


Informatik II
Einführung in Python, Beyond the Basics

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de


Höhere Datenstrukturen

- Eines der Features, das Python so mächtig macht (→ "very high level language")
- Enthält wichtige Klassen von höheren Datenstrukturen
  - Sequenzen, Dictionaries, Tupel, ...
- Sequenz := geordnete Mengen von Objekten
  - Werden immer mit natürlichen Zahlen indiziert
    - Index >= 0 (außer bei Fortran, wo Index >= 1!)
  - Unterklassen: Strings, Tupel, Listen
  - Liste = veränderbare (*mutable*) Sequenz
  - String & Tupel = nicht-veränderbare (*immutable*) Sequenz
- Dictionary := "assoziatives Array"
  - Indizierung mit einem Key, z.B. String

Strings

- Strings sind Zeichen-Sequenzen

```
a = "Python ist toll"
```

- Zugriff auf die Zeichen erfolgt durch den Index-Operator []

```
b = a[3]    # b = 'h'
```

- Teilstrings erhält man mit dem *Slice*-Operator [i:j]

```
c = a[3:5]    # b = "hon"
```

- Strings lassen sich mit dem +-Operator konkatenieren

```
d = a + " sagt der Professor"
```


- Strings sind in Python nicht veränderbar, wenn sie einmal festgelegt wurden

```
# Folgendes ist in Python nicht möglich
a = "Qython ist toll"
a[0] = 'P'
```

- Operationen auf Strings:

Operationen auf Strings	
`s[i]`	Ergibt Element i der Sequenz s
`s[i:j]`	Ergibt einen Teilstring (*slice*)
`len(s)`	Ergibt die Anzahl der Elemente in s
`min(s)`	Ergibt Minimum
`max(s)`	Ergibt Maximum
`float(s)`	String nach Float konvertieren (analog int)
`str(4.2)`	Konvertiert die Zahl in einen String

Funktionen

- Umfangreiche Programme werden in Funktionen aufgeteilt, dadurch werden sie modularer und einfacher zu warten.
- Funktionen werden mit der **`def`**-Anweisung definiert:

```
def add( x, y ):
      return x+y
```

  - Achtung: es gibt keine separate Deklaration!
    (wie in C++ durch den sog. *Function Prototype*, z.B. in Header-Files)
- Funktionsaufrufe erfolgen durch Angabe des Funktionsnamens und der Funktionsargumente

```
a = add( 3, 5 )
```

  - Achtung: Anzahl der Argumente muß mit der Funktionsdefinition übereinstimmen, sonst wird ein Type-Error ausgelöst

Achtung

- Es wird kein Rückgabetyp deklariert!
  - Funktion kann jedesmal Objekte von verschiedenem Typ liefern!
  - Große Flexibilität, große Gefahr

"With great power comes great responsibility"

- Parameter haben keinen Typ deklariert!
  - Dito

Parameterübergabe

- Ist in Python prinzipiell *Call-by-Reference*
  - Funktion bekommt Referenz (= Zeiger), keine Kopie des Wertes
  - D.h., Parameter können in der Funktion geändert werden!

```
def set( x ):
  x[0] = 3

a = [ 1, 2, 3 ]
print a
set( a )
print a
```

Ausgabe

```
123
323
```

- Ausnahmen: "einfache" Datentypen (Integer, Float, String)!
  - Hier findet Call-by-Value statt!
  - D.h., der Parameter ist eine Kopie des Argumentes

```
def set( x ):
  x = 3

a = 1
print a
set( a )
print a
```

Ausgabe

```
1
1
```


Rückgabewerte

- Die return-Anweisung gibt einen Wert aus der Funktion zurück

```
a = 3
def square( x ):
  square = x*x
  return square

print a
a = square( a )
print a
```

Ausgabe

```
3
9
```

- Mehrere Werte können als Tupel zurückgegeben werden:

```
def square_cube( x ):
  square = x*x
  cube = x*x*x
  return ( square, cube )

a = 3
x, y = square( a )        # keine Klammern auf lhs!
```

Keyword-Parameter

- Die Parameter in einem "klassischen" Aufruf der Art

```
foo( 1.0, "hello", [1,2,3] )
```

  heißen *"positional arguments"*, weil ihre Zuordnung zu den formalen Parametern durch die Position in der Liste aller Argumente gegeben ist

- Alternative: Parameter-Übergabe durch *Key-Value*-Paare:

```
def new_frame(text, name, bg_color, fg_color, font, size):
    … code …

new_frame("Hello World", name="hello", font="Helvetica",
          size=4, bg_color="blue", fg_color="red" )
```


Default-Werte

- Angabe aller Parameter bei langer Parameterliste ist manchmal mühsam
- Lösung: Default-Argumente

```
def new_frame( text, name="upper_left", bg_color="white",
               fgcolor="black", font="Ariel", size=2 ):
    … code …

new_frame( "Hello World", font="Helvetica" )
new_frame( "Our products", "index_frame", size=4,
           bg_color="blue")
```

  - Alle Parameter, die im Aufruf nicht angegeben werden (*positional* oder *key/value*), werden mit Default-Argumenten belegt

- Regeln für Argumente
- Beispiel-Funktion:

```
def f(name, age=30):
    … code …
```

- Argument darf nicht sowohl *positional* als auch als *Key/Value* gegeben werden:

```
f("aaron", name="sam") --> ValueError
```

- *Positional* Argumente müssen *Key/Value*-Argumenten voranstehen

```
f(name="aaron", 34) --> SyntaxError
```

- Alle Argumente ohne Default-Werte müssen definiert werden

```
f() --> ValueError
```


Funktionen als Parameter

- Funktionen sind vollwertige Objekte!

→ Funktionen können auch Funktionen als Parameter erhalten

  - Analoges Konstrukt in C: Funktionszeiger
  - Analoges Konstrukt in C++: Funktoren

- Beispiel: Sortieren von Listen

```
list.sort( cmpfunc )
```

Beispiel **map**

- Die Funktion **`t = map( func, s)`** wendet die Funktion **`func()`** auf alle Elemente eines Arrays **`s`** an und gibt eine neue Liste **`t`** zurück

```
a = [1, 2, 3, 4, 5, 6]
def triple(x):
      return 3*x
b = map( triple, a ) # b = [3,6,9,12,15,18]
```

- Weiteres map-Beispiel: alle *command line arguments* als int's lesen

```
import sys
int_args = map( int, sys.argv[1:] )
```


Scope von Variablennamen

- Scope := Gültigkeitsbereich eines Identifiers
- Variablennamen in Funktionen nur innerhalb der Funktion gültig:

```
a = 3
def foo( x ):
  a = 5

print a
foo( a )
print a
```

Ausgabe
3
3

- Auf Variablen außerhalb greift man per **`global`**-Anweisung zu:

```
a = 3
def foo( x ):
  global a
  a = 5

print a
foo( a )
print a
```

Ausgabe
3
5

Beispiel: Berechnung des Wochentages

```
def Day( day, month, year ):
       days = ["Mo","Di","Mi","Do","Fr","Sa","So"]
       y = year - (14 - month) / 12
       x = y + y/4 - y/100 + y/400
       m = month + 12 * ((14 - month) / 12) - 2
       d = (day + x + (31*m)/12) % 7
       return days[d]

print Day( 24, 12, 2005 )
```

Ausgabe

```
Sa
```


Module

- Wenn Programme zu lang werden, kann man sie in mehrere Dateien unterteilen, um sie besser warten zu können
- Python erlaubt es, Definitionen in eine Datei zu setzen und sie als Modul zu benutzen
- Um ein Modul zu erzeugen schreibt man die Def's in einen File, der denselben Namen wie das Modul und Suffix **`.py`** hat
- Beispiel:

```
# File: div.py          # bildet divmod() nach
def divide( a, b ):
       q = a/b
       r = a-q*b        # Wenn a und b ganzzahlig, dann auch q
       return (q, r)    # liefert ein Tuple
```

Verwendung von Modulen

- Um ein Modul zu verwenden benutzt man die **`import`**-Anweisung
- Um auf eine Funktion aus einem Modul zuzugreifen, stellt man ihr den Modulnamen voran:

```
import div
a, b = div.divide( 100, 35  )
```

- Um spezielle, einzelne Definitionen in den aktuellen *name space* (Namensraum) zu importieren benutzt man die **`from`**-Anweisung:

```
from div import divide
a, b = divide( 100, 35  )
```

  - Alle Definitionen eines Moduls kann man mit einem Wildcard in den aktuellen Namensraum importieren:

```
from div import *
```


Der Modul-Suchpfad

- Beim Laden der Module sucht der Interpreter in einer Liste von Verzeichnissen, die in der Liste **`sys.path`** definiert ist:

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python1.5/',
'/usr/local/lib/python1.5/test',
'/usr/local/lib/python1.5/plat-sunos5',
'/usr/local/lib/python1.5/lib-tk',
'/usr/local/lib/python1.5/lib-dynload',
'/usr/local/lib/site-python']
```

- Neue Verzeichnisse fügt man dem Suchpfad durch einen Eintrag in die Liste hinzu, z.B.

```
>>> sys.path.append( "." )
```

Das Modul math

- Das Modul math definiert mathematische Standarfunktionen für Floating-Point-Zahlen

Einige Funktionen des math-Moduls	
`ceil(x)`	Ergibt nächstgrößere ganze Zahl von x.
`cos(x)`	Ergibt Cosinus von x.
`exp(x)`	Ergibt e ** x.
`fabs(x)`	Ergibt Betrag von x.
`floor(x)`	Ergibt nächstkleinere ganze Zahl von x.
`fmod(x, y)`	Ergibt x % y.
`frexp(x)`	Ergibt positive Mantisse und Exponenten von x.
`hypot(x, y)`	Ergibt Euklidischen Abstand, sqrt(x*x+y*y).
`ldexp(x, i)`	Ergibt x * (2 ** i).
`log(x)`	Ergibt natürlichen Logarithmus von x.
`log10(x)`	Ergibt Logarithmus zur Basis 10 von x.
`pow(x, y)`	Ergibt x ** y.
`sin(x)`	Ergibt Sinus von x.
`sqrt(x)`	Ergibt Quadratwurzel von x.
`tan(x)`	Ergibt Tangens von x.

Komplexe Zahlen und das Modul cmath

- Python hat neben Ganzzahlen und FP-Zahlen auch komplexe Zahlen als Typ direkt eingebaut (**`complex`**)
- Zahlen mit j am Ende interpretiert Python als komplexe Zahlen
- Komplexe Zahlen mit Real- und Imaginärteil erzeugt man durch
- Addition, also z.B. c = 1.2 + 12.24j
- Das Modul cmath definiert mathematische Standarfunktionen für komplexe Zahlen

Einige Funktionen des cmath-Moduls	
`cos(x)`	Ergibt Cosinus von x.
`exp(x)`	Ergibt e ** x.
`log(x)`	Ergibt natürlichen Logarithmus von x.
`log10(x)`	Ergibt Logarithmus zur Basis 10 von x.
`sin(x)`	Ergibt Sinus von x.
`sqrt(x)`	Ergibt Quadratwurzel von x.
`tan(x)`	Ergibt Tangens von x.

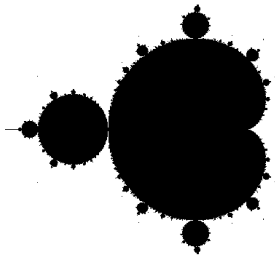
Beispiel: Mandelbrotmenge

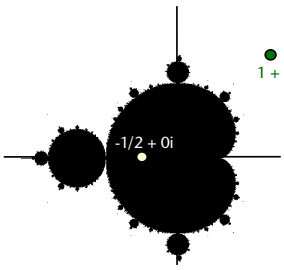
- Menge von Punkten M in der komplexen Ebene:
  - Bilde zu jedem $c \in \mathbb{C}$ die (unendliche) Folge

$$z_{i+1} = z_i^2 + c\,, \quad z_0 = 0$$

  - Definiere die Mandelbrot-Menge

$$\mathbb{M} = \{c \in \mathbb{C} \mid \text{Folge } (z_i) \text{ bleibt beschränkt } \}$$





i	z_i
0	1 + i
1	1 + 3i
2	-7 + 7i
3	1 - 97i
4	-9407 – 193i
5	88454401 + 3631103i

c = 1 + i ist nicht in der Mandelbrotmenge enthalten

i	z_i
0	-1/2
1	-1/4
2	-7/16
3	1
4	-79/256
5	-26527/65536

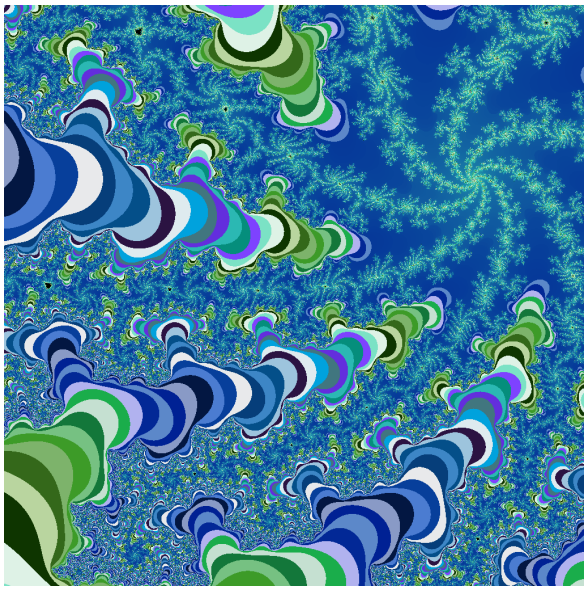
c = -1/2 ist in der Mandelbrotmenge enthalten

Visualisierung der Mandelbrotmenge

- Färbe Pixel (x, y) schwarz falls z = x + iy in der Menge ist, sonst weiß
- Einschränkungen in der Praxis:
  - Man kann nicht unendlich viele Punkte zeichnen
  - Man kann nicht unendlich oft iterieren
- Deswegen: approximative Lösung
  - Wähle eine endliche Menge von Punkten aus
  - Iteriere N mal
    - Satz (o. Bew.): Ist $|z_t| > 2$ für ein t, dann ist c nicht in der Mandelbrotmenge
    - Es gilt (fast immer): Ist $|z_{1000}| \leq 2$ dann ist c "wahrscheinlich" in der Mandelbrotmenge enthalten
- Schönere Bilder erhält man, wenn man die Punkte zusätzlich färbt:
  - Färbe c abhängig von der Anzahl an Iterationen t die nötig waren, bis $|z_t| > 2$ wurde.

Mandelbrotmengen-Impressionen

Mandelbrotmengen-Impressionen



Das Modul `random`

- Das Modul **`random`** erzeugt Pseudo-Zufallszahlen

Einige Funktionen des random-Moduls

`choice(seq)`	Gibt zufälliges Element einer Sequenz seq zurück
`random()`	Gibt Zufallszahl zwischen 0 und 1 aus
`uniform(a, b)`	Gibt normalverteilte Zufallszahl aus dem Intervall [a, b)
`randint(a, b)`	Gibt ganzzahlige Zufallszahl aus dem Intervall [a, b]
`seed([x])`	Initialisiert den Zufallszahl-Generator. Falls x nicht explizit angegeben wird, wird einfach die aktuelle Systemzeit verwendet

Die Python Imaging Library (PIL)

- Bibliothek zum Erzeugen, Konvertieren, Bearbeiten, usw von Bildern
- Die Bibliothek enthält mehrere Module
  - Image Modul
  - ImageDraw Modul
  - ImageFile Modul
  - ImageFilter Modul
  - ImageColor Modul
  - ImageWin Modul
  - ...

Das Modul `Image` der PIL

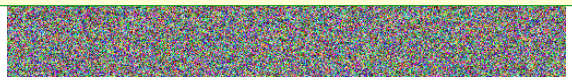
- Stellt grundlegende Funktionen zur Bilderzeugung zur Verfügung:

Einige Funktionen des Image-Moduls

`new(mode, size)`	Erzeugt neues Bild. mode ist ein String der das verwendete Pixelformat beschreibt (z.B. "RGB", "CMYK"), size ist ein 2-Tupel, durch welches Höhe und Breite des Bildes in Pixeln angegeben werden
`new(mode, size, color)`	Wie oben mit einem zusätzlichen 3-Tupel für die Farbtiefe.
`putpixel( xy, color)`	Setzt den Pixel an der Position (x, y) auf den angegebenen Farbwert
`show()`	Zeigt das Bild an. Die Ausgabe ist abhängig vom verwendeten Betriebssystem
`save(outfile, options)`	Speichert ein Bild in der Datei mit dem Namen outfile. Zusätzlich können noch Optionen angegeben werden

Beispiel

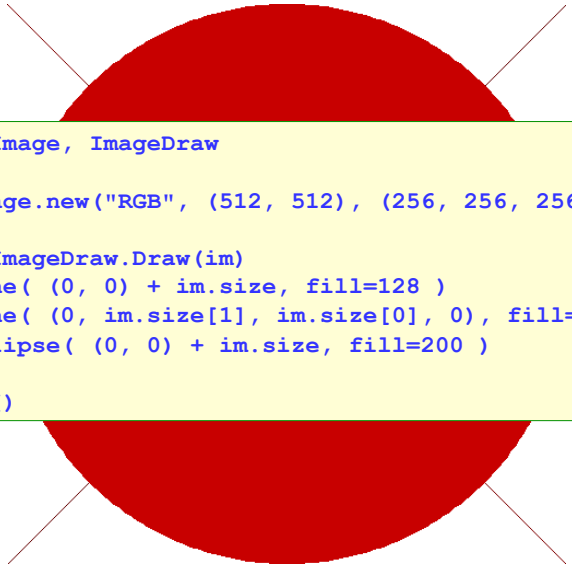
```
# import Libraries
import Image
import random

# Create new Image
im = Image.new("RGB", (512, 512), (256, 256, 256) )

# Set some Pixels randomly in the Image
for i in range( 0, 512 ):
    for j in range( 0, 512 ):
        r = random.randint(0, 256)
        g = random.randint(0, 256)
        b = random.randint(0, 256)
        im.putpixel( (i, j), (r, g, b) )

# Finally: Show the image
im.show()
```



Das Modul ImageDraw der PIL

- Einfache Funktionen zum Erzeugen von 2D-Grafiken:
  - **`ellipse`**(xy, options) Erzeugt eine Ellipse
  - **`line`**(xy, options) Erzeugt eine Linie
  - **`point`**(xy, options) Erzeugt einen Punkt
  - **`polygon`**(xy, options) Erzeugt ein Polygon
  - **`rectangle`**(box, options) Erzeugt ein Rechteck
  - **`text`**(position, string, options) Erzeugt eine Text

Beispiel



```
import Image, ImageDraw

im = Image.new("RGB", (512, 512), (256, 256, 256) )

draw = ImageDraw.Draw(im)
draw.line( (0, 0) + im.size, fill=128 )
draw.line( (0, im.size[1], im.size[0], 0), fill=100 )
draw.ellipse( (0, 0) + im.size, fill=200 )

im.show()
```


Unit-Tests in Python

- Unit-Test
  - Einfachste Form von Software-Test aus dem Software-Engineering
  - Unit = Funktion oder Klasse
  - Testet, ob Ist-Ausgabe der Soll-Ausgabe entspricht
  - Wird normalerweise vom Programmierer der Funktion/Klasse gleich mitgeschrieben
    - er weiß am besten, was rauskommen muß
    - Dient gleichzeitig der gedanklichen Unterstützung beim Aufstellen der Spezifikation der Funktion / Klasse
- Unit-Tests können später automatisiert im Batch ablaufen
  - Stellt sicher, daß Einzelteile der Software noch das tun, was sie sollen
  - Stellt sicher, daß im Code-Repository immer eine korrekte Version ist

Integration von Unit-Tests im Modul selbst (der `__name__`-Trick)

- Jedes Python-Modul besitzt einen eigenen Namen
  - Innerhalb eines Moduls ist der Modulname (als String) als Wert der globalen Variablen **`__name__`** verfügbar.
  - Im Hauptprogramm enthält diese Variable "**`__main__`**"
- Dadurch lassen sich in Python sehr leicht Unit-Tests direkt im Modul implementieren:
  - Bestimmte Teile eines Moduls werden nur dann ausgeführt, wenn man es als eigenständiges Programm startet
  - Beim Import in ein anderes Modul werden diese Teile nicht ausgeführt

```
if __name__ == "__main__":
      print 'This program is being run by itself'
else:
      print 'I am being imported from another module'
```


Beispiel

```
def ggt(a,b):
    while b != 0:
        a, b = b, a%b
    return a

def test_ggt():
    if ggt(100, 0) == 100:
        print "test1 passed"
    else:
        print "test1 failed"
    if ggt(43, 51) == 1:
        print "test2 passed"
    else:
        print "test2 failed"
    if ggt(10, 5) == 5:
        print "test3 passed"
    else:
        print "test3 failed"

if __name__ == "__main__":
    test_ggt()
```

Object-Oriented Analysis / Design (OOAD)

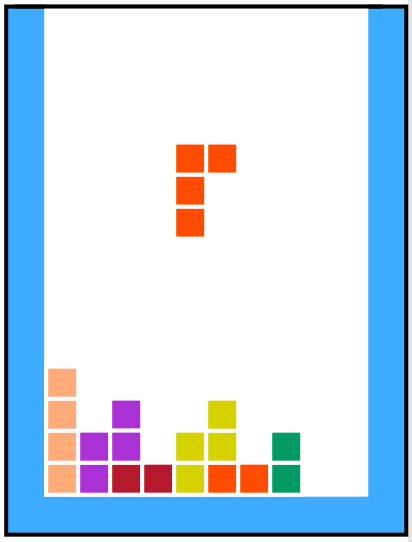
- Angemessene Weise, ein komplexes System zu modellieren
- Modelliere Software-System als Menge kooperierender Objekte
  - Programmverhalten bestimmt durch *Gruppenverhalten*
  - Entsteht aus Verhalten einzelner Objekte
- Objekte werden *antropomorph* betrachtet
  - Jedes hat gewisse "Intelligenz" (Auto kann selbst fahren, Tür kann sich selbst öffnen, ...)
  - Trigger dazu muß von außen kommen (→ Methodenaufruf)
- Jedes Objekt ist "black box":
  - Versteckt Details
  - Erleichtert die Entwicklung / Wartung eines komplexen Systems

Was ist ein Objekt?

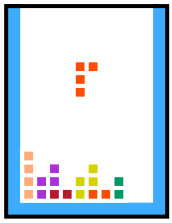
- Typische Kandidaten für Objekte:
  - Dinge: Stift, Buch, Mensch
  - Rollen: Autor, Leser, Benutzerhandbuch
  - Ereignisse: Fehler, Autopanne
  - Aktionen (manchmal!): Telefongespräch, Meeting
- Keine Objekte sind:
  - Algorithmen (z.B. Sortieren),
- Ein Objekt hat
  - eine Struktur (= interne "objekt-eigene" Variablen → Instanzvariablen)
  - einen Zustand (aktuelle Belegung der Instanzvariablen)
  - ein Verhalten / Fähigkeiten (→ Methoden)
  - eine Identität (= Nummer, Zeiger, ...)

Beispiel: Tetris

- Was sind die Objekte?
- Was müssen die Objekte können?
- Welche Eigenschaften haben die Objekte?



- Objekte:
  - Brett, Spielsteine
- Fähigkeiten:
  - Steine:
    - Erzeugt werden
    - Fallen
    - Rotieren
    - Stoppen
  - Brett:
    - Erzeugt werden
    - Zeilen löschen
    - Spielende feststellen



- Eigenschaften:
  - Steine:
    - Orientierung
    - Position
    - Form
    - Farbe
  - Brett:
    - Größe
    - Belegung der Zeilen

Definition von Klassen

- Allgemeine Form:

```
class name( object ):
    "documentation"
    Anweisungen
```

- Anweisungen sind i.A. Methodendeklarationen von der Form:

```
def name(self, arg1, arg2, ...):
        ...
```

  - erster Parameter jeder Methode ist eine Referenz auf die aktuelle Instanz der Klasse
  - Konvention: **`self`** nennen!
  - Ähnlich dem Keyword **`this`** in Java oder C++

Zugriff auf Instanzvariablen

- **`self`** muß man in der Methoden-Deklaration immer angeben, nicht aber im Aufruf:

```
def set_age(self, num):
```

```
x.set_age(23)
```

- Zugriff auf Instanzvariablen innerhalb einer Instanzmethode immer über **`self`**:

```
def set_age( self, num ):
    self.age = num
```

- Zugriff von außerhalb einer Instanzmethode geht über den Namen der Instanz selbst:

```
x.set_age( 23 )
x.age = 17
```

Data Hiding in Python

- Achtung: in Python gibt es keine Sprachkonstrukte, um *Data Hiding* zu implementieren!
- Etwas Analoges zu diesem gibt es nicht in Python:

```
class Name
{
public:
   int public_var;
private:
   float private_var;
};
```

```
class Name
{
public int public_var;
private float private_var;
};
```

- Kommentar dazu aus den Foren:

> Python culture tends towards "we're all consenting adults here". If you attempt to shoot yourself in the foot, you should get some kind of warning that perhaps it is not what you really want to do, but if you insist, hey, go ahead, it's your foot!

Erzeugung von Instanzen

- Syntax: `x = ClassName()`
- Beispiel:

```
class Vector2D:
    ...
x = Vector2D( 1, 2 )
z = x
```

- Ordentliche Initialisierung:
  - Die spezielle Methode `__init__` wird bei der Erzeugung einer Instanz aufgerufen (falls sie definiert wurde)
  - Dient (hauptsächlich) zur Initialisierung der Instanzvariablen
- Beispiel:

```
class Name:
    def __init__( self ):
        self.var = 0
```

- `__init__` heißt Konstruktor:
- Kann, wie jede andere Funktion, beliebig viele Parameter nehmen zur Initialisierung einer neuen Instanz
- Beispiel:

```
class Atom:
    def __init__( self, id, x,y,z ):
        self.id = id
        self.position = (x,y,z)
```

- Es gibt nur diesen einen Konstruktor!
  - (In C++ kann man viele deklarieren)
  - Keine wesentliche Einschränkung, da man ja Default-Argumente und Key/Value-Parameter hat

Beispiel: Atom class

```
class Atom:
    """A class representing an atom."""
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.position = (x,y,z)
    def __repr__(self):           # overloads printing
        return '%d %10.4f %10.4f %10.4f' %
               (self.atno, self.position[0],
                self.position[1],self.position[2])

>>> atom = Atom(6,0.0,1.0,2.0)
>>> print atom              # ruft __repr__ auf
6  0.0000  1.0000 2.0000
>>> atom.atno               # Zugriff auf ein Attribut
6
```

```
class Molecule:
    def __init__(self, name='Generic'):
        self.name = name
        self.atomlist = []

    def addatom(self,atom):
        self.atomlist.append(atom)

    def __repr__(self):
        str = 'Molecule named %s\n' % self.name
        str += 'Has %d atoms\n' % len(self.atomlist)
        for atom in self.atomlist:
            str += str(atom) + '\n'
            return str
```



```
>>> mol = Molecule('Water')
>>> at = Atom(8,0.,0.,0.)
>>> mol.addatom(at)
>>> mol.addatom( atom(1,0.0,0.0,1.0) )
>>> mol.addatom( atom(1,0.0,1.0,0.0) )
>>> print mol
Molecule named Water
Has 3 atoms
8   0.000 0.000 0.000
1   0.000 0.000 1.000
1   0.000 1.000 0.000
```

- Bemerkung: **`__repr__`** wird immer dann aufgerufen, wenn ein Objekt in einen lesbaren String umgewandelt werden soll (z.B. durch **`print`** oder **`str()`**)

Öffentliche (public) und private Daten

- Zur Zeit ist alles in **`Atom/Molecule`** öffentlich, so könnten wir etwas richtig Dummes machen wie

```
>>> at = Atom(6,0.0,0.0,0.0)
>>> at.position = 'Grape Jelly'
```

  dies würde jede Funktion, die **`at.position`** benutzt, abbrechen
- Aus diesem Grund sollten wir **`at.position`** schützen und Zugriffsmethoden auf dessen Daten bieten
  - *Encapsulation* oder *Data Hiding*
  - Zugriffsmethoden sind *"Getters"* und *"Setters"*
- Leider: in Python existiert (noch) kein schöner Mechanismus dafür!
  - Konvention: Instanzvariablen, deren Name mit 2 Underscore beginnt, sind privat; Bsp.: **`__a`** , **`__my_name`**
  - Üblich ist die Konvention: prinzipiell keinen direkten Zugriff von außen

Klassen, die wie Arrays und Listen aussehen

- Durch Überladen von **`__getitem__(self,index)`** damit die Klasse sich wie ein Array/Liste verhält, d.h., der Index-Operator def. ist:

```
class Molecule:
    def __getitem__(self,index):
        return self.atomlist[index]

>>> mol = Molecule('Water') # definiert wie vorhin
>>> for atom in mol:        # benutze wie eine Liste!
        print atom
>>> mol[0].translate(1.,1.,1.)
```

- Bestehende Operatoren in einer Klasse neu/anders zu definieren nennt man Überladen (*Overloading*)

Klassen, die wie Funktionen aussehen (Funktoren)

- Überladen von **`__call__(self,arg)`** damit sich die Klasse wie eine Funktion verhält, m.a.W., damit der ()-Operator für Instanzen definiert ist:

```
class gaussian:
    def __init__(self,exponent):
        self.exponent = exponent
    def __call__(self,arg):
        return math.exp(-self.exponent*arg*arg)

>>> func = gaussian(1.0)
>>> func(3.0)
0.0001234
```


Andere Dinge zum Überladen

- **`__setitem__(self,index,value)`**
  - Analogon zu **`__getitem__`** für Zuweisung der Form **`a[index] = value`**
- **`__add__(self,other)`**
  - Überlädt den "+" Operator: **`molecule = molecule + atom`**
- **`__mul__(self,number)`**
  - Überlädt den "*" Operator: **`molecule = molecule * 3`**
- **`__del__(self)`**
  - Überlädt den Standarddestruktor
  - Wird aufgerufen, wenn das Objekt nirgendwo im Programm mehr benötigt wird (keine Referenz darauf mehr exisitert)

Zwei Arten von Attributen

- Die Daten, die von einem Objekt gespeichert werden und keine Methoden sind, heißen Attribute. Es gibt zwei Arten:
  - Instanzattribute (= Instanzvariablen):
    Variable, die einer bestimmten Instanz einer Klasse gehört. Jede Instanz kann ihren eigenen Wert für diese Variable haben. Dies ist die gebräuchlichste Art von Attributen.
  - Class attributes (=Klassenvariablen):
    Gehört einer Klasse.
    Für alle Instanzen dieser Klasse hat dieses Attribut den gleichen Wert. In manchen Programmiersprachen als "*static*" bezeichnet.
    Nützlich für Konstanten oder als Counter für die Anzahl der Instanzen, die bereits erstellt wurden.

Klassenvariablen

- Bisher: Variablen waren Instanzvariablen, d.h., jede Instanz hat eigene "Kopie"
- Klassenvariable := Variablen, die innerhalb der Klasse genau 1x existieren
  - Alle Instanzen einer Klasse haben eine Referenz auf das gemeinsame Klassenattribut,
  - wenn eine Instanz es verändert, so wird der Wert für alle Instanzen verändert.
- In Python: definiere Klassenvariable außerhalb einer Methode
- Notation zum Zugriff: **`self.__class__.name`**

```
class sample:                       >>> a = sample()
    x = 23                          >>> a.increment()
    def increment(self):            >>> a.__class__.x
        self.__class__.x += 1       24
```

Introspektion

- Was tun, wenn Sie den Namen des Attributs oder der Methode einer Klasse nicht kennen (z.B., weil die Klassendefinition erst zur Laufzeit dazugeladen wurde), aber trotzdem auf das Element zur Laufzeit zugreifen wollen...
- Lösung: Introspektion
  - Introspektion := Methoden und Konstrukte in der Programmiersprache, um alle Attribute (Instanzvariablen und Methoden) einer Klasse aufzulisten und so zugänglich zu machen, daß man sie verwenden kann (z.B. die Methoden aufrufen kann)
  - Zu einer Zeichenkette, die den Namen des Attributs oder der Methode beinhaltet, eine Referenz zu bekommen (die man verwenden kann)
  - Theoretisch: man könnte sogar zur Laufzeit Methoden hinzufügen
- Im folgenden nur 2 (von vielen!) Möglicheiten der Introspektion in Python

getattr(object_instance, string)

```
>>> f = student("Bob Smith", 23)

>>> getattr(f, "full_name")
"Bob Smith"

>>> getattr(f, "get_age")
 <method get_age of class studentClass at 010B3C2>

>>> getattr(f, "get_age")()    #Das können wir aufrufen.
23

>>> getattr(f, "get_birthday")
      # Verursacht AttributeError - No method exists.
```

```
class student(object):
    def __init__( self,
                  name = "",
                  age = 0 ):
        self.name = name
        self.age = age
```

hasattr(object_instance, string)

```
>>> f = student("Bob Smith", 23)

>>> hasattr(f, "full_name")
True

>>> hasattr(f, "get_age")
True

>>> hasattr(f, "get_birthday")
False
```


Identitätsfindung

- Ein Objekt hat mehrere "Identitäten"
- Gleichheit mit anderen Objekten: `x == y`
  - Liefert True oder False (kann in der Klasse umdefiniert werden!)
  - Überprüft Inhalt (= Instanzvariable) auf Gleichheit
- Eindeutige ID: `id(x)`
  - Liefert eine eindeutige Zahl für jedes Objekt zu einem best. Zeitpunkt
- "*is a*"-Beziehung ("Instanz von" = Klassenzugehörigkeit)

```
isinstance( x, (ClassName1, ClassName2, …) )
```

  - Liefert True oder False (liefert True auch, falls x Instanz einer Unterklasse)

Anwendungsbeispiel

- Das Herausfinden der "is-a"-Identität dient z.B. dazu, verschiedene Aktionen innerhalb einer Methode durchzuführen, abhängig vom Typ des tatsächlichen Parameters

```
class MyClass( object ):
    def __init__( self, other ):
        if isinstance( other, MyClass ):
            ...                         # make a copy
        elsif isinstance( other, (int, float) ):
            inst_var = other            # create inst.var
        else:
            inst_var = 0                # default
```


Kein Bedarf für Freigaben

- Objekte braucht man nicht zu löschen oder freizugeben
- Python hat eine automatische *Garbage Collection* (Speicherbereinigung)
- Funktioniert mit *Reference Counting*
- Python ermittelt automatisch, wann alle Referenzen auf ein Objekt verschwunden sind und gibt dann diesen Speicherbereich frei
- Funktioniert im Allgemeinen gut, wenige memory leaks

Dokumentation

- Häufige Haltung: Source-Code sei die beste Dokumentation
  - "UTSL" ("use the source, luke")
- Aber:
  - ist viel zu umfangreich für einen schnellen Überblick
  - unterstützt das Navigieren zu gesuchten Informationen nur wenig
  - gibt keine Auskunft, welche Annahmen / Voraussetzungen einzelne Systemteile über das Verhalten anderer Teile machen (Schnittstellen)
  - gibt keine Auskunft, warum gerade *diese* Lösung gewählt wurde (warnt also nicht vor subtilen Fallen)
  - ...
- Deswegen: Korrekte (d.h. auch aktuelle) und hilfreiche Dokumentation ist extrem wichtig für die Entwicklung und Wartung eines Programms!

In Python integrierte Dokumentation

- In Python ist Doku fester Bestandteil der Sprache!
  - Geht noch weiter als in Java

```
def f():
    """
    blub
    bla
    """
    ...

help(f)
```

Ausgabe

```
Help on function f in module __main__:

f()
    blub
    bla
```

- Diese Kommentare heißen *Docstrings* in Python

Vollständiges *Boilerplate* für Docstrings

```
"""
Documentation for this module.

More details.
"""

def func():
    """ Documentation for a function.

        More details.
    """
    . . .

class MyClass:
    """ Documentation for a class.

        More details.
    """

    def __init__(self):
        """ Docu for the constructor. """
        . . .

    def method(self):
        """ Documentation for a method. """
        . . .
```


Automatische Generierung aus Source

- In allen Sprachen kann die Dokumentation zum großen Teil als Kommentar in den Source eingebettet werden
- Tools (wie z.B. Doxygen www.doxygen.org) erzeugen daraus automatisch sehr ansprechende und effiziente HTML-Seiten (oder LaTeX, oder ...)
  - Einzige Bedingung: Markup der Doku durch sog. *Tags*
- Vorteile:
  - Hoher Automatisierungsgrad (also Arbeitsersparnis)
  - durch enge Kopplung an Implementierung leichter aktuell zu halten als separate Dokumente
  - Sowohl interne (Developer-) als auch externe (API-) Doku kann aus demselben Source generiert werden

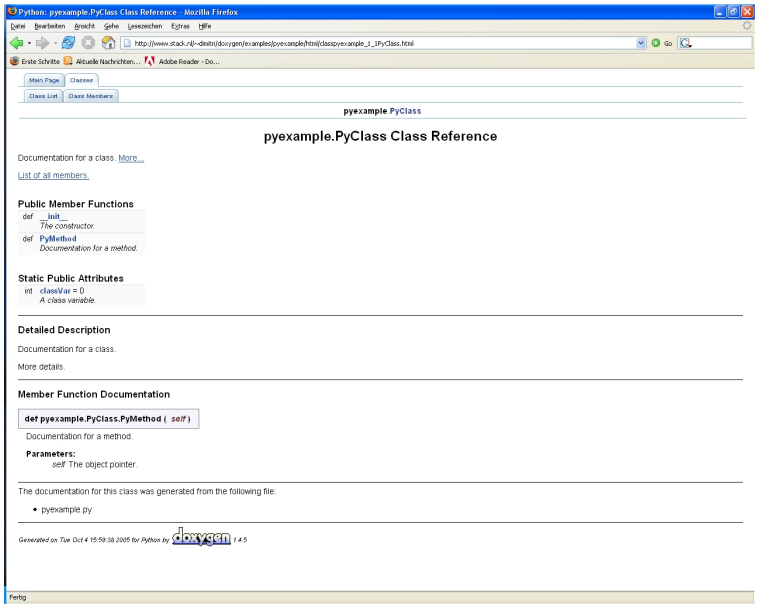
Doxygen-Boilerplate für die Dokumentation einer Funktion

```
def func( a ):
    """!  @brief Einzeilige Beschreibung

          @param param1       Beschreibung von param1
          @param param2       Beschreibung von param2

          @return
            Beschreibung des Return-Wertes.

          Detaillierte Beschreibung ...

          @pre
            Annahmnen, die die Funktion macht...
            Dinge, Aufrufer unbedingt beachten muss...

          @todo
            Was noch getan werden muss

          @bug
            Bekannte Bugs dieser Funktion

          @see
            andere Methoden / Klassen
     """
```


Dokumentation von Klassen, Moduln, ...

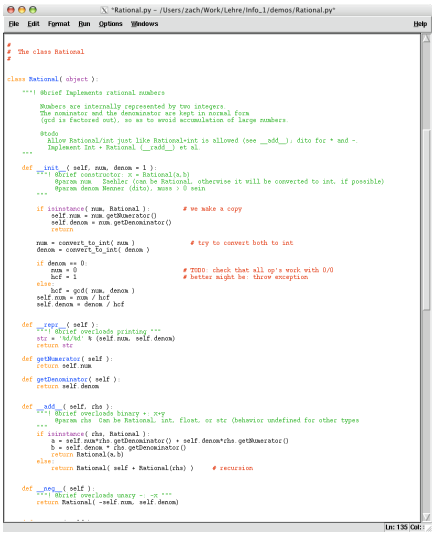
```
"""! @brief Documentation for a module.

     More details.
"""

def func():
    """! @brief Blub """
    . . .

class MyClass:
    """! @brief Class Blub …
    """

    def __init__(self):
        """! @brief the constructor """
        . . .

    def method( self, p1 ):
        """! @brief Documentation for a method
             @param p1   blubber
        """
        . . .

    ## A class variable.
    classVar = 0;

    ## @var _memVar
    #  a member variable
```

Doxygen-Dokumentation als HTML



Interaktives Beispiel: Rationale Zahlen

- Ziel: Neue Klasse (= Typ) **`Rational`** mit allen Operatoren



Unbehandelte Features

- Vererbung, Unterklassen,
- Exceptions
- Lambda-Funktionen
- Compilieren von Code zur Laufzeit durch das Programm selbst
- Unit-Testing (fest integriert in Python)
- ...