Python & Statistische Methoden der Datenanalyse

A. Schälicke

version 1.2 - 14. April 2010

Zusammenfassung

Dieser Artikel dient als Begleitmaterial zur Übung & Vorlesung "Statistische Methoden der Datenanalyse" . Es stellt kein vollwertiges Pythontutorial dar, sondern konzentriert sich auf die Eigenschaften die zur Bearbeitung der Übungsaufgaben besonders nützlich sind.

Inhaltsverzeichnis

1 Einführung in Python

1.1 Python

Python ist eine Programmiersprache, die mit dem Ziel entworfen wurde möglichst einfach und übersichtlich zu sein. Sie wurde 1990 von *Guido van Rossum* am Centrum voor Wiskunde en Informatica in Amsterdam entwickelt. Die Entwicklung ist aber nicht abgeschlossen. Die aktuelle Version ist 2.6.5 (19. März 2010).

Programme in *Python* sind in der Regel deutlich kürzer als vergleichbare C/C++/Java-Programme. Die Benutzung ist sehr einfach, im Gegensatz zu Shell-Skripten ist *Python* aber auch ein Programmiersprache die strukturierte Lösungen für große oder komplexe Probleme ermöglicht.

1.2 Ein erstes Beispiel

Die Kompaktheit von *Python*-Programmen ist insbesondere auf folgende Punkte zurück zu führen:

- high-level Datentypen ermöglichen komplexe Vorgänge kompakt darzustellen,
- Funktionsblöcke werden durch Einrückungen dargestellt (statt Klammern, oder `BEGIN` / `END`)
- keine Deklaration von Variablen oder Argumenten
- *Smart-pointers*

Ein kleines Beispiel zur Berechnung der Fibonacci-Folge soll einen ersten Eindruck geben.[1].

```
>>> # Fibonacci Folge:
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
```

1.3 Listen, for-Schleifen und List-Comprehension

Listen werden in *Python* in '`[`' und '`]`' eingeschlossen. Eine leere Liste wird durch '`[]`' repräsentiert. Das Anhängen von Elementen an eine Liste geschieht durch '`append`'. Ein `Integer`-Liste kann einfach mit dem Befehl `range([start,]stop[,step])` angelegt werden. Der `stop` Parameter ist nicht Teil der erhaltenen Liste.

```
>>> teens=range(10,20)
>>> teens
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Listen stellen in *Python* auch die Grundlage von `for`-Schleifen dar.

[1] Zum Ausprobieren `python` an der Kommandozeile eingeben. Verlassen wird durch die Tastenkombination `CTRL-D`

```
>>> for item in ['apple', 'grapefruit', 'banana']:
...      print item,
...
apple grapefruit banana
```

Das abschließende Komma des `print` Befehls bewirkt, dass alle Ausgabe in einer Zeile erfolgen.

Eine häufige Anwendung von `for`-Schleifen beruht auf der Verwendung von Integer-Listen

```
>>> i2 = []
>>> for i in range(10):
...      i2.append(i*i)
...
>>> print i2
```

In diesem Beispiel wurde ein Iterator `i` angelegt der über alle Elemente der List läuft, welche wiederum durch den Befehl `range(10)` erzeugt wurde.

Eine besonders mächtige Eigenschaft von *Python* ist die so genannte *List-Comprehension*. Dadurch ist es möglich Listen in einer einzigen Programmzeile zuerzeugen. Eine *List-Comprehension* besteht aus eine Ausdruck gefolgt von einer `for`-Anweisung, welcher beliebig viele weitere `if`-Bedingungen oder `for`-Anweisungen folgen können. Das Ergebnis besteht aus einer Liste, die der bei Abarbeitung der `for`-Schleifen under Berücksichtigung der `if`-Bedingungen entsteht.

Das obige Beispiel ändert sich damit zu:

```
>>> i2 = [i*i for i in range(10)]
>>> i2
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Wenn über mehr als eine Liste gleichzeitig iteriert werden soll, gibt es zwei Möglichkeiten. Die erste Möglichkeit ist einen Laufindex mittels `range` zu Erzeugen.

```
a = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
b = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

for i in range(len(a)):
    print b[i]-a[i]
```

Mit Hilfe des Befehls `zip` ist es aber auch möglich über beide Listen gleichzeitig zu iterieren:

```
for ai,bi in zip(a,b):
    print bi-ai
```

1.4 Funktionen

In Python können natürlich auch Funktionen (oder sogar Klassen, was aber hier nicht besprochen werden soll) definiert werden.

```
>>> def f(x,y):
...     temp=x*y
...     return temp
...
>>> f(3.,4.)
```

```
12.0
>>> from numpy import *
>>> f(array([1.,2.,3.]),array([1.,2.,3.]))
array([ 1.,  4.,  9.])
```

Sehr einfache Funktionen lassen sich kompakter mittels der (aus funktionaler Programmierung entlehnten) `lambda` - Schreibweise formulieren:

```
>>> f = lambda x,y : x*y
```

Diese Variante ist besonders interessant, wenn ein Befehl als Argument eine Funktion erwartet, da dann – als temporäre, unbenannte Funktion – direkt der `lambda` Ausdruck verwendet werden kann.

1.5 Py4Science

Eine weiter große Stärke von *Python* ist die einfache Verknüpfungsmöglichkeit mit Code-Entwicklungen anderer Programmiersprachen. So gibt es Schnittstellen die Programme geschrieben in Fortran, C, C++, u.a. in Python einbinden. Bereits in der Standardbibliothek sind eine Vielzahl von Funktionen insbesondere für Internetanwendungen bereitgestellt. Im Internet gibt es bereits eine Vielzahl von Modulen für spezielle Anwendungen.

In wissenschaftlichen Anwendungen haben sich, insbesondere mit dem verbesserten Interpreter `ipython`, eine Reihe von Modulen als besonders geeignet erwiesen:

- NumPy
- SciPy
- PyLab (MatPlotLib)

Diese Kombination ermöglicht komplexe Operationen durch Vektor- bzw. Matrixschreibweise kompakt auszudrücken, stellt eine umfangreiche Funktionsbibliothek bereit, und beinhaltet auch Routinen zur graphischen Veranschaulichung. Der Syntax ist dem des kommerziellen MatLab-Systems sehr ähnlich.

Module werden durch den `import` Befehl geladen. Das kann in verschiedenen Variationen geschehen:

- laden eines kompletten Moduls

```
>>> import numpy
>>> numpy.random.uniform()
```

- laden eines kompletten Moduls unter einem Alias

```
>>> import numpy as N
>>> N.random.uniform()
```

- laden eines Submoduls

```
>>> from numpy import random
>>> random.uniform()
```

*Hinweis statt dem Namen eines Submoduls kann auch der Platzhalter * verwendet werden, um alle Bestandteile eines Moduls ohne Angabe des Modulprefix verfügbar zu machen.*

- laden eines Submoduls unter einem alias

```
>>> from numpy import random as R
>>> R.uniform()
```

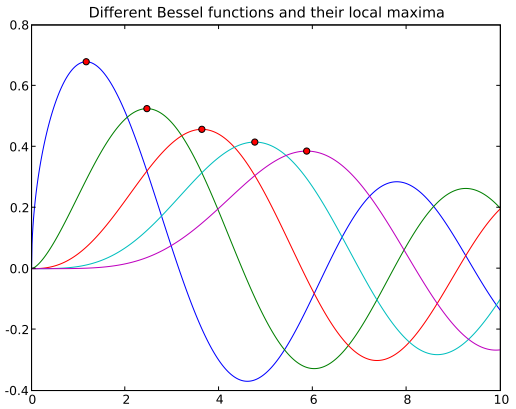
Das folgende Beispiel soll die Fähigkeiten der Wissenschaftsmodule illustrieren. Detailierte Informationen finden man in den nachfolgenden Abschnitten, oder in der Literatur.

```
from scipy import *
from numpy import *
from pylab import *

x = arange(0,10,0.01)

for k in arange(0.5,5.5):
     y = special.jv(k,x)
     plot(x,y)
     f = lambda x: -special.jv(k,x)
     x_max = optimize.fminbound(f,0,6)
     plot([x_max], [special.jv(k,x_max)],'ro')

title('Different Bessel functions and their local maxima')
show()
```



1.6 Literatur

- Python Homepage
- Python Tutorial
- Python Einführung (deutsch)
- Python Quick Reference Guide Version 2.5
- Py4Science Homepage
- How to Think Like a Computer Scientist
- NumPy homepage
- SciPy Cookbook
- SciPy Tutorial
- MatPlotLib Homepage

2 Python im Physik-Pool

2.1 Getting Started - IPython

Zum starten des *Python*-Interpreters muss man nur `ipython` an der Kommandozeile eingeben. Beim ersten Start werden Konfigurationsdateien in Home-Verzeichnis unter `/.ipython`, abgelegt.

Verlassen kann man `ipython` oder `python` durch die Tastenkombination `CTRL-D`. Alternativ steht auch der Befehl `exit()` zum Verlassen des Pyhonprompts zur Verfügung. IPython kann man auch mit `Exit` beenden.

```
pool:~>ipython
**********************************************************************
Welcome to IPython. I will try to create a personal configuration directory
where you can customize many aspects of IPython's functionality in:

~/.ipython
Initializing from configuration /var/lib/python-support/python2.5/IPython/UserConfig

Successful installation!

Please read the sections 'Initial Configuration' and 'Quick Tips' in the
IPython manual (there are both HTML and PDF versions supplied with the
distribution) to make sure that your system environment is properly configured
to take advantage of IPython's features.

Important note: the configuration system has changed! The old system is
still in place, but its setting may be partly overridden by the settings in
"~/.ipython/ipy_user_conf.py" config file. Please take a look at the file
if some of the new settings bother you.


Please press <RETURN> to start IPython.
**********************************************************************
Python 2.5.2 (r252:60911, Jan 24 2010, 17:44:40)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.4 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]: ^D
Do you really want to exit ([y]/n)? y
```

2.2 Personalisierung

Die Standard-Farbpalette von `ipython` ist für dunkle Hintergrundfarbe optimiert, bei Verwendung von hellen Hintergrundfarben, wie in den meisten Terminalfenstern, sollte man das ändern. Es ist auch möglich die Abfrage beim Beenden auszuschalten. Dazu editiert man die Konfigurationsdatei `~/.ipython/ipythonrc` an den entsprechenden Stellen wie folgt:

ipythonrc

```
....
# keep uncommented only the one you want:
#colors Linux
colors LightBG
#colors NoColor

....
confirm_exit 0
```

Zudem ist beim interaktiven Arbeiten sinnvoll `ipython` mit der Option `-pylab` aufzurufen, siehe auch Abschnitt 5.2.

2.3 Hilfe & Dokumentation

Der Python-Interpreter `ipython` ist besonders für Einsteiger in die Pythonprogrammierung hilfreich, aber auch fortgeschrittenen Anwendern ermöglicht es komfortable ausgewählte Programmteile interaktive zu testen, bevor sie ins Hauptprogramm eingebaut werden.

Hilfe kann man z.B. durch durch folgende Vorgehensweisen erhalten

- Standard-Python Hifesystem
  `help()`, `help(<comando>)`, z.B. `help(range)`
- IPython Hilfe
  `?`, `<befehl>?`, z.B. `range?`
- `pdoc` System
  `pdoc <befehl>`
- *Auto-completition*
  `[TAB]`
- *Python-History*
  `%history`, `history` (*IPython Magic command*)
  Up/Down-Pfeiltasten (wenn bereits text in der aktuellen Zeile eingeben ist, geht Up-Pfeiltaste zum letzten auftreten dieses Textes, d.h. entspricht einer Suchfunktion)
- einige weitere *IPython Magic commands*

  `%magic` gibt Hilfe zum Magic-system
  `%Exit` verlässt IPython
  `%ls` gibt den Inhalt des aktuellen Verzeichnisses aus
  `%cd` wechselt das aktuelle Verzeichnis

3 Numpy

3.1 Die erste Session - Arrays

In Anlehnung an die Standard-Listen in *Python* stellt das *NumPy* Modul eine `array`-erweiterung zur Verfügung. Erzeugt werden `array`s entweder durch expliziten aufruf des Konstruktors, z.B. `array([1.,2.,3.])`, durch Verwendung des Befehls `arange`, oder auch automatisch bei Verwendung von Numpy-versionen von mathematischen Funktionen, z.B. `sin`.

```
>>> from numpy import *
>>> x = arange(0.,pi,0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
        2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1])
>>> sin(x)
array([ 0.        ,  0.09983342,  0.19866933,  0.29552021,  0.38941834,
        0.47942554,  0.56464247,  0.64421769,  0.71735609,  0.78332691,
        0.84147098,  0.89120736,  0.93203909,  0.96355819,  0.98544973,
        0.99749499,  0.9995736 ,  0.99166481,  0.97384763,  0.94630009,
        0.90929743,  0.86320937,  0.8084964 ,  0.74570521,  0.67546318,
```

```
        0.59847214,  0.51550137,  0.42737988,  0.33498815,  0.23924933,
        0.14112001,  0.04158066])
>>> exp([0.,1., 2.])
array([ 1.        ,  2.71828183,  7.3890561 ])
```

3.2 Zufallszahlen

Mit dem Module *NumPy* ist es sehr einfach Arrays von Zufallszahlen anzulegen. Zum Beispiel können diskrete Zufallsverteilungen generiert werden.

```
import numpy
numpy.random.poisson(size=10)
```

Das funktioniert natürlich auch mit kontinuierlichen Verteilungen.

```
numpy.random.uniform(size=10)
```

3.3 Statistik Funktionen

- Summe:

$$\bar{x} = \sum_i x_i \tag{1}$$

```
from numpy import *
N = 100000
x = random.uniform(size=N)
sum(x)
```

- Mittelwert & Varianz:

$$\bar{x} = \frac{1}{N} \sum_i x_i \tag{2}$$

$$v = \frac{1}{N} \sum_i (x_i - \bar{x})^2 \tag{3}$$

```
mean(x)
var(x)
```

3.4 Extra-Dimensionen

Ein `numpy`-Array muss nicht eindimensional sein. Die Verteilung der Element auf die Dimensionen kann (auch im Nachhinein) über den `shape` Parameter festgelegt werden.

Funktionen wie `sum` oder `mean`, die einen array zu einer Zahl kontrahieren, kann ein optionaler Parameter `axis` mitgegeben werden. In diesem Fall wird die entsprechende Operation nur entlang der angegeben Dimension durchgeführt, das Ergebnis ist dann ein wieder eine Array mit reduzierter Dimension.

```
>>> x.shape
(100000,)
>>> x.shape=(10000,10)
>>> sum(x,axis=0)
```

```
array([ 4968.56447055,  4993.76044577,  4951.46844308,  5059.7819245 ,
        4972.54312824,  5007.45446391,  4993.63990046,  5011.02797777,
        5013.06217959,  5030.526914  ])
```

4 SciPy

Das *SciPy* Modul erweitert *Python* um eine Vielzahl von Funktionen für wissenschaftliches Arbeiten. Es beinhaltet bereits viele Funktionen des *NumPy* Moduls.

```
from scipy import *
N = 10000
x = random.uniform(size=N)
patches, bins = histogram(x,bins=10,range=(0.,1.), normed=True)

print patches

# use MatPlotLib to view the histogram
from pylab import *
bar(bins, patches, width=0.1)
show()
```

Für Anwendungen im Rahmen der Statistik-Vorlesung sind folgende Module von interesse:

- `stats` – Statistical functions and tests
- `special` – Collection mathematical functions
- `optimize` – General-purpose optimization routines
- `ndimage` – Multidimensional image processing
- `integrate` – Integration routines

4.1 Statistik-Funktionen

Das Modul `scipy.stats` enthält u.a.

- Funktionen zur Bestimmung statischer Momente von Verteilungen
- Routinen für diverse statische standard Tests und
- diverse kontinuierliche und diskrete Verteilungsfunktionen.

Für alle Verteilungsfunktions(klassen) sind die Methoden:

rvs – random variates with the distribution

pdf/pmf – probability density/mass function

cdf – cummulative distribution function

sf – survival function (1.0 - cdf)

ppf – percent-point function (inverse of cdf)

isf – inverse survival function

stats – mean, variance, and optionally skew and kurtosis

definiert.

In `ipython` kann man mit `<keyword>?` leicht Hilfe zu allen bereitgestellten Funktionen bekommen.

```
import scipy.stats
scipy.stats?       # summary help

from scipy.stats import *
norm?              # gaus normal distribution
expon?             # exponential distribution
cauchy?            # Cauchy distribution

binom?             # binominal distribution
poisson?           # poisson distribution
```

Die folgenden statistische Standard-Funktionen sind ebenfalls implementiert: `gmean`, `hmean`, `mean`, `cmedian`, `median`, `mode`, `tmean`, `tvar`, `tmin`, `tmax`, `tstd`, `tsem`, `moment`, `variation`, `skew`, `kurtosis`, `describe`, `skewtest`, `kurtosistest`, `normaltest`.

5 PyLab

Das *PyLab* Modul verbindet die Fähigkeiten von *NumPy* und *SciPy* mit einem Graphik-System. Der Syntax ist ähnlich dem kommerziellen MatLab Programm.

5.1 Beispiele

```
from pylab import *
from scipy.stats import *

entries=poisson.rvs(3.,size=100)    # generate random variable
bins=arange(10)                     # define 10 bins
hist(entries,bins,normed=True)      # create, fill and draw histogram
title(r'Poisson-Verteilung $f(n)=\lambda e^{-\lambda n}$ mit $\lambda=1/3$')
ylabel('$f(n)$')
xlabel('$n$')

show()                    # show figure (not needed in interactive mode)
```

```
figure()

x=linspace(-10,10,101)          # define 101 x values in [-10,10]
y1=norm.pdf(x)                  # calc Gaus with default paramter
y2=norm.pdf(x,loc=1,scale=2)
y3=norm.pdf(x,loc=3,scale=5)

plot(x,y1,color='red',label='normal')
plot(x,y2,color='blue',linestyle='--',label=r'$\mu=1$, $\sigma=2$')
plot(x,y3,color='green',linestyle=':',label=r'$\mu=3$,$\sigma=5$')

title('Gauss-Normal-Verteilung')
```

```
ylabel('$f(x)$')
xlabel('$x$')
legend()

# increase maximum of yaxis
ymin,ymax = ylim()
ylim(ymax=ymax*1.1)

show()                    # show figure (not needed in interactive mode)
```

```
from pylab import *
from scipy import *
from numpy import *

x = arange(0,10,0.01)

for k in arange(0.5,5.5):
     y = special.jv(k,x)
     plot(x,y)
     f = lambda x: -special.jv(k,x)
     x_max = optimize.fminbound(f,0,6)
     plot([x_max], [special.jv(k,x_max)],'ro')

title('Different Bessel functions and their local maxima')
show()
```

5.2 Konfiguration

In den Standardeinstellungen arbeitet das Graphik-Modul *MatPlotLib* im *Batch-Modus*, d.h. die graphische Ausgabe der vorangegangenen Befehle erfolgt erst wenn `show()` aufgerufen wird.

```
from pylab import *
hist(rand(100000))
show()        # im batch modus wird der plot erst jetzt ausgegeben
```

Auch ist ein Weiterarbeiten solange nicht möglich, bis das Graphikfenster wieder geschlossen wird. Eine Alternative bietet der interaktive Modus. Dieser kann mit dem Befehl `ion()` aktiviert werden.

```
from pylab import *
ion()         # aktiviere interaktiven modus
hist(rand(100000))  # plot wird sofort dargestellt
```

Alternative kann man ipython mit der option `-pylab` aufrufen. Dadurch wird ipython im interactiven modus mit Thread-support gestartet. Zusatzlich werden auch alle functionen vom matplotlib.pylab Packet importiert. Das obige Beispiel sieht dann wie folgt aus.

```
pool:~>ipython -pylab
...
In [1]: hist(rand(100000))  # plot wird sofort dargestellt
```

A Quick Reference Guide

A.1 Python interpreter

A.1.1 Ausführen von Python-Skripts

An der Komandozeile können Python-Skripte einfach durch Anhängen and den Standard Python-Interpreter

```
pool:~> python myskript.py
```

oder natürlich auch an den *iPython*-Interpreter gestartet werden.

```
pool:~> ipython myskript.py
```

`iPython` wechselt anschliessend in den interaktiven Modus (wenn Graphik ausgegeben wurde).
Alternative kann ein Skript aus iPython mit dem `run` Befehl aufgerufen werden:

```
In [1]: run myskript.py
```

A.2 NumPy package

random.uniform `(low=0.0, high=1.0, size=None)`
array von gleichverteilten Zufallszahlen

random.rand `(d0, d1, ..., dn)`
array von gleichverteilten Zufallszahlen der angegebenen Dimension(en)

arange `([start,] stop[, step])`
array gegebener Schrittweite (ohne 'stop')

linspace `(start, stop, num[, endpoint=True ])`
array mit linearen Abständen (ohne 'stop' wenn 'endpoint=False')

sum `(a [,axis=None])`
Summe über alle Werte eines array (oder einer Achse, bei verwendung des `axis` parameter)

mean `(a [,axis=None])`
Mittelwert alle Werte eines array (oder einer Achse, bei verwendung des `axis` parameter)

histogram `(a[, bins=10][, range=(down,up)][, normed=False])`
erzeugt Histogram im gegeben Bereich

histogram2d `(x,y,[, bins=10][, normed=False])`
erzeugt ein 2D Histogram der arrays.

A.2.1 Array-Arithmetik

```
# Erzeugen
a = array([1., 2., 5., 13., 21.])     # von einer Liste
x = arange(0.,1.,0.2)                 # = array([ 0., 0.2, 0.4, 0.6, 0.8])
r = random.rand(5,2)                  # von Zufallszahlen
m = diag(ones(3))                     # 3 x 3 Einheitsmatrix

# Auschschneiden
a[1:3]                                # im Bereich = array([ 2., 5.])
x[:-2]                                # vom Ende = array([ 0., 0.2, 0.4])
```

```
r[:,0]                                  # komplette erste Spalte

# Elementweise Operationen
b = a**2                                # = array([ 1., 4., 25., 169., 441.])
y = sin(x)                              # = array([ 0., 0.199, 0.389, 0.565, 0.717])

# Dimensionsreduktion
w = mean(r,axis=0)                      # z.B. = array([ 0.52594254,  0.66878539])

# Matrixalgebra
A = array([ [ 1., 4.], [3., 0.] ])      # 2 x 2 dimensional
u = array([ [1.], [0.] ])               # Spaltenvektor
dot(A,u)                                # Matrix-multiplikation

Ainv = linalg.inv(A)                    # Inverse einer Matrix
dot(Ainv,A)                             # check: = diag(ones(2))

s,v=linalg.eig(A)                       # berechnet Eigenwerte und Vektoren
v1, v2 = v[:,0:1], v[:,1:2]             # def. (Spalten-)Eigenvektoren
dot(A,v1) - s[0]*v1                     # check: = array([[ 0.], [ 0.]])

vInv = linalg.inv(v)                    # Inverse der Transformationsmatrix
dot(vInv, dot(A, v))                    # check: = diag(s)
```

A.3 PyLab Package

plot `(x,y,**kwargs)`
Zeichnen einer Funktion, z.B.

```
x = linspace(0.,2.*pi,100)
plot(x,sin(x))
```

hist `(r,bins,**kwargs)`
Zeichnet Histogramm des gegeben Ereignis-array,z.B.

```
hist(random.normal(size=100000),arange(-3.,3.,0.1))
```

bar `(left,heigt[, width=0.8,align='edge'])`
Zeichnet Bar-Plot, z.B. ein Histogramm erzeugt durch `histogram`

```
r = random.normal(size=10000)
h,left = histogram(r,20,range=(-2.,2.),normed=True)
bar(left,h,width=left[1]-left[0])
```

figure `([num = None][, figsize=(8, 6)])`
Erzeugt ein neues Fenster (oder wechselt zur angegeben Nummer)

Farben

Einige Farben:
`black, silver, gray, white, maroon, red, purple, fuchsia, green, lime, olive, yellow, navy, blue, teal, aqua, lightblue, ...`
Kürzel:
`b (blue), g (green), r (red), c (cyan), m (magenta), y (yellow), k (black), w (white)`

Zahlen:
`0.75` (a grayscale intensity), `#2F4F4F` an RGB hex color string, `(0.18, 0.31, 0.31)` an RGB tuple.

A.4 Scipy package

A.4.1 Packages:

- `cluster` – Vector quantization / Kmeans
- `fftpack` – Discrete Fourier transform algorithms
- `integrate` – Integration routines
- `interpolate` – Interpolation tools
- `io` – Read write of vector/matrix files (also MatLab)
- `lib` – Lapack/Blas wrapper
- `linalg` – Linear algebra basics
- `linsolve` – Linear solvers
- `maxentropy` - Maximum entropy modelling tools
- `misc` – Various utilities that don't have another home
- `ndimage` – Multidimensional image processing
- `odr` – Orthogonal Distance Regression (ODR)
- `optimize` – General-purpose optimization routines
- `sandbox` – Tools that are not mature enough
- `signal` – Signal processing
- `sparse` – Rudimentary sparse matrix class
- `special` – Collection of mathematical functions
- `stats` – Statistical functions and tests

B MatPlotLib

B.1 Plotting commands

acorr plot the autocorrelation function

annotate annotate something in the figure

arrow add an arrow to the axes

axes Create a new axes

axhline draw a horizontal line across axes

axvline draw a vertical line across axes

axhspan draw a horizontal bar across axes

axvspan draw a vertical bar across axes

axis Set or return the current axis limits

bar make a bar chart

barh a horizontal bar chart

broken_barh a set of horizontal bars with gaps

box set the axes frame on/off state

boxplot make a box and whisker plot

cla clear current axes

clabel label a contour plot

clf clear a figure window

clim adjust the color limits of the current image

close close a figure window

colorbar add a colorbar to the current figure

cohere make a plot of coherence

contour make a contour plot

contourf make a filled contour plot

csd make a plot of cross spectral density

delaxes delete an axes from the current figure

draw Force a redraw of the current figure

errorbar make an errorbar graph

figlegend make legend on the figure rather than the axes

figimage make a figure image

figtext add text in figure coords

figure create or change active figure

fill make filled polygons

gca return the current axes

gcf return the current figure

gci get the current image, or None

getp get a handle graphics property

grid set whether gridding is on

hist make a histogram

hold set the axes hold state

ioff turn interaction mode off

ion turn interaction mode on

isinteractive return True if interaction mode is on

imread load image file into array

imshow plot image data

ishold return the hold state of the current axes

legend make an axes legend

loglog a log log plot

matshow display a matrix in a new figure preserving aspect

pcolor make a pseudocolor plot

pcolormesh make a pseudocolor plot using a quadrilateral mesh

pie make a pie chart

plot make a line plot

plot_date plot dates

plotfile plot column data from an ASCII tab/space/comma delimited file

polar make a polar plot on a PolarAxes

psd make a plot of power spectral density

quiver make a direction field (arrows) plot

rc control the default params

rgrids customize the radial grids and labels for polar

savefig save the current figure

scatter make a scatter plot

setp set a handle graphics property

semilogx log x axis

semilogy log y axis

show show the figures

specgram a spectrogram plot

spy plot sparsity pattern using markers or image

stem make a stem plot

subplot make a subplot (numrows, numcols, axesnum)

subplots_adjust change the params controlling the subplot positions of current figure

subplot_tool launch the subplot configuration tool

table add a table to the plot

text add some text at location x,y to the current axes

thetagrids customize the radial theta grids and labels for polar

title add a title to the current axes

xcorr plot the autocorrelation function of x and y

xlim set/get the xlimits

ylim set/get the ylimits

xticks set/get the xticks

yticks set/get the yticks

xlabel add an xlabel to the current axes

ylabel add a ylabel to the current axes

B.2 colormaps

autumn set the default colormap to autumn

bone set the default colormap to bone

cool set the default colormap to cool

copper set the default colormap to copper

flag set the default colormap to flag

gray set the default colormap to gray

hot set the default colormap to hot

hsv set the default colormap to hsv

jet set the default colormap to jet

pink set the default colormap to pink

prism set the default colormap to prism

spring set the default colormap to spring

summer set the default colormap to summer

winter set the default colormap to winter

spectral set the default colormap to spectral