

Physik auf dem Computer

Die Programmiersprache Python

Methodik der Programmierung

Jens Smiatek und Axel Arnold

Institut für Computerphysik
Universität Stuttgart

Sommersemester 2017

Python



- schnell zu erlernende Programmiersprache
- objektorientierte Programmierung ist möglich
- viele Standardfunktionen
- breite Auswahl an Bibliotheken
- freie Software mit aktiver Gemeinde
- portabel, gibt es für fast jedes Betriebssystem
- entwickelt 1991 von Guido van Rossum, CWI, Amsterdam



Informationen zu Python

- aktuelle Versionen 3.6.1 bzw. 2.7.13
- 2.x ist *noch* weiter verbreitet (z.B. Python 2.7 im CIP-Pool)
- Fokus in dieser Vorlesung auf Version 2.x

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>

Python starten

Aus der Shell:

```
> python
Python 2.7.12 (default, Jul 01 2016, 15:34:22)
  [GCC] on linux2
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** mit Tab-Ergänzung, History usw.

Python-Skripte

Beispiel: Python-Skript helloworld.py:

```
#!/usr/bin/python
```

```
# unsere erste Python-Anweisung
```

```
print "Hello World"
```

- mit `python helloworld.py` starten
- oder ausführbar machen (`chmod a+x helloworld.py`)
- **Umlaute vermeiden** oder Encoding-Cookie einfügen
- „`#!`“ funktioniert genauso wie beim Shell-Skript
- Zeilen, die mit „`#`“ starten, sind Kommentare

**Kommentare sind wichtig,
um ein Programm verständlich zu machen!**



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
- 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. $1 + 2 \rightarrow 3$, $"a" + "b" \rightarrow "ab"$
-	Subtraktion, z.B. $1 - 2 \rightarrow -1$
*	Multiplikation, Strings vervielfältigen, z.B. $2 * 3 = 6$, $"ab" * 2 \rightarrow "abab"$
/	Division, bei ganzen Zahlen ganzzahlig, z.B. $3 / 2 \rightarrow 1$, $-3 / 2 \rightarrow -2$, $3.0 / 2 \rightarrow 1.5$
%	Rest bei Division, z.B. $5 \% 2 \rightarrow 1$
**	Exponent, z.B. $3**2 \rightarrow 9$, $.1**3 \rightarrow 0.001$

- 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

<code>==, !=</code>	Test auf (Un-)Gleichheit, z.B. $2 == 2 \rightarrow \text{True}$, $1 == 1.0 \rightarrow \text{True}$, $2 == 1 \rightarrow \text{False}$
<code><, >, <=, >=</code>	Vergleich, z.B. $2 > 1 \rightarrow \text{True}$, $1 <= -1 \rightarrow \text{False}$
<code>or, and</code>	Logische Verknüpfungen „oder“ bzw. „und“
<code>not</code>	Logische Verneinung, z.B. not <code>False</code> == <code>True</code>

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

<i>a</i>	<i>b</i>	<i>a</i> und <i>b</i>	<i>a</i> oder <i>b</i>
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** $5 < 7 \rightarrow \text{True}$,
 $1 < 1$ **or** $2 >= 3 \rightarrow \text{False}$

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 Einrückungen

```
>>> 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 stehen
- Einrücken durch Leerzeichen oder Tabulatoren (einfacher)

Blöcke und Einrückungen 2

- ein Block kann nicht leer sein, Befehl **pass** ist keine Anweisung

```
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 $k \leq a < l$
- später lernen wir Listen zu erstellen und verändern

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 Anfang

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

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

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



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
- Ob eine Rekursion endet, ist nicht immer offensichtlich
- Jeder rekursive Algorithmus kann auch **iterativ** als verschachtelte Schleifen formuliert 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 Klammern
- können Daten *verschiedenen* Typs enthalten
- `liste[i]` bezeichnet das *i*-te Listenelement, negative Indizes starten vom Ende
- `liste.append()` fügt ein Element an eine Liste an
- **del** löscht ein Listenelement

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])
3
```

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

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

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: Ganzzahlen (Integers)
- %e, %f, %g: Fließkommazahl mit oder ohne Exponent oder als 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()
```

- Mögliche Modi (Wert von mode):

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

- Dateien sind Sequenzen von Zeilen (wie eine Liste von Zeilen)
- Nur beim Schließen der Datei werden alle Daten geschrieben

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

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!

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 *pseudozufällige* 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 wie ein Tupel
 - Alle Elemente vom selben (einfachen) Datentyp
 - Aber sehr schneller Zugriff
 - 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
>>> print np.array([1.0, 2, 3])
array([ 1.,  2.,  3.])
>>> print np.ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
>>> print 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

```

>>> print np.array([[1, 2, 3], [4, 5, 6]])
array([[1, 2, 3],
       [4, 5, 6]])
>>> print 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]]])
>>> print 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

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

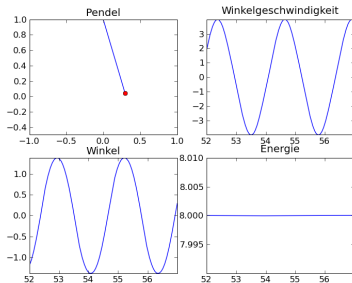
Lineare Algebra

```
>>> 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
```

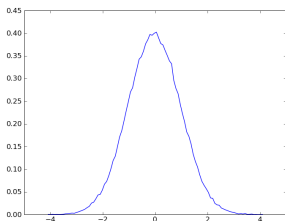
- Arithmetischer **Durchschnitt**

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

- **Varianz**

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

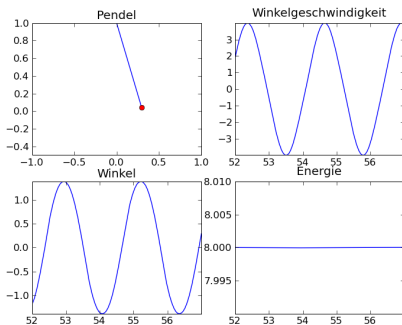
Histogramme



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

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

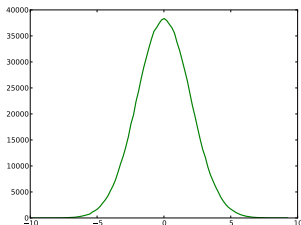
Malen nach Zahlen – matplotlib



- Ein Modul zum Erstellen von Graphen
- 2D oder 3D, mehrere Graphen in einem
- Kann auch animierte Kurven darstellen

2D-Plots

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



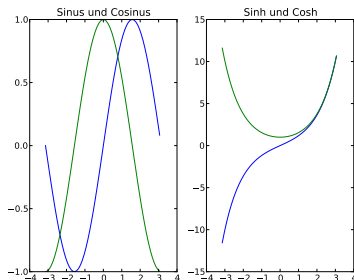
- `pyplot.plot` erzeugt einen 2D-Graphen
- `pyplot.text` schreibt beliebigen Text in den Graphen
- `pyplot.show()` zeigt den Graphen an
- Parametrische Plots mit Punkten ($x[t]$, $y[t]$)
- für Funktionen Punkte ($x[t]$, $y(x[t])$) mit x Bereich
- Farbe und Form über String und Parameter – ausprobieren

Mehrfache Graphen

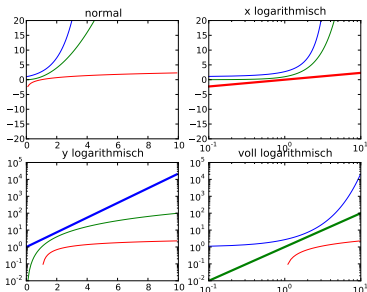
```
bild = pyplot.figure()
```

```
graph_1 = bild.add_subplot(121, title="Sinus und Cosinus")  
graph_1.plot(x, np.sin(x))  
graph_1.plot(x, np.cos(x))  
graph_2 = bild.add_subplot(122, title="Sinh und Cosh")  
graph_2.plot(x, np.sinh(x), x, np.cosh(x))
```

- Mehrere Kurven in einem Graphen:
`plot(x_1,y_1 [,"stil"], x_2,y_2 ,...)!`
- Oder mehrere `plot`-Befehle
- Mehrere Graphen in einem Bild mit Hilfe von `add_subplot`



Logarithmische Skalen



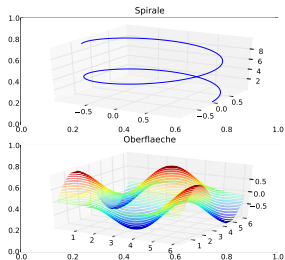
$$y = \exp(x)$$
$$y = x^2$$
$$y = \log(x)$$

- `set_xscale("log")` bzw. `set_yscale("log")`
- y logarithmisch: $y = \exp(x)$ wird zur Geraden $y' = \log(y) = x$
- x logarithmisch: $y = \log(x) = x'$ ist eine Gerade
- x + y logarithmisch: Potenzgesetze $y = x^n$ werden zu Geraden, da $y' = \log(x^n) = n \log(x) = nx'$

3D-Plots

```
import matplotlib
import matplotlib.pyplot as pyplot
import mpl_toolkits.mplot3d as p3d
...
bild = pyplot.figure()
z = np.arange(0, 10, 0.1)
x, y = np.cos(z), np.sin(z)
graph = p3d.Axes3D(bild)
graph.plot(x, y, z)
```

- plot: wie 2D, nur mit 3 Koordinaten x, y, z
- plot_wireframe: Gitteroberflächen
- contourf3D: farbige Höhenkodierung



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?
- Wahl der Rechenstrukturen (Algorithmen)
z. B. Newton-Verfahren,...

Methodik des Programmierens

Schritte bei der Entwicklung eines Programms

- **Implementation und Dokumentation**

- Programmieren und *gleichzeitig* dokumentieren
- Kommentare und externe Dokumentation (z. B. Formeln)

- **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?

Methodik des Programmierens

- Teillösungen sollen wiederverwendbar sein
 - möglichst allgemein formulieren
- 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 wird Problemanalyse wichtiger
- *Software Engineering* bei umfangreichen Projekten

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!
⇒ mehr Tests für unzulässige Eingaben

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 $a^2 + b^2 = c^2$ erfüllt, oder None, wenn keine Lösung 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

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