

## 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

## 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: 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

## 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") → '123d123abe'
- Groß-/Kleinschreibung ändern  
"hallo Welt".**capitalize**() → 'Hallo welt'  
"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..".**rstrip**(".") → 'Hallo..'

## 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

## 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()
```

---

- `datei.read()`: Lesen der *gesamten* Datei als Zeichenkette
  - `datei.readline()`: Lesen einer Zeile als Zeichenkette
  - Je nach Bedarf mittels `split`, `map`, `int` oder `float` verarbeiten
- 

```
l = '3. 4. 1.'
map(float, l.split())
```

---

## 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()
```

---

- `datei.write(data)`: *Zeichenkette* data zur Datei hinzufügen
- Anders als **print** *kein* automatisches Zeilenende
- Bei Bedarf Zeilenumbruch mit „\n“
- Daten, die keine Zeichenketten sind, mittels %-Operator oder **str** umwandeln



## Dateien als Sequenzen

```

input = open("in.txt")
output = open("out.txt", "w")
linenr = 0
for line in input:
    linenr += 1
    output.write(str(linenr) + ": " + line + "\n")
output.close()
  
```

- Alternative Implementation zum vorigen Beispiel
- Dateien verhalten sich in **for** wie Listen von Zeilen
- Einfache zeilenweise Verarbeitung
- Aber kein Elementzugriff usw.!
- **write**: alternative, umständlichere Ausgabe mittels **str**-Umwandlung



## Standarddateien

- wie in der bash gibt es auch Dateien für Standard-Eingabe, -Ausgabe und Fehler-Ausgabe
- Die Dateivariablen sind

<code>sys.stdin</code>	Eingabe (etwa Tastatur)
<code>sys.stdout</code>	Standard-Ausgabe
<code>sys.stderr</code>	Fehler-Ausgabe

```
import sys
line = sys.stdin.readline()
sys.stderr.write("don't know what to do with {}\n".format(line))
for i in range(10):
    sys.stdout.write("{} ".format(i))
sys.stdout.write("\n")
```

Ausgabe:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

## Ein-/Ausgabe mittels `raw_input`

---

```
>>> passwd = raw_input("enter password to continue: ")
enter a password to continue: secret
>>> control = input("please repeat the password: ")
please repeat the password: passwd
>>> if passwd == control: print "both are the same!"
both are the same!
```

---

- Tastatureingaben können einfach über `raw_input` in eine Zeichenkette gelesen werden
  - `input` wertet diese hingegen als Python-Ausdruck aus
  - Dies ist eine potentielle Fehlerquelle:
- 

```
>>> passwd = input("enter a password to continue: ")
enter a password to continue: secret
NameError: name 'secret' is not defined
```

---

- Eingaben über die Kommandozeile sind meist praktischer — oder wäre Dir ein `mv` lieber, dass nach den Dateien fragt?

## Umlaute — Encoding-Cookie

```
#!/usr/bin/python
# encoding: utf-8
# Zufällige Konstante  $\alpha$ 
alpha = 0.5
#  $\alpha^2$  ausgeben
print "Mir dünkt, dass  $\alpha^2 = \{:g\}$ ".format(alpha**2)
```

Ausgabe:

Mir dünkt, dass  $\alpha^2 = 0.25$

- Umlaute funktionieren bei Angabe der Codierung
- Muss in den ersten beiden Zeilen stehen
- Variablennamen trotzdem in ASCII!

## 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 sys
>>> print "program name is {}".format(sys.argv[0])
program name is ""
>>> from random import random
>>> print random()
0.296915031568
```

- Bis jetzt haben wir einen Teil der Basisfunktionalität von Python gesehen.
- Weitere Funktionen sind in **Module** ausgelagert
- Manche sind nicht Teil von Python und müssen erst nachinstalliert werden
- Die Benutzung eines installierten Moduls muss per **import** angekündigt werden („Modul laden“)
- Hilfe: **help(modul)**, alle Funktionen: **dir(modul)**

## Das sys-Modul

- Schon vorher für Eingaben benutzt
- Stellt Informationen über Python und das laufende Programm selber zur Verfügung
- `sys.argv`: Kommandozeilenparameter, `sys.argv[0]` ist der Programmname
- `sys.stdin`,  
`sys.stdout`,  
`sys.stderr`: Standard-Ein-/Ausgabedateien

---

```
import sys
sys.stdout.write("running {}\n".format(sys.argv[0]))
line = sys.stdin.readline()
sys.stderr.write("some error message\n")
```

---

## argparse-Modul: Parameter in Python 2.7

---

```
from argparse import ArgumentParser
parser = ArgumentParser()
parser.add_argument("-f", "--file", dest="filename",
                    help="write to FILE", metavar="FILE")
parser.add_argument("number", type=int, help="the number")
args = parser.parse_args()
```

---

- Einlesen von Kommandozeilenflags
- add\_argument spezifiziert Parameter
  - kurzer + langer Name („-f“, „--file“),  
ohne Minus positionsabhängiger Parameter
  - dest: Zielvariable für den vom Benutzer gegebenen Wert
  - **type**: geforderter Datentyp (**type**="int")
- Bei Aufruf python parse.py -f test 1 ist  
args.filename = 'test', args.number = 1
- python parse.py -f test a gibt Fehler, da „a“ keine Zahl



## math- und random-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  $s$

## os-Modul: Betriebssystemfunktionen

---

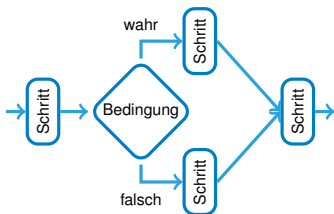
```
import os  
import os.path  
# Datei in Verzeichnis "alt" verschieben  
dir = os.path.dirname(file)  
name = os.path.basename(file)  
altdir = os.path.join(dir, "alt")  
# Verzeichnis "alt" erstellen, falls es nicht existiert  
if not os.path.isdir(altdir): os.mkdir(altdir)  
# Verschieben, falls nicht schon existent  
newpath = os.path.join(altdir, name)  
if not os.path.exists(newpath): os.rename(file, newpath)
```

---

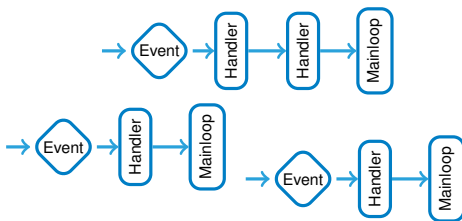
- betriebssystemunabhängige Pfadtools im Untermodul `os.path`:  
z.B. `dirname`, `basename`, `join`, `exists`, `isdir`
- `os.system`: Programme wie von der Shell aufrufen
- `os.rename/os.remove`: Dateien umbenennen / löschen
- `os.mkdir/os.rmdir`: erzeugen / entfernen von Verzeichnissen

# Graphical User Interfaces (GUI)

## Klassisch



## Eventgetrieben (GUI)



- Klassisch: Programm bestimmt, was passiert
- Warten auf Benutzereingaben
- GUI: Programm reagiert auf Eingaben (Events)
- Hauptschleife ruft Codestücke (Handler) auf
- Reihenfolge durch Benutzereingaben oder Timer bestimmt
- Programm muss Datenkonsistenz sicherstellen

## Das Tkinter-Modul

---

```
import Tkinter
root = Tkinter.Tk() # main window and connection to Tk
root.title("test program")
def quit():
    print text.get()
    root.quit()
# text input
text = Tkinter.Entry(root)
text.pack()
# button ending the program
end = Tkinter.Button(root, text = "Quit", command = quit)
end.pack({"side": "bottom"})
root.mainloop()
```

---

- bietet Knöpfe, Textfenster, Menüs, einfache Graphik usw.
- mit `Tk.mainloop` geht die Kontrolle an das Tk-Toolkit
- danach eventgesteuertes Programm



## Numerik mit Python – numpy

- numpy ist ein Modul für effiziente numerische Rechnungen
- Nicht fester Bestandteil von Python, aber Paket in allen Linux-Distributionen
- Alles nötige unter <http://numpy.scipy.org>
- Bietet unter anderem
  - mathematische Grundoperationen
  - Sortieren, Auswahl von Spalten, Zeilen usw.
  - Eigenwerte, -vektoren, Diagonalisierung
  - diskrete Fouriertransformation
  - statistische Auswertung
  - Zufallsgeneratoren
- Wird bei `ipython --pylab` automatisch unter dem Kürzel `np` geladen

## np.ndarray – $n$ -dimensionale Arrays

```
>>> import numpy as np
>>> A = np.identity(2)
>>> print A
[[ 1.  0.]
 [ 0.  1.]]
>>> v = np.zeros(5)
>>> print v
[ 0.  0.  0.  0.  0.]
>>> print type(A), type(v)
<type 'numpy.ndarray'> <type 'numpy.ndarray'>
```

- NumPy basiert auf  $n$ -dimensionalem Array `numpy.ndarray`
- Technisch zwischen Array und Tupel
  - Kein `append/remove`
  - Aber elementweiser lesender und schreibender Zugriff
  - Alle Elemente vom selben (einfachen) Datentyp
- Entspricht mathematischen Vektoren, Arrays, Tensoren, ...

## Eindimensionale Arrays – Vektoren

---

```
>>> import numpy as np
>>> print np.array([1.0, 2, 3])
[ 1.,  2.,  3.]
>>> print np.zeros(2)
[ 0.,  0.]
>>> print np.ones(5)
[ 1.,  1.,  1.,  1.,  1.]
>>> print np.arange(2.2, 3, 0.2)
[ 2.2,  2.4,  2.6,  2.8]
```

---

- `np.array` erzeugt ein `ndarray` aus einer (geschachtelten) Liste
- `np.arange` entspricht **range** für Fließkomma
- `np.zeros/ones` erzeugen 0er/1er-Arrays

## 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





## Array-Informationen

```
>>> v = np.zeros(4)
>>> print v.shape
(4,)
>>> I = np.identity(2)
>>> print I.shape
(2, 2)
>>> print I.dtype
float64
```

- `array.shape` gibt die Größen der Dimensionen als Tupel zurück
- Anzahl der Dimensionen (Vektor, Matrix, ...) ergibt sich aus Länge des Tupels
- `array.dtype` gibt den gemeinsamen Datentyp aller Elemente
- Wichtig beim Debuggen (Warum passen die Matrix und der Vektor nicht zusammen?)

## Elementzugriff

```

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

- [ ] indiziert Zeilen und Elemente usw.
- Anders als bei Pythonlisten können geschachtelte Indizes auftreten (wie bei Matrizen)
- $a[1,2]$  entspricht mathematisch  $a_{2,3}$  wegen der verschiedenen Zählweisen (ab 0 bzw. ab 1)
- Achtung: in der Mathematik bezeichnet  $a_1$  meist einen *Spaltenvektor*, hier  $a[1]$  einen *Zeilenvektor*
- Es gilt wie in der Mathematik: **Z**eilen **z**uerst, **S**palten **s**päter!

## Subarrays

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

- Wie bei Listen lassen sich auch Bereiche wählen, in allen Dimensionen
- `a[1:, 1:]` beschreibt die 1,1-Untermatrix, also ab der 2. Zeile und Spalte
- `a[:, 2]` beschreibt den 3. Spaltenvektor
- Achtung, dies sind immer *flache* Kopien!

## Flache und tiefe Kopien von ndarray

---

```
>>> a = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> b = a.copy()           # tiefe Kopie
>>> a[:,0] = np.zeros(3) # flache Kopie, ändert a
>>> print a
[[0 2 3]
 [0 5 6]
 [0 8 9]]
>>> print b
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

---

- Nützlich, weil häufig Untermatrizen verändert werden
- Anders als bei Python-Arrays sind Unterlisten *keine* Kopien!
- Kopien gibt es nur explizit durch `copy`

## Manipulation von ndarrays

---

```
>>> a = np.array([[1,2], [3,4]])
>>> a = np.concatenate((a, [[5,6]]))
>>> print a
[[1 2]
 [3 4]
 [5 6]]
>>> print a.transpose()
[[1 3 5]
 [2 4 6]]
>>> a = np.array([1 + 2j])
>>> print a.conjugate()
[ 1.-2.j]
```

---

- `np.concatenate` hängt Matrizen aneinander
- `transpose()`: Transponierte (Spalten und Zeilen vertauschen)
- `conjugate()`: Komplex Konjugierte

## np.dot: Matrix-Matrix-Multiplikation

---

```
>>> a = np.array([[1,2],[3,4]])
>>> i = np.identity(2)           # 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

# Lineare Algebra

---

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

---

- **min, max**: Minimum und Maximum aller Elemente
- **cross**: Vektorkreuzprodukt
- **linalg.det, .trace**: Determinante und Spur
- **linalg.norm**: (2-)Norm
- **linalg.eig**: Eigenwerte und -vektoren
- **linalg.inv**: Matrixinverse
- **linalg.solve(A, b)**: Lösen von  $Ax = b$

## Beispiel: Rücksubstitution

### Problem

Gegeben: Rechte obere Dreiecksmatrix  $A$ , Vektor  $b$

Gesucht: Lösung des linearen Gleichungssystems  $Ax = b$

$$\begin{array}{cccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ & & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ & & & & \ddots & & \vdots & & \vdots \\ & & & & & & a_{nn}x_n & = & b_n \end{array}$$

Zum Beispiel aus dem Gaußschen Eliminationsverfahren  
(dazu mehr in „Physik auf dem Computer“)

### Methode: Rücksubstitution

- Letzte Gleichung:  $a_{nn}x_n = b_n \implies x_n = b_n/a_{nn}$
- Vorletzte Gleichung:

$$x_{n-1} = (b_{n-1} - a_{n-1,n}x_n)/a_{n-1,n-1}$$

- Und so weiter...



## Implementation

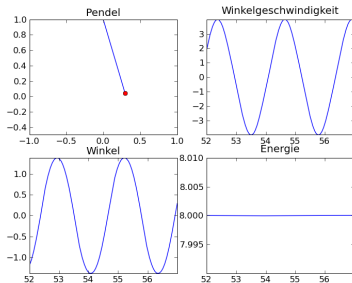
```
import numpy as np
def backsubstitute(A, b):
    rows = b.shape[0]           # length of the problem
    x = np.zeros(rows)         # solution, same size as b
    for i in range(1, rows + 1): # loop rows reversely
        row = rows - i
        x[row] = b[row] - np.dot(A[row,row+1:], x[row+1:])
        x[row] /= A[row,row]
    return x
A = np.array([[1, 2, 3], [0, 4, 5], [0, 0, 6]])
b = np.array([1, 2, 3])
print backsubstitute(A, b), np.linalg.solve(A, b)
```

Ausgabe

```
[-0.25 -0.125  0.5  ] [-0.25 -0.125  0.5  ]
```

# 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
```

---

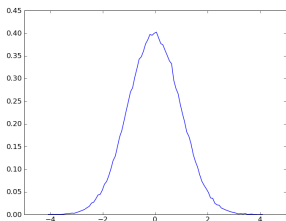
- Arithmetischer **Durchschnitt**

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

- **Varianz**

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

# Histogramme



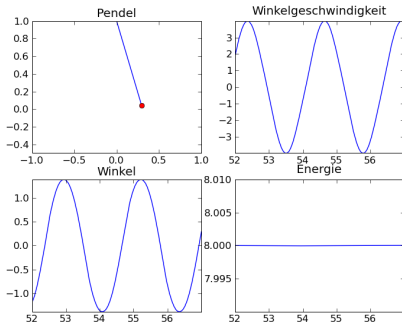
---

```
>>> zz = np.random.normal(0,1,100000)
>>> werte, raender = 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`)

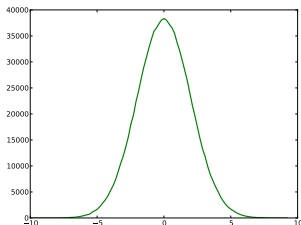
## Malen nach Zahlen – matplotlib



- Ein Modul zum Erstellen von Graphen
- 2D oder 3D, mehrere Graphen in einem
- Speichern als Bitmap
- 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()
```



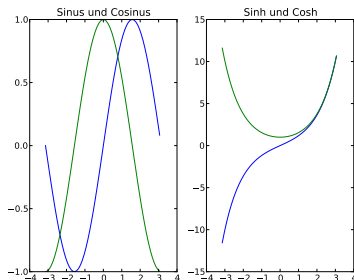
- `pyplot.plot` erzeugt einen 2D-Graphen
- `pyplot.text` schreibt beliebigen Text in den Graphen
- `pyplot.show()` zeigt den Graphen an
- Parametrische Plots mit Punkten ( $x[t]$ ,  $y[t]$ )
- für Funktionen Punkte ( $x[t]$ ,  $y(x[t])$ ) mit  $x$  Bereich
- Farbe und Form über String und Parameter –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

## Mehrfache Graphen

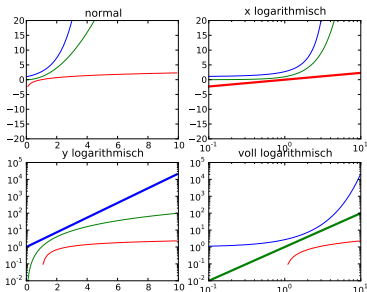
```
bild = pyplot.figure()
```

```
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



$$y = \exp(x)$$

$$y = x^2$$

$$y = \log(x)$$

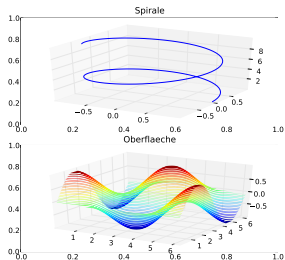
- `set_xscale("log")` bzw. `set_yscale("log")`
- $y$  logarithmisch:  $y = \exp(x)$  wird zur Geraden  $y' = \log(y) = x$
- $x$  logarithmisch:  $y = \log(x) = x'$  ist eine Gerade
- $x + y$  logarithmisch: Potenzgesetze  $y = x^n$  werden zu Geraden, da  $y' = \log(x^n) = n \log(x) = nx'$



## 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
- Achtung! 3D ist neu und das Interface ändert sich noch


 INSTITUTE FOR  
 COMPUTATIONAL  
 PHYSICS
 

## Interaktive Visualisierung

```

import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as pyplot
...
abb = pyplot.figure()
plot = abb.add_subplot(111)
kurve, = plot.plot([],[])
def weiter():
    abb.canvas.manager.window.after(1000, weiter)
    kurve.set_data(x, np.sin(x))
    abb.canvas.draw()
...
abb.canvas.manager.window.after(100, weiter)
pyplot.show()
  
```

- Update und Timing durch GUI (hier TkInter)
- set\_data um die Daten zu verändern