Physik auf dem Computer
Wiederholung: Python, NumPy und C

Axel Arnold Olaf Lenz

Institut für Computerphysik
Universität Stuttgart

Sommersemester 2012

Python

- Wiederholung der wichtigsten Eckpunkte
- interpretierte Programmiersprache
- aktuelle Versionen 3.2 bzw. 2.7.2 (CIP-Pool: 2.6)

Hilfe zu Python

- Homepage: `http://www.python.org`
- „A Byte of Python“: `http://abop-german.berlios.de`
- „Dive into Python“: `http://diveintopython.net`

Python starten

```
#!/usr/bin/python

print "Hello World"
```

- Skript mit `python helloworld.py` starten
- oder Skript ausführbar machen (`chmod a+x helloworld.py`)
- OS erkennt an „ #! “ (*shebang*) in erster Zeile, welcher Interpreter benutzt werden soll
- Python interaktiv mit `ipython` (Tab-Vervollständigung, History, ...)

Syntax

```
#!/usr/bin/python

# Eine sinnlose Bedingung
if 1 != 1:
    pass
else:
    print "Hello World"
```

- Umlaute vermeiden oder Encoding-Cookie einfügen
- Kommentare mit „#“
- Alle gleich eingerückten Zeilen gehören zum selben Block
- Ein „:“ am Ende der vorigen Zeile beginnt den neuen Block
- `pass` macht nix

Zahlen, Zeichenketten, Vergleiche, Variablen

```
>>> print -12345
-12345
>>> print 13.8E-24
1.38e-23
```

```
>>> print "Hello World"
Hello World
>>> print Hello World
Hello World
>>> print """Hello
... World"""
Hello
World
```

```
>>> print 1 == 2 or 1 != 2
True
>>> print 1 < 2 and not 1 > 2
True
>>> print '1' == 1
False
```

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

Arithmetische Ausdrücke, Bedingungen, Schleifen

```
>>> print ((1.+2.)*3./4.)**5.
57.6650390625
>>> print '1'+'2'
12
>>> print 1/2, 1%2
0 1
>>> print 1./2
0.5
```

```
if a < 3:
   print "a < 3"
elif a > 7:
   print "a > 7"
else:
   print "3 < a < 7"
```

```
i = 1
while i < 10:
    if i % 7 == 0: break
    if i % 3 == 0: continue
    print i
    i += 1
# Ausgabe: 1 2 4 5 6
```

```
>>> for s in ["Axel", "Olaf"]:
...   print "Hello", s
...
Hello Axel
Hello Olaf
```

Funktionen

```
def max2(a, b):
    if a > b: return a
    else: return b
```

```
pi = 3.14159
def circle_area(r):
    return pi*r**2
```

```
def max(*seq):
    maxv = seq[0]
    for v in seq:
        if v > maxv:
            maxv = v
    return maxv
mymax = max(1, 22, 5, 8, 5)
```

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

Dokumentation, Funktionen als Werte, Rekursion


```
def print_f(f, seq):
    """Berechnet Funktion f fuer alle Elemente von seq
       und gibt sie aus."""
    for v in seq:
        print v, f(v)

def myfunc(x):
    return x*x

print_f(myfunc, range(10))
```

```
def fib(a):
    if a <= 1: return 1
    else: return fib(a-2) + fib(a-1)
```

Listen

```
>>> mylist = ["Olaf", "Axel", "Floh"]
>>> mylist[0]
'Olaf'
>>> mylist + ["Dominic"]
['Olaf', 'Axel', 'Floh', 'Dominic']
>>> mylist[1:3]
['Axel', 'Floh']
>>> mylist.append('Dominic')
>>> mylist
['Olaf', 'Axel', 'Floh', 'Dominic']
>>> del mylist[-1]
>>> mylist
['Olaf', 'Axel', 'Floh']
>>> len(mylist)
3
```

Tupel, Flache und tiefe Kopien

```
>>> 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')
```

```
>>> list1 = [ 1, 2, 3 ]
>>> list2 = list1
>>> del list1[-1]
>>> list2
[1, 2]
```

```
>>> list1 = [ 1, 2, 3 ]
>>> list2 = list1[:]
>>> del list1[-1]
>>> list2
[1, 2, 3]
```

Assoziative Listen: Dictionaries

```
>>> de_en = { "Milch": "milk", "Mehl": "flour" }
>>> 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
Mehl => flour
Milch => milk
Oel => oil
>>> if "Mehl" in de_en:
...     print "Kann \"Mehl\" uebersetzen"
Kann "Mehl" uebersetzen
```

Formatierte Ausgabe, Objekte

```
>>> 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
```

```
>>> mylist = list()
>>> mylist.append(2)
>>> mylist.append(3)
>>> mylist.append(1)
>>> mylist.sort()
>>> mylist
[1, 2, 3]
```

```
>>> "Hallo Welt!".find("Welt")
6
```

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()
```

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!"
```

Module

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

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

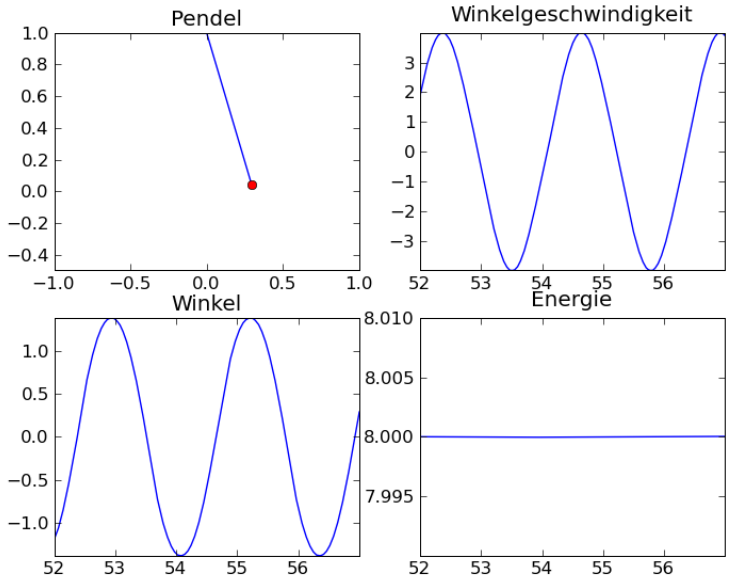
Lineare Algebra

```
>>> 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
```

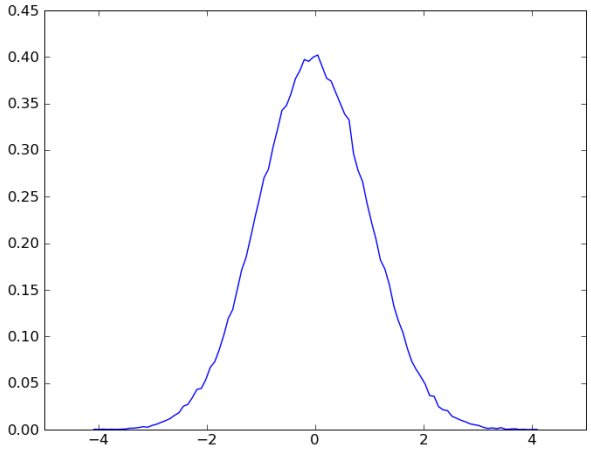
- Arithmetischer **Durchschnitt**

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

- **Varianz**

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

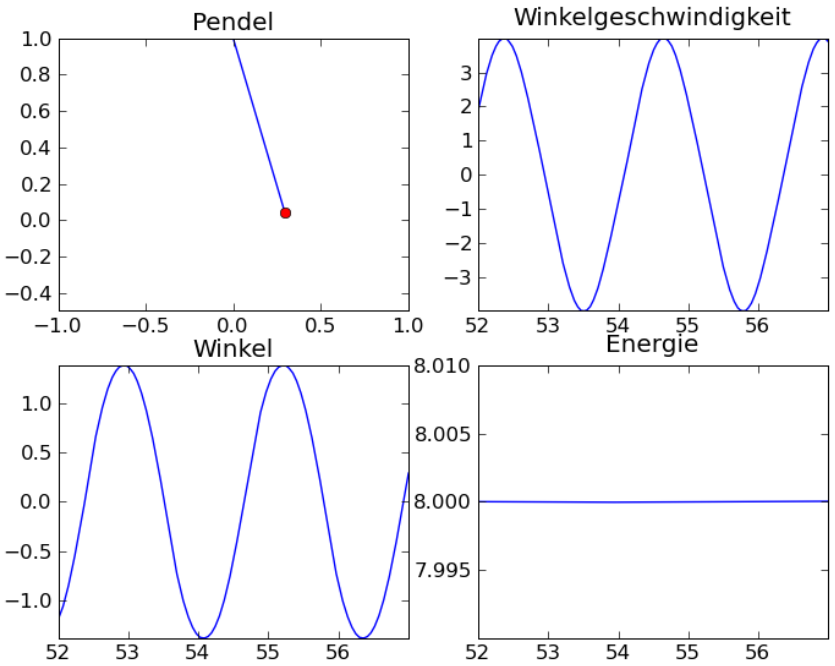
Histogramme




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

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

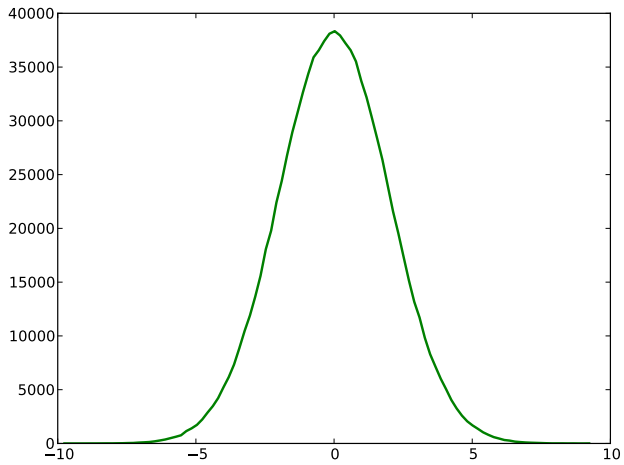
Malen nach Zahlen – matplotlib



- 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

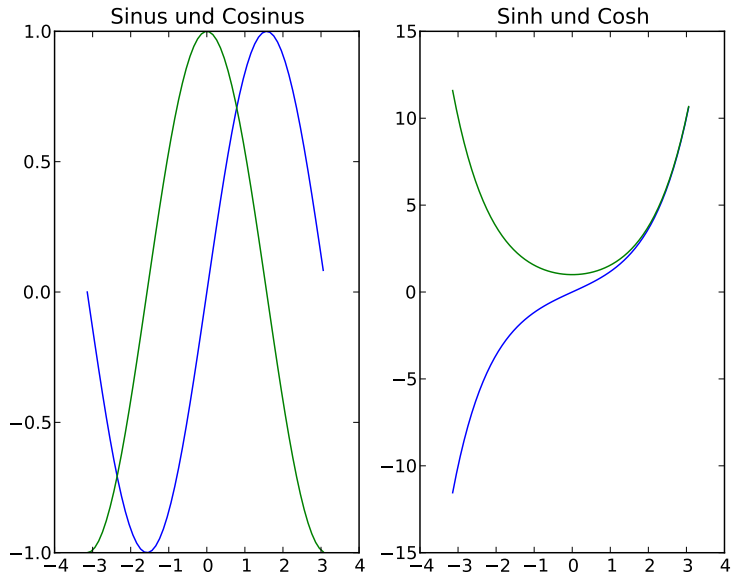
```
bild = pyplot.figure()

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))
```

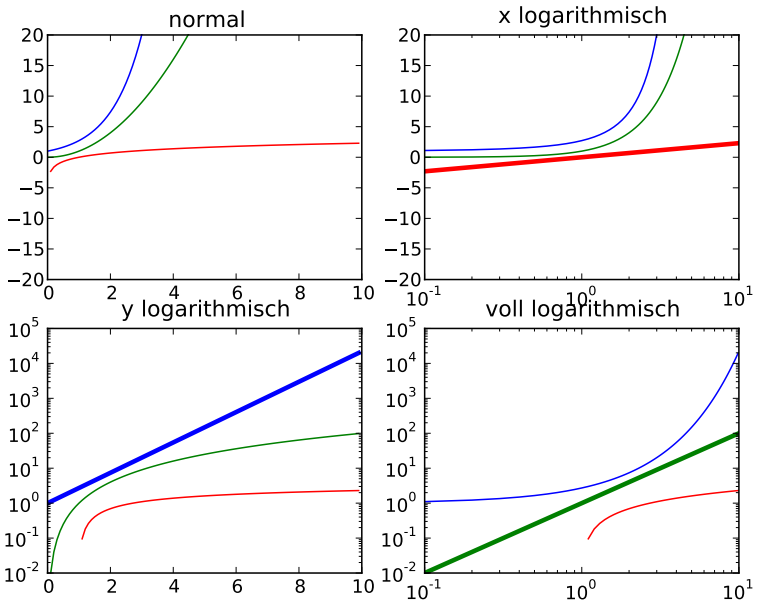
- Mehrere Kurven in einem Graphen:

  `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

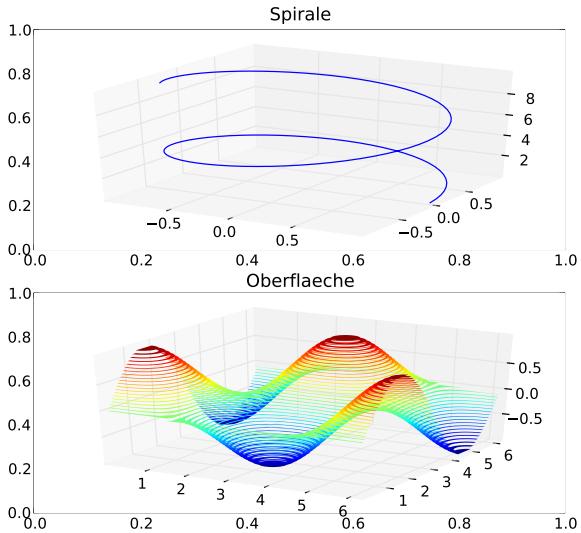


$y = exp(x)$
$y = x^2$
$y = log(x)$

- `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