Workshop Python

Teil II

1. **Wiederholung Teil 1**
2. **Kapitel 4: Funktionen, Module, Packages**
3. **Kapitel 5: Objektorientierte Programmierung**

Wiederholung

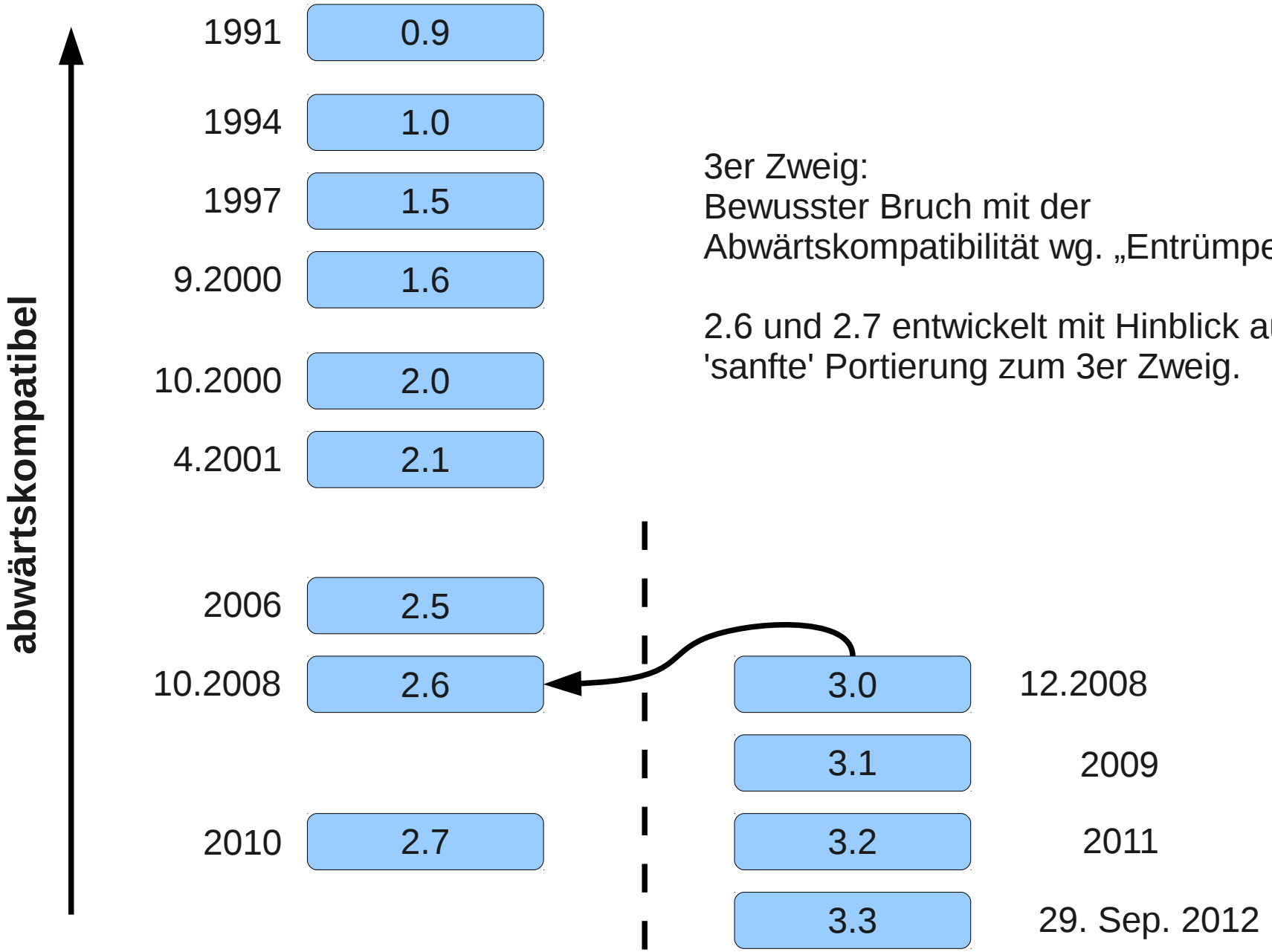
Workshop 1

- Entwicklung ab Dezember 1989 (Guido van Rossum)
- Ziele: Einfachheit und Übersichtlichkeit
- Compiler nach Bytecode / Python Virtual Machine / Garbage Collection

- Paradigmen:
  - Prozedural
  - Objektorientiert
  - Funktional

- Typsystem:
  - Stark Typisiert
  - Dynamisch Typisiert
  - Implizit Typisiert ('Duck-Typing')

- Einsatzgebiete:
  - Scripting (Internet; Extensions *GIMP, Blender* etc)
  - Rapid Prototyping
  - Systemprogrammierung
  - Component Integration



3er Zweig:
Bewusster Bruch mit der Abwärtskompatibilität wg. „Entrümpelung“

2.6 und 2.7 entwickelt mit Hinblick auf 'sanfte' Portierung zum 3er Zweig.

Statements:

Statements enden mit dem Ende der Zeile.

Mehrere Statements auf einer Zeile getrennt per Strichpunkt. `;`

Mehrzeiliges Statement falls die Zeile beendet wird mit:

- Einem Backslash `\`
- Einer nicht geschlossenen Klammer `( [ {`
- Einem nicht geschlossenen Triple-Quote String `""" '''`

Compound Statements:

Bestehen aus einem *Block Header* und einem *Block*

Der Block Header endet mit einem Doppelpunkt, `:`
und muss auf einer eigenen Zeile beginnen.

Der Block wird durch Einrückung mit der gleichen Anzahl Whitespace gekennzeichnet (*Indentation*)

Kommentare:

Von einem Hash bis zum Ende der Zeile. `#`

Alternativ auch per Triple-Quote über mehrere Zeilen (inoffiziell)

Assignments:

- Normal Assignments `a = 5`
- Augmented Assignments `a += 10`
- Sequence Assignments `a, b = 5, 15`
- Extended Assignments (Python 3) `a, *b, c = [1, 2, 3, 4, 5]`

Expressions:

Expressions liefern eine Referenz auf ein Objekt.

Assignments sind keine Expressions (Unterschied zu C)

- Objekt `5`
- Operator expression `6 + 7`
- Funktionsaufruf `berechne(5, 6)`
- Objektmethode `calculator.add(7, 8)`

Arithmetische Operatoren: `+ - * / ** % //`

Vergleichsoperatoren: `< > <= >= == !=`

Logische Operatoren: `and or not`

Bitoperatoren: `& | ^ << >> ~`

Ternärer Operator: `x if <test> else y`

Membership Operatoren: `in not in`

Identity Operatoren: `is is not`

Bedingte Ausführung:

```
if <condition>:
    <block>
elif <condition>:
    <block>
else:
    <block>
```

Iteration über alle Elemente

```
for <a> in <iterable>:
    <block>
```

Schleife solange Bedingung erfüllt

```
while <condition>:
    <block>
```

for / while Block sofort verlassen

```
break
```

Zum Beginn for / while Block springen

```
    continue
```

Nach for / while Block,
falls nicht mit break beendet

```
else:
    <block>
```

None ist ein eigener Datentyp

```
n = None
```

Boolean dient für Wahr / Falsch Werte

```
a = True; b = False
```

Integer sind Ganzzahlen

```
i = 123
```

Floats sind Fließkommazahlen

```
f = 2.6e12
```

Complex sind komplexe Zahlen aus Real- und Imaginärteil

```
c = 1.5 + 2.6j
```

Strings sind *unveränderbare* und *indizierte* Arrays von Zeichen.

```
s = "Hello, World!"
```

Listen sind *veränderbare* und *indizierte* Arrays von *Referenzen* auf *beliebige* Objekte.

```
a = [1, 2, 3]
```

Tupel sind *unveränderbare* und *indizierte* Arrays von *Referenzen* auf *beliebige* Objekte.

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

Dictionaries sind *veränderbare* Arrays von *Referenzen* auf *beliebige* Objekte, indiziert über einen *hashbaren key*:

```
c = { "Name" : "Albert", "Nachname" : "Einstein" }
```

Sets sind *ungeordnete*, *iterierbare* Mengen *eindeutiger* Refenzen auf *beliebige* Objekte

```
d = { "Hund", "Katze", "Maus" }          # Ab Python 2.7
d = set(["Hund", "Katze", "Maus"])       # Python 2.6
```

Zugriff auf Teile einer indizierbaren Sequenz (Strings, Listen, Tupel) mittels Index:

```
s = "ABCDEF"
```

Indexing: `s[i] # -len(s) <= i < len(s)`

```
s[0]     # 'A'
s[2]     # 'C'
s[-1]    # 'F'
s[-6]    # 'A'
```

```
 A  B  C  D  E  F
 0  1  2  3  4  5
-6 -5 -4 -3 -2 -1
```

Slicing: `s[i:j]`

```
s[1:3]   # 'BC'
```

Slicing mit Stride: `s[i:j:k]`

```
s[1:5:2] # 'BD'
s[::2]   # 'ACE'
s[::-1]  # 'FEDCBA'
```

Tupel / Listen Methoden

```
len(t)          # Länge des Tupels / Liste
a in t          # Test ob a in t enthalten
t.count(a)      # Zählt Anzahl von a in t
t.index(a)      # Gibt die erste Position von a in t zurück
min(t), max(t)  # Gibt kleinstes / größtes Element von t
```

Listen Methoden

```
l[index] = a    # Ersetzt Element an Stelle <index> durch a
del l[index]    # Entfernt Element an Stelle <index>
l.append(a)     # Fügt Element a an Liste an
l.extend(a)     # Fügt Elemente des Iterables a einzeln an l
l.insert(a,p)   # Fügt a an Stelle p ein
l.reverse()     # Dreht die Liste um (in-place)
l.sort()        # Sortiert die Liste (in-place)
l.remove(a)     # Löscht das erste Vorkommen von a
l = []          # Erzeugt eine leere Liste
```

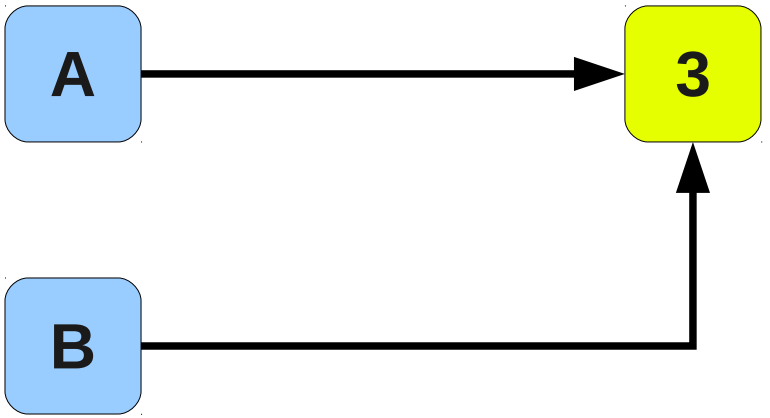
Dictionary Methoden

```
d.keys()          # liefert Liste der Keys
d.values()        # liefert Liste der Values
d.items()         # liefert Liste der (Keys, Values)

key in d          # Test ob key in d
d[key] = value    # Erzeugt oder Ändert den Eintrag key
del d[key]        # Löscht das Paar (key,value) aus d
d = {}            # Erzeugt ein neues, leeres Dictionary
```

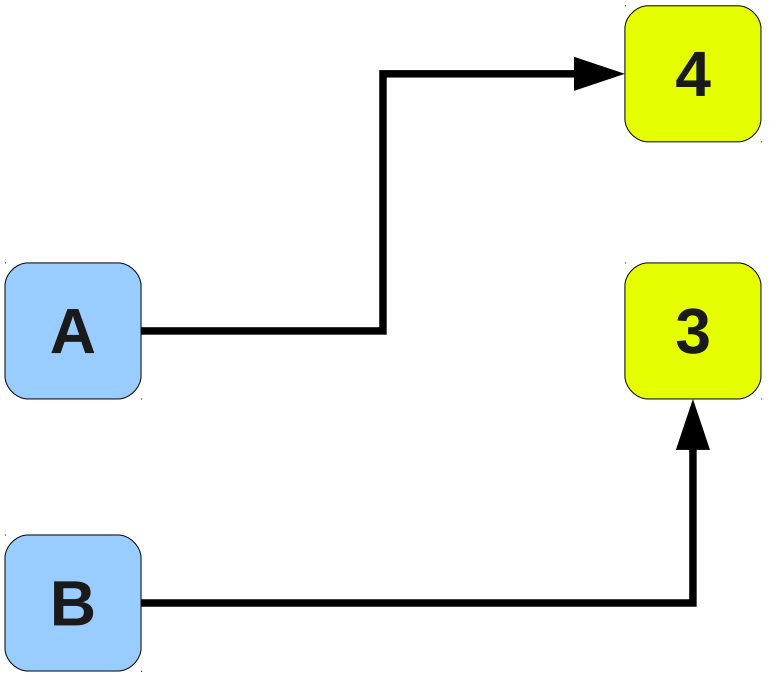
Type conversions

```
int(<string> [,base])  # String -> Int
float(<string | int>)  # String/Int -> Float
str(<object>)          # Object -> String (print-friendly)
repr(<object>)         # Object -> String (eval-friendly)

ord(<char>)            # Single Char -> Char Code
chr(<int>)             # Char Code -> Single Char

hex(<int>)             # Int -> Hexadecimal als String
bin(<int>)             # Int -> Binary als String
oct(<int>)             # Int -> Octal als String

list(<iterable>)       # Iterable -> List
tuple(<iterable>)      # Iterable -> Tupel
dict(<iterable>)       # Iterable of length 2 elements -> Dict
set(<iterable>)        # Iterable -> Set
```

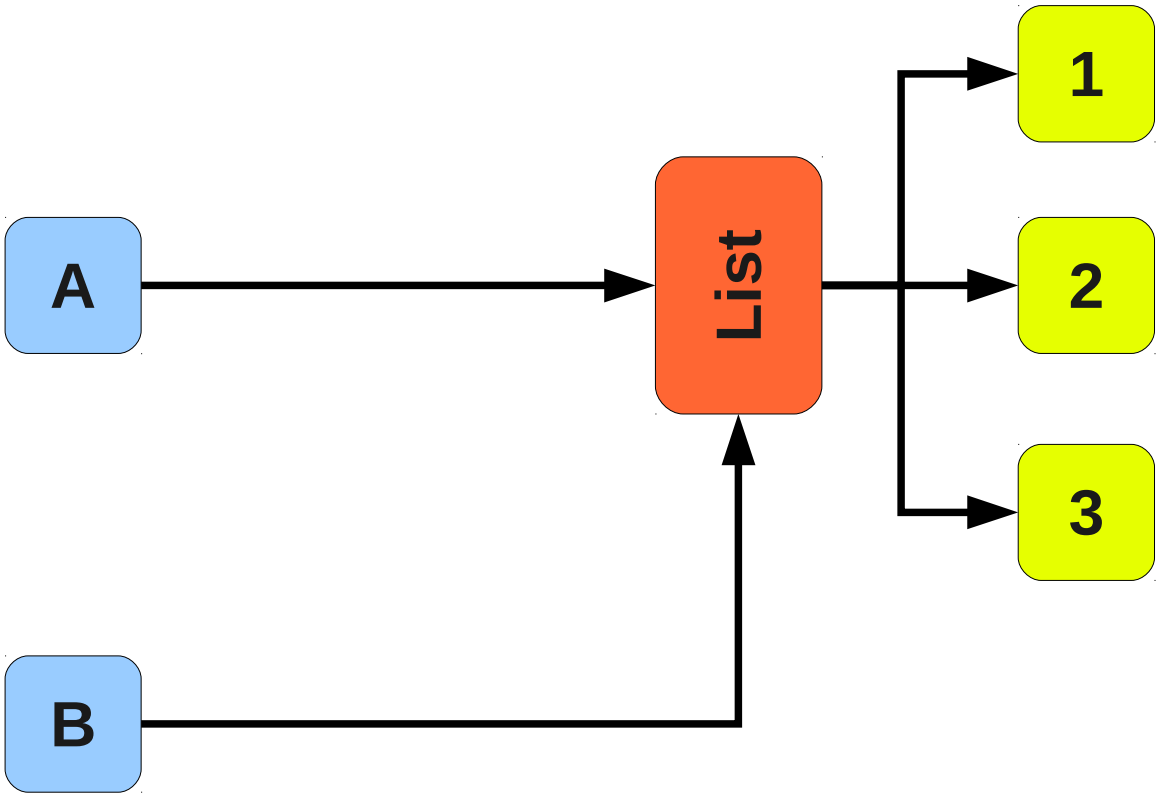
```
>>> a = 3
>>> b = a
```



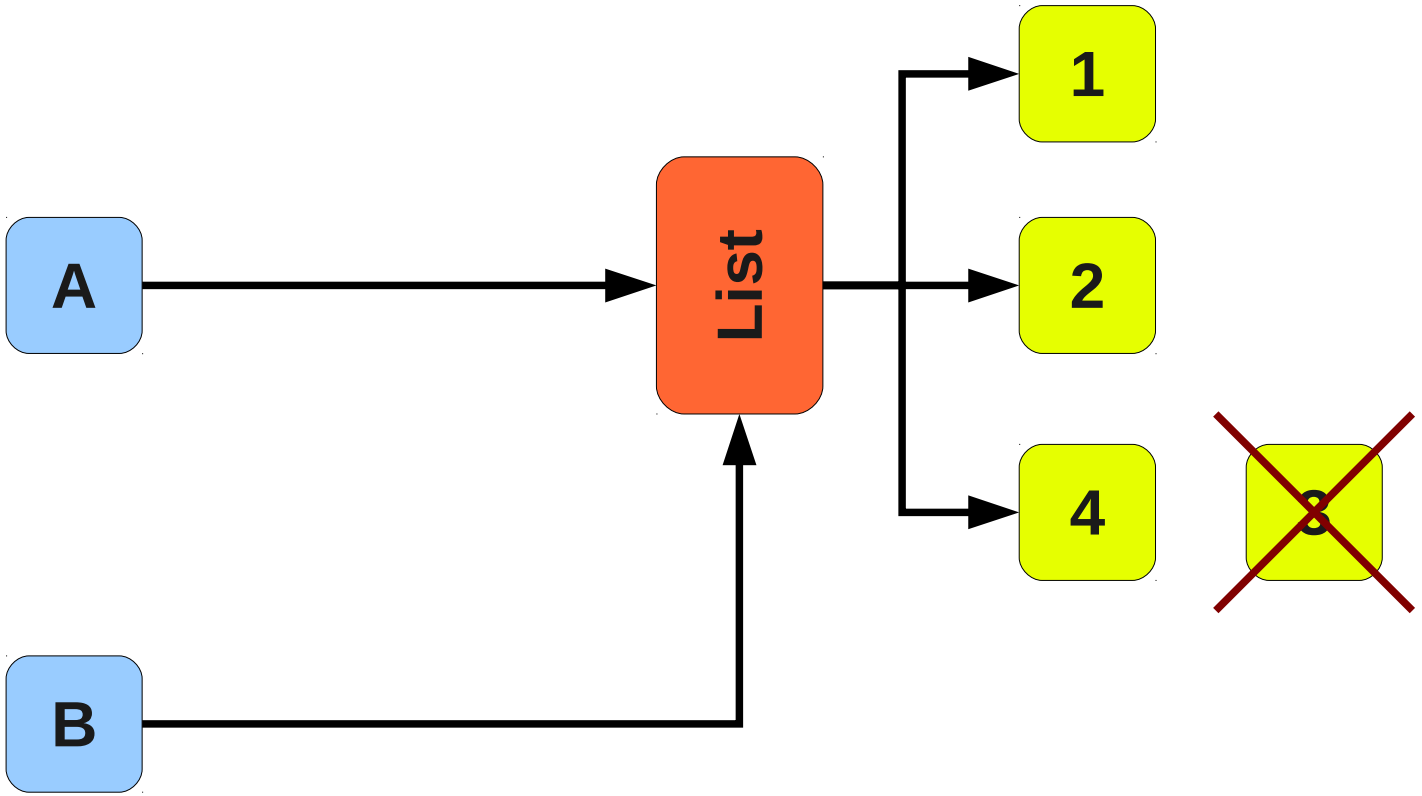
```
>>> a = 3
>>> b = a
>>> a = 4
```



```
>>> a = [1, 2, 3]
>>> b = a
```



```
>>> a = [1, 2, 3]
>>> b = a
>>> a[2] = 4
```



Funktionsargumente werden als *Objektreferenz* übergeben und in den lokalen Namespace eingefügt.

Immutable Objekte bleiben dadurch nach außen unverändert.
Allerdings können *mutable* Objekte nach außen verändert werden:

```
def func1(a):
    a = 4

a = 3
func1(a)
print a             # ergibt 3


def func2(b):
    b[0] = 4

b = [1,2,3]
func2(b)
print b             # ergibt [4,2,3]
```

Wie beim letzten Mal erzeugen wir ein File exercises.py, und machen es executable:

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

def uebung1(args)
    ...

class klasse_test(object):
    ...

if __name__ == "__main__":
    a = uebung1(1,5,6)

    b = klasse_test()
```

Aufruf entweder über Kommandozeile ...

```
> chmod +x exercises.py
> ./exercises.py
```

Wie beim letzten Mal erzeugen wir ein File exercises.py, und machen es executable:

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

def uebung1(args)
    ...

class klasse_test(object):
    ...

if __name__ == "__main__":
    pass
```

... oder über die Python Shell:

```
>>> import exercises
>>> exercises.uebung1(1,5,6)
>>> b = exercises.klasse_test()
>>> reload(exercises)
```

Teil 4

Funktionen, Module, Packages

def definiert eine Funktion:

```
def <function-name>([arg,... arg=value,... *args, **kwargs]):
    <function-body>
    [return <return-value>]    # Optional; Funktion ohne return
                               # gibt None zurück
```

```
def circle_area(radius):
    area = 3.1415 * radius * radius
    return area

print circle_area(5.0)     # Berechnet Kreisfläche r = 5.0
```

Argumente in Funktions*definitionen*:

`arg`	Zuweisung durch Name oder Position
`arg=value`	Default Wert falls *arg* nicht vorhanden
`*args`	Übergibt zusätzliche Positional arguments als Tupel
`**kwargs`	Übergibt zusätzliche Keyword arguments als dictionary

Argumente in Funktions*aufrufen*:

`arg`	Positional argument (Match by position)
`arg=value`	Keyword argument (Match by name)
`*args`	Sequenz die nach Position zugeordnet wird
`**kwargs`	Dictionary das nach Keyword zugeordnet wird

```
def parrot(voltage, state="a stiff", action="voom",
           type="Norwegian Blue"):
```

Korrekte Aufrufe:

```
parrot(1000)
parrot(voltage=1000)
parrot(voltage=2000, action="VROOOM")
parrot(action="VROOOOOOMMMM", voltage=3000)
parrot("a million", "bereft of life", "jump")
parrot("a thousand", state="pushing up the daisies")
```

Fehlerhafte Aufrufe:

```
parrot()                     # Erforderliches Argument fehlt
parrot(voltage=5.0, "dead")  # Non-keyword nach keyword
parrot(110, voltage=220)     # 2 Werte für gleiches Argument
parrot(actor="John Cleese")  # Unbekanntes keyword
```

```
def sum(*args):
    sum = 0
    for a in args:
        sum += a
    return sum

print sum(1,2,3)              # ergibt 6
print sum(1,2,3,7,9,10,13)    # ergibt 45


def kwfunc(**kwargs):
    print kwargs["name"]
    print kwargs["surname"]

print kwfunc(name="Albert", surname="Einstein")
```

```
def posfunc(a, b, c):
    print a, b, c

seq = [1, 4, 5]
posfunc(*seq)


def kfunc(name, surname):
    print name
    print surname

person = {"name" : "Stephen", "surname" : "Hawking"}
kfunc(**person)
```

Generelle Form für Funktionsargumente:

```
def <function-name>(*args, **kwargs):
    print args
    print kwargs
```

```
def general_func(*args, **kwargs):
    print args
    print kwargs

l = [4,5,6]
d = {"name" : "Albert", "surname" : "Einstein"}

general_func(1,3,arg="abc",*l,**d)
# (1, 3, 4, 5, 6)
# {'arg' : 'abc', 'name' : 'Albert', 'surname' : 'Einstein'}
```

Auch Funktionen können als **Referenz auf ein Objekt** übergeben werden, sowohl als Funktionsargument als auch als Ergebnis einer Funktion.

("Higher-Order Functions" -> wichtig für funktionale Programmierung)

```
def square(x):
    return x * x

# Functions can be assigned like any other object

g = square                      # g is now a reference to square
print g(4)                      # 16


# Functions can be passed as arguments to other functions

def funcsum(f, x, y):
    return f(x) + f(y)

print funcsum(square, 3, 4)   # 25
```

Dekoratoren agieren als **Wrapper** zur Modifizierung einer Funktion

```
def decorator(fn):
   <statements>
   return fn_new

def func():
   <statements>
func = decorator(func)        # Argument ist Funktionsobjekt!
```

In Dekoratorschreibweise:

```
def decorator(fn):
   <statements>
   return fn_new

@decorator
def func():
   <statements>
```

```
def shout(fn):
    def wrap():
        return fn().upper() + "!!!"
    return wrap

def whisper(fn):
    def wrap():
        return "psssssst ... " + fn().lower() + " ..."
    return wrap

@shout
def hello_world():
    return "Hello, World!"

print hello_world()    # "HELLO, WORLD!!!!"
```

Der *Funktionsbody* des Dekorators wird einmal während des Dekorierens aufgerufen:

```
def decorator(fn):
    print "Funktion " + fn.__name__ + " wurde dekoriert"
    return fn

@decorator
def do_something(st):
    print st

do_something("Mache irgendetwas")
do_something("Mache irgendetwas anderes")
do_something("Mache irgendetwas ganz anderes")
```

Die Modifizierung der dekorierten Funktion erfolgt i.d.R. über eine *Wrapperfunktion,* die anstelle der ursprünglichen Funktion ausgeführt wird:

```
def decorator(fn):
    print "Funktion " + fn.__name__ + " wurde dekoriert"
    def wrap(st):
        print "Dekorierte Funktion " + fn.__name__ + \
        " wurde aufgerufen"
    return wrap

@decorator
def do_something(st):
    print st

do_something("Mache irgendetwas")
do_something("Mache irgendetwas anderes")
do_something("Mache irgendetwas ganz anderes")
```

Der Wrapper muss sich um *Ausführung, Argumente und Rückgabewert* der ursprünglichen Funktion kümmern:

```
def decorator(fn):
    print "Funktion " + fn.__name__ + " wurde dekoriert"
    def wrap(st):
        print "Dekorierte Funktion " + fn.__name__ + \
        " wurde aufgerufen mit Argument: " + st
        return fn(st)
    return wrap

@decorator
def do_something(st):
    print st

do_something("Mache irgendetwas")
do_something("Mache irgendetwas anderes")
do_something("Mache irgendetwas ganz anderes")
```

Anwendungen für Dekoratoren:

- Debugging
- Tracer
- Caches
- Thread programming
- Depreciation Warnungen in Modulen
- OOP: Special methods (@property, @staticmethod, @classmethod)
- OOP: Singleton classes

Beispiele: http://wiki.python.org/moin/PythonDecoratorLibrary

Modularisierung ist die Aufteilung des Quelltextes in einzelne Teile.

Generell zwei Arten von Modulen:

- Programmbibliotheken (z.B. Python Standardbibliothek)
- Lokale Module zur Kapselung von Programmteilen in verschiedenen Dateien

Python Module bestehen aus einer Sammlung von Definitionen und Statements

- Die Definitionen sind nach *Import* verfügbar
- Statements werden das *erste* Mal ausgeführt wenn das Modul importiert wird; dient zur Initialisierung des Modules.

Moduldateien werden *modulname.py* benannt. (Nach Konvention in Lowercase)

Allgemeiner Aufbau eines Moduls:

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

"""
Module docstring
"""

<imports>                       # Imports from other modules

<definitions>                   # Constants, functions, classes

<statements>                    # Module initialization

if __name__ == "__main__":      # If module is executed
   <statements>                 # as a script
```

Sondernamen in Modulen:

`__name__`	Name des Moduls (= `__main__` falls Modul direkt ausgeführt wird)
`__file__`	Filename des Moduls
`__doc__`	Documentation String des Moduls
`_<name>`	Modul-Interna. Konvention für (Pseudo)-Private. Sollte nicht von außerhalb aufgerufen werden!

```
import <module> [, <module2> [, ...] ]
```

Importiert das Module *module*, dessen Elemente dann mittels *module.attribute* aufgerufen werden können:

```
import random, math

a = random.randrange(1,5) # Liefert Zufallszahl von 1 bis 4
b = math.cos(4.5)
```

Module werden beim Importieren gesucht in:

- Dem aktuellen Verzeichnis
- Nach der Umgebungsvariablen PYTHONPATH (gleiche Syntax wie PATH)
- Dem internen Default-Path (abrufbar und modifizierbar über *sys.path*)

Ein **Package** ist eine Sammlung von Modulen, organisiert in einer Directorystruktur:

```
sound/                          Top-Level Package
    __init__.py                 Package initializer
    formats/                    Subpackage "formats"
          __init__.py           Subpackage initializer
          mp3.py                Subpackage module
          wav.py
    effects/                    Subpackage "effects"
          __init__.py
          echo.py
          surround.py
    filters/
      ...
```

Import aus einem Package:

```
import sound.formats.mp3
```

Aufruf von einem importierten Package:

```
sound.formats.mp3.encode(...)
```

Package Initializer `__init__.py`

- Muss in jedem Verzeichnis des Packages vorhanden sein
- Kann leer bleiben
- Wird ausgeführt, falls das Paketmodul zum ersten Mal importiert wird.

Teil 5

Objektorientierte Programmierung

Grundlegende Begriffe der Objektorientierten Programmierung:

- **Klassen**: "Baupläne" für Objekte
- **Instanz***:* Konkrete Manifestation einer Klasse ('Objekt')
- **Attribut**: Einer Instanz / Klasse zugeordnete Daten
- **Methode**: Einer Instanz / Klasse zugeordnete Algorithmen
- **Vererbung**: Ableitung einer Subklasse, die die Attribute / Methoden ihrer Superklasse(n) erbt
- **Kapselung**: Verbergen der internen Struktur einer Klasse
- **Polymorphie**: Fähigkeit, unterschiedliche Datentypen anzunehmen
- **Abstrakte Klasse**: Klasse, die nur "Rumpf"methoden enthält und die nicht instanziiert werden kann (dient als Bauplan für Subklassen)
- **Virtuelle Methode**: Dynamisch gebundene Methode (späte Bindung); erlaubt Überschreiben / Überlagern von Methoden

Besonderheiten von Python:

- Dynamisches Typsystem / Bindung zur Laufzeit
  - Instanzen können beliebig erweitert werden
  - Methoden sind prinzipiell *virtuell*
  - Operatoren-Überladung möglich (Überschreiben von 'Special Methods')
- *self* muss explizit angegeben werden
- Multiple Vererbung wird unterstützt
- Nur Pseudo-Private (über Konvention / Name mangling. Zugriff bleibt möglich)
- Keine echten abstrakten Klassen (bzw. nur mit Delegatoren oder Metaklassen)
- Properties werden unterstützt
- Metaklassen erlauben eigene Klassenerzeugungssysteme (in Python ist alles ein Objekt; sogar Klassen) (Very advanced Python wizardry!)

Die Definition einer Klasse erfolgt durch das Keyword *class*.

Eine Instanz wird durch Aufrufen des Klassennamens mit Argumentliste erzeugt.

```
class <name>:                    # Class definition
   <statements>

<instance> = <classname>()    #  Instantiation
```

```
class Person:
   pass

professor = Person()
artist = Person()
student = Person()
```

Aufruf / Zugriff auf Instanzmember erfolgt über `.` (Punkt-Syntax)

```
class <name>:
   def method(self, args):  # Defining an instance method
      <statements>

<instance>.method(args)     # Calling an instance method
<instance>.attribute        # Accessing an instance attribute
```

```
class Person:
   def set_age(self, age):
      self.age = age

professor = Person()
professor.set_age(72)
print professor.age
```

Beim Erzeugen einer neuen Instanz wird die Sondermethode `__init__` aufgerufen (Konstruktor).

```
class <name>:
   def __init__(self, args):     # Instance constructor
      <statements>

<instance> = <classname>(args)   # Generating an instance
```

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

professor = Person("Stephen Hawking")
print professor.name
```

Als erstes Argument jeder Definition einer Methode ***muss*** die Referenz auf die Instanz stehen (Python-Philosophie 'Explicit is better than Implicit').

Per Übereinkunft heisst diese immer ***self***.

Beim Aufruf einer Instanzmethode wird die Instanzreferenz vom Python-Interpreter als erstes Element in die Argumentliste eingefügt.

```
instance.method(args)  ->  cls.method(instance,args)
```

```
class Person:
   def __init__(self, who):
      self.name = who
   def set_age(self, age):
      self.age = age

professor = Person("Stephen Hawking")
professor.set_age(70)     # Person.set_age(professor,70)
```

Genaueres siehe Essay von Guido van Rossum:
http://neopythonic.blogspot.de/2008/10/why-explicit-self-has-to-stay.html

Ab Python 2.2 wurden sog. 'New Style' Klassen eingeführt.

Um in Python 2.x eine 'New Style' Klasse zu erhalten muss sie explizit von `object` erben

Ab Python 3.0 sind prinzipiell alle Klassen 'New Style'.

Unterschiede sind i.d.R. minimal für die meisten Programmierprobleme; genaueres -> später. (Diamant-Vererbung, manche built-in Methoden, etc.)

In diesem Workshop werden nur New-Style Klassen verwendet (generell empfohlen)

```
class ClassicStyle:         # Classic (Python 2.x)

class NewStyle(object):     # New Style (Python 2.x, ab 2.2)

class NewStyle:             # New Style (Python 3.x)
```

Klassennamen sollten im **`CamelCase`** stehen.

Funktionen und Methoden sollten **`lowercase`** sein,
mit zusätzlichen Underscores `_` , falls es der Lesbarkeit dient.

`self` niemals anders nennen!

Konstanten prinzipiell **UPPERCASE**,
mit Underscores `_` , falls die Lesbarkeit damit erhöht wird

```
class Person(object):                    # Pythonic style
    BMI_NORMAL = 24.9
    def set_age(self, age):
        self.age = age

class person_Buddy(object):              # Very unpythonic!
    normalBmi = 24.9
    def SetAge(this_inst, age):
        this_inst.age = age
```

Vererbung: Elternklassen werden in der Parameterliste des class-Statements angegeben (multiple Vererbung wird unterstützt)

```
class <subclass>(<superclass>,...):
```

```
class Person(object):
   def __init__(self, who):
       self.name = who

class Professor(Person):
   <professor specific members>

class Student(Person):
   <student specific members>

prof = Professor("Stephen Hawking")
stud = Student("John Somebody")
```

```
class Super(object):
    def method(self):
        print "In Super.method"
    def action(self):
        raise NotImplementedError("Subclass must implement
                                   abstract method!")

class Inheritor(Super): pass

class Replacer(Super):
    def method(self):
        print "In Replacer.method"

class Extender(Super):
    def method(self):
        Super.method(self)
        print "In Extender.method"

class Provider(Super):
    def action(self):
        print "In Provider.action"
```

Elternmethoden können auf zwei Arten aufgerufen werden:

a) Direkt über `<Superclass>.method(self,args)`

b) Über die built-in Funktion `super(baseclass,self).method(args)`

```
class Parent(object):
   def method(self):
      print "In Super.method"

class Child1(Parent):
   def method2(self):
      Parent.method(self)

class Child2(Parent):
   def method2(self):
      super(Child2,self).method()
```

Attribut / Methodenname	Bedeutung	Kommentar
`name`	public	
`_name`	protected	Nur Konvention! Zugriff weiterhin möglich
`__name`	private	wird intern zu `_classname__name`
`__name__`	special	

`type(object)`	Ergibt den Typ (=Klasse) eines Objekts
`isinstance(object, cls)`	Test ob `object` eine Instanz von `cls` oder einer Subklasse davon ist
`issubclass(cls, supercls)`	Test ob Klasse `cls` eine Unterklasse von `supercls` ist
`hasattr(object, "name")`	Test ob `object` das Attribut / Methode `name` besitzt
`id(object)`	Ergibt die interne ID von `object`
`super(BaseClass, instance)`	Gibt ein Proxyobjekt der Elternklasse der `instance` aus `BaseClass`
`<class>.mro()`	Gibt den Suchbaum der Klassenhierarchie von `class`

(funktionieren i.d.R. nur mit New-Style Classes)

Klassen:

`__name__`	Klassenname
`__bases__`	Elternklassen (Tupel)
`__mro__`	Vererbungssuchpfad

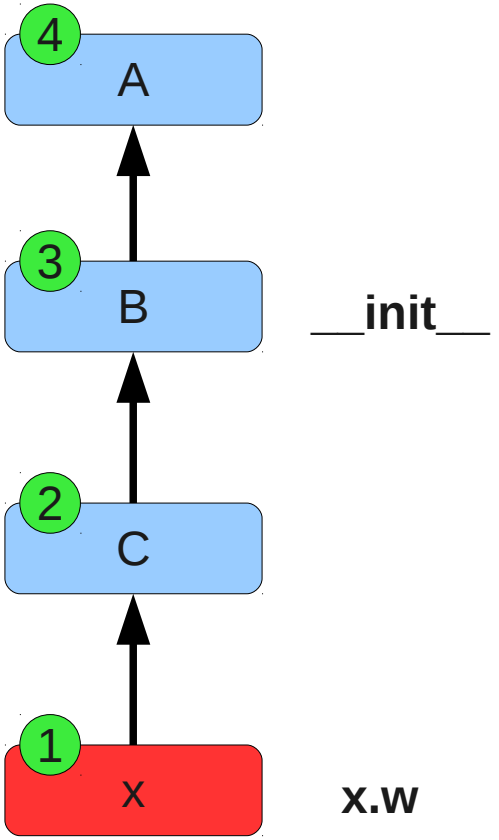
Klassen und Instanzen:

`__class__`	Klassenzugehörigkeit einer Instanz
`__dict__`	Methoden und Attribute (Dictionary)
`__doc__`	Documentation String der Klassendefinition

(funktionieren i.d.R. nur mit New-Style Classes)

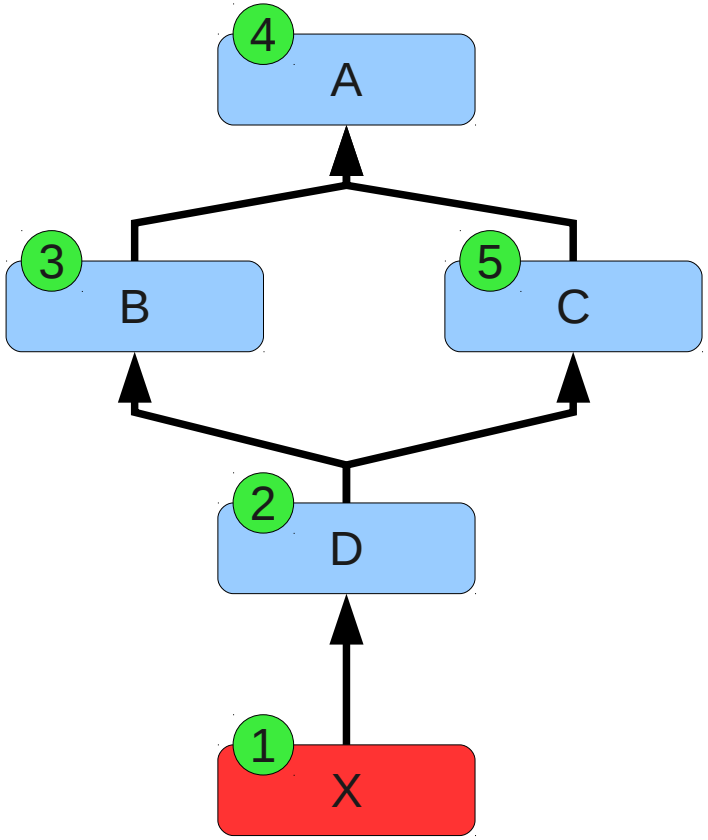
Der **Vererbungspfad** in Python ist im Prinzip eine Frage des Namespace:

```
class A(object):
    def __init__(self):
        self.w = "Instanz Klasse A"

class B(A):
    def __init__(self):
        self.w = "Instanz Klasse B"

class C(B):
    pass

x = C()         # Welcher Konstruktor?
print x.w       # Wo ist jetzt x.w?
```

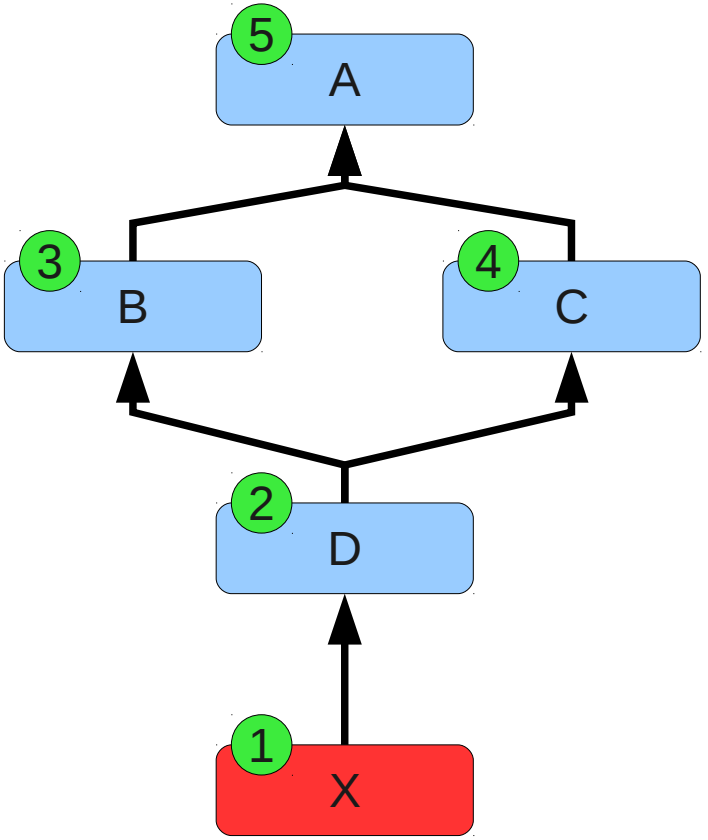


```
class A:
   def __init__(self):
      self.w = "Klasse A"

class B(A):
   pass

class C(A):
   def __init__(self):
      self.w = "Klasse C"

class D(B,C):
   pass

x = D()

print x.w
# Was wird ausgegeben?
```

```
class A(object):
   def __init__(self):
      self.w = "Klasse A"

class B(A):
   pass

class C(A):
   def __init__(self):
      self.w = "Klasse C"

class D(B,C):
   pass

x = D()

print x.w
# Was wird ausgegeben?
```



Classic Classes



New-style Classes

Genauer Suchpfad bei Mehrfachvererbung (nur New-Style):

```
classname.__mro__
```

Besondere Methoden und **Operatorenüberladung**

Pythonobjekte implementieren eine Vielzahl von *Special Methods.*

Operatorenüberladung erfolgt durch Überschreiben:

```
class MyNumber(object):

   def __init__(self, value):
      self.data = value

   def __add__(self, other):
      return MyNumber(self.data + other.data)

   def __str__(self):
      return str(self.data)

x = MyNumber(5)
y = x + MyNumber(6)
print y.data            # Ergibt 11
print x, y              # Ergibt 5 11
```

Besondere Methoden: **Allgemeine Methoden**

`__init__(self [,args])`	Konstruktor
`__del__(self)`	Finalizer
`__str__(self)`	`str(self)` ('User-Friendly' String)
`__repr__(self)`	`repr(self)` ('Representative' String)
`__hash__(self)`	`hash(self)`
`__nonzero__(self)`	Test des Objekts auf Wahrheitswert (Python 2)
`__bool__(self)`	Test des Objekts auf Wahrheitswert (Python 3)
`__cmp__(self,other)`	Vergleich Objekt `self` mit `other`
`__getattr__(self,name)`	`self.name` (*name* existiert nicht)
`__getattribute__(self,name)`	`self.name` (*name* existiert)
`__setattr__(self,name,value)`	`self.name = value`
`__delattr__(self,name)`	`del self.name`
`__call__(self [,args])`	`self([args])` Aufruf des Objekts als Funktion
`__new__(cls [,args])`	Erzeugt neues Klassenobjekt (!= Konstruktor!)

Was ist mit dem **Destruktor**?

Antwort: In Python gibt es den *Finalizer* __del__

Aufruf erfolgt durch den Garbage Collector, muss deshalb i.d.R. nicht explizit aufgerufen werden.

Verwendung wird abgeraten, da der Aufruf nicht vorhersehbar erfolgt

```
class Person(object):
   def __init__(self, who):
      self.name = who
   def __del__(self):            # Kann zu Problemen führen!
      <do_something>
```

Besondere Methoden: **Collection Objects**

```
__len__(self)                  len(self)
__contains__(self, item)       item in self
__getitem__(self,key)          self[key]
__setitem__(self,key,value)    self[key] = value
__delitem__(self,key)          del self[key]

__reversed__(self)             self.reversed()

__iter__(self)                 iter(self)
__next__(self [,default])      next(self [,default])
```

Besondere Methoden: **Mathematische Objekte**

```
__add__(self, other)          self + other
__sub__(self, other)          self - other
__mul__(self, other)          self * other
__div__(self, other)          self / other          (Python 2)
__truediv__(self, other)      self / other          (Python 3)
__floordiv__(self,other)      self // other
__mod__(self, other)          self % other
__divmod__(self,other)        divmod(self, other)
__pow__(self,other[,modulo])  self ** other [% modulo])

__and__(self,other)           self & other
__or__(self,other)            self | other
__xor__(self,other)           self ^ other

__radd__(self,other)          other + self
__iadd__(self, other)         self += other
```

Ein wichtiges Paradigma von OOP ist *Encapsulation*.

Klasseninterna sollen nach außen verborgen bleiben und nur über Zugriffsschnittstellen erreichbar sein.

```
class Person(object):
    def __init__(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, value):
        self.__age = value
```

Problem: Unübersichtliche Syntax beim Zugriff über getter / setter Methoden

```
p = Person(23)

p.set_age(p.get_age() + 1)    # Mache p ein Jahr älter
```

Lösung: **Properties**

```
<attr> = property(<getter>, <setter>, <deleter>, 'docstring')

# Defaultwerte:
<attr> = property(fget=None, fset=None, fdel=None, doc=None)
```

```
class Person(object):

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

    def getName(self):
        print "Fetching the name"
        return self._name

    def setName(self, value):
        print "Changing the name"
        self._name = value

    def delName(self):
        print "Removing the name"
        del self._name

    name = property(getName, setName, delName, "Name property")

p = Person("Stephen Hawking")
p.name = "Albert Einstein"
print p.name
del p.name
print Person.name.__doc__
```

Properties per **Dekoratoren:**

```
class Person(object):

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

    @property
    def name(self):
        " Name property docstring "
        print "Fetching the name"
        return self._name

    @name.setter
    def name(self, value):
        print "Changing the name"
        self._name = value

    @name.deleter
    def name(self):
        print "Removing the name"
        del self._name
```

Zuweisungen an Objekte außerhalb von Methodendefinitionen erzeugen Attribute, die nicht einer Instanz, sondern der *Klasse* zugewiesen sind, und auf die jede Instanz Zugriff hat.

Der Zugriff erfolgt über `<classname>.attribute`

```
class Person(object):
    counter = 0
    def __init__ (self, who):
        self.name = who
        Person.counter += 1

professor = Person("Stephen Hawking")
artist = Person("van Gogh")

print Person.counter          #  2
```

An die Klasse gebundene Methoden gibt in den Variationen
Static Method und *Class Method.*

- *Static Methods* sind an keine Instanz gebunden und haben daher keinen Parameter *self.*
- *Class Methods* haben anstelle von *self* als ersten Parameter **cls**, der nicht die jeweilige Instanz, sondern die aufrufende Klasse übergibt.

```
class <name>(object):

   def a_static_method(args):
      <statements>

   a_static_method = staticmethod(a_static_method)

   def a_class_method(cls,args):
      <statements>

   a_class_method = classmethod(a_class_method)
```

An die Klasse gebundene Methoden gibt in den Variationen
Static Method und *Class Method.*

- *Static Methods* sind an keine Instanz gebunden und haben daher keinen Parameter *self.*
- *Class Methods* haben anstelle von *self* als ersten Parameter **cls**, der nicht die jeweilige Instanz, sondern die aufrufende Klasse übergibt.

Syntax in Dekoratorenschreibweise:

```
class <name>(object):

    @staticmethod
    def a_static_method(args):
        <statements>

    @classmethod
    def a_class_method(cls,args):
        <statements>
```

```
class C(object):
    value = 20

    def __init__(self,val):           self.value = val

    def a_instance_method(self,val): self.value = val

    @staticmethod
    def a_static_method(val):         C.value = val

    @classmethod
    def a_class_method(cls,val):      cls.value = val
```

```
a = C(1)
b = C(10)

print a.value, b.value, C.value                                  # 1 10 20

a.a_instance_method(2);  print a.value, b.value, C.value # 2 10 20
a.a_static_method(21);   print a.value, b.value, C.value # 2 10 21
a.a_class_method(22);    print a.value, b.value, C.value # 2 10 22

b.a_instance_method(11); print a.value, b.value, C.value # 2 11 22
b.a_static_method(23);   print a.value, b.value, C.value # 2 11 23
b.a_class_method(24);    print a.value, b.value, C.value # 2 11 24
```

Member einer Instanz können auch außerhalb einer Klassendefinition erzeugt werden.

Es wird aber i.d.R. davon abgeraten (Python lässt einem alle Freiheiten)

Erzeugen eines Instanzmembers überlagert schon vorhandene Member des gleichen Namens in tieferer Namespaceebene

```
class Person (object):
    age = 20

p = Person()
s = Person()
print p.age, s.age      # 20, 20  (Zugriff auf Klassenmember)

p.age = 40              # Erzeugt Instanzmember (wie self.age)
print p.age, s.age      # 40, 20

del p.age               # Entfernt den Instanzmember
print p.age, s.age      # 20, 20
```

Bei Klassenattributen treten die gleichen Effekte auf wie bei Variablen und Funktionen, falls das Attribut ein *Mutable Object* ist:

```
class A(object):
   w = [1, 2, 3]

x = A()
y = A()

x.w[2] = 4
print x.w    # [1, 2, 4]
print y.w    # [1, 2, 4]
```

Eine Vererbungshierarchie ist in Python im Prinzip ein Namespace.

Achtung! Dies kann bei der Verwendung von Klassenattributen aufgrund der dynamischen Laufzeitbindung unerwünschte Seiteneffekte haben:

```
class A(object):
    w = 0

class B(A):
    pass

x = B()
print x.w, A.w   # (0, 0) Greift auf Klassenattribut A.w zu!
x.w += 1         #        Erzeugt Instanzattribut x.w
print x.w, A.w   # (1, 0)

A.w += 1         #         A.w ist jetzt 1 ...
y = B()          #         ... B erbt *jetzt* neu von A!
y.w += 1         #
print x.w, y.w, A.w  # (1, 2, 1)
```

Very advanced Python Wizardry!

Metaklassen sind Klassen um Klassen zu erzeugen.

Mit einer Metaklasse kann man den Prozess der Klassenerzeugung und -verwaltung neu definieren. (meist über die Sondermethode __new__)

Verwendung einer Metaklasse erfolgt über das Keyword *metaclass*:

```
class <classname>(Object):                  # Python 2.x
    __metaclass__ = <meta>


class <classname>(metaclass=<meta>)     # Python 3
```

```
from abc import ABCMeta, abstractmethod

class Super:
   __metaclass__ = ABCMeta          # Python 2.6 / 2.7

   @abstractmethod
   def action(self):
      pass
```

```
>>> x = Super()
(Fehler: Abstrakte Superklasse kann nicht instantiiert werden)

>>> class Sub(Super): pass
>>> x = Sub()
(Fehler: Implementierung der abstrakten Methode fehlt)

>>> class Sub(Super):
...    def action(self): print ("Action!")
...
>>> x = Sub()
>>> x.action()
(Jetzt funktioniert es!)
```

```
class <name>(object):                # Definition (New-Style)
   """ docstring """                 # Class docstring

   def method(self, args):           # Class method definition
      """ docstring """              # Method docstring
      self.member = value            # Per-instance data
      self._protected = value        # Pseudo-protected data
      self.__private = value         # Pseudo-private data

   def __init__(self, args):         # Constructor
      <statements>

   def __str__(self):                # Operator Overloading
      <statements>

<instance> = <classname>(args)       # Instantiation
<instance>.member                    # Attribute access
<instance>.method(args)              # Method access
```

```
class <name>(Superclass1, ...):  # Definition & Inheritance
   __metaclass__ = <metaclass>   # Metaclass invocation

   static_data = value           # Static class data

   @property
   def name(self):               # Property access method
      "Docstring Property"
   name = property(fget, fset, fdel, doc)

   <Superclass>.method(self, args)  # Parent method call
   super(<name>,self).method(args)

   @staticmethod
   def static(args):             # Static method
   static = staticmethod(static)

   @classmethod
   def clsmethod(cls, args):     # Class method
   clsmethod = classmethod(clsmethod)
```

Pseudo-Structs:

Übersichtliches Organisieren von Daten ähnlich einer C Struct

In diesem Fall ist es ok, Instanzattribute außerhalb der Klasse zu erzeugen.

```
class Konto: pass

giro = Konto()
giro.blz = "667 800 00"
giro.number = 452890000
giro.balance = 45.38
giro.owner = "Max Mustermüller"

print giro.number
print giro.owner
...
```

Caching:

Zwischenspeichern von zeitaufwendigen Berechnungen, um Ergebnis bei erneuter Berechnung schnell verfügbar zu haben

```
def fibonacci(n):
    if n in (0, 1):
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

```
@Cache
def fibonacci(n):
    if n in (0, 1):
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Caching mittels Dekorator-Objekt

```
class Cache(object):

   def __init__(self):
      self.cache = {}
      self.func = None

   def cachedFunc(self, *args):
      if args not in self.cache:
         self.cache[args] = self.func(*args)
      return self.cache[args]

   def __call__(self, func):
      self.func = func
      return self.cachedFunc
```

Exceptions sind eigene Klassen.
Eigene Exceptions können definiert werden:

```
class MyError(Exception):

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

   def __str__(self):
      return "A custom error occurred with code: " \
             + str(self.value)
```

```
>>> from examples import MyError
>>> raise MyError(5)
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
examples.MyError: A custom error occurred with code: 5
```

(Näheres zum Exception-System von Python kommt in Workshop III)

Geplant für den Workshop Python III:

- Exceptions
- Funktionale Programmierung
- Die Standard Library