try :  
    Anweisung(en)  
except :  
    Anweisung(en)
```

```
try :  
    # hier wird mit Fehlern gerechnet  
    text = input("Bitte eine Zahl eingeben: ")  
    zahl = int(text)  
    print("Danke, die Zahl ist", zahl)  
except :  
    # Sprung hierher falls Fehler aufgetreten  
    print("Fehler bei der Zahleneingabe!")  
  
.... # Ausführung wird fortgesetzt
```

Ausnahmen

- Fehler lassen sich durch den Typ unterscheiden
- Fehlerdaten können als Variable "gefangen" werden

```
try :  
    Anweisung(en)  
except ZeroDivisionError as e :  
    Anweisung(en)  
except ValueError as e :  
    Anweisung(en)
```

```
try : # hier wird mit Fehlern gerechnet  
    a = int(input("Bitte eine Zahl eingeben: "))  
    ...  
    erg = a / b  
except ValueError as e : # falls Fehler bei Eingabe  
    print("Fehler bei der Zahleneingabe!")  
    print(e)  
except ZeroDivisionError as e : # falls Fehler bei Division  
    print("Fehler, kann nicht durch 0 dividieren!")  
  
.... # Ausführung wird fortgesetzt
```

Ausnahmen melden

- Fehler können auch vom eigenen Code "geworfen" werden
- Standardtypen oder Benutzerdefinierte

```
try :  
    ...  
    raise ValueError  
except :  
    Anweisung(en)
```

```
try :  
    ...  
    if a < 0 :  
        raise ValueError # hier den Fehler melden  
    ...  
except ValueError as e : # konkreter Fehlertyp  
    ...  
except : # alle anderen Typen  
    print("Fehler im Programm!")  
    print(sys.exc_info()[0])      # Ersatz für variable e  
    .... # Ausführung wird fortgesetzt
```

Dateioperationen

» Dateien und Directorys

Dateioperationen

- ▶ `open()` – Parameter: Filename, Mode
 - (r)read, (a)ppend, (w)rite, (c)reate
 - Zusätzlich: (t)ext oder (b)inär
- ▶ `close()`
- ▶ `read()`, `readline()`, `write()`, `writelines()`

```
file = open("text.txt", "r")
line = None
while line != "":
    line = file.readline()
    print(line)
file.close()
```

Files

Optionen bei open():

- ▶ 'r' This is the default mode. It Opens file for reading.
- ▶ 'w' This Mode Opens file for writing.
 - If file does not exist, it creates a new file.
 - If file exists it truncates the file.
- ▶ 'x' Creates a new file.
 - If file already exists, the operation fails.
- ▶ 'a' Open file in append mode.
 - If file does not exist, it creates a new file.
- ▶ 't' This is the default mode. It opens in text mode.
- ▶ 'b' This opens in binary mode.
- ▶ '+' This will open a file for reading and writing (updating)

Directory-Operationen

► Modul os:

- os.readdir() ... Einträge aus einem Directory lesen
- Os.walk() ... Directory – Bäume durchgehen
- os.stat(filename) ... Filestatus
- os.path ... Operationen mit dem File-Pfad
- os.remove(filename) ... File löschen

```
import os
for file in os.listdir(".") :
    print(file)
    print(os.stat(file))
    print(os.path.isfile(file))
```

Module

» Python Bibliotheken nutzen
und erstellen

Module verwenden

- ▶ Module können aus folgenden Quellen stammen:
 - Standardbibliothek
 - Eigene Bibliotheken und Module
 - Von Drittanbietern (python -m pip install SomePackage)
 - Lokale Module
- ▶ Ein Modul ist im einfachsten Fall ein eigenes Python-File (Endung .py)
- ▶ "Aktivierung" mit import

```
import math          # alle Inhalte importieren
wert = math.cos(34.7) # jedes Modul bildet einen Namespace

import datetime as dt      # Alias für den Namespace
jetzt = dt.datetime.now()  # Verwendung des Namespace

from random import randrange # nur ein Element importieren
zahl = randrange(100)
```

Module erstellen

- ▶ Jedes Python-File kann als Modul verwendet werden
- ▶ Enthalten meist
 - Klassendefinitionen
 - Funktionsdefinitionen
 - Konstanten
- ▶ Modulname = Dateiname

Datei myfunc.py:

```
def foo(text) :          # Funktionsdefinition
    print("Hello from foo", text)
```

Datei main.py:

```
import myfunc           # Modul importieren (=Namespace)
myfunc.foo("Module-Demo") # Aufruf über Namespace.Funktion
```

Module

- ▶ Die Funktion `dir(modulename)` erstellt eine Liste aller Elemente eines Moduls
- ▶ Module können/sollten Doc-Kommentare enthalten

Datei `myfunc.py`:

```
"""
Funktionssammlung myfunc

Enthält verschiedene nützliche Funktionen
"""

def foo(text) :
    """ Gibt eine Meldung und den "text" aus """
    print("Hello from foo", text)
```

Hilfe zu `myfunc`:

NAME
myfunc - Funktionssammlung myfunc

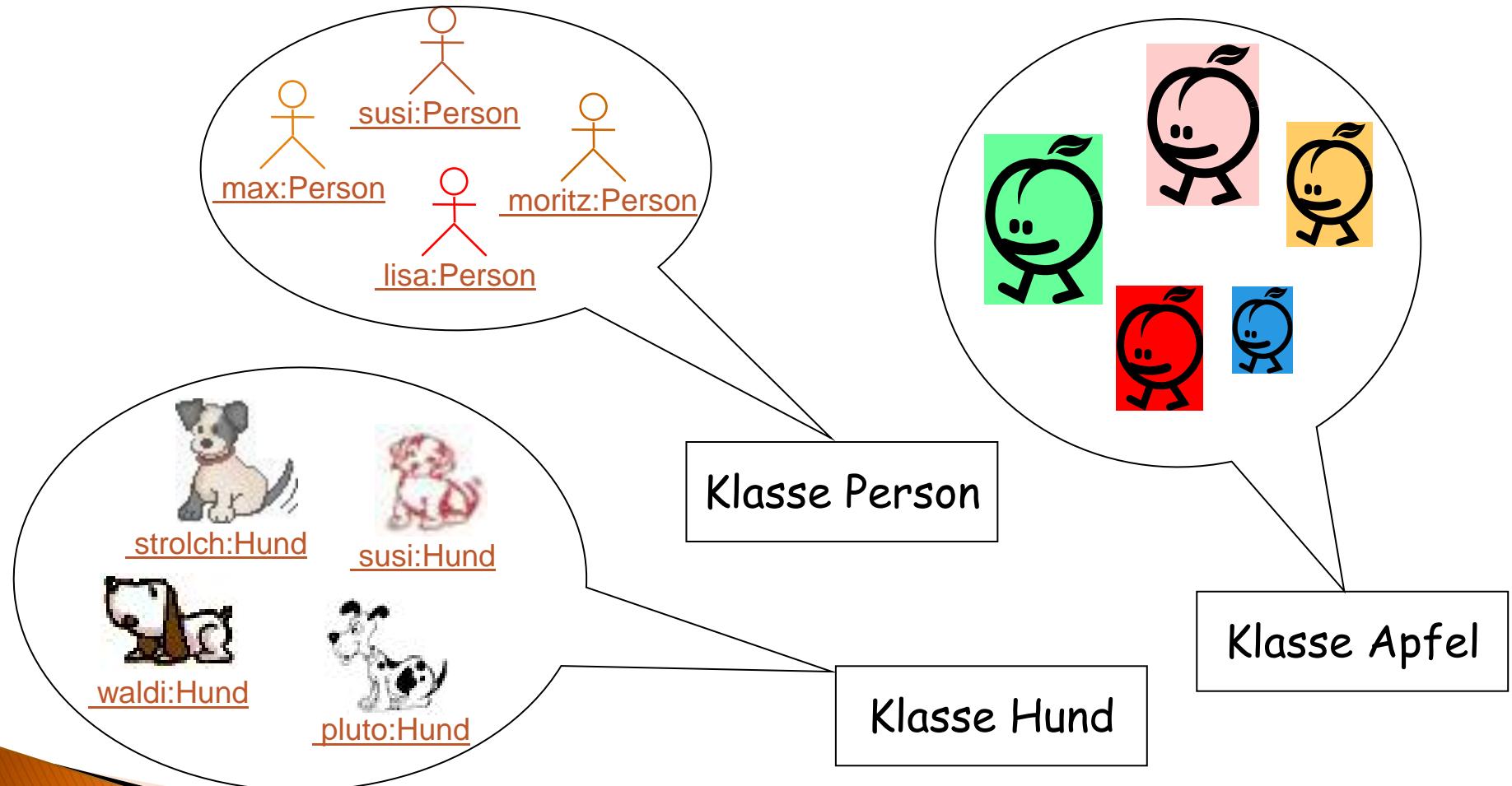
DESCRIPTION
Enthält verschiedene nützliche Funktionen

FUNCTIONS
`foo(text)`
Gibt eine Meldung und den "text" aus

Objektorientierte Konzepte

» Klassen und Objekte
definieren und verwenden

Klassen und Objekte



Klassen und Objekte

**Klasse
(Class)**



Typ

**Objekt
(Object)**



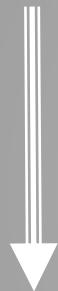
Wert

- Datentyp
- Enthält Attribute und Methoden
- Schablone für Objekte

- Exemplar ("Instanz") einer Klasse ("Wert")

Klassen und Objekte

Klasse



str (String)

Objekt



"Herr Nilsson"

Klassen und Objekte

Klasse



Person

Objekt



Name: Susi

Alter: 12

Hobby: Reiten

Klassen und Objekte

Klasse



Hund

Objekt



Name: Struppi
MarkenNr:
W4711

Klassen und Objekte

- ▶ Eine **Klasse** beschreibt eine Menge von Objekten mit gleicher Struktur (Attributen), gleichem Verhalten (Methoden)
- ▶ Die **Attribute** (Felder) einer Klasse definieren den Status ihrer Objekte
- ▶ Die **Methoden** einer Klasse definieren die Fähigkeiten ihrer Objekte
- ▶ Jede **Klasse** hat einen Namen;
- ▶ Jede **Klasse** muss ausführlich dokumentiert werden

Klassen und Objekte

Es soll eine Klasse erstellt werden,
welche eine Uhrzeit als Stunde und
Minute kapselt.

Die Werte für Stunde und Minute
müssen gesetzt werden können.
Eine beliebige Anzahl von Minuten soll
dazugezählt werden können

Klassendefinition

Klassename

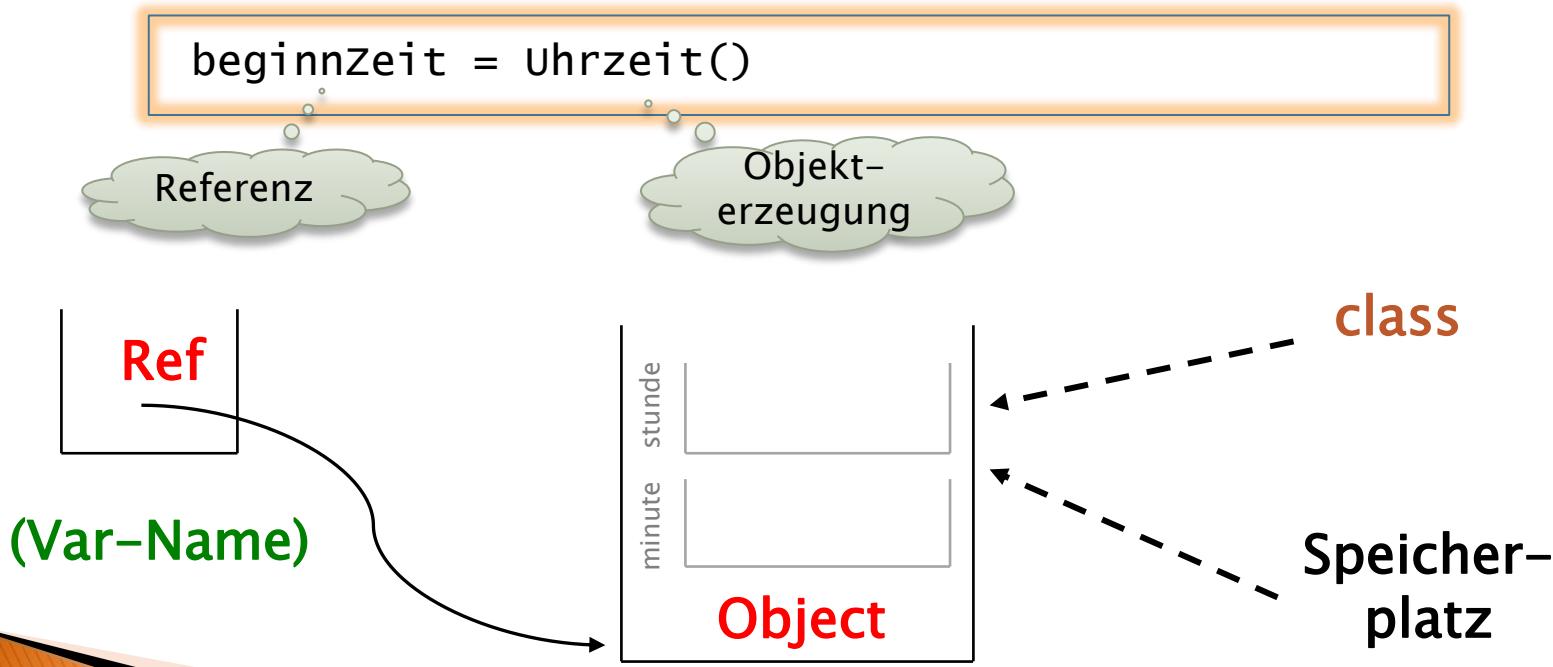
Attribute legen fest welche Daten die Objekte haben

```
class Uhrzeit :  
    # eine Uhrzeit setzen  
    def setzen(self, std, min) :  
        # Attribute entstehen bei der Zuweisung  
        self.stunde = std  
        self.minute = min  
  
    # die Uhrzeit anzeigen  
    def anzeigen(self) :  
        print(f"{self.stunde}:{self.minute}")  
  
    # Minuten dazuzählen  
    def zeitDazu(self, minuten) :  
        ...
```

Methoden bestimmen welche Aktionen für die Objekte ausgeführt werden können

Instanziierung von Objekten

- ▶ Erzeugung von Objekten (Instanziierung) erfolgt in Python ausschließlich **dynamisch**
- ▶ Weitere Verwendung immer über Referenz

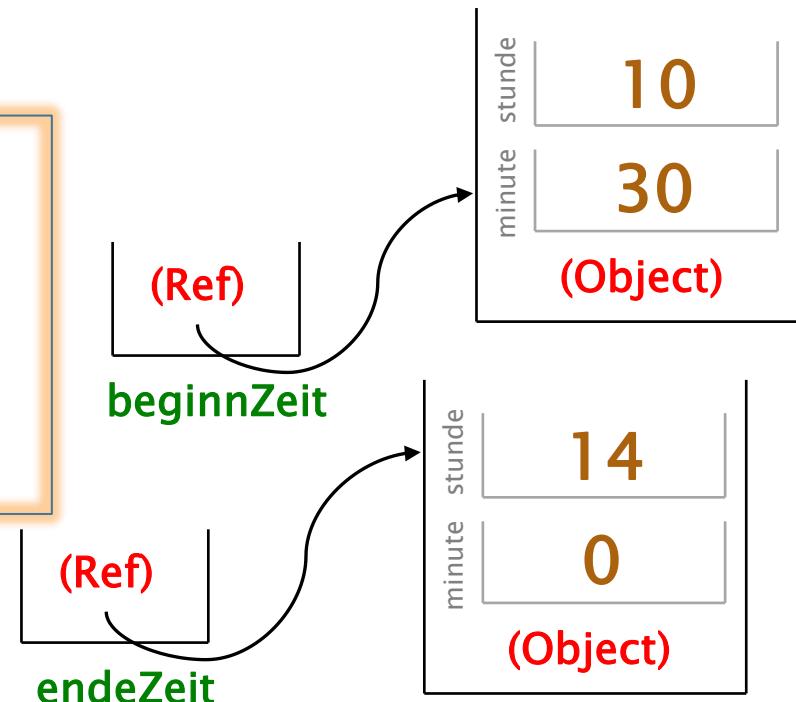


Verwenden von Objekten

► Aufrufen der Methoden

- erfolgt immer über eine Referenz
- mit dem Operator für den Memberzugriff `.`

```
beginnZeit = Uhrzeit()  
beginnZeit.setzen(10, 30)  
beginnZeit.anzeigen()  
endeZeit = Uhrzeit()  
endeZeit.setzen(14, 0)
```



Zugriffsattribute

- ▶ Datenkapselung ist ein Grundkonzept der OOP
 - In Python nur über Konvention/Name Mangling gelöst (nicht sicher)

Zugriff	Syntax	Wer soll zugreifen	Umsetzung
private	<code>__attribute</code>	nur die Klasse selbst	Name Mangling
protected	<code>_attribute</code>	die Klasse und abgeleitete Klassen	Konvention
public	<code>attribute</code>	Jeder, Default in der Klasse	

Initialisierung



- ▶ In Python können neue Objekte initialisiert werden
 - dazu dient die Funktion `__init__()`
 - wird beim Erzeugen des Objekts automatisch aufgerufen
 - muss den Parameter `self` definieren
 - kann weitere Parameter definieren
 - kann Defaultparameter (Standardwerte) verwenden
 - darf keinen Wert zurückliefern (`return`)

Initialisierung

```
class Uhrzeit :  
  
    # Initialisierungsfunktion mit Parametern  
    def __init__(self, std = 0, min = 0) :  
        self.__stunde = std # Definition als "private" Attribut  
        self.__minute = min
```

beginn = Uhrzeit()
beginn.setzen(10, 30)

ende = Uhrzeit(14, 0)

Operator-Overloading

In Python-Klassen können die normalen Operatoren mit Klassen-Methoden überladen werden. Damit sind Operationen zwischen Klassen möglich, genau so wie man die Basis-Datentypen verwenden würde.

```
class A:  
    def __init__(self, a):  
        self.a = a  
  
    # adding two objects  
    def __add__(self, o):  
        return self.a + o.a  
  
ob1 = A(1)  
ob2 = A(2)  
print(ob1 + ob2)
```

Operator-Overloading

+	<code>_add_(self, other)</code>
-	<code>_sub_(self, other)</code>
*	<code>_mul_(self, other)</code>
/	<code>_truediv_(self, other)</code>
//	<code>_floordiv_(self, other)</code>
%	<code>_mod_(self, other)</code>
**	<code>_pow_(self, other)</code>
>>	<code>_rshift_(self, other)</code>
<<	<code>_lshift_(self, other)</code>
&	<code>_and_(self, other)</code>
	<code>_or_(self, other)</code>
^	<code>_xor_(self, other)</code>

--	<code>_ISUB_(SELF, OTHER)</code>
+=	<code>_IADD_(SELF, OTHER)</code>
*=	<code>_IMUL_(SELF, OTHER)</code>
/=	<code>_IDIV_(SELF, OTHER)</code>
//=	<code>_IFLOORDIV_(SELF, OTHER)</code>
%=	<code>_IMOD_(SELF, OTHER)</code>
**=	<code>_IPOW_(SELF, OTHER)</code>
>>=	<code>_IRSHIFT_(SELF, OTHER)</code>
<<=	<code>_ILSHIFT_(SELF, OTHER)</code>
&=	<code>_IAND_(SELF, OTHER)</code>
=	<code>_IOR_(SELF, OTHER)</code>
^=	<code>_IXOR_(SELF, OTHER)</code>

<	<code>_LT_(SELF, OTHER)</code>
>	<code>_GT_(SELF, OTHER)</code>
<=	<code>_LE_(SELF, OTHER)</code>
>=	<code>_GE_(SELF, OTHER)</code>
==	<code>_EQ_(SELF, OTHER)</code>
!=	<code>_NE_(SELF, OTHER)</code>

Instanz- vs Klassen-Attribute

	Instanzmember	Statische Member
Zugehörigkeit	sind an ein Objekt gebunden	gehören direkt zur Klasse
Verwendung	es muss zuvor ein Objekt erzeugt und initialisiert worden sein	es ist kein Objekt erforderlich
Zugriff	nur über eine Referenz/Variable	ohne Referenz, über den Klassennamen
Definition	in einer Methode bzw. <code>__init__()</code>	direkt im Klassenblock
Einsatz	<ul style="list-style-type: none">○ mehrere Exemplare der Klasse mit jeweils eigenen Werten;○ fortgeschrittene OOP-Techniken nutzen (z.B. Vererbung)	<ul style="list-style-type: none">○ Utility-Methoden, die ohne vorherige Instanziierung verwendbar sein sollen;○ "Globale" Methoden

Instanz-Member

► Instanz-Attribut

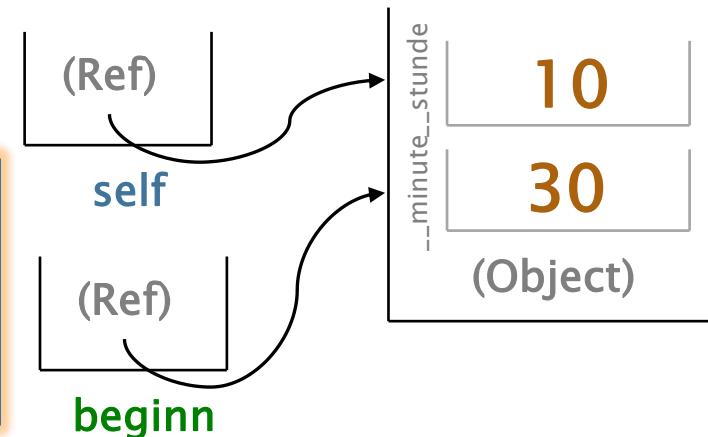
- "Gewöhnliches" Attribut: existiert 1x pro Objekt

► Instanz-Methode

- kann auch die Instanz-Attribute lesen und ändern
- dazu wird beim Aufruf die Referenz auf das Objekt implizit mitgegeben: **self**

```
beginn = Uhrzeit()  
beginn.setzen(10, 30)
```

```
def setzen(self, stunde, minute) :  
    self.__stunde = stunde  
    self.__minute = minute
```



Statische Member

- ▶ werden direkt im Block der Klasse definiert
- ▶ **Statische Attribute (Klassen-Attribute)**
 - existieren genau 1x für die Klasse
 - existieren unabhängig von Instanzen der Klasse
- ▶ **Statische Methoden (Klassen-Methoden)**
 - können diese Attribute und andere statische Methoden verwenden
 - können keine Instanzmember verwenden (**kein self**)
- ▶ **Zugriff von außen**
 - erfolgt über den Klassennamen

Statische Member

Das Instanzattribut "__id"
existiert 1x pro Objekt

```
class CountClass :  
  
    __objectCount = 0  
  
    def __init__(self, id) :  
        self.__id = id  
        CountClass.__objectCount += 1  
  
    def getId(self) :  
        return self.__id  
  
    def getObjectCount() :  
        return CountClass.__objectCount
```

In Klassenmethoden kann nur auf
Klassenattribute zugegriffen werden

Das Klassenattribut
"__objectCount" gibt es nur
einmal für die ganze Klasse!

```
c1 = CountClass(5)  
c2 = CountClass(7)  
  
id1 = c1.getId()      # == 5  
id2 = c2.getId()      # == 7  
  
count = CountClass.  
    getObjectCount() # == 2
```

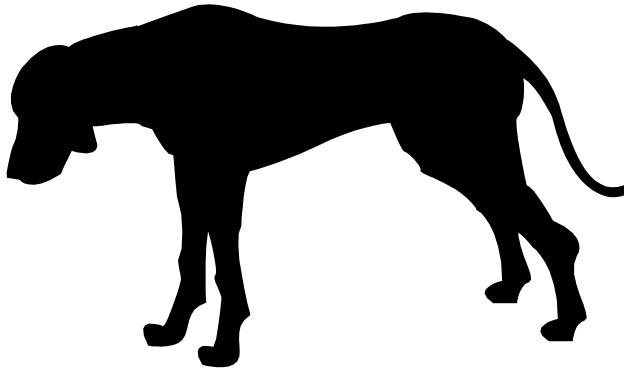
Der Zugriff von außen erfolgt über
den Klassennamen

Vererbung

» Spezialisierung von Klassen

Vererbung - Ableitung

- Die abgeleitete Klasse ist eine Spezialisierung einer (Basis-)Klasse



Hund

- Basisklasse
 - Grundattribute und -methoden von Hunden

Dackel ist abgeleitet
von Hund



Dackel

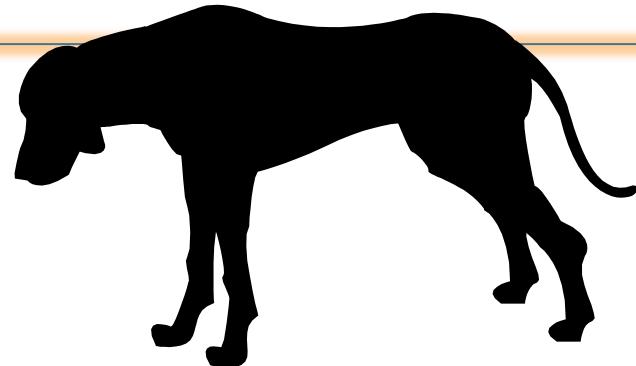
- Wie Hund, aber:
 - einige Dinge anders
 - zusätzliche Funktionalität

Vererbung - Ableitung

class Hund



class Dackel(Hund)



Dackel wird von
Hund abgeleitet

Vererbung

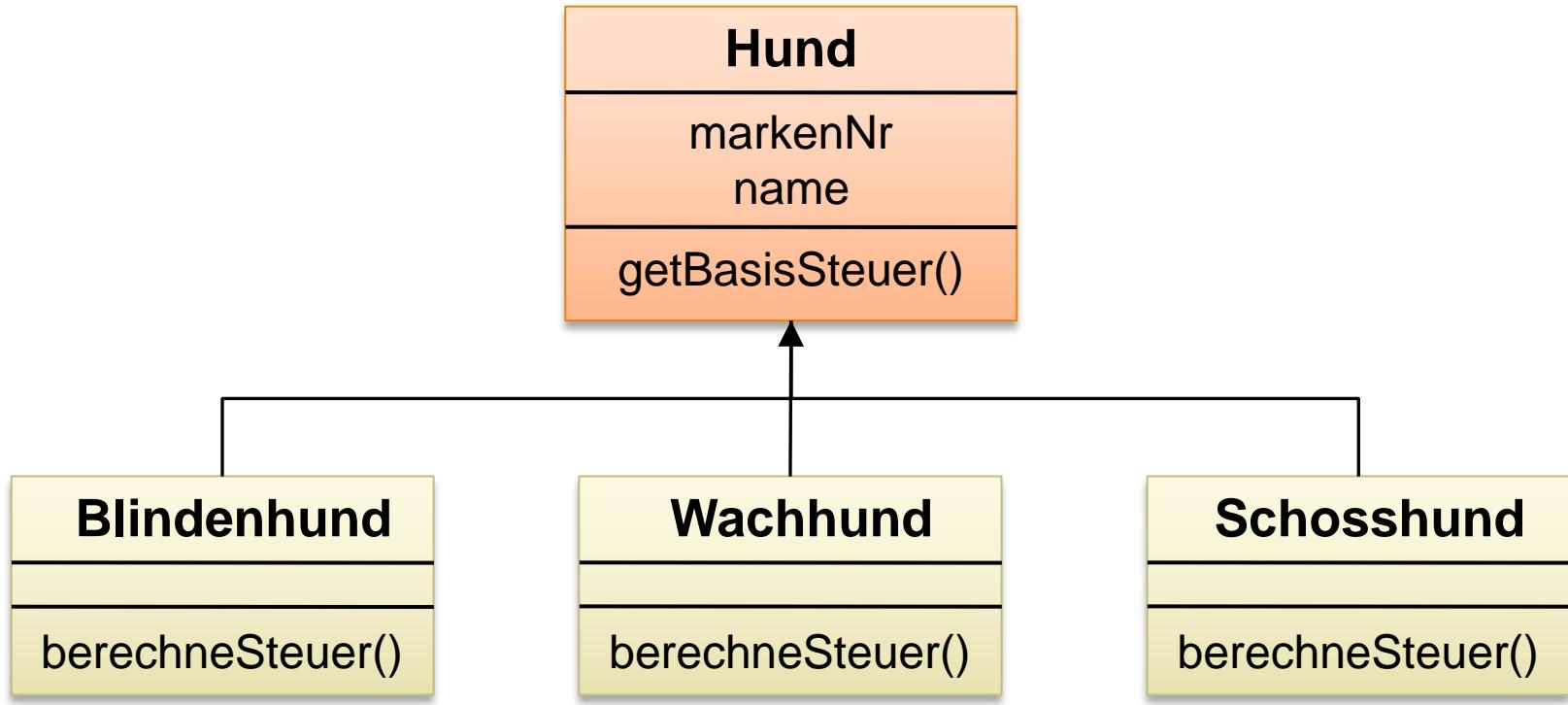
- ▶ Beispiel "Hundesteuer berechnen"

*Die Hundesteuer beträgt 100 EUR.
Wachhunde zahlen nur die Hälfte.
Und Blindenhunde sind überhaupt
von der Steuer befreit.*

Berechnet werden soll die Steuer für
Blindenhunde, Wachhunde und Schoßhunde

Vererbung

► Beispiel "Hundesteuer berechnen"



Blindenhund, Wachhund und Schoßhund sind Spezialisierungen von Hund, mit jeweils eigener Berechnungsmethode

Vererbung

```
Hund :
```

```
def __init__(self, ...) :  
    self.name = ...  
    self.markenNr = ...  
  
# Normale Steuer berechnen  
def getBasisSteuer(self) :  
    # "Normale" Steuer  
    return 100  
  
def getName(self) :  
    return self.name  
  
def getMarkenNr(self) :  
    return self.markenNr
```

Basisklasse

```
class Wachhund(Hund) :
```

```
def __init__(self, ...) :  
    self.einsatzort = ...  
  
# Hundesteuer für Wachhunde  
def berechneSteuer(self) :  
    # zahlen nur die Hälfte  
    return self.getBasisSteuer()/2
```

```
def getEinsatzort(self) :  
    return self.einsatzort
```

abgeleitete
Klasse

Vererbung - Initialisierung

- ▶ Beim Erzeugen eines Objekts einer abgeleiteten Kasse wird die Initialisierung der Basisklasse NICHT automatisch aufgerufen
 - kann/soll in der **Initialisierung der Basisklasse** aufgerufen werden
 - erst dann existieren die Attribute!

```
class Hund :  
    def __init__(self) :  
        self.name =
```

```
rex = Wachhund()
```

```
class Wachhund(Hund) :  
    def __init__(self) :  
        super().__init__()  
        # oder  
        Hund.__init__(self)
```



1. Initialisierung von Hund
2. Initialisierung von Wachhund

Vererbung: super-Aufruf

- Falls die Basisklassen-Initialisierung Parameter hat
 - werden die Parameter beim Aufruf an die `__init__()` Methode der Basisklasse übergeben

```
class Hund :  
    def __init__(self, name) :  
        self.name = name
```

```
class Wachhund(Hund) :  
    def __init__(self, n, nr) :  
        super().__init__(n)  
        self.nummer = nr
```

```
rex = Wachhund("Rex", 4711)
```

Datum und Zeit

» Modul datetime

date, time

- ▶ Klassen die ein einfaches Datum (Tag, Monat, Jahr) bzw. eine einfache Zeit speichern
 - Aktueller Wert: `today()` für date
 - Initialisierungen mit Parametern
 - Zugriff auf die einzelnen Komponenten: `hour`, `minute`, `second` bzw. `day`, `month`, `year` etc.

```
import datetime # benötigtes Modul

heute = datetime.date.today()
print("Heute ist der", heute)    # => "Heute ist der 2019-05-31"

zeit = datetime.time(11, 36, 23)
print("Jetzt ist es", zeit)      # => "Jetzt ist es 11:36:23"
print("Stunden =", zeit.hour)   # => "Stunden = 11"
```

datetime

- ▶ Klasse für einen kompletten Zeitpunkt (Datum und Zeit)
 - Aktueller Wert: `now()` für `date`
 - Initialisierungen mit Parametern
 - Zugriff auf die einzelnen Komponenten: `hour`, `minute`, `second` bzw. `day`, `month`, `year` etc.

```
import datetime # benötigtes Modul

jetzt = datetime.datetime.now()
print("Aktuell", jetzt)    # => "Aktuell 2019-05-31 11:48:19"

print("Stunden =", jetzt.hour) # => "Stunden = 11"
print("Jahr =", jetzt.year)   # => "Jahr = 2019"
```

Umwandlung datetime – String

- ▶ Unterstützt das Standard ISO Format:
`2019-04-18T06:35:13.546`
- ▶ Erzeugen eines ISO String `.isoformat()`
- ▶ Aus ISO String parsen
`datetime.fromisoformat(string)`
- ▶ Einen beliebig formatierten String erzeugen
`.strftime(format)`
- ▶ Aus einem beliebig formatierten String parsen
`datetime.strptime(string, format)`

List, Dictionary, String

»> Sequentieller Datenspeicher

List

- ▶ Sequentieller Datenspeicher von Werten beliebigen Typs
 - Anzahl der Elemente: `len(list)`
 - Zugriff auf Elemente: per **Index** (beginnt bei 0)
- ▶ Ersatz für klassisches Array

```
# Liste mit Strings definieren
obst = ["Apfel", "Birne", "Orange", "Banane", "Kiwi"]
# Iteration mit for-Schleife und Index
for i in range(len(obst)) :
    # Zugriff mit Index von 0 bis 4
    print(obst[i])
```

Listen-Iteration (for)

Ausdruck 1:
Laufvariable

Ausdruck 2: Container
mit mehreren Elementen

```
# Liste mit Strings definieren
obst = ["Apfel", "Birne", "Orange", "Banane", "Kiwi"]

...
# über alle Elemente mit for-Schleife iterieren
for frucht in obst :
    # frucht enthält einen Wert
    print(frukt)
```

List

```
liste = []
```

Leere Liste

```
zahlen = [5, 13, 55, 42, 7]
```

Definition

[-5]	[-4]	[-3]	[-2]	[-1]
5	13	55	42	7
[0]	[1]	[2]	[3]	[4]

```
zahlen[0] = 10
```

Zugriff per Index

[-5]	[-4]	[-3]	[-2]	[-1]
10	13	55	42	7
[0]	[1]	[2]	[3]	[4]

```
size = len(zahlen)
```

Anzahl der Elemente

[-5]	[-4]	[-3]	[-2]	[-1]
5	13	55	42	7
[0]	[1]	[2]	[3]	[4]



Manipulation von Listen

- ▶ Kombinieren von Listen mit `+` und `+=`
- ▶ Test auf Vorkommen mit `in` und `not in`
- ▶ Am Ende anhängen `append(obj)`
- ▶ An Position einfügen `insert(index, obj)`
- ▶ Element löschen `remove(obj)`
- ▶ An Position entfernen `pop()`, `pop(index)`
- ▶ Liste leeren `clear()`
- ▶ Elemente sortieren `sort()`

Dictionary

- ▶ Liste von Key–Value Paaren
- ▶ Datenstruktur mit schnellen Zugriff über den Key

```
# Dictionary
vokabeln = { "Apfel": "apple",
             "Brot": "bread" }
# Zugriff mittels key
print(vokabeln["Apfel"]) # => apple
# Zugriff auf alle Elemente
for key in vokabeln.keys():
    print(key)
```

String (str)

- ▶ Sequentielle Liste von Unicode-Zeichen
 - Anzahl der Elemente: `len(string)`
 - Zugriff auf Elemente: per **Index** (beginnt bei 0)
- ▶ Ist unveränderlich, d.h. jede Manipulation erzeugt einen neuen String

```
# einen String definieren
text = "Hallo Welt"
# String erweitern
text += " mit Python"
# in Kleinbuchstaben umwandeln und anzeigen
print(text.lower())
```

Manipulation von Strings

- ▶ Kombinieren von Strings mit `+` und `+=`
- ▶ Vervielfältigen mit `*`
- ▶ Test auf Vorkommen mit `in` und `not in`
- ▶ Slicing = Ausschneiden `[begin:end:step]`

`text = "Hallo Welt"`

`text[6:8] => "We"`

`text[:4] => "Hall"`

`text[6:] => "Welt"`

`text[::-2] => "HloWl"`

- ▶ diverse Funktionen `lower()`, `upper()`, `capitalize()`
- ▶ Position suchen `find(substr[,start[,end]])`
- ▶ Formatieren `format()` (siehe Ausgabe)
- ▶ Teile ersetzen `replace()`
- ▶ Zerteilen, Zusammenfügen `split()`, `join()`

Datenstrukturen: Tupel, Sets

» mit Python

Datenstrukturen: Vergleich

Python	Java	C#
Liste, kein Array	Array, ArrayList	Array, List
Dictionary	Dictionary	Dictionary
Tupel	javatuples – class	Tuple types (C# 7)
Set	Set	Set

Python Datentyp	Syntax
Liste	[... , ...]
Dictionary	{ „key“: „value“ }
Tupel	(... , ...)
Set	{ ... , ... }

Datenstrukturen: Tupel

- Ein Tupel besteht aus mehreren Werten, die durch Kommas voneinander getrennt sind
- Eine Mischung der Datentypen innerhalb eines Tupels ist möglich
- Klammern können, müssen aber nicht verwendet werden. Bei der Ausgabe werden immer Klammern verwendet
- Tupel können ineinander geschachtelt werden
- Ein Tupel ist nach der Erzeugung nicht mehr veränderbar

Anwendungen:

- Zusammenhängende Daten wie Koordinaten
- Return-Werte von Funktionen, wo mehr als ein Parameter rückgegeben werden soll

Datenstrukturen: Tupel

```
# Definition einer Funktion mit Tupel als Rückgabewert
def vektorAdd(a,b):
    return ( a[0] + b[0], a[1] + b[1] )

# Tupel mit Klammern
v1 = (1,2)
# Tupel ohne Klammern
v2 = 2,3

v3 = vektorAdd(v1, v2)

# Ausgabe des Tupels mit Klammern
print (f"Ergebnis: {v3}")
```

Datenstrukturen: Sets

- Eine Menge ist eine ungeordnete Sammlung ohne doppelte Elemente.
- Sie werden vor allem dazu benutzt, um zu testen, ob ein Element in der Menge vertreten ist und doppelte Einträge zu beseitigen.
- Mengenobjekte unterstützen ebenfalls mathematische Operationen wie Vereinigungsmenge, Schnittmenge, Differenz und symmetrische Differenz.
- Mengen werden mittels geschweifter Klammern definiert

Anwendungen:

- Eindeutigkeit von Elementen in einem Set
- Mengenoperationen

Datenstrukturen: Sets

```
basket = {'Apfel', 'Orange', 'Apfel', 'Birne', 'Orange', 'Banane'}
# zeigt, dass die Duplikate entfernt wurden
print(f"Ausgabe des Sets, Duplikate sind entfernt: {basket}")

# schnelles Testen auf Mitgliedschaft
print('Orange' in basket)
print('Hirse' in basket)

a = set('abracadabra')
b = set('alacazam')
# einzelne Buchstaben in a
print(f'Menge a: {a}, Menge b: {b}')

# Mengenoperation in a aber nicht in b
print(f'Mengenoperation in a aber nicht in b: {a - b}')
# Mengenoperation in a oder b
print(f'Mengenoperation in a oder b: {a | b}')
# Mengenoperation sowohl in a, als auch in b
print(f'Mengenoperation sowohl in a, als auch in b: {a & b}')
# Mengenoperation entweder in a oder b
print(f'Mengenoperation entweder in a oder b: {a ^ b}')
```

Serialisieren/ Deserialisieren: Json, XML

» mit Python

Serialisieren/Deserialisieren: Json

- JSON (JavaScript Object Notation) ist ein Datenaustauschformat. Spezifiziert in RFC 7159 und durch ECMA-404. Der Syntax ist an JavaScript Object Literal Syntax angelehnt.
- Verwendung:
 - Datentransfer im Internet
 - Definition von Konfigurations-Parametern
- Mit Hilfe des Python-Packages „json“ kann Serialisieren (Umwandlung Datenstruktur => String) und Deserialisierung (Umwandlung String => Datenstruktur) gemacht werden.
- Json-Datenstrukturen in Python sind eine Mischung aus Listen (JSON Arrays) und Dictionaries (JSON Objekte)

Serialisieren/Deserialisieren: Json

Beispiel Json:

```
{  
  "name": "Georg",  
  "alter": 47,  
  "verheiratet": false,  
  "beruf": null,  
  "kinder": [  
    {  
      "name": "Lukas",  
      "alter": 19,  
      "schulabschluss": "Gymnasium"  
    },  
    {  
      "name": "Lisa",  
      "alter": 14,  
      "schulabschluss": null  
    }  
  ]  
}
```

Serialisieren/Deserialisieren: Json

Die folgende Tabelle zeigt die Umwandlungen zwischen JSON und Python Datentypen:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Serialisieren/Deserialisieren: Json

```
import json

aText = '["foo", {"bar": 1.0, "bool": true}]'
# Deserialisieren string => Python Objekt
aJson = json.loads(aText)
# Serialisieren Python Objekt => string
print(json.dumps(aJson))
# Zugriff auf einzelne Elemente
print(aJson[1]["bool"])

# Unicode
print(json.dumps('\u1234'))

# Pretty Print
print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True, indent=4))
```

Serialisieren/Deserialisieren: XML

- XML (Extensible Markup Language) ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten im Format einer Textdatei, die sowohl von Menschen als auch von Maschinen lesbar ist. Daten werden durch die Verwendung von Tags gekennzeichnet.
- Verwendung:
 - Datentransfer im Internet (z.B. News-Feeds)
 - Definition von Konfigurations-Parametern
- Mit Hilfe des Python-Packages „xml“ kann Serialisieren (Umwandlung Datenstruktur => String) und Deserialisierung (Umwandlung String => Datenstruktur)
- XML-Datenstrukturen in Python werden durch Dictionaries abgebildet

Serialisieren/Deserialisieren: XML

Beispiel XML:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

Serialisieren/Deserialisieren: XML

```
import xml.etree.ElementTree as ET

xmlText = """<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>"""

# Deserialisation
root = ET.fromstring(xmlText)
print(root.tag)
for child_of_root in root:
    print (child_of_root.tag, child_of_root.attrib)
    for child_of_node in child_of_root:
        print (child_of_node.tag, child_of_node.text , child_of_node.attrib)

# Zugriff auf einzelne Teile
print(root[1][2].text)

# Serialisation
print(ET.dump(root))
```