Skriptsprachen

Numpy und Scipy

Kai Dührkop

Lehrstuhl fuer Bioinformatik
Friedrich-Schiller-Universitaet Jena
kai.duehrkop@uni-jena.de

24. September 2015

Numpy

numpy und scipy

```
import numpy as np
import scipy as sp
```

- numpy
  - Library für numerische Vektoren und Matrizen
  - effiziente Operationen auf Arrays
  - ermöglicht speichereffiziente Darstellung von numerischen Arrays
- scipy
  - Sammlung mathematischer Funktionen (numerisch, analytisch, stochastisch)
  - arbeitet eng mit numpy zusammen
  - historisch sind einige Funktionen in numpy verteilt, die eigentlich in scipy gehören sollten

- numpy arrays sind **homogene** arrays eines Datentyps
- können Datentypen darstellen, die Python nicht kennt (z.B. 16-bit unsigned Integer)

convert python list to numpy array

```
np.array([1,2,3,4,5])
np.array([1,2,3,4,5], dtype=np.int32)
np.array([[True, False],
          [False, True]], dtype=np.bool8)
```

convert numpy array to python list

```
ary = np.array([[1,2],[3,4]])
alist = ary.tolist() #=> [[1,2],[3,4]]
```

allocating a new array

```
# 10x10 array filled with 0
np.zeros((10, 10), dtype=np.float64)

# 100x100x2 array filled with 1
np.ones((100,100,2), dtype=np.float32)
```

allocating random array

```
# random array with values from 0 to 10
# with size 5x5
ary = np.random.randint(0,10,(5,5))
# use poisson distribution instead
ary = sp.random.poisson(5.0, (5,5))
```

slicing

```
ary[0, 1] #=> entry in first row, second column
ary[:, 1] #=> second column
ary[0, :] #=> first row
ary[0:2, 0:2] #=> submatrix
```

slicing with indizes

```
np.array([ary[0,0], ary[1,1], ary[2,2]])
# is equivalent to
indizes = [0,1,2]
ary[indizes,indizes]

# another example
ary[[0,1],[3,4]]
#=> returns elements ary[0,3] and ary[1,4]
```

slicing with ix

```
# pick all elements which are from row 1,2 OR 3
# and column 0, 1 OR 5
ary[np.ix_([0,1,3], [0,1,5])]
```

slicing with booleans

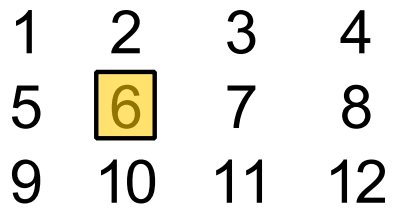
```
# returns all elements greater than 5
ary[ary > 5]
```

- indizes, booleans, lists and slices are possible for selecting elements
- to keep the dimension/shape of the array you need slices or ix_ function

1	2	3	4
5	6	7	8
9	10	11	12

Creating the array

```
ary = np.array([
   [1,2,3,4],
[5,6,7,8],
[9,10,11,12]
])
```



Accessing single element

```
ary[1,1]
```

Output

```
6
```

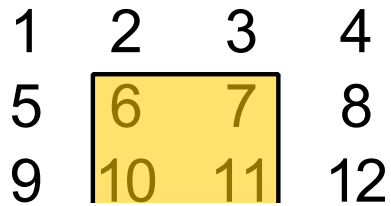
1	2	3	4
5	6	7	8
9	10	11	12

Accessing column

```
ary[:,1]
```

Output

$$\begin{pmatrix} 2 \\ 6 \\ 10 \end{pmatrix}$$

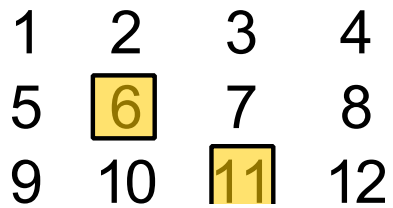


Accessing submatrix

```
ary[1:3,1:3]
```

Output

$$\begin{pmatrix} 6 & 7 \\ 10 & 11 \end{pmatrix}$$

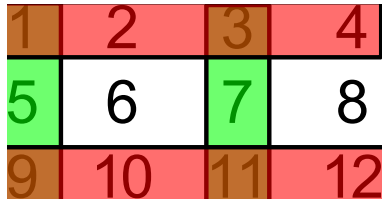


Accessing several elements

```
ary[[1, 2], [1, 2]]
```

Output

$$\begin{pmatrix} 6 & 11 \end{pmatrix}$$



Accessing several rows and cols

```
ary[np.ix_([0,2], [0,2])]
```

Output

$$\begin{pmatrix} 1 & 3 \\ 9 & 11 \end{pmatrix}$$

component-wise operations

```
# returns new array with each element
# is increased by one
newAry = ary + 1

# many unary and binary operations
np.sqrt(ary) + np.square(ary)

# returns a new array with each element
# is multiplied with the corresponding element
# in the second array
ary + ary2
```

- standard operations (+, - , *, ...) can be applied to arrays
- for other mathematical operations, use the numpy module instead of the math module
- operations always operate on each element separately and return a new array

axis-wide operations

```
# get max element from array
maximum = np.max(ary)

# get max element from each column
maximumAry = np.max(ary, 0)

# get max element from each row
maximumAry = np.max(ary, 1)

# sum up all columns and return an array of sums
summedCols = np.sum(ary, 0)
```

- axis-wide operations either apply on all elements or move along the given axis

transposing arrays

```
ary.T
```

concatenating arrays

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), 0)
# array([[1, 2],
#        [3, 4],
#        [5, 6]])
np.concatenate((a, b.T), 1)
# array([[1, 2, 5],
#        [3, 4, 6]])
```

matrices

```
A = np.matrix(ary)
A*A #=> matrix multiplication
A**2 #=> A*A

ary.dot(ary.T) == A * A.T
```

- matrices behave **almost** like arrays
- some operations are different: * and ** are matrix multiplication/power, so they are not applied element-wise
- for arrays **dot** can be used for matrix multiplication and dot product

linear algebra

```
import numpy.linalg as lg
eigenvalue, eigenvectors = lg.eig(ary)

determinante = lg.det(ary)

# solve linear equation
variables = np.array([[1,3], [3, 4]])
coefficients = np.array([30, 20])
solution = lg.solve(variables, coefficients)
#=> [-12, 14]
```

$$x_0 + 3x_1 = 30$$
$$3x_0 + 4x_1 = 20$$

statistics

```
import numpy.linalg as lg
# mean, variance, standard deviation, covariance
mean(ary), var(ary), std(ary), cov(ary)

# median, 20% and 80% percentile
median(ary)
quantile(ary, (20, 80))
```

distributions

```
from scipy import stats as st

# Normal distribution with mean=5, std=2
N = st.norm(5, 2)

# get culmulative probability
N.cdf(2)

# get probability density for some points
ary = N.pdf([1, 5, 7])
```

distributions

```
from scipy import stats as st

# pareto distribution with b=1, k=2
p = st.pareto(1, 0, 2)

# generate 100 random values
ary = p.rvs(100)

# learn parameters from data
params = st.pareto.fit(ary)
p = st.pareto(*params)
```

matplotlib

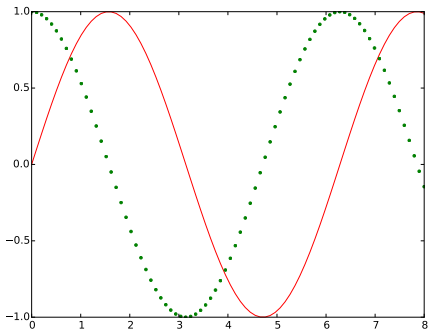
matplotlib

```
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
import scipy.stats as st
# in ipython for interactive work:
%matplotlib
```

- plotting library
- can work with numpy arrays
- can be used interactively in ipython

matplotlib

```
# get array of x values (similar to xrange)
# from 0 to 8 with 80 steps
xs = np.linspace(0, 8, 80)

# plot sinus and cosinus function
plt.plot(xs, np.sin(xs), "r")
plt.plot(xs, np.cos(xs), "g.")

# save plot to file
plt.savefig("sinuskosinus.pdf")
```



simple plotting

- plot() is a allrounder, allowing for different plotting styles
- takes x and y values as parameters together with a format string
- format string defines color and line style
  - color can be **r**ed, **g**reen, **blue**, blac**k** and many others
  - line styles: solid (-), dashed (--), dotted (:)
  - marker styles: points (.), circles (o), stars (*), triangle ($\hat{}$)
- example format string: **r^** is red triangles. **b-o** is blue solid line with circles

other optional parameters

- linewidth
- color and marker can be used instead of format strings
- markersize
- label - binds a name on data (can be used later to refer to specific datapoints)

other plotting functions

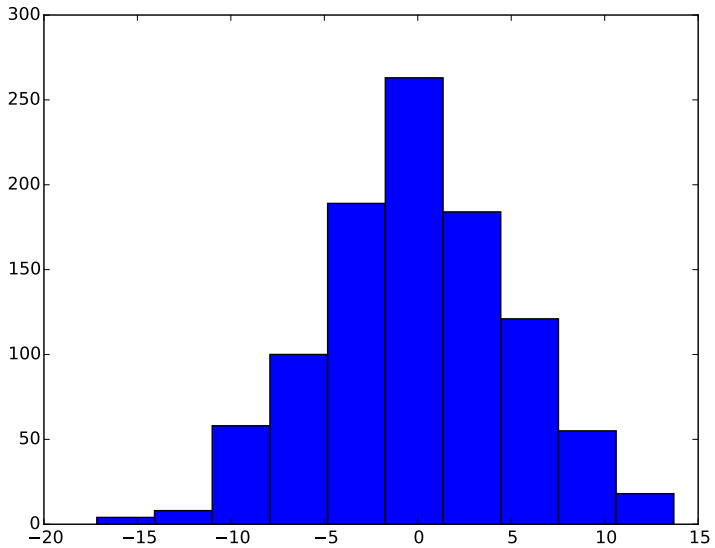
- **hist** for plotting histograms
- **lines** for plotting line curves
- **vlines**, **hlines** for vertical and horizontal lines
- **fill** for filled polygons (area under the curve is filled with a color)
- **pie**
- a lot more in `http://matplotlib.org/examples/`

other plotting functions

- **hist** for plotting histograms
- **lines** for plotting line curves
- **vlines**, **hlines** for vertical and horizontal lines
- **fill** for filled polygons (area under the curve is filled with a color)
- **pie**
- a lot more in `http://matplotlib.org/examples/`

histograms

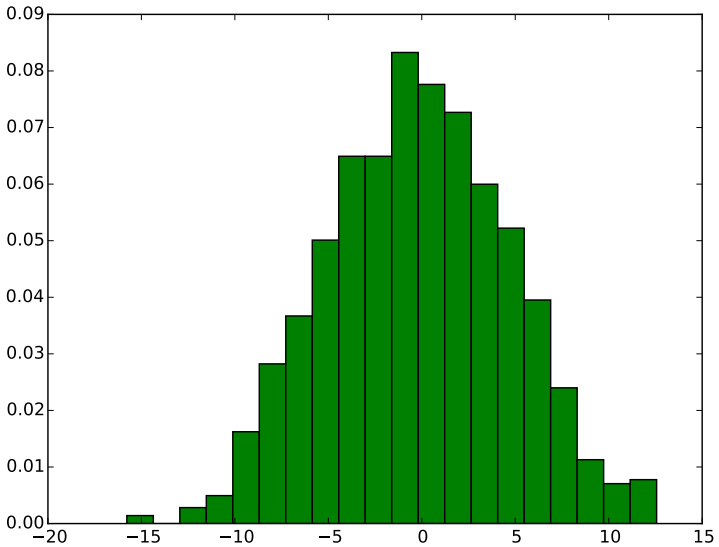
```
observations = st.norm(0, 5).rvs(1000)
plt.hist(observations)
```



- simple case: histogram from an array of observations

histograms

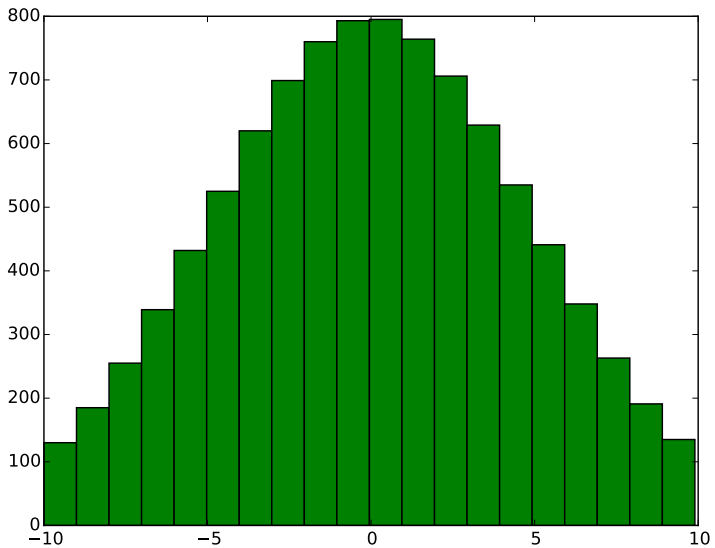
```
observations = st.norm(0, 5).rvs(1000)
plt.hist(observations, color="green",
         bins=20, normed=True)
```



- specify color and number of bins
- use frequencies instead of counts

histograms

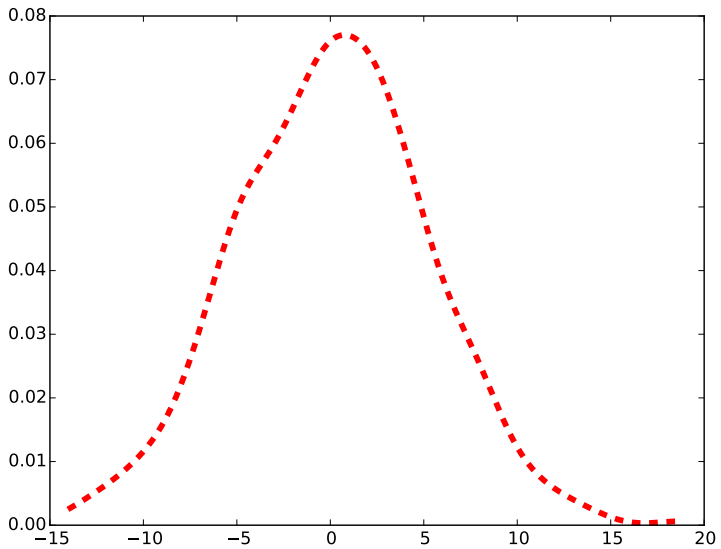
```
values = np.arange(-10,10,0.1)
occurences = np.round(
   st.norm(0,5).pdf(values) * 1000)

plt.hist(values, weights=occurences,
          color="green", bins=20)
```



- we have tuples with observation value and number of

kernel density

```
observations = st.norm(0, 5).rvs(1000)
xs = np.linspace(np.min(observations),
   np.max(observations),1000)

kernel = st.gaussian_kde(observations)

plt.plot(xs, kernel(xs), "r--", linewidth=4)
```

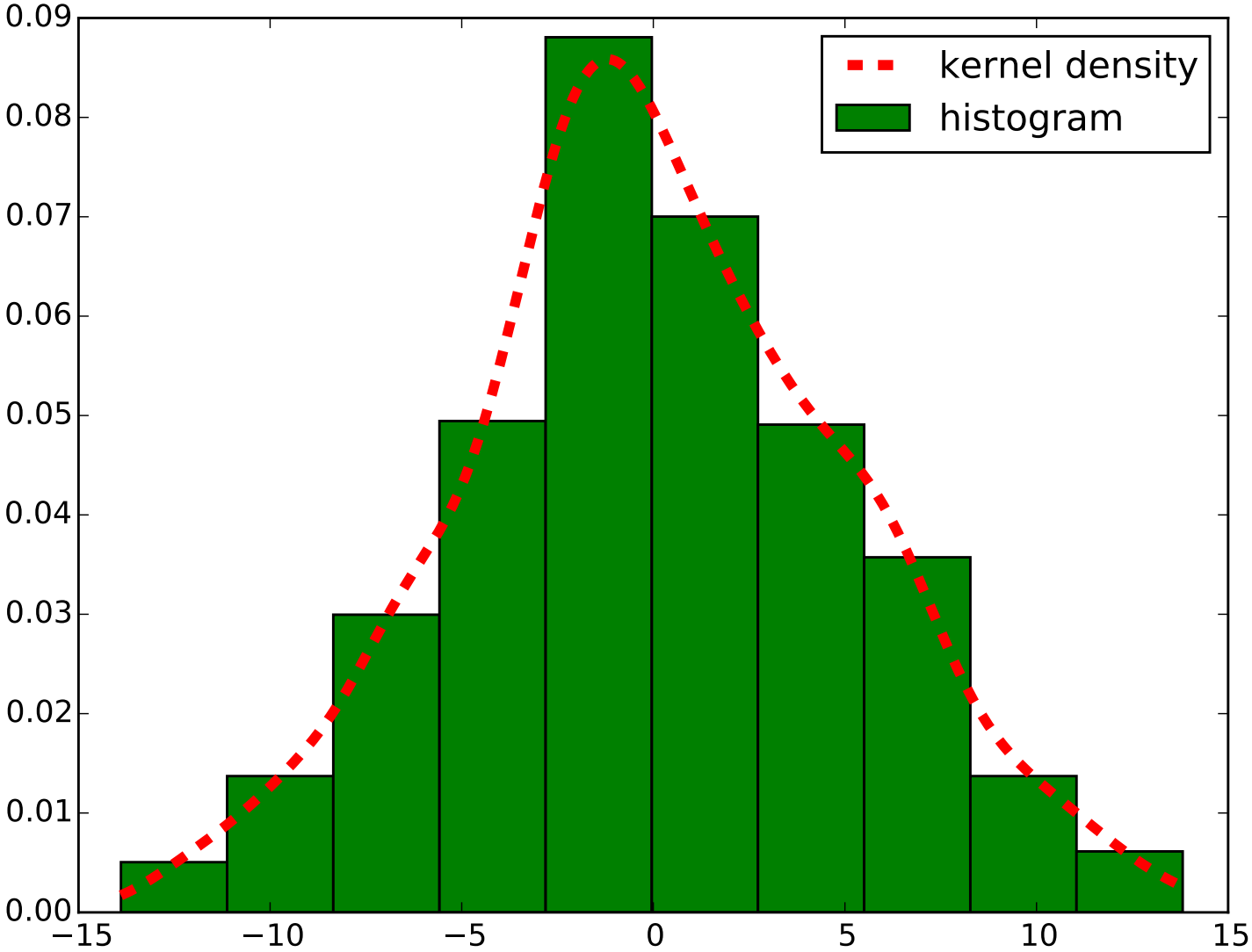


labels

```
plt.xlabel("mass")
plt.ylabel("frequency")
plt.title("A histogram of masses")

# print a label directly into the plot
plt.text(5, 3, "The datapoint at position 5/3 is in

# enable tex math mode
plt.rc('text', usetex=True)
plt.xlabel(r"$\frac{m}{z}$")
```

legend

```
observations = st.norm(0, 5).rvs(1000)
xs = np.linspace(np.min(observations),
   np.max(observations),1000)

kernel = st.gaussian_kde(observations)
plt.hist(observations, color="green",
   normed=True, label="histogram")
plt.plot(xs, kernel(xs), "r--",
   linewidth=4, label="kernel density")
plt.legend()
```


kernel density
histogram
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
−15
−10
−5
0
5
10
15

other important functions

```
# display plot from x-values 0 to 10
plt.xlim(0, 10)
# and y values 0 1
plt.ylim(0,1)

# clear plot
plt.clf()
```