Freie Universität Berlin

Computerphysik WS 2015/2016

Einführung in die Programmierung mit Python und Jupyter

Alexander Schlaich

AG Netz, FU Berlin

14. Oktober 2015

Hinweise zu den Übungen

Freie Universität Berlin

- Die Übungsblätter werden Donnerstags auf der Vorlesungswebseite zum Download bereit gestellt.
- Die Aufgaben sollen in IPython Notebooks implementiert und sinnvoll kommentiert werden.
- Die Abgabe der Lösungen erfolgt als Notebook (*.ipynb). Name, Nummer des Übungsblattes und Tutor sollten im Dateinamen und im Notebook enthalten sein. Plots (auf Beschriftungen achten!) und Rechnungen sind möglichst in das Notebook einzubinden oder ggf. anzuhängen.
- Die Abgabe kopierter Lösungen führt zu null Punkten für alle Einsender.
- Die bearbeiteten Aufgaben müssen Sie im Tutorium präsentieren und erklären können. Andernfalls führt dies zum Verlust der Punkte für diese Aufgabe.
- Die Lösungen müssen bis Sonntag 24 Uhr der darauf folgenden Woche per Email an Ihren Tutor geschickt werden.

Übungsgruppen

Freie Universität Berlin

- Die Anwesenheit in den Übungsgruppen ist grundsätzlich erforderlich. Bei Abwesenheit werden die abgegebene Lösungen nicht berücksichtigt (krankheitsbedingtes Fehlen wird entschuldigt).
- Die Abgegebenen Lösungen messen stetig Ihren Lernfortschritt und bilden die Grundlage für die Benotung!
- In der ersten Übungswoche: Grundlagen zu Python (0. Übungsblatt, freiwillig)
- Zum Bestehen des Moduls sind mindestens 50% der Punkte nötig.
- Die Übungsgruppen finden zu folgenden Terminen im Rechnerraum (1.3.01) statt:
  - Di. 14:15-15:45 Dr. Markus Miettinen (englisch)
  - Do. 10:15-11:45 Alexander Schlaich
  - Do. 14:15-15:45 Julian Kappler
  - Fr. 10:15-11:45 Johann Hansing
- 14 Übungsblätter zu je 20 Punkten
  (280 Punkte $\rightarrow$ 140 Punkte zum Bestehen).

Einführung in die Programmierung mit Python

Basierend auf der Vorlesug "Computerphysik" von Axel Arnold
Beiträge von Christopher Mielack und Julius Schulz

Python

- schnell zu erlernende Programmiersprache
  – tut, was man erwartet
- objektorientierte Programmierung ist möglich
- viele Standardfunktionen („all batteries included“)
- breite Auswahl an Bibliotheken
- freie Software mit aktiver Gemeinde
- portabel, gibt es für fast jedes Betriebssystem
- entwickelt von Guido van Rossum, CWI, Amsterdam

Informationen zu Python

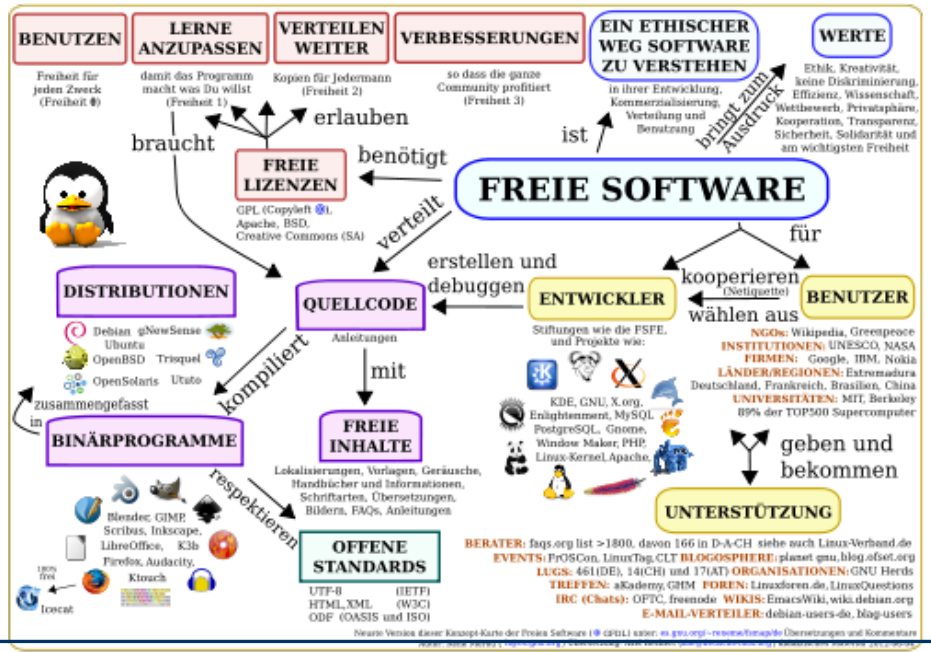
- aktuelle Versionen 3.5.0 bzw. 2.7.10
- mächtige numerische Bibliotheken verfügbar (NumPy)
- effizientes wissenschaftliches Arbeiten mit SciPy, MatPlotLib, etc.
- Komplettpaket: z.B. Anaconda `http://www.continuum.io/downloads` verfügbar für alle Betriebssysteme (Windows, Mac, Linux)

Hilfe zu Python

- offizielle Homepage
  `http://www.python.org`
- Einsteigerkurs „A Byte of Python“
  `http://swaroopch.com/notes/Python` (englisch)
  `http://abop-german.berlios.de` (deutsch)
- mit Programmiererfahrung „Dive into Python“
  `http://diveintopython.net`
- „A Primer on Scientific Programming with Python“
  von Hans Petter Langtangen (verfügbar über `SpringerLink`)

Was ist Python?

- **Skriptsprache**, d.h. keine Kompilierung
  (im Gegensatz zu Programmiersprachen wie z.B. C)
- Interaktiv über Python-Konsole
- Effiziente Routinen häufig auf C-Ebene implementiert (z.B. NumPy)
- **Ziel:** möglichst einfach und übersichtlich
- Skriptsprache u.a. für Blender, Cinema 4D, GIMP, OpenOffice.org, PyMOL
- Laufzeitkritische Skripts lassen sich relativ einfach in C übersetzen bzw. an bestehende Projekte in C anbinden, z.B. mittels cython



Python2 vs Python3

- Python 3.0 veröffentlicht 2008 (2.7 Mitte of 2010)
- Nur kleine Inkompabilitäten übrig (z.B. Unicode Strings in 3.x)
- `print` ist nun eine Funktion
- Per default keine Integerdivision (`2/3` vs `2//3`)

In Python 2 kann schon größtenteils Kompabilität erreicht werden:

```
from __future__ import print_function
from __future__ import division
```

Empfehlung für die Bearbeitung der Übungsaufgaben:

- Möglichst aktuelles Release von Python und allen Paketen (Distributionsverwaltung oder `pip`)
- Falls Sie Python 2 verwenden möchten verwenden Sie die beiden Funktionen aus `__future__`
- im Rechnerraum z.B. `python` → `python2.7` aber auch `python3` → `python3.4` vorhanden

Python starten

Aus der Shell:

```
> python3
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more...
>>> print ("Hello World")
Hello World
>>> help("print")
>>> exit()
```

- >>> markiert Eingaben
- `print()`: Ausgabe auf das Terminal
- `help()`: interaktive Hilfe, wird mit "q" beendet
- statt `exit()` reicht auch Control-d
- oder `ipython` (`ipython3`) mit Tab-Ergänzung, History usw.

Python-Skripte

Als Python-Skript helloworld.py:

```
#!/usr/bin/python3

# unsere erste Python-Anweisung
print ("Hello World")
```

- mit `python3 helloworld.py` starten
- oder ausführbar machen (`chmod a+x helloworld.py`)
- Umlaute vermeiden (v.a. Python2 und unter Windows)
- „ #! “ funktioniert genauso wie beim Shell-Skript (*Shebang*)
- Zeilen, die mit „ # “ starten, sind Kommentare

Kommentare sind wichtig,
um ein Programm verständlich zu machen!

- und nicht, um es zu verlängern!

IPython

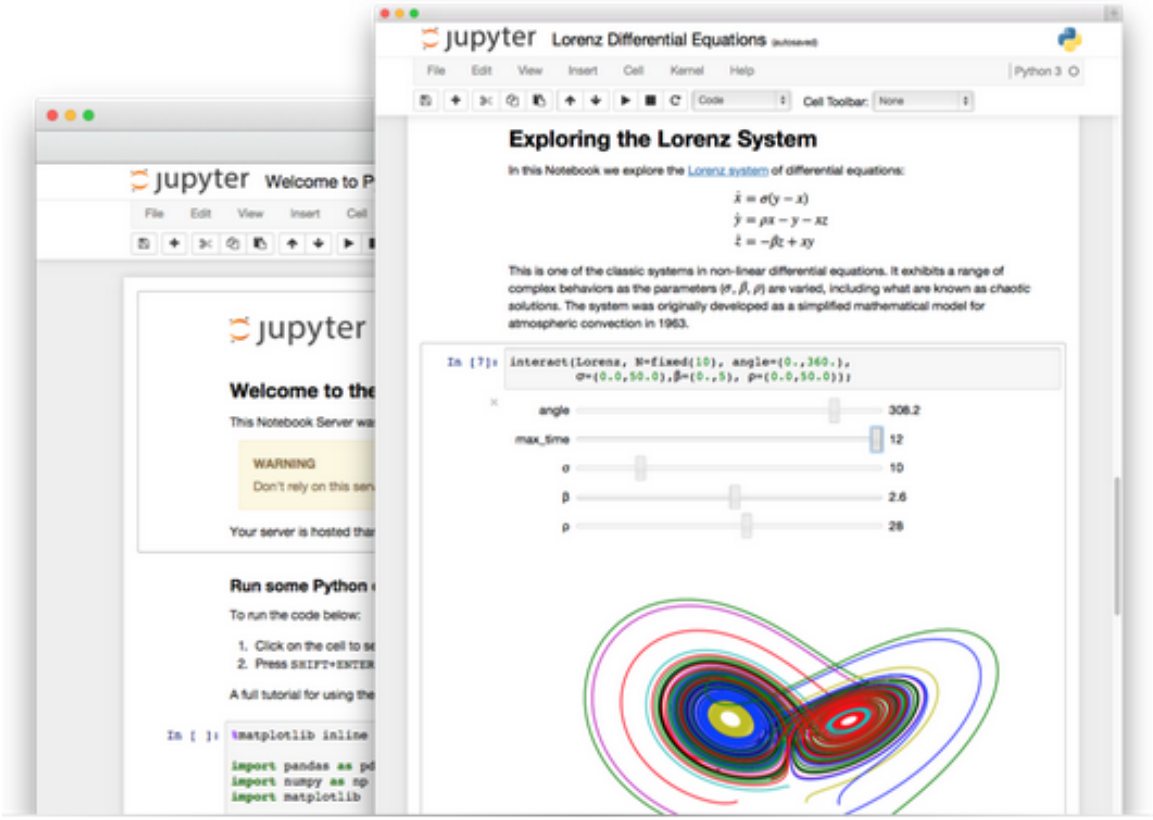
- interaktiver, komfortabler python prompt (z.B. von bash aus gestartet):

```
> ipython3
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
Type "copyright", "credits" or "license" for more information.

[...]

In [1]: print ("Hello World")
Hello World
```

- erkennt einige shell-Befehle, wie z.B. `"ls"`, `"cd"` , ...
- beherrscht Syntax Highlighting und Tab-Completion
- Hilfe zu jedem beliebigen Befehl/Objekt bekommt man durch Anhängen eines `"?"`, z.B. `"range?"`
- beenden mit Strg-D, y

IPython Notebook

jupyter



- Interaktive Nutzung und inline Darstellung mit Matplotlib magic `%matplotlib inline` (später)
- Textsatz mittels Markdown und Latex-Gleichungen → wissenschaftliche Dokumentation on-the-fly bis hin zum Paper-draft

IPython Notebook / Jupyter

- Notebook Server wird gestartet mit
  `ipython notebook` bzw. `jupyter notebook`
- ...und kann im Browser geöffnet werden unter `http://127.0.0.1:8888`
- legt `*.ipynb` Dateien in dem Verzeichnis ab von dem aus ihr den Server gestartet habt
- Code in einer Zelle kann mit Shift-Enter ausgeführt werden
- Speichern mit Strg-S, Hilfe mit h
- Output (inklusive Plots!) erscheint direkt unter eurem Code, bleibt im Notebook gespeichert
- Notebooks können exportiert werden, z.b. als HTML, PDF, ...mit

```
ipython nbconvert --to FORMAT notebook.ipynb
```

- *Wichtig:* Vor Abgabe der Hausaufgaben als *.ipynb Datei zum Testen Kernel ⟩ Restart ausführen und dann Cell ⟩ Run All

Datentypen 1

- ganze Zahlen:

```
>>> print (42)
42
>>> print (-12345)
-12345
```

- Fließkommazahlen:

```
>>> print (12345.000)
12345.0
>>> print (6.023e23)
6.023e+23
>>> print (13.8E-24)
1.38e-23
```

  - `1.38e-23` steht z. B. für 1.38×10^{-23}
  - `12345` $\neq$ `12345.0` (z. B. bei der Ausgabe)

Datentypen 2

- Zeichenketten (Strings)

```
>>> print ("Hello World")
Hello World
>>> print ('Hello World')
Hello World
>>> print ("""Hello
... World""")
Hello
World
```

  - zwischen einfachen (') oder doppelten (") Anführungszeichen
  - Über mehrere Zeilen mit dreifachen Anführungszeichen (auch als Kommentarfunktion verwendbar)
  - Leerzeichen sind normale Zeichen! `"Hello␣World"` $\neq$ `"Hello␣␣World"`
  - Zeichenketten sind keine Zahlen! `"1"` $\neq$ `1`

Variablen

```
>>> number1 = 1
>>> number2 = number1 + 5
>>> print (number1, number2)
1 6
>>> number2 = "Keine Zahl"
>>> print (number2)
Keine Zahl
```

- Variablennamen bestehen aus Buchstaben, Ziffern oder „ _ “ (Unterstrich)
- am Anfang keine Ziffer
- Groß-/Kleinschreibung ist relevant: Hase $\neq$ hase
- Richtig: `i`, `some_value`, `SomeValue`, `v123`, `_hidden`, `_1`
- **Falsch:** `1_value`, `some␣value`, `some-value`

Arithmetische Ausdrücke

+	Addition, bei Strings aneinanderfügen, z.B. 1 + 2 → 3, "a" + "b" → "ab"
-	Subtraktion, z.B. 1 - 2 → -1
*	Multiplikation, Strings vervielfältigen, z.B. 2 * 3 = 6, "ab" * 2 → "abab"
/	Division, z.B. 3 / 2 → 1.5, -3 / 2 → -1.5, 3.0 / 2 → 1.5 cave: Python 2 Ganzzahldivision (bei Python 3 mittels //): 3 / 2 → 1, -3 / 2 → -2, 3.0 / 2 → 1.5
%	Rest bei Division, z.B. 5 % 2 → 1
**	Exponent, z.B. 3**2 → 9, .1**3 → 0.001

- mathematische Präzedenz (Exponent vor Punkt vor Strich),
  z. B. `2**3 * 3 + 5` $\rightarrow 2^3 \cdot 3 + 5 = 29$
- Präzedenz kann durch runde Klammern geändert werden:
  `2**(3 * (3 + 5))` $\rightarrow 2^{3 \cdot 8} = 16\,777\,216$

Logische Ausdrücke

`==, !=`	Test auf (Un-)Gleichheit, z.B. `2 == 2` → `True`, `1 == 1.0` → `True`, `2 == 1` → `False`
`<,>,<=,>=`	Vergleich, z.B. `2 > 1` → `True`, `1 <= -1` → `False`
`or, and`	Logische Verknüpfungen „oder“ bzw. „und“
`not`	Logische Verneinung, z.B. `not False == True`

- Vergleiche liefern Wahrheitswerte: `True` oder `False`
- Wahrheitstabelle für die logische Verknüpfungen:

a	b	a `and` b	a `or` b
True	True	True	True
False	True	False	True
True	False	False	True
False	False	False	False

- Präzedenz: logische Verknüpfungen vor Vergleichen
- Beispiele: `3 > 2 and 5 < 7` → `True`, `1 < 1 or 2 >= 3` → `False`

if: bedingte Ausführung

```
>>> a = 1
>>> if a < 5:
...     print ("a ist kleiner als 5")
... elif a > 5:
...     print ("a ist groesser als 5")
... else:
...     print ("a ist 5")
a ist kleiner als 5
>>> if a < 5 and a > 5:
...     print ("Das kann nie passieren")
```

- `if-elif-else` führt den Block nach der ersten erfüllten Bedingung (logischer Wert `True`) aus
- Trifft keine Bedingung zu, wird der `else`-Block ausgeführt
- `elif` oder `else` sind optional

Blöcke und Einrückung

```
>>> a=5
>>> if a < 5:
...     # wenn a kleiner als 5 ist
...     b = -1
... else: b = 1
>>> # aber hier geht es immer weiter
... print (b)
1
```

- Alle *gleich eingerückten* Befehle gehören zum Block
- Nach dem `if`-Befehl geht es auf Einrückungsebene des `if` weiter, egal welcher `if`-Block ausgeführt wurde
- Einzeilige Blöcke können auch direkt hinter den Doppelpunkt
- Einrücken durch Leerzeichen (Achtung: immer gleich!)
  oder Tabulatoren (Vorsicht bei Dateiformaten und Editoren)

Blöcke und Einrückung 2

- ein Block kann nicht leer sein, aber der Befehl `pass` tut nichts:

```
if a < 5:
    pass
else:
    print ("a ist groesser gleich 5")
```

- `IndentationError` bei ungleichmäßiger Einrückung:

```
>>> print ("Hallo")
Hallo
>>>     print ("Hallo")
  File "<stdin>", line 1
    print ("Hallo")
    ^
IndentationError: unexpected indent
```

- Falsche Einrückung führt im allgemeinen zu Programmfehlern!

while: Schleifen

```
>>> a = 1
>>> while a < 5:
...     a = a + 1
>>> print (a)
5
```

- Führt den Block solange aus, wie die Bedingung wahr ist
- kann auch nicht ausgeführt werden:

```
>>> a = 6
>>> while a < 5:
...     a = a + 1
...     print ("erhoehe a um eins")
>>> print (a)
6
```

for: Sequenz-Schleifen

```
>>> for a in range(1, 3): print (a)
1
2
>>> b = 0
>>> for a in range(1, 100):
...     b = b + a
>>> print (b)
4950
>>> print (100 * (100 - 1) / 2)
4950
```

- `for` führt einen Block für jedes Element einer Sequenz aus
- Das aktuelle Element steht in `a`
- `range(k,l)` ist eine Liste der Zahlen a mit $k \leq a < l$
- Kann auch für eine Liste von Werten verwendet werden:
  `for i in [3, 5, -8, 95]:`

break und continue: Schleifen beenden

```
>>> for a in range(1, 10):
...     if a == 2 or a == 4 or a == 6: continue
...     elif a == 5: break
...     print (a)
1
3
>>> a = 1
>>> while True:
...     a = a + 1
...     if a > 5: break
>>> print (a)
6
```

- beide überspringen den Rest des Schleifenkörpers
- `break` bricht die Schleife ganz ab
- `continue` springt zum nächsten Schleifendurchlauf

Funktionen

```
>>> def printPi():
...     print ("pi ist ungefaehr 3.14159")
>>> printPi()
pi ist ungefaehr 3.14159

>>> def printMax(a, b):
...     if a > b: print (a)
...     else:     print (b)
>>> printMax(3, 2)
3
```

- eine Funktion kann beliebig viele Argumente haben
- Argumente sind Variablen der Funktion
- Beim Aufruf bekommen die Argumentvariablen Werte in der Reihenfolge der Definition
- Der Funktionskörper ist wieder ein Block

return: eine Funktion beenden

```
def printMax(a, b):
    if a > b:
      print (a)
      return
    print (b)
```

- `return` beendet die Funktion sofort

Rückgabewert

```
>>> def max(a, b):
...     if a > b: return a
...     else:     return b
>>> print (max(3, 2))
3
```

- eine Funktion kann einen Wert zurückliefern
- der Wert wird bei `return` spezifiziert

Lokale Variablen

```
>>> def max(a, b):
...     if a > b: maxVal=a
...     else:     maxVal=b
...     return maxVal
>>> print (max(3, 2))
3
>>> print (maxVal)
NameError: name 'maxVal' is not defined
```

- Variablen innerhalb einer Funktion sind *lokal*
- lokale Variablen existieren nur während der Funktionsausführung
- globale Variablen können aber gelesen werden

```
>>> faktor=2
>>> def strecken(a): return faktor*a
>>> print (strecken(1.5))
3.0
```

Vorgabewerte und Argumente benennen

```
>>> def lj(r, epsilon = 1.0, sigma = 1.0):
...     return 4*epsilon*( (sigma/r)**6 - (sigma/r)**12 )
>>> print (lj(2**(1./6.)))
1.0
>>> print (lj(2**(1./6.), 1, 1))
1.0
```

- Argumentvariablen können mit Standardwerten vorbelegt werden
- diese müssen dann beim Aufruf nicht angegeben werden

```
>>> print (lj(r = 1.0, sigma = 0.5))
0.0615234375
>>> print (lj(epsilon=1.0, sigma = 1.0, r = 2.0))
0.0615234375
```

- beim Aufruf können die Argumente auch explizit belegt werden
- dann ist die Reihenfolge egal

Funktionen als Werte

```
def printGrid(f, a, b, step):
    """
Gibt x, f(x) an den Punkten
x= a, a + step, a + 2*step, ..., b aus.
    """
    x = a
    while x < b:
        print (x, f(x))
        x = x + step

def test(x): return x*x

printGrid(test, 0, 1, 0.1)
```

- Funktionen ohne Argumentliste „(..)“ sind normale Werte
- Funktionen können in Variablen gespeichert werden
- ... oder als Argumente an andere Funktionen übergeben werden

Komplexe Datentypen

- Komplexe Datentypen sind zusammengesetzte Datentypen
- Beispiel: Eine Zeichenkette besteht aus beliebig vielen Zeichen
- die wichtigsten komplexen Datentypen in Python:
  - Strings (Zeichenketten)
  - Listen
  - Tupel
  - Dictionaries (Wörterbücher)
- diese können als Sequenzen in `for` eingesetzt werden:

```
>>> for x in "bla": print ("->", x)
-> b
-> l
-> a
```

Listen

```
>>> kaufen = [ "Muesli", "Milch", "Obst" ]
>>> kaufen[1] = "Sahne"
>>> print (kaufen[-1])
Obst
>>> kaufen.append(42)
>>> del kaufen[-1]
>>> print (kaufen)
['Muesli', 'Sahne', 'Obst']
```

- komma-getrennt in eckigen Klammmern
- können Daten *verschiedenen* Typs enthalten
- `liste[i]` bezeichnet das `i`-te Listenelement,
  negative Indizes starten vom Ende Achtung: Zählung beginnt bei 0!
- `liste.append()` fügt ein Element an eine Liste an
- `del` löscht ein Listenelement
- haben wir bereits in `for i in [3, 5, -8, 95]:` gesehen

Listen 2

```
>>> kaufen = kaufen + [ "Oel", "Mehl" ]
>>> print (kaufen)
['Muesli', 'Sahne', 'Obst', 'Oel', 'Mehl']
>>> for l in kaufen[1:3]:
...     print (l)
Sahne
Obst
>>> print (len(kaufen[:4]))
4
```

- „+“ fügt zwei Listen aneinander
- `[i:j+1]` ist die Subliste vom `i`-ten bis zum `j`-ten Element
- Leere Sublisten-Grenzen entsprechen Anfang bzw. Ende, also stets `liste == liste[:] == liste[0:]`
- `for`-Schleife über alle Elemente
- `len()` berechnet die Listenlänge

Formatierte Ausgabe: der %-Operator


```
>>> print ("Integer %d %05d" % (42, 42))
Integer 42 00042
>>> print ("Fliesskomma %e |%+8.4f| %g" % (3.14, 3.14, 3.14))
Fliesskomma 3.140000e+00 | +3.1400| 3.14
>>> print ("Strings %s %10s" % ("Hallo", "Welt"))
Strings Hallo       Welt
```

- Der %-Operator ersetzt %-Platzhalter in einem String
- %d: Ganzahlen (Integers)
- %e, %f, %g: Fliesskomma mit oder ohne Exponent oder wenn nötig (Standardformat)
- %s: einen String einfügen
- %x[defgs]: auf x Stellen mit Leerzeichen auffüllen
- %0x[defg]: mit Nullen auffüllen
- %$x.y$[efg]: x gesamt, y Nachkommastellen

Ein-/Ausgabe: Dateien in Python

```
eingabe = open("ein.txt")
ausgabe = open("aus.txt", "w")
nr = 0
ausgabe.write("Datei %s mit Zeilennummern\n" % eingabe.name)
for zeile in eingabe:
    nr += 1
    ausgabe.write("%d: %s" % (nr, zeile))
ausgabe.close()
```

- Dateien sind mit `open(datei, mode)` erzeugte Objekte
- Mögliche Modi (Wert von `mode`):

`r` oder leer	lesen
`w`	schreiben, Datei zuvor leeren
`a`	schreiben, an existierende Datei anhängen

- sind Sequenzen von Zeilen (wie eine Liste von Zeilen)
- Nur beim Schließen der Datei werden alle Daten geschrieben.
- Für Dateien mit numerischen Daten siehe `numpy.loadtxt` (später).

Module

```
>>> import random

>>> print (random.random(), random.randint(0, 100))
0.576899996137, 42

>>> from random import randint

>>> print (randint(0, 100))
14
```

- enthalten nützliche Funktionen, Klassen, usw.
- sind nach Funktionalität zusammengefasst
- werden per `import` zugänglich gemacht
- Hilfe: `help(modul)`, alle Funktionen: `dir(modul)`
- einzelne Bestandteile kann man mit `from ... import ...` importieren – bei Namensgleichheit kann es zu Kollisionen kommen!
- Abkürzungen: `import matplotlib.pyplot as plt`

Das math-Modul

```
import math
import random
def boxmueller():
    """
liefert normalverteilte Zufallszahlen
nach dem Box-Mueller-Verfahren
    """
    r1, r2 = random.random(), random.random()
    return math.sqrt(-2*math.log(r1))*math.cos(2*math.pi*r2)
```

- `math` stellt viele mathematische Grundfunktionen zur Verfügung, z.B. floor/ceil, exp/log, sin/cos, pi
- `random` erzeugt *pseudo*zufällige Zahlen
  - `random()`: gleichverteilt in $[0, 1)$
  - `randint(a, b)`: gleichverteilt ganze Zahlen in $[a, b)$
  - `gauss(m, s)`: normalverteilt mit Mittelwert m und Varianz s

Numerik mit Python – numpy

- `numpy` ist ein Modul für effiziente numerische Rechnungen
- Baut auf n-dimensionalem Feld-Datentyp `numpy.array` auf
  - Feste Größe und Form
  - Alle Elemente vom selben (einfachen) Datentyp
  - Aber sehr schneller Zugriff (C-Niveau)
  - Viele Transformationen
- Bietet
  - mathematische Grundoperationen
  - Sortieren, Auswahl von Spalten, Zeilen usw.
  - Eigenwerte, -vektoren, Diagonalisierung
  - diskrete Fouriertransformation
  - statistische Auswertung
  - Zufallsgeneratoren
- Hilfe unter `http://docs.scipy.org`

`array` – eindimensionale Arrays

```
>>> import numpy as np
>>> np.array([1.0, 2, 3])
array([ 1.,  2.,  3.])
>>> np.ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
>>> np.arange(2.2, 3, 0.2, dtype=float)
array([ 2.2,  2.4,  2.6,  2.8])
```

- `np.array` erzeugt ein Array (Feld) aus einer Liste
- `np.arange` entspricht `range` für beliebige Datentypen
- `np.zeros` / `ones` erzeugen 1er/0er-Arrays
- `dtype` setzt den Datentyp *aller* Elemente explizit

`bool`	Wahrheitswerte
`int`	ganze Zahlen

`float`	IEEE-Fließkommazahlen
`complex`	Komplexe Fließkommazahlen

- ansonsten der einfachste für alle Elemente passende Typ

Mehrdimensionale Arrays

```
>>> np.array([[1, 2, 3], [4, 5, 6]])
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.array([[[1,2,3], [4,5,6]], [[7,8,9], [0,1,2]]])
array([[[1, 2, 3],
        [4, 5, 6]],

       [[7, 8, 9],
        [0, 1, 2]]])
>>> np.zeros((2, 2))
array([[ 0.,  0.],
       [ 0.,  0.]])
```

- Mehrdimensionale Arrays entsprechen verschachtelten Listen
- Alle Zeilen müssen die gleiche Länge haben
- `np.zeros` / `ones`: Größe als Tupel von Dimensionen
  (Tupel: unveränderbare Listen, runde Klammern)

Elementzugriff und Subarrays

```
>>> a = np.array([[1,2,3,4,5,6], [7,8,9,0,1,2]])
>>> print (a.shape, a[1,2], a[1])
(2, 6) 9 [7 8 9 0 1 2]
>>> print (a[0,1::2])
[2, 4, 6]
>>> print (a[1:,1:])
[[8, 9, 0, 1, 2]]
>>> print (a[0, np.array([1,2,5])])
[2, 3, 6]
```

- `[]` indiziert Elemente und Zeilen usw.
- Auch Bereiche wie bei Listen
- `a.shape` ist die aktuelle Form (Länge der Dimensionen)
- `int`-Arrays, um beliebige Elemente zu selektieren

Methoden: Matrixoperationen

```
>>> a = np.array([[1,2], [3,4]])
>>> a = np.concatenate((a, [[5,6]]), axis=0)
>>> print (a.transpose())
[[1 3 5]
 [2 4 6]]
 >>> print (a.shape, a.transpose().shape)
(3, 2) (2, 3)
>>> print (a[1].reshape((2,1)))
[[3]
 [4]]
```

- `reshape()` kann die Form eines Arrays ändern, solange die Gesamtanzahl an Element gleich bleibt
- `concatenate()` hängt zwei Matrizen aneinander, `axis` bestimmt die Dimension, entlang der angefügt wird
- `transpose()`, `conjugate()`: Transponierte, Konjugierte
- `min()`, `max()` berechnen Minimum und Maximum aller Elemente

Lineare Algebra: Vektoren und Matrizen

```
>>> a = np.array([[1,2],[3,4]])
>>> i = np.array([[1,0],[0,1]]) # Einheitsmatrix
>>> print (a*i)                 # punktweises Produkt
[[1 0]
 [0 4]]
>>> print (np.dot(a,i))         # echtes Matrixprodukt
[[1 2]
 [3 4]]
>>> print (np.dot(a[0], a[1]))  # Skalarprodukt der Zeilen
11
```

- Arrays werden normalerweise *punktweise* multipliziert
- `np.dot` entspricht
  - bei zwei eindimensionalen Arrays dem Vektor-Skalarprodukt
  - bei zwei zweidimensionalen Arrays der Matrix-Multiplikation
  - bei ein- und zweidim. Arrays der Vektor-Matrix-Multiplikation
- Die Dimensionen müssen passen

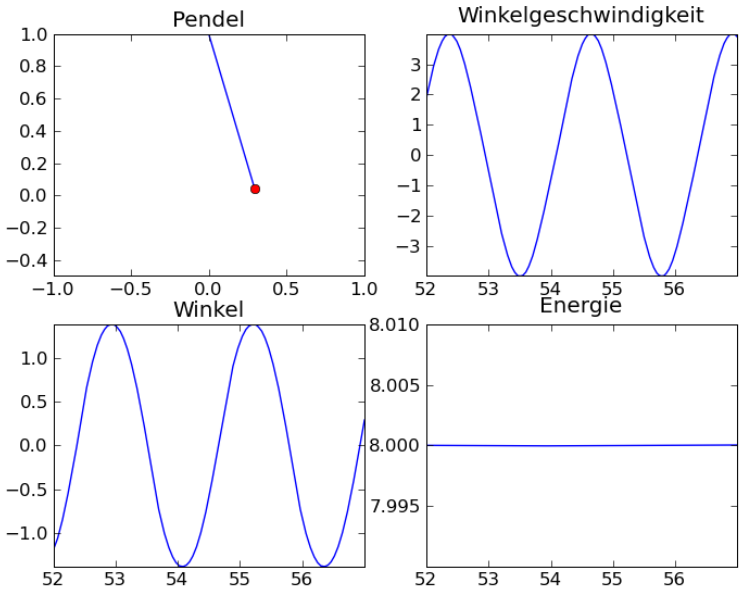
Lineare Algebra

```
>>> a = np.array([[1,0],[0,1]])
>>> print (np.linalg.det(a))
1
>>> print (np.linalg.eig(a))
(array([ 1.,  1.]), array([[ 1.,  0.],
       [ 0.,  1.]]))
```

- `numpy.cross`: Vektorkreuzprodukt
- `numpy.linalg.det`, `.trace`: Determinante und Spur
- `numpy.linalg.norm`, `.cond`: Norm und Kondition
- `numpy.linalg.eig`: Eigenwerte und -vektoren
- `numpy.linalg.inv`: Inverse einer Matrix berechnen
- `numpy.linalg.cholesky`, `.qr`, `.svd`: Matrixzerlegungen
- `numpy.linalg.solve(A, b)`: Lösen von $Ax = b$

Analyse und Visualisierung

Zeit	Winkel	Geschw.	Energie
55.0	1.1605	2.0509	8.000015
55.2	1.3839	0.1625	8.000017
55.4	1.2245	-1.7434	8.000016
55.6	0.7040	-3.3668	8.000008
55.8	-0.0556	-3.9962	8.000000
56.0	-0.7951	-3.1810	8.000009
56.2	-1.2694	-1.4849	8.000016
56.4	-1.3756	0.43024	8.000017
56.6	-1.1001	2.29749	8.000014
56.8	-0.4860	3.70518	8.000004



- Zahlen anschauen ist langweilig!
- Graphen sind besser geeignet
- Statistik hilft, Ergebnisse einzuschätzen (Fehlerbalken)
- Histogramme, Durchschnitt, Varianz

Durchschnitt und Varianz

```
>>> samples=100000
>>> z = np.random.normal(0, 2, samples)

>>> print (np.mean(z))
-0.00123299611634
>>> print (np.var(z))
4.03344753342
```

- Arithmetischer Durchschnitt

$$\langle z \rangle = \sum_{i=1}^{\mathsf{len}(z)} z_i / \mathsf{len}(z)$$

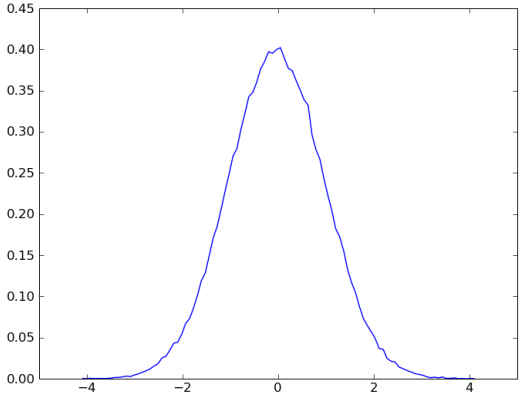
- Varianz

$$\sigma(z) = \left\langle (z - \langle z \rangle)^2 \right\rangle$$

Ein-/Ausgabe: numerische Daten

- Möchte man Dateien mit numerischen Daten einlesen, könnte man diese mit `open` laden und dann *parsen*, d.h. Strings in numerische Daten umwandeln.
- Einen einfacheren Weg bietet `numpy.loadtxt(file)` bzw. `numpy.savetxt(file, array)`
- Dabei werden die Zeilen und Spalten der Datei als zwei-dimensionales `numpy.array` gespeichert
- Die einzelnen Einträge müssen ein numerisches Format haben (“.” als Dezimaltrenner)
- Zeilen die mit # beginnen, werden ignoriert.
- Viele Optionen zum Anpassen des Dateiformats.

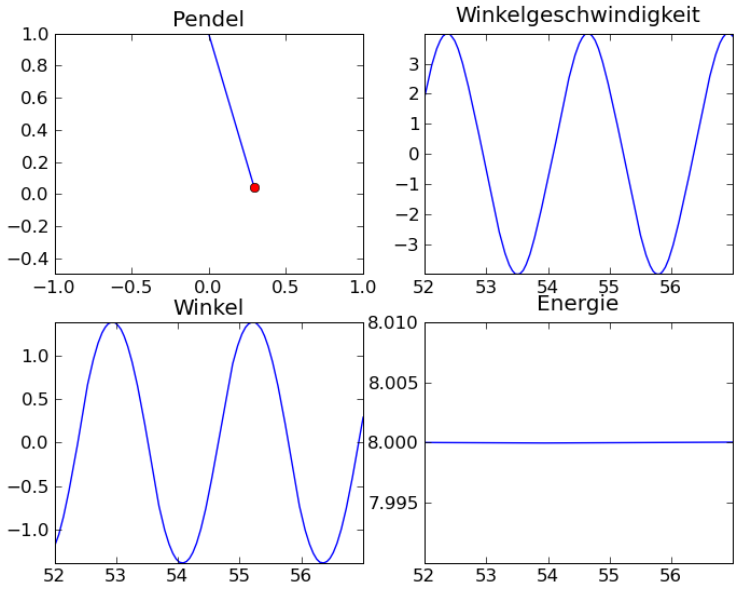
Histogramme




```
>>> zz = np.random.normal(0,1,100000)
>>> werte, rand = np.histogram(zz, bins=100, normed=True)
```

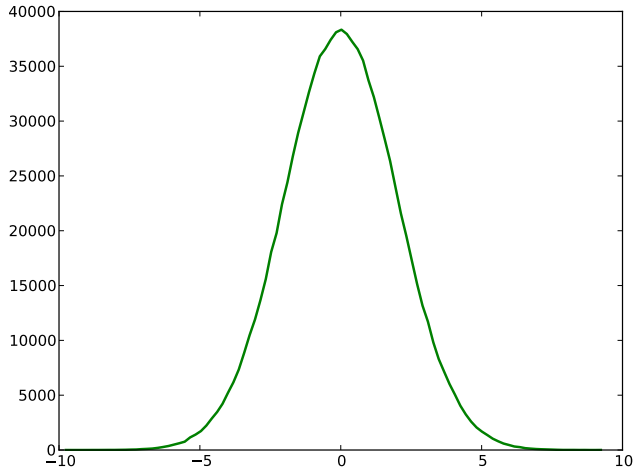
- Histogramme geben die Häufigkeit von Werten an
- In `bins` vielen gleich breiten Intervallen
- `werte` sind die Häufigkeiten, `raender` die Grenzen der Intervalle (ein Wert mehr als in `werte`)

Malen nach Zahlen – matplotlib



2D-Plots

```
import matplotlib
import matplotlib.pyplot as plt
...
x = np.arange(0, 2*np.pi, 0.01)
y = np.sin(x)
plt.plot(x, y, "g", linewidth=2)
plt.text(1, -0.5, "sin(2*pi*x)")
plt.show()
```

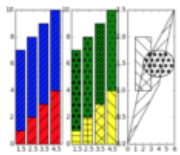


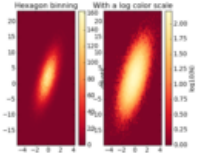
- `pyplot.plot` erzeugt einen 2D-Graphen
- `pyplot.text` schreibt beliebigen Text in den Graphen
- `pyplot.show()` zeigt den Graphen an
- Parametrische Plots mit Punkten (`x[t]`, `y[t]`)
- für Funktionen Punkte (`x[t]`, `y(x[t])`) mit `x` Bereich
- Farbe und Form über String und Parameter – ausprobieren

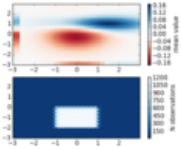
Matplotlib Übersicht

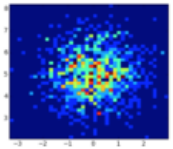
- 3D-Plots
- Scatter plots
- Colormaps
- Mehrfach-Plots
- Polar-Koordinaten
- LaTeX-Labels
- Grafische Übersicht mit Code-Beispielen auf

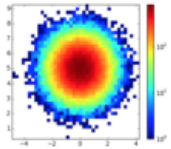
  `http://matplotlib.org/gallery.html`

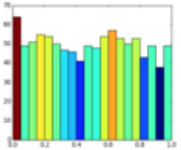

hatch_demo

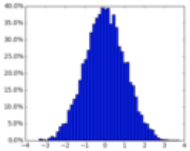

hexbin_demo

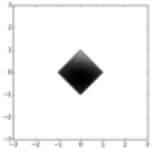

hexbin_demo2

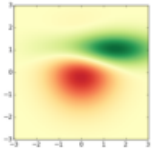

hist2d_demo

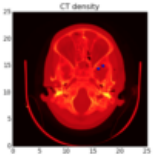

hist2d_log_demo

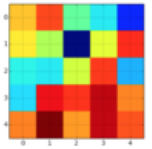

hist_colormapped

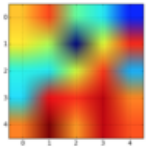

histogram_percent_demo

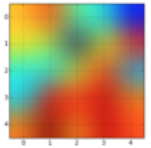

image_clip_path

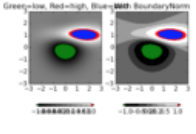

image_demo

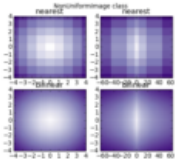

image_demo2


image_interp


image_interp


image_interp


image_masked


image_nonuniform

Numpy.Vectorize

- Häufig (beispielsweise für Plots) müssen Werte als `numpy.array` vorliegen. Will man Funktionen plotten, müssen sie ein solches zurückgeben.
- Solange man nur Standardoperatoren und Funktionen aus numpy verwendet, arbeiten Funktionen problemlos auf Arrays
- Folgendes Beispiel funktioniert nicht ohne Probleme:

```
def func(a, b):
    if a > b:
        return a - b
    else:
        return a + b
```

- In diesem Fall biete die Funktion `numpy.vectorize` die Möglichkeit, diese Funktion mit numpy arrays kompatibel zu machen:

```
g = vectorize(func)
p.plot(x, g(x,a))
```

Aussagekräftige Plots enthalten immer sinnvolle Achsenbeschriftungen, Titel und Legenden!

weitergehende Themen

Die hier behandelten Themen sind für die ersten Übungen nicht relevant, zeigen aber einige eindrucksvolle Einsatzmöglichkeiten für Python, in der zur Verfügung stehend Zeit aber nicht behandelt werden können.

Außerdem werden typische Stolpersteine angesprochen und die Herangehensweise an Probleme erläutert, weshalb wir allen Übungsteilnehmern die Lektüre sehr ans Herz legen.

Methodik des Programmierens

Schritte bei der Entwicklung eines Programms

- Problemanalyse
  - Was soll das Programm leisten?
    Z.B. eine Nullstelle finden, Molekulardynamik simulieren
  - Was sind Nebenbedingungen?
    Z.B. ist die Funktion reellwertig? Wieviele Atome?
- Methodenwahl
  - Schrittweises Zerlegen in Teilprobleme (Top-Down-Analyse)
    Z.B. Propagation, Kraftberechnung, Ausgabe
  - Wahl von Datentypen und -strukturen
    Z.B. Listen oder Tupel? Wörterbuch?
  - Wahl der Rechenstrukturen (Algorithmen)
    Z.B. Newton-Verfahren, Regula falsi,...
- Implementation und Dokumentation
  - Programmieren und *gleichzeitig* dokumentieren
  - Kommentare und externe Dokumentation (z.B. Formeln)

Methodik des Programmierens

- Testen auf Korrektheit
  - Funktioniert das Programm?
    Z.B. findet es eine bekannte Lösung?
  - Terminiert das Programm?
    D.h. hält es immer an?
- Testen auf Effizienz
  - Wie lange braucht das Programm bei beliebigen Eingaben?
  - Wieviel Speicher braucht es?
- Meistens wiederholt sich der Prozess:
  - Bei der Methodenwahl stellen sich weitere Einschränkungen als nötig heraus: *z.B. Funktion darf keine Singularitäten aufweisen*
  - Bei der Implementation zeigt sich, dass die gewählte Methode nicht umzusetzen ist: *z.B. weil implizite Gleichungen gelöst werden müssten*
  - Beim Testen zeigt sich, dass die Methode ungeeignet oder nicht schnell genug ist: *z.B. zu langsam, numerisch instabil*
- Mit wachsender Projektgröße zunehmend wichtig: *Software Engineering*

Allgemeine Programmier Empfehlungen

- Redundanz vermeiden
  - Tut eine Library vielleicht schon das, was ich implementieren will?
  - wiederkehrenden Code als Funktion auslagern
- Sinnvolle Benennung von Variablen und Funktionen ist besser als Kommentare

  gut: `calculateVelocity (...)`

  schlecht: `calc(...): # berechnet Geschwindigkeit`

  gut: `numberOfParticles = 10`

  schlecht: `n = 10 # Anzahl der Teilchen`
- Funktionsargumente verwenden statt globaler Variablen

```
def createDuplicates( inputList, numberOfCopies):
    return inputList * numberOfCopies
```

... ist besser als ...

```
numberOfCopies = 10
inputList = [1,2,3]
def createDuplicates():
    return inputList*numberOfCopies
```

Testen auf Korrektheit und Effizienz

Korrektheit

- Extremwerte überprüfen (leere Eingabe, 0)
- *Generische* Fälle prüfen, d.h. alle Vorzeichen, bei reellen Zahlen nicht nur ganzzahlige Eingaben
- Alle Fehlermeldungen sollten getestet, also ausgelöst werden!
  $\Rightarrow$ mehr Tests für unzulässige Eingaben als für korrekte

Effizienz

- Wie viele elementare Schritte (Schleifendurchläufe) sind nötig?
- Möglichst langsam wachsende Anzahl elementarer Schritte
- Ebenso möglichst langsam wachsender Speicherbedarf
- Sonst können nur sehr kleine Aufgaben behandelt werden
- Beispiel: bei $N = 10^6$ ist selbst $0,1\mu s \times N^2 = 1$ Tag

Beispiel: Pythagoreische Zahlentripel

- Problemanalyse

  Gegeben: eine ganze Zahl c

  Gesucht: Zahlen a, b mit $a^2 + b^2 = c^2$

  1. Verfeinerung: $a = 0$, $b = c$ geht immer $\Rightarrow a, b > 0$

  2. Verfeinerung: Was, wenn es keine Lösung gibt? Fehlermeldung

- Methodenwahl
  - Es muss gelten: $0 < a < c$ und $0 < b < c$
  - Wir können also alle Paare a, b mit $0 < a < c$ und $0 < b < c$ durchprobieren – verschachtelte Schleifen
  - Unterteilung in Teilprobleme nicht sinnvoll

- Effizienz?

  Rechenzeit wächst wie $|c|^2$ – das ist langsam!

Beispiel: Pythagoreische Zahlentripel

- Implementation

```
def zahlentripel(c):
    """
Liefert ein Ganzzahlpaar (a, b), dass a^2 + b^2 = c^2
erfuellt, oder None, wenn keine Loesung existiert.
    """
    # Durchprobieren aller Paare
    for a in range(1,c):
        for b in range(1,c):
            if a**2 + b**2 == c**2: return (a, b)
    return None
```

- Test der Effizienz

Zahl (alle ohne Lösung)	1236	12343	123456
Zeit	0.5s	41s	1:20h

- Das ist tatsächlich sehr langsam! Geht es besser?

Beispiel: Pythagoreische Zahlentripel

- Methodenwahl zur Effizienzverbesserung
  O.B.d.A. $a \leq b$
  - Sei zunächst $a = 1$ und $b = c - 1$
  - Ist $a^2 + b^2 > c^2$, so müssen wir b verringern,
    und wir wissen, dass es keine Lösung mit $b = c - 1$ gibt
  - Ist $a^2 + b^2 < c^2$, so müssen wir a erhöhen und wir wissen, dass es keine Lösung mit $a = 1$ gibt
  - Ist nun $a^2 + b^2 > c^2$, so müssen wir wieder b verringern,
    egal ob wir zuvor a erhöht oder b verringert haben
  - Wir haben alle Möglichkeiten getestet, wenn $a > b$
- Effizienz?
  Wir verringern oder erhöhen a bzw. b in jedem Schritt. Daher sind es nur maximal $|c|/2$ viele Schritte.

Beispiel: Pythagoreische Zahlentripel

- Implementation der effizienten Vorgehensweise

```
def zahlentripel(c):
    "loest a^2 + b^2 = c^2 oder liefert None zurueck"
    # Einschachteln der moeglichen Loesung
    a = 1
    b = c - 1
    while a <= b:
        if   a**2 + b**2 < c**2: a += 1
        elif a**2 + b**2 > c**2: b -= 1
        else: return (a, b)
    return None
```

- Demonstration der Effizienz

Zahl	12343	123456	1234561	12345676	123456789
Zeit	0.01s	0.08s	0.63s	6.1s	62s

Dokumentation von Funktionen

```
def max(a, b):
    "Gibt das Maximum von a und b zurueck."
    if a > b: return a
    else:     return b

def min(a, b):
    """
Gibt das Minimum von a und b zurueck. Funktioniert
ansonsten genau wie die Funktion max.
    """
    if a < b: return a
    else:     return b
```

- Dokumentation optionale Zeichenkette vor dem Funktionskörper
- wird bei `help(funktion)` ausgegeben

Rekursion

```
def fakultaet(n):
    # stellt sicher, das die Rekursion irgendwann stoppt
    if n <= 1:
        return 1
    # n! = n * (n-1)!
    return n * fakultaet(n-1)
```

- Funktionen können andere Funktionen aufrufen, insbesondere sich selber
- Eine Funktion, die sich selber aufruft, heißt rekursiv
- Rekursionen treten in der Mathematik häufig auf
- sind aber oft nicht einfach zu verstehen
- Ob eine Rekursion endet, ist nicht immer offensichtlich
- Jeder rekursive Algorithmus kann auch iterativ als verschachtelte Schleifen formuliert werden

Shallow copies

Shallow copy:

```
>>> bezahlen = kaufen
>>> del kaufen[2:]
>>> print (bezahlen)
['Muesli', 'Sahne']
```

Subliste, deep copy:

```
>>> merken = kaufen[1:]
>>> del kaufen[2:]
>>> print (merken)
['Sahne', 'Obst', 'Oel', 'Mehl']
```

„=“ macht in Python flache Kopien komplexer Datentypen!

- Flache Kopien (shallow copies) verweisen auf dieselben Daten
- Änderungen an einer flachen Kopie betreffen auch das Original
- Sublisten sind echte Kopien
- daher ist `l[:]` eine echte Kopie von `l`

Shallow copies 2 – Listenelemente

```
>>> elementliste=[]
>>> liste = [ elementliste, elementliste ]
>>> liste[0].append("Hallo")
>>> print (liste)
[['Hallo'], ['Hallo']]
```

Mit echten Kopien (deep copies)

```
>>> liste = [ elementliste[:], elementliste[:] ]
>>> liste[0].append("Welt")
>>> print (liste)
[['Hallo', 'Welt'], ['Hallo']]
```

- komplexe Listenelemente sind flache Kopien
  und können daher mehrmals auf dieselben Daten verweisen
- kann zu unerwarteten Ergebnissen führen

Tupel: unveränderbare Listen

```
>>> kaufen = ("Muesli", "Kaese", "Milch")
>>> print (kaufen[1])
Kaese
>>> for f in kaufen[:2]: print (f)
Muesli
Kaese
>>> kaufen[1] = "Camembert"
TypeError: 'tuple' object does not support item assignment
>>> print (k + k)
('Muesli', 'Kaese', 'Milch', 'Muesli', 'Kaese', 'Milch')
```

- komma-getrennt in runden Klammern
- können nicht verändert werden
- ansonsten wie Listen einsetzbar
- Strings sind Tupel von Zeichen

Dictionaries

```
>>> de_en = { "Milch": "milk", "Mehl": "flour" }
>>> print (de_en)
{'Mehl': 'flour', 'Milch': 'milk'}
>>> de_en["Oel"]="oil"
>>> for de in de_en: print (de, "=>", de_en[de])
Mehl => flour
Milch => milk
Oel => oil
>>> for de, en in de_en.iteritems(): (print de, "=>", en)
>>> if "Mehl" in de_en: (print "Kann \"Mehl\" uebersetzen")
```

- komma-getrennt in geschweiften Klammern
- speichert Paare von Schlüsseln (Keys) und Werten
- Speicher-Reihenfolge der Werte ist nicht festgelegt
- daher Indizierung über die Keys, nicht Listenindex o.ä.
- mit `in` kann nach Schlüsseln gesucht werden

Objekte in Python


```
>>> original = list()
>>> original.append(3)
>>> original.append(2)

>>> kopie = list(original)
```

```
>>> original.append(1)
>>> original.sort()

>>> print (original, kopie)
[1, 2, 3], [3, 2]
```

- In Python können komplexe Datentypen wie Objekte im Sinne der objekt-orientierten Programmierung verwendet werden
- Datentypen entsprechen Klassen (hier `list`)
- Variablen entsprechen Objekten (hier `original` und `kopie`)
- Objekte werden durch durch Aufruf von Klasse() erzeugt
- Methoden eines Objekts werden in der Form Objekt.Methode() aufgerufen (hier `list.append()` und `list.sort()`)
- `help`(Klasse/Datentyp) informiert über vorhandene Methoden
- Per `class` kann man selber Klassen erstellen

Stringmethoden

- Zeichenkette in Zeichenkette suchen

  `"Hallo Welt".find("Welt")` → `6`

  `"Hallo Welt".find("Mond")` → `-1`

- Zeichenkette in Zeichenkette ersetzen

  `" abcdabcabe ".replace("abc", "123")` → `'123 d123abe '`

- Groß-/Kleinschreibung ändern

  `"hallo".capitalize()` → `'Hallo'`

  `"Hallo Welt".upper()` → `'HALLO WELT'`

  `"Hallo Welt".lower()` → `'hallo welt'`

- in eine Liste zerlegen

  `"1, 2, 3, 4".split(",")` → `['1', ' 2', ' 3', ' 4']`

- zuschneiden

  `"  Hallo ".strip()` → `'Hallo'`

  `"..Hallo..".lstrip(".")` → `'Hallo..'`

Fehlermeldungen: raise

```
def loesungen_der_quad_gln(a, b, c):
    "loest a x^2 + b x + c = 0"
    det = (0.5*b/a)**2 - c
    if det < 0: raise Exception("Es gibt keine Loesung!")
    return (-0.5*b/a + det**0.5, -0.5*b/a - det**0.5)

try:
    loesungen_der_quad_gln(1,0,1)
except:
    print ("es gibt keine Loesung, versuch was anderes!")
```

- `raise Exception("Meldung")` wirft eine Ausnahme (Exception)
- Funktionen werden nacheinander an der aktuellen Stelle beendet
- mit `try: ... except: ...` lassen sich Fehler abfangen, dann geht es im `except`-Block weiter

Das sys-Modul

stellt Informationen über Python und das laufende Programm selber zur Verfügung.

- `sys.argv`: Kommandozeilenparameter,
  `sys.argv[0]` ist der Programmname
- `sys.path`: Liste der Verzeichnisse, in denen Python Module sucht
- `sys.exit("Fehlermeldung")`: bricht das Programm ab
- `sys.stdin`,
  `sys.stdout`,
  `sys.stderr`: Dateiobjekte für Standard-Ein-/Ausgabe

```
zeile = sys.stdin.readline()
sys.stdout.write("gerade eingegeben: %s" % zeile)
sys.stderr.write("Meldung auf der Fehlerausgabe")
```

Parameter: das optparse-Modul

```
from optparse import OptionParser

parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="Ausgabe in FILE", metavar="FILE")
parser.add_option("-q", "--quiet", dest="verbose",
                  action="store_false", default=True,
                  help="So wenige wie moeglich Ausgaben")
(options, args) = parser.parse_args()
```

- `optparse` liest Kommandozeilenparameter wie unter UNIX üblich
- liefert ein Objekt (hier `options`) mit allen Argumenten
- und eine Liste (`args`) mit den verbliebenen Argumente
- Bei Aufruf `python parse.py -f test a b c` ist:
  - `options.filename = "test"`
  - `options.verbose = True`, da Standardwert (default)
  - `args = ['a', 'b', 'c']`

BS-Abstraktion: das os-Modul

Freie Universität Berlin

```
import os
import os.path
dir = os.path.dirname(file)
name = os.path.basename(file)
altdir = os.path.join(dir, "alt")
if not os.path.isdir(altdir):
    if os.path.exists(altdir):
        raise Exception("\"alt\" kein Verzeichnis")
    else: os.mkdir(altdir)
os.rename(file, os.path.join(altdir, name))
```

- betriebssystemunabhängige Pfadtools im Untermodul `os.path`: z.B. `dirname`, `basename`, `join`, `exists`, `isdir`
- `os.system`: Programme wie von der Shell aufrufen
- `os.rename` / `os.remove`: Dateien umbenennen / löschen
- `os.mkdir` / `os.rmdir`: erzeugen / entfernen von Verzeichnissen
- `os.fork`: Skript ab dieser Stelle zweimal ausführen