

Einführung in Python

10. Vorlesung



NumPy

SciPy

Matplotlib



Wiederholung letztes Mal

- Im Grunde gibt es drei Arten von Fehlern:
 - *Syntaxfehler* *Laufzeitfehler* *Semantikfehler*
- Im Allgemeinen kann man solchen Fehlern folgendermaßen begegnen:
 - Gute und sehr ausführliche Dokumentation des Programmcodes
 - Kontrollen an kritischen Stellen des Programms einbauen, wenn sie nicht erfüllt sind wird ein Programmabbruch ausgelöst (**Exceptions** oder **Asserts**)
 - Starten des Programms in einem speziellem Testmodus, so dass es seine Arbeitsweise dokumentiert (**Logging**)
 - Schritt-für-Schritt Analyse des Programms mit Hilfe eines **Debuggers**



Wiederholung letztes Mal

- Vor- und Nachbedingungen mit ***assert***: Ist eine assert-Funktion nicht erfüllt, so wird eine *AssertionError* Ausnahme erzeugt

```
def primzahl_faktoren(zahl):  
    #Prüfe Vorbedingung  
    assert (type(zahl) == int) and (zahl > 0)  
    fak = [1]  
    faktor = 2  
    while zahl > 1:  
        while zahl % faktor == 0:  
            fak.append(faktor)  
            zahl /= faktor  
            faktor += 1  
    #Prüfe Nachbedingung  
    produkt = 1  
    for i in fak:  
        produkt *= i  
    assert produkt == zahl  
    return(fak)
```

- Im optimierten Modus werden alle *assert*-Anweisungen übersprungen

```
user@pc:~$ python3.6 -O meinSkript.py
```



Wiederholung letztes Mal

- Selbstdokumentation des Programms durch Log-Dateien

```
>>> import logging  
  
>>> logging.basicConfig(filename='/tmp/logFile.txt', level=logging.DEBUG)  
>>> logging.debug('Erster Eintrag')  
>>> x = 19  
>>> logging.debug(x)  
>>> logging.info('Zweiter Eintrag')
```

logFile.txt

```
DEBUG:root:Erster Eintrag  
DEBUG:root:19  
INFO:root:Zweiter Eintrag
```

Bedeutsamkeit

DEBUG

INFO

WARNING

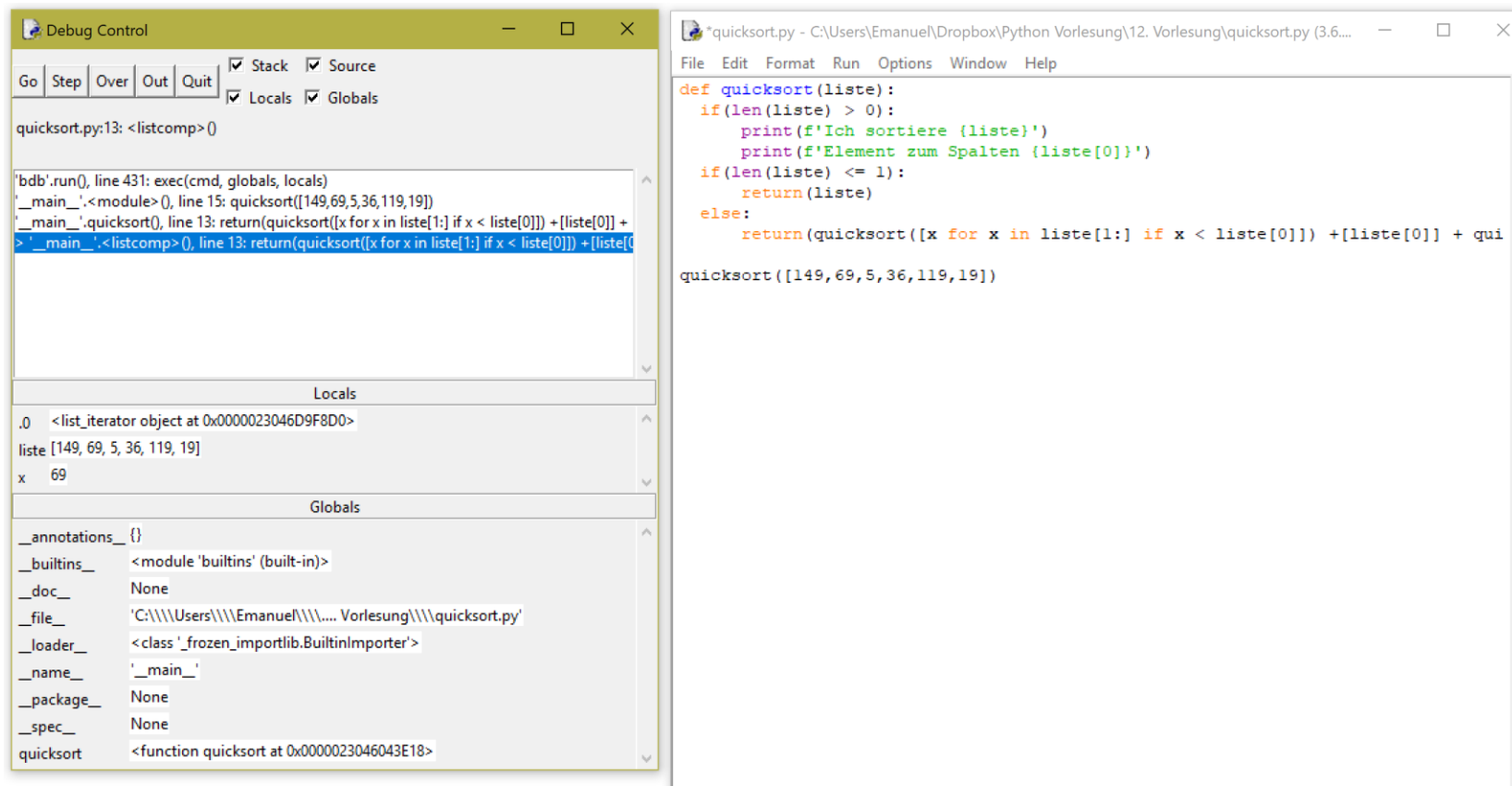
ERROR

CRITICAL



Wiederholung letztes Mal

- Der IDLE Debugger von Python ist ein mächtiges Werkzeug, um Programme schrittweise auszuführen und den Inhalt aller Variablen im Auge zu behalten



The image displays the Python IDLE environment with two windows. The left window, titled 'Debug Control', shows the execution state of a program. The 'Stack' pane indicates the current frame is 'quick.py:13: <listcomp>()'. The 'Locals' pane shows the variable 'liste' with the value '[149, 69, 5, 36, 119, 19]' and 'x' with the value '69'. The 'Globals' pane shows the function 'quicksort' defined at '0x0000023046043E18'. The right window, titled '*quick.py - C:\Users\Emanuel\Dropbox\Python Vorlesung\12. Vorlesung\quick.py (3.6...', shows the source code of the QuickSort algorithm. The code is as follows:

```
def quicksort(liste):  
    if len(liste) > 0:  
        print(f'Ich sortiere {liste}')        print(f'Element zum Spalten {liste[0]}')        if len(liste) <= 1:  
            return liste  
        else:  
            return (quicksort([x for x in liste[1:] if x < liste[0]]) + [liste[0]] + qui  
quicksort([149, 69, 5, 36, 119, 19])
```



Wiederholung letztes Mal

- Automatisiertes Testen mit **Doctest** und **Unittest**

```
def quadratsumme(liste):  
    """ Summer der Quadrate der Elemente einer Zahlenliste  
    >>> quadratsumme([1,2,3])  
    14  
    >>> quadratsumme([10,10,10])  
    300  
    >>> quadratsumme([])  
    0  
    """  
  
    summe = 0  
    for x in liste:  
        summe += x**2  
    return(summe)
```

```
import unittest  
  
class testFibonacci(unittest.TestCase):  
    def setUp(self):  
        self.testZahlen = [1,2,5,25,35]  
  
    def testElementarfall(self):  
        self.assertEqual(fibonacci(self.testZahlen[0]), 1)  
  
    def testEinfacheFall(self):  
        self.assertEqual(fibonacci(self.testZahlen[1]), 1)  
  
    def testDreiSchwereFälle(self):  
        self.assertEqual(fibonacci(self.testZahlen[2]), 5)  
        self.assertEqual(fibonacci(self.testZahlen[3]), 75025)  
        self.assertEqual(fibonacci(self.testZahlen[4]), 9227465)
```

- Somit lassen sich generalisierte, automatische Testroutinen schreiben



Wiederholung letztes Mal

- Laufzeitanalyse mittels **profile** um mögliche Performanzschwächen feststellen zu können

```
import profile, random

profile.run('[sort(x) for x in [[random.randint(0,1000) for i in range(10000)] for y in range(100)]]')
```

8640192 function calls (8640092 primitive calls) in 150.125 seconds
Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000000	1.984	0.000	1.984	0.000	:0(append)
1000000	2.562	0.000	2.562	0.000	:0(bit_length)
1		0.000	0.000	150.125	150.125 :0(exec)
1639986	4.031	0.000	4.031	0.000	:0(getrandbits)
1000000	78.828	0.000	78.828	0.000	:0(min)
1000000	33.062	0.000	33.062	0.000	:0(remove)
1		0.000	0.000	0.000	0.000 :0(setprofile)
100		5.672	0.057	119.547	1.195 <ipython-input-4-5c83a5>:1(sort)
102/2		2.328	0.023	150.109	75.055 <string>:1(<listcomp>)



Wissenschaftliches Arbeiten mit Python



NumPy

+



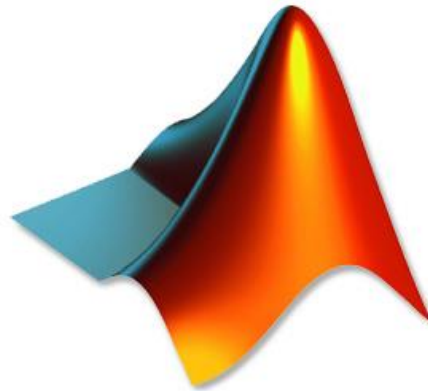
SciPy

+



matplotlib

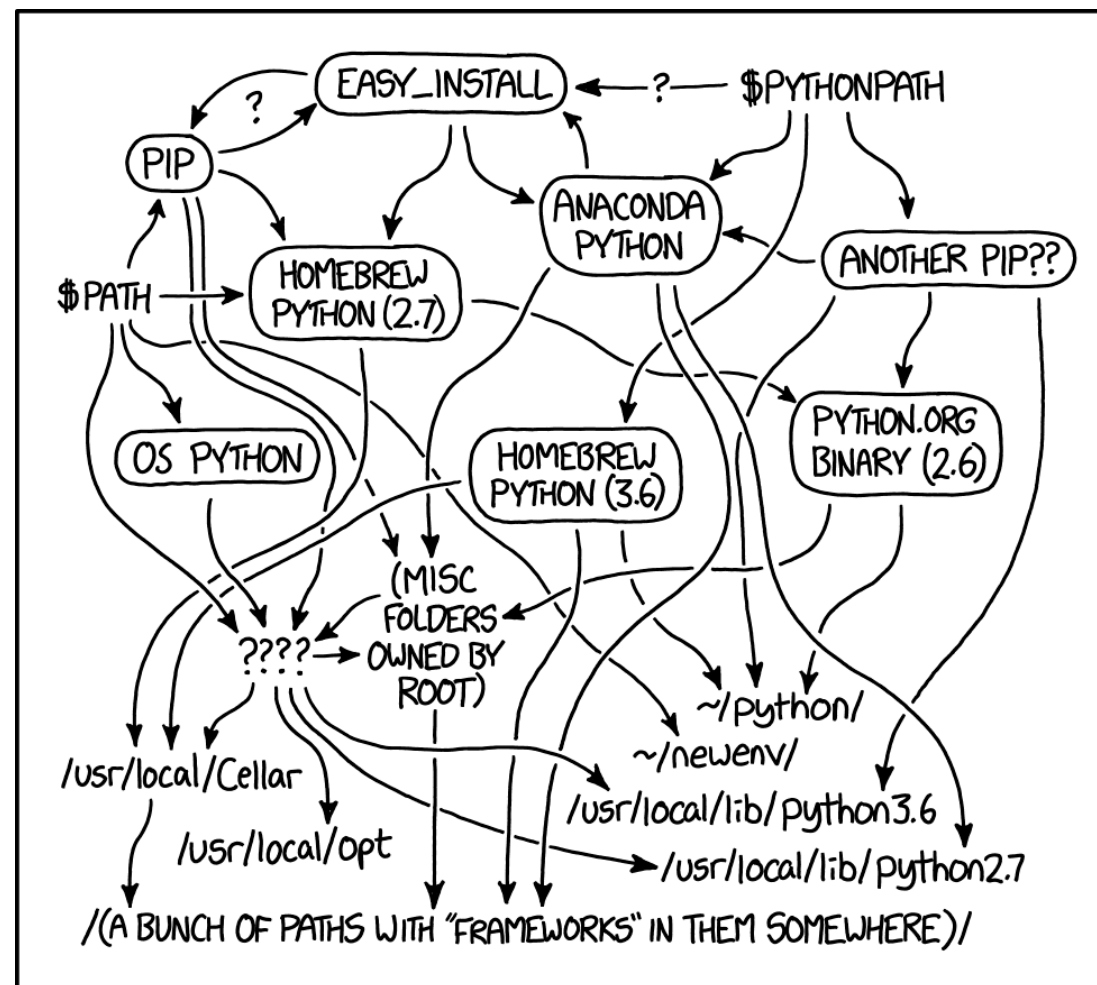
=



MATLAB



Neue Pakete für Python installieren



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.



Wiederholung: Importieren von Modulen

- Module bzw. Bibliotheken werden mittels der *import*-Funktion eingebunden

```
import random
```

- Dadurch lassen sich alle Klassen, Funktionen und Konstanten des eingebundenen Moduls nutzen, da sie nun in den globalen Namensraum geladen wurden

```
import random, numpy
```

- Man kann mit einem import-Befehl auch mehrere Module laden



Wiederholung: Importieren von Modulen

- Lädt man ein Modul nur mittels seines Namens in den eigenen Namensraum, so lassen sich dessen Funktionen etc. nur über den vollen Namen nutzen

```
import random  
x = random.randint(5,19)
```

- Es lassen sich mit dem *from*-Befehl auch gezielt Komponenten eines Moduls laden und dann ohne den vollständigen Modulnamen nutzen

```
from random import randint  
x = randint(5,19)
```

- Mit dem *as-Schlüsselwort* lässt sich beim importieren auch ein neuer Name wählen und der *-Operator lädt **alle** Funktionen eines Moduls

```
import random as r  
from random import *  
x = r.randint(5,19)  
y = randrange(36,69)
```



Modul Suchpfade

- Importiert man ein Modul, so sucht der Python-Interpreter in folgender Reihenfolge an diesen Orten nach einer gleichnamigen Datei mit der Endung **.py**:
 1. Im aktuellen Verzeichnis
 2. Im Pfad der PYTHONPATH Variable
 3. Falls diese Variable nicht vorhanden ist, im Standard Installationspfad, z. B. /usr/lib/python3.6
- Mit **sys.path** erhält man alle Suchpfade des Interpreters

```
import sys
sys.path
['',
 '/usr/local/bin',
 '/home/Emanuel/programs/Python-3.6.2/lib',
 '/mnt/c/Users/Emanuel/Desktop/scripts',
 '/usr/local/lib/python3.5',
 '/usr/local/lib/python3.5/site-packages']
```



Pakete installieren über die Kommandozeile

- Normalerweise wird mit jeder Python-Installation das Programm **pip** mit installiert (falls nicht: <https://pip.pypa.io/en/stable/installing/>)
- pip ist ein Verwaltungsprogramm für Pythonpakete und greift auf den PyPI (Python Package Index -- <https://pypi.org/>) zu

```
user@pc:~$ pip install PAKETNAME
```

- pip lädt und installiert automatisch das gewünschte Modul und alle seine Abhängigkeiten

```
user@pc:~$ pip install -index-url http://my.package.repo/test/ PAKETNAME
```

```
user@pc:~$ pip install ./downloads/SomePackage-1.9.0.tar.gz
```



NumPy & SciPy

$$\begin{bmatrix} \cos 90^\circ & \sin 90^\circ \\ -\sin 90^\circ & \cos 90^\circ \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0 \\ a_1 \end{bmatrix}$$



NumPy & SciPy

- *Numeric Python* ist ein Modul welches mathematische und numerische Datentypen und Funktionen bereitstellt
- Insbesondere stellt es effiziente Datenstrukturen für große Arrays und Matrizen sowie Operationen auf diesen bereit
- *Scientific Python* erweitert NumPy um weitere effiziente mathematische Algorithmen und Funktionen wie Minimierung, Regression, Fouriertransformation und mehr



NumPy versus Python

- Als alternative zur *range*-Funktion bietet numpy die *arange*- und *linspace*-Funktionen

```
import numpy as np

>>> range(5, 19)
range(5,19)
>>> type(range(5, 19))
<class 'range'>                                     #Ein range-Objekt ist ein Iterator

>>> np.arange(5, 19)
array([5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18])
>>> type(np.arange(5,19))
<class 'numpy.ndarray'>

>>> np.arange(5, 19, 3.6)
array([5. ,  8.6, 12.2, 15.8])

>>> np.linspace(5, 19, 10)
array([5.,  6.55555556,  8.11111111,  9.66666667, 11.22222222, 12.77777778,
 14.33333333, 15.88888889, 17.44444444, 19.]])
```



NumPy versus Python

- Zeitvergleich zwischen Python-Listen und NumPy-Listen

```
import numpy as np
import time

def normal_array():
    t = time.time()
    x, y = range(10000000), range(10000000)
    z = []
    for i in range(len(x)):
        z.append(x[i] + y[i])
    print(time.time() - t)
    return(z)

def numpy_array():
    t = time.time()
    x, y = np.arange(10000000), np.arange(10000000)
    z = x + y
    print(time.time() - t)
    return(z)
```



NumPy Arrays

- Das *ndarray* ist das wichtigste Objekt von NumPy, um welches das ganze Modul aufgebaut ist
- Es stellt ein homogenes multidimensionales Array (oder Matrix) dar, d. h. alle Elemente dieser Arrays haben immer den selben Typ
- Ein *ndarray* wird durch ein Tupel positiver Ganzzahlen indiziert, wobei man oft von Achsen statt Dimension spricht

```
[5, 19, 36]          #Array vom Rang 1 mit Achsenlänge 3

[[1,2,3],            #Array vom Rang 2; erste Achse hat Länge 2, zweite Achse Länge 3
 [4,5,6]]

[[[1,2],[3,4]],      #Array vom Rang 3; erste Achse hat Länge 3, die beiden anderen 2
 [[5,6],[7,8]],
 [[9,0],[1,2]]]
```



NumPy Arrays erzeugen

```
import numpy as np

>>> x = np.array(19)                                #0-Dimensionales Skalar

>>> y = np.array([5,19,36,69,119,149])

>>> z = np.array([1.2, 3.4, 5.6, 7.8, 9.0])

>>> np.ndim(x)                                       #Gibt den Rang des Arrays wieder
0

>>> np.ndim(y)
1

>>> y.dtype                                         #Gibt den Typ der Array Elemente wieder
dtype('int32')
>>> z.dtype
dtype('float64')
```



NumPy Arrays erzeugen

```
import numpy as np

>>> x = np.array([ [5.1, 9.3, 6.6],
                   [9.1, 1.9, 1.4]])

>>> x.ndim
2

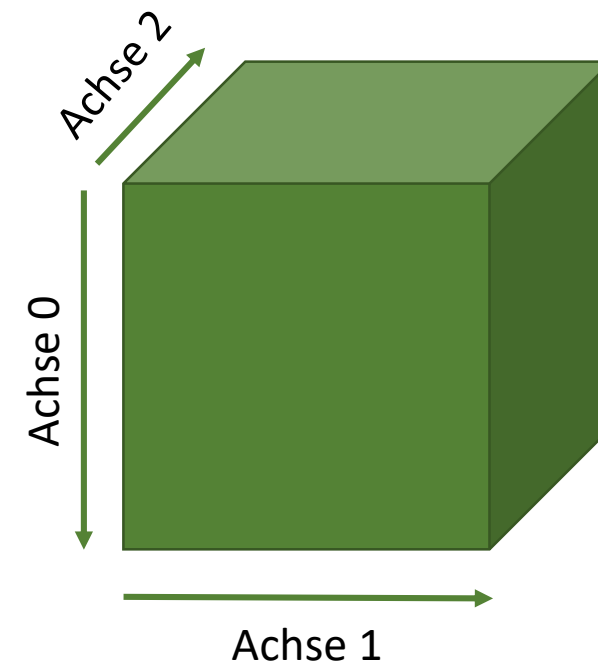
>>> y = np.array([ [1,2,3],
                   [4,5,6],
                   [7,8,9],
                   [0,1,2],
                   [3,4,5],
                   [6,7,8]])

>>> np.shape(y)
(6,3)

>>> y.reshape(3,6)
[[1,2,3,4,5,6],
 [7,8,9,0,1,2],
 [3,4,5,6,7,8]]

>>> y.reshape(4,4)
ValueError: cannot reshape array of size 18 into shape (4,4)

>>> y.reshape(2,3,3)
```



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf ein einzelnes Array Element  
x = np.arange(1,13).reshape(3,4)  
x[1:1]
```

Ausgabe:

6



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf ganze Spalten per slicing  
x = np.arange(1,13).reshape(3,4)  
x[:,1]
```

Ausgabe:

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



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf ganze zeilen per slicing  
x = np.arange(1,13).reshape(3,4)  
x[2,:]
```

Ausgabe: (9 10 11 12)



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf eine Teilmatrix per slicing  
x = np.arange(1,13).reshape(3,4)  
x[1:3,1:3]
```

Ausgabe:

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



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf eine Teilmatrix per slicing  
x = np.arange(1,13).reshape(3,4)  
np.array([x[0,0], x[1,1], x[2,2], x[1,3]])
```

```
#Zugriff auf eine Teilmatrix über Indize-Listen  
x[[0,1,2,1],[0,1,2,3]]
```

Ausgabe: (1 6 11 8)



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

#Zugriff auf eine Teilmatrix über Indize-Listen

```
x = np.arange(1,13).reshape(3,4)  
np.array([x[[0,0],[0,3]], x[[2,2],[0,3]])
```

#Zugriff auf eine Teilmatrix per slicing

```
x[::2,::3]
```

Ausgabe:

$$\begin{pmatrix} 1 & 4 \\ 9 & 12 \end{pmatrix}$$



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf eine Teilmatrix über Indize-Listen  
x = np.arange(1,13).reshape(3,4)  
np.array([x[[0,0],[0,3]], x[[2,2],[0,3]])  
  
#Zugriff auf eine Teilmatrix über die ix-Methode  
x[np.ix_([0,2],[0,3])]
```

Ausgabe:

$$\begin{pmatrix} 1 & 4 \\ 9 & 12 \end{pmatrix}$$



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf eine Teilmatrix über die ix-Methode  
x[np.ix_([0,2],[0,3])]
```



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf eine Teilmatrix über die ix-Methode  
x[np.ix_([0,2],[0,3])]
```



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf eine Teilmatrix über die ix-Methode  
x[np.ix_([0,2],[0,3])]
```



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf eine Teilmatrix über die ix-Methode  
x[np.ix_([0,2],[0,3])]
```



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf eine Teilmatrix über die ix-Methode  
x[np.ix_([0,2],[0,3])]
```

Ausgabe:

$$\begin{pmatrix} 1 & 4 \\ 9 & 12 \end{pmatrix}$$



NumPy Arrays indizieren und slicen

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
#Zugriff auf eine Teilmatrix über einen booleschen Ausdruck  
x[x % 2 == 0]
```

Ausgabe: (2 4 6 8 10 12)



NumPy Arrays indizieren und slicen

- Indize-Listen sind effizienter als Slicing, geben aber immer nur **eindimensionale** Arrays wieder
- Auch boolesche Ausdrücke können nur **eindimensionale** Arrays zurückgeben
- Slices und die `ix_()`-Methode geben **mehrdimensionale** Arrays wieder



NumPy Arrays flache und tiefe Kopien

```
import numpy as np

>>> y = [4,18,35,68,118,148]
>>> y2 = y[:3]
>>> y2
[4,18,35]
>>> y2[1] = 19
>>> y
[4,18,35,68,118,148]

>>> x = np.array([4,18,35,68,118,148])
>>> x2 = x[:3]
>>> x2
array([ 4, 18, 35])
>>> x2[1] = 19
>>> x
[4,19,35,68,118,148]
```

- Im Gegensatz zu Standardarrays erzeugt ein Slice eines NumPy-Arrays **immer** nur eine flache Kopie, d. h. kein neues Objekt wird erzeugt



NumPy Arrays flache und tiefe Kopien

```
import numpy as np

>>> x = np.array([4,18,35,68,118,148])
>>> x2 = x[3:]

>>> x == x2
False
#Ab der nächsten Pythonversion wird
#hier eine Exception geschmissen

>>> x.data == x2.data
False
#Das data-Attribut zeigt
#auf den Speicher des Arrays

>>> print(f'{x.data}\n{x2.data}')
<memory at 0x0000021C5ED04F48>
<memory at 0x0000021C5ED04E88>

>>> np.may_share_memory(x, x2)
True
```

- Die Funktion `np.may_share_memory()` vergleicht ob sich der Speicherbereich zweier NumPy-Arrays überlappt



NumPy Arrays flache und tiefe Kopien

```
import numpy as np

>>> x = np.array([4,18,35,68,118,148])
>>> x2 = np.ndarray.copy(x)

>>> x2[1] = 19
>>> x2
array([ 4, 19, 35, 68, 118, 148])

>>> x
array([ 4, 18, 35, 68, 118, 148])

>>> np.may_share_memory(x, x2)
False
```

- Die NumPy Methode *ndarray.copy()* erzeugt eine tiefe Kopie eines NumPy-Arrays, d. h. es wird ein neues Objekt mit den gleichen Werten angelegt



NumPy Arrays erzeugen

```
import numpy as np

>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])

>>> np.ones((4,5), dtype=int)
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])

>>> x = np.arange(1,13).reshape(2,2,3)

>>> np.zeros_like(x, dtype=np.float64)
array([[[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]],
       [[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
```

- Es lassen sich 0er- und 1er-Matrizen beliebiger Form erstellen



NumPy Arrays erzeugen

```
import numpy as np

>>> np.random.randint(0, 10, (5,5))
array([[8, 6, 6, 1, 5],
       [6, 2, 9, 0, 4],
       [5, 5, 5, 3, 9],
       [4, 6, 0, 3, 2],
       [5, 8, 8, 4, 5]])

>>> np.random.poisson(10, (5,5))
array([[13, 11, 6, 16, 7],
       [10, 11, 8, 10, 12],
       [8, 8, 12, 8, 8],
       [11, 13, 20, 5, 15],
       [6, 20, 13, 11, 14]])
```

- Es lassen sich Matrizen mit zufälligen Werten nach bestimmten Verteilungen erstellen



NumPy Arrays erzeugen

```
import numpy as np

>>> np.identity(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

>>> np.eye(4,5,0, dtype=np.int64)
array([[1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0]], dtype=int64)
```

- Es lassen sich auch Einheitsmatrizen beliebiger Form erzeugen



Strukturierte Arrays mit dtype erzeugen

Name	Alter	Anzahl Projekte	Größe
NM	29	6	163,2
SK	29	4	189,8
EB	27	10	189,1
KL	25	2	175,9

```
import numpy as np

>>> x = np.array([[ 'NM', 29, 6, 163.2], [ 'SK', 29, 4, 189.8], [ 'EB', 27, 10, 189.1], [ 'KL', 25, 2, 175.9]])
>>> x
array([[ 'NM', '29', '6', '163.2'],
       [ 'SK', '29', '4', '189.8'],
       [ 'EB', '27', '10', '189.1'],
       [ 'KL', '25', '2', '175.9']],
      dtype='<U5')

>>> x[0][1] + 1
TypeError: Can't convert 'int' object to str implicitly
```



NumPy Arrays erzeugen

- Mit einem *dtype*-Objekt lassen sich (fast) beliebige Datentypen definieren und einem Array zuweisen

```
import numpy as np

>>> i16 = np.dtype(np.int16)

>>> person = np.dtype([('name', 's20'), ('alter', np.int8),
                        ('projekte', np.int8), ('größe', np.float16)])

>>> person
dtype([('name', '<U'), ('alter', 'i1'), ('projekte', 'i1'), ('größe', '<f2')])

>>> x = np.array([('NM', 29, 6, 163.2), ('SK', 29, 4, 189.8),
                  ('EB', 27, 10, 189.1), ('KL', 25, 2, 175.9)], dtype=person)

>>> x[0][1] + 1
30

>>> x['projekte']
array([ 6,  4, 10,  2], dtype=int8)
```



Operationen auf NumPy Arrays

- Skalaroperationen werden wie erwartet Komponentenweise durchgeführt, und liefern immer ein **neues** Array-Objekt

```
import numpy as np

>>> x = np.array([5.1, 9.3, 6.6, 9.1, 1.9, 1.4, 9.0])
>>> x - 2.3
array([ 2.8,  7. ,  4.3,  6.8, -0.4, -0.9,  6.7])
>>> x * 6
array([ 30.6,  55.8,  39.6,  54.6,  11.4,   8.4,  54. ])

>>> x = np.arange(9).reshape(3,3)
>>> x + np.arange(9,18).reshape(3,3)
array([[ 9, 11, 13],
       [15, 17, 19],
       [21, 23, 25]])
>>> x * 1.9
array([[ 0. ,  1.9,  3.8],
       [ 5.7,  7.6,  9.5],
       [11.4, 13.3, 15.2]])
```



Operationen auf NumPy Arrays

- Fehlende Zeilen/Spalten werden automatisch aufgefüllt, in dem der Operand wiederholt wird, es wird aber immer nur **eine** Dimension aufgefüllt (um Fehler zu vermeiden)

```
import numpy as np

>>> x = np.array([[1,2,2],[1,0,1],[2,2,1]])
>>> y = np.ones(3, dtype=np.int32)
>>> x + y
array([[2, 3, 3],
       [2, 1, 2],
       [3, 3, 2]])

>>> z = np.ones(2, dtype=np.int32)
>>> x + z
ValueError: operands could not be broadcast together with shapes (3,3) (2,)
```



Operationen auf NumPy Arrays

- Der `*`-Operator führt nur eine Komponentenweise Multiplikation zwischen zwei mehrdimensionalen Matrizen aus
- Für die echte Matrixmultiplikation gibt es die `np.dot()`-Funktion

```
import numpy as np

>>> x = np.arange(9).reshape(3,3)
>>> y = np.arange(9,18).reshape(3,3)
>>> x * y
array([[ 0, 10, 22],
       [36, 52, 70],
       [90, 112, 136]])

>>> np.dot(x,y)
array([[ 42,  45,  48],
       [150, 162, 174],
       [258, 279, 300]])
```



Achsenweite Funktionen auf NumPy Arrays

- Achsenweite Funktionen werden über eine oder mehrere (oder alle) Dimensionen hinweg angewandt
- Als Parameter werden immer die Dimensionen angegeben, über die die Funktion angewandt werden soll

```
import numpy as np

>>> x = np.array([[2,3,1,9],[5,6,12,8],[1,2,5,0]])
>>> x
array([[ 2,  3,  1,  9],
       [ 5,  6, 12,  8],
       [ 1,  2,  5,  0]])

>>> np.max(x)
12
>>> np.max(x, 0)
array([ 5,  6, 12,  9])
>>> np.max(x, 1)
array([ 9, 12,  5])
>>> np.sum(x, (0,1))
54
```



Achsenweite Funktionen auf NumPy Arrays

- Achsenweite Funktionen werden über eine oder mehrere (oder alle) Dimensionen hinweg angewandt
- Als Parameter werden immer die Dimensionen angegeben, über die die Funktion angewandt werden soll

```
import numpy as np

>>> x = np.array([[2,3,1,9],[5,6,12,8],[1,2,5,0]])
>>> x
array([[ 2,  3,  1,  9],
       [ 5,  6, 12,  8],
       [ 1,  2,  5,  0]])

>>> np.all(x > 19)
False
>>> np.any(x < 1, 2)
array([False, False,  True], dtype=bool)
```



NumPy Arrays transponieren

```
import numpy as np

>>> x = np.array([[2,3,1,9],[5,6,12,8],[1,2,5,0]])
>>> np.transpose(x)
array([[ 2,  5,  1],
       [ 3,  6, 12],
       [ 1, 12,  5],
       [ 9,  8,  0]])

>>> x.T
array([[ 2,  5,  1],
       [ 3,  6, 12],
       [ 1, 12,  5],
       [ 9,  8,  0]])
```



NumPy Arrays konkatenieren

- Mit *np.concatenate()* lassen sich Arrays um andere Arrays in einer beliebigen Dimension erweitern, solange alle anderen Dimensionen die gleiche Länge haben

```
import numpy as np

>>> x = np.array([[2,3,1,9],[5,6,12,8],[1,2,5,0]])
>>> y = np.arange(9,18).reshape(3,3)
>>> np.concatenate((x,y),1)
array([[ 2,  3,  1,  9,  9, 10, 11],
       [ 5,  6, 12,  8, 12, 13, 14],
       [ 1,  2,  5,  0, 15, 16, 17]])

>>> np.concatenate((x,y),0)
ValueError: all the input array dimensions except for the concatenation axis
must match exactly
```



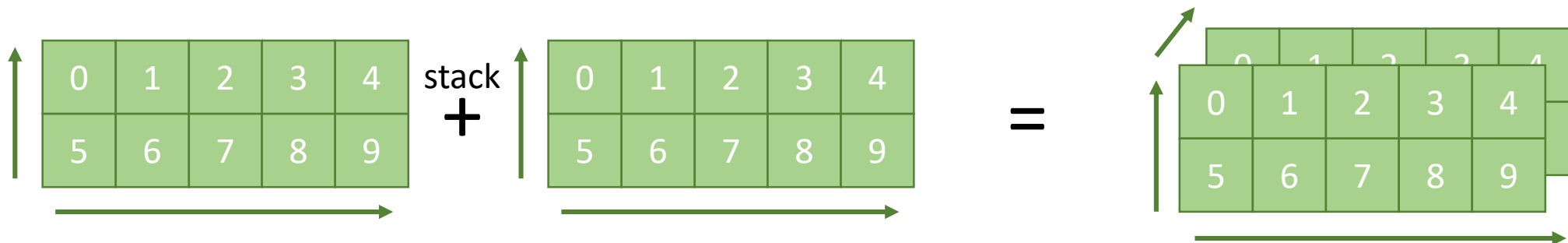
NumPy Arrays konkatenieren

- Mit *np.stack()* lassen sich zwei Arrays mit dem gleichem Shape zu einem Array mit einer weiteren Dimension zusammenfügen

```
import numpy as np

>>> x = np.arange(10).reshape(2,5)
>>> y = np.arange(10).reshape(2,5)
>>> np.stack((x,y))
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]],

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



NumPy Arrays konkatenieren

- Mit *np.newaxis* lässt sich ein Array um eine Dimension erweitern

```
import numpy as np

>>> x = np.array([5,19,36,69,119,149])
>>> y = x[:,np.newaxis]
>>> y
array([[ 5],
       [19],
       [36],
       [69],
       [119],
       [149]])
```



NumPy Arrays sortieren

- Mit *np.sort()* lassen sich Arrays achsenweise sortieren

```
import numpy as np

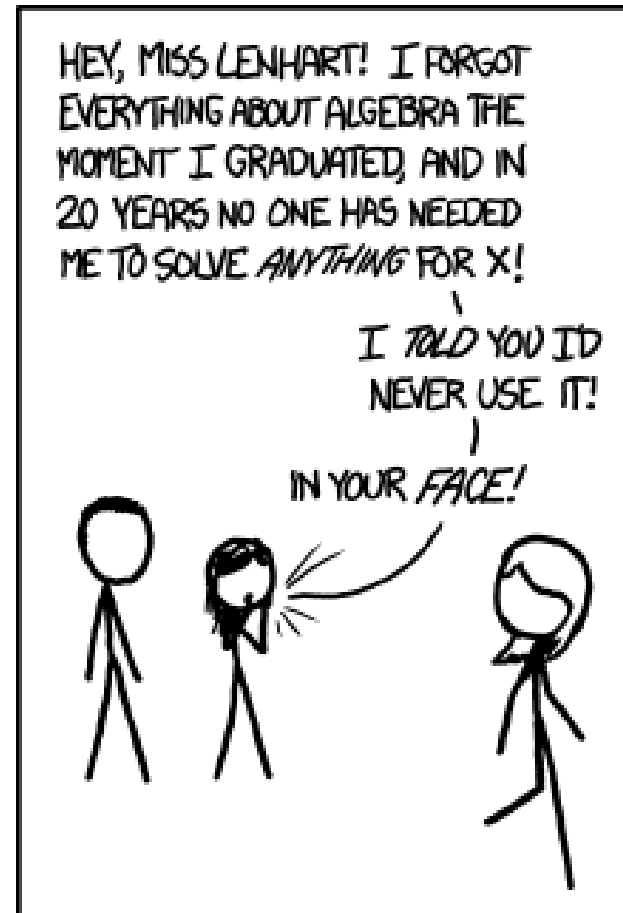
>>> x = np.array([[2,3,1,9],[5,6,12,8],[1,2,5,0]])
>>> x
array([[ 2,  3,  1,  9],
       [ 5,  6, 12,  8],
       [ 1,  2,  5,  0]])

>>> np.sort(x)
array([[ 1,  2,  3,  9],
       [ 5,  6,  8, 12],
       [ 0,  1,  2,  5]])

>>> np.sort(x, 0)
array([[ 1,  2,  1,  0],
       [ 2,  3,  5,  8],
       [ 5,  6, 12,  9]])
```



Lineare Algebra mit NumPy



IT'S WEIRD HOW PROUD PEOPLE ARE OF NOT LEARNING MATH WHEN THE SAME ARGUMENTS APPLY TO LEARNING TO PLAY MUSIC, COOK, OR SPEAK A FOREIGN LANGUAGE.



Lineare Algebra mit NumPy

- Die *matrix*-Klasse ist ein Wrapper über die *array*-Klasse und kann nahezu gleich verwendet werden
- Die ***- und ****-Operationen sind so überschrieben das sie Matrixmultiplikationen ausführen (nicht Komponentenweise)
- Oft wird (leider) viel zwischen Matrix und Arrays hin- und hergecastet

```
import numpy as np

>>> x = np.matrix([[2,3,1],[5,6,12],[1,2,5]])
>>> x*x
matrix([[ 20,  26,  43],
        [ 52,  75, 137],
        [ 17,  25,  50]])

>>> x**3
matrix([[ 213,  302,  547],
        [ 616,  880, 1637],
        [ 209,  301,  567]])
```



Lineare Algebra mit NumPy

- NumPy unterstützt viele Funktionen und Algorithmen der Linearen Algebra, die sich direkt auf den Array- und Matrixobjekten anwenden lassen

```
import numpy as np

>>> x = np.matrix([[2,3,1],[5,6,12],[1,2,5]])
>>> np.linalg.eig(x)
(array([ 11.82016663, -0.92462244,  2.10445581]),
matrix([[ -0.30650748, -0.68940485, -0.88797034],
        [ -0.900331   ,  0.71359219, -0.17296339],
        [ -0.30896158, -0.12452769,  0.42613652]]))

>>> np.linalg.det(x)
-23.0

>>> np.linalg.inv(x)
matrix([[ -0.26086957,  0.56521739, -1.30434783],
        [  0.56521739, -0.39130435,  0.82608696],
        [ -0.17391304,  0.04347826,  0.13043478]])
```



Lineare Algebra mit NumPy

- NumPy unterstützt viele Funktionen und Algorithmen der Linearen Algebra, die sich direkt auf den Array- und Matrixobjekten anwenden lassen

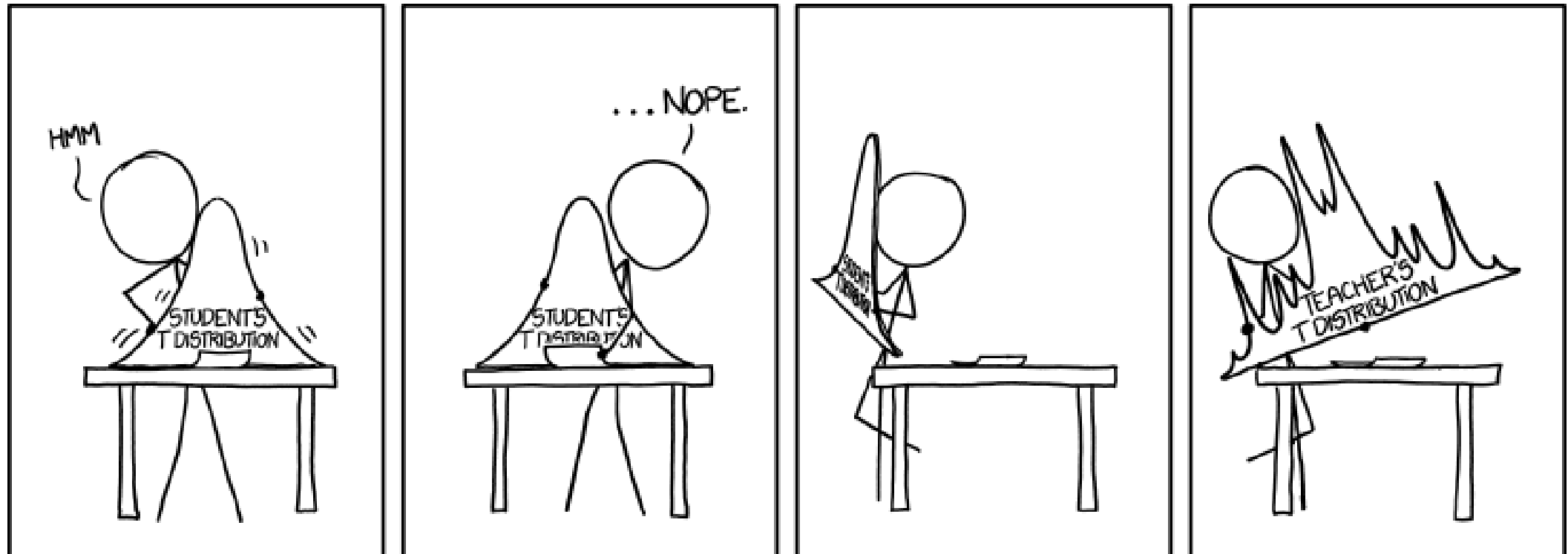
```
import numpy as np

>>> variablen = np.array([[1,3],[3,4]])
>>> koeffizienten = np.array([30, 20])
>>> lösung = np.linalg.solve(variablen, koeffizienten)
>>> lösung
array([-12.,  14.])
```

$$\begin{aligned}x_0 + 3x_1 &= 30 \\ 3x_0 + 4x_1 &= 20\end{aligned}$$



Statistik mit SciPy



Statistik mit SciPy

- SciPy ist auch ein Modul voller mathematischer Funktionen und Algorithmen
- Enthält alle möglichen Wahrscheinlichkeitsverteilungen und statistische Methoden

```
import scipy.stats as st

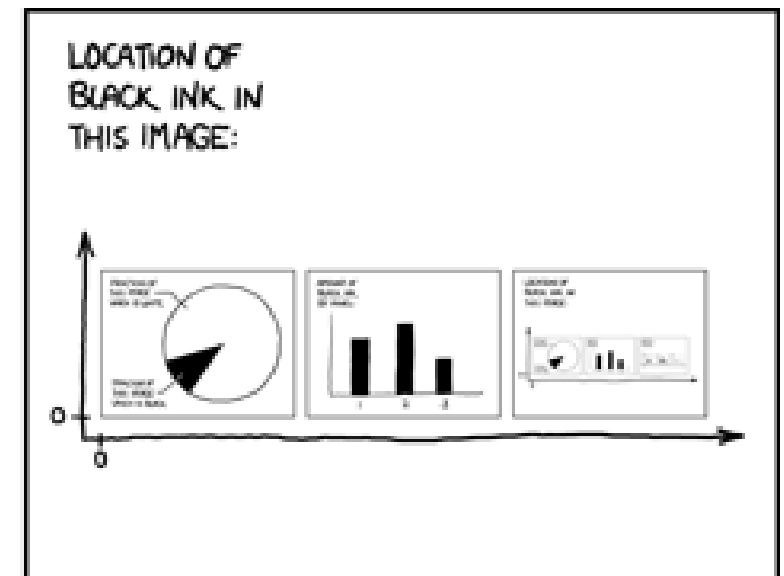
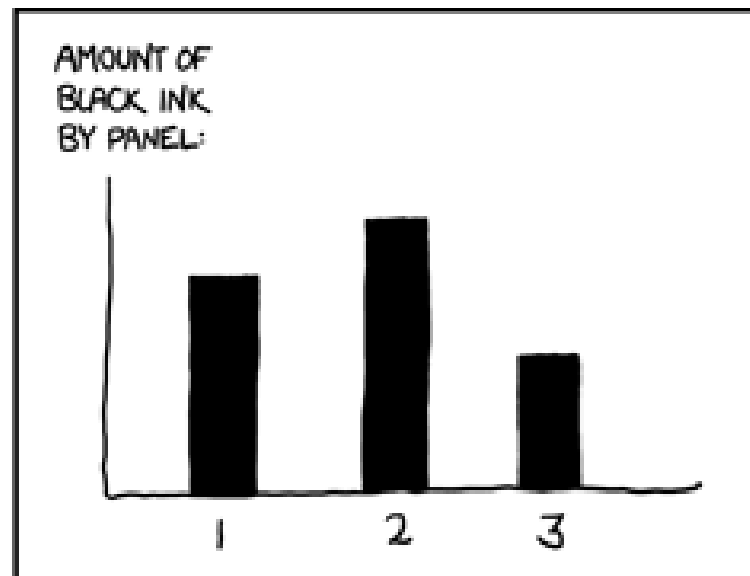
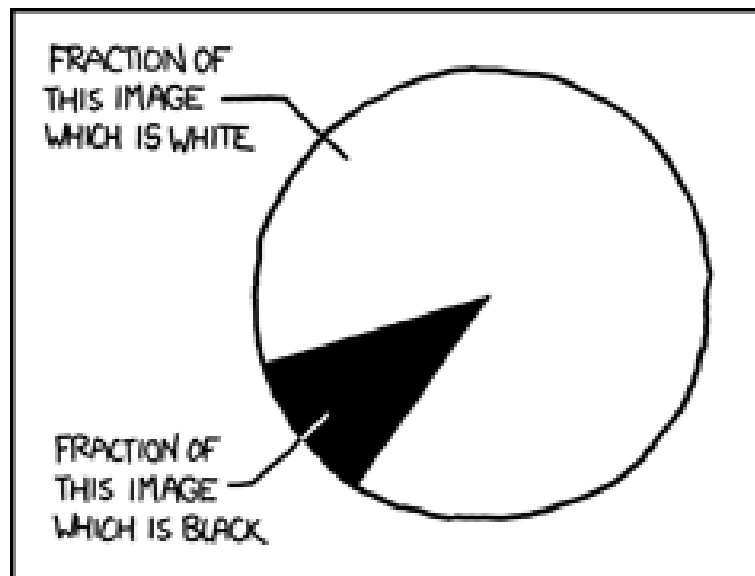
>>> N = st.norm(5, 2)           #Normalverteilung mit mean=5 und std=2
>>> N.cdf(2)                    #Kumulative Wahrscheinlichkeit von N
>>> N.pdf([1,5,7])              #Wahrscheinlichkeitsdichte für bestimmte Punkte

>>> P = st.pareto(1, 0, 2)      #Paretoverteilung
>>> a = P.rvs(100)              #Generiere 100 Zahlen aus der Verteilung P

>>> parameter = st.pareto.fit(someData) #Lerne die Verteilungsparameter aus someData
```



Matplotlib



Matplotlib

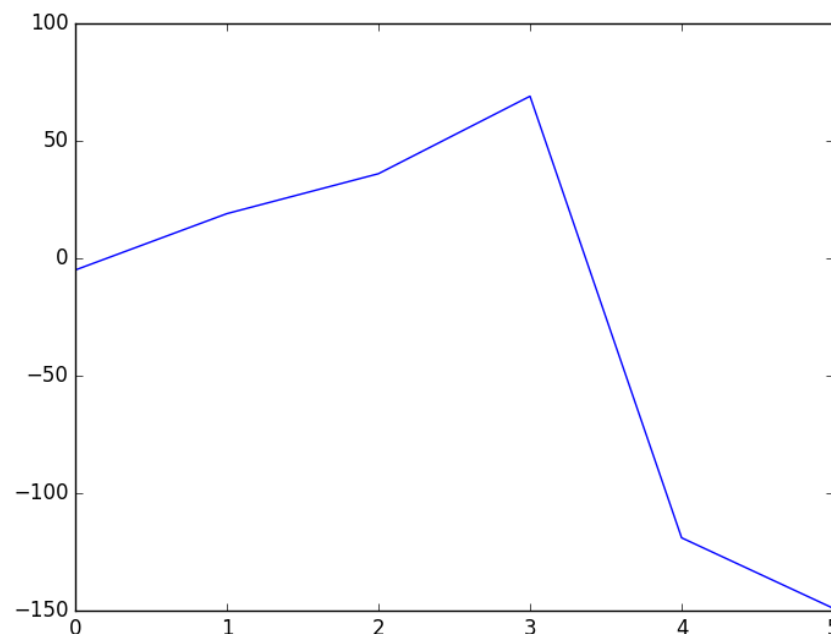
- Ist ein Modul zur Erzeugung von Grafiken und Diagrammen und ist stark an MATLAB angelehnt
- Vorteil gegenüber MATLAB:
 - Kostenlos
 - Quelloffen
 - Objektorientiert
- „Matplotlib versucht Einfaches einfach und Schweres möglich zu machen. Man kann mit nur wenigen Codezeilen Plots, Histogramme, Leistungsspektren, Balkendiagramme, Fehlerdiagramme, Streudiagramme / Punktwolken, und so weiter erzeugen.“ – www.matplotlib.org



Matplotlib

- Die `plot()`-Funktion interpretiert die übergebenen Werte als Y-Koordinaten und die Indizes der Werte als X-Koordinaten

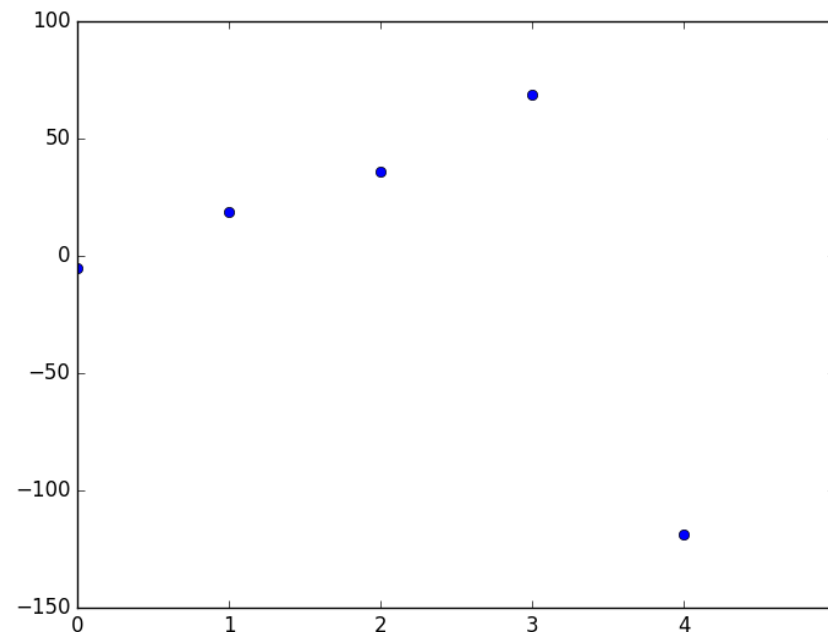
```
import matplotlib.pyplot as plt  
  
>>> plt.plot([-5, 19, 36, 69, -119, -149])  
[<matplotlib.lines.Line2D object at 0x0000021c630747f0>]  
>>> plt.show()
```



Matplotlib

- Mit Formatierungsstrings kann man das Erscheinungsbild der geplotteten Daten verändern

```
import matplotlib.pyplot as plt  
  
>>> plt.plot([-5, 19, 36, 69, -119, -149], 'ob')  
[<matplotlib.lines.Line2D object at 0x0000021C630747F0>]  
>>> plt.show()
```



Matplotlib

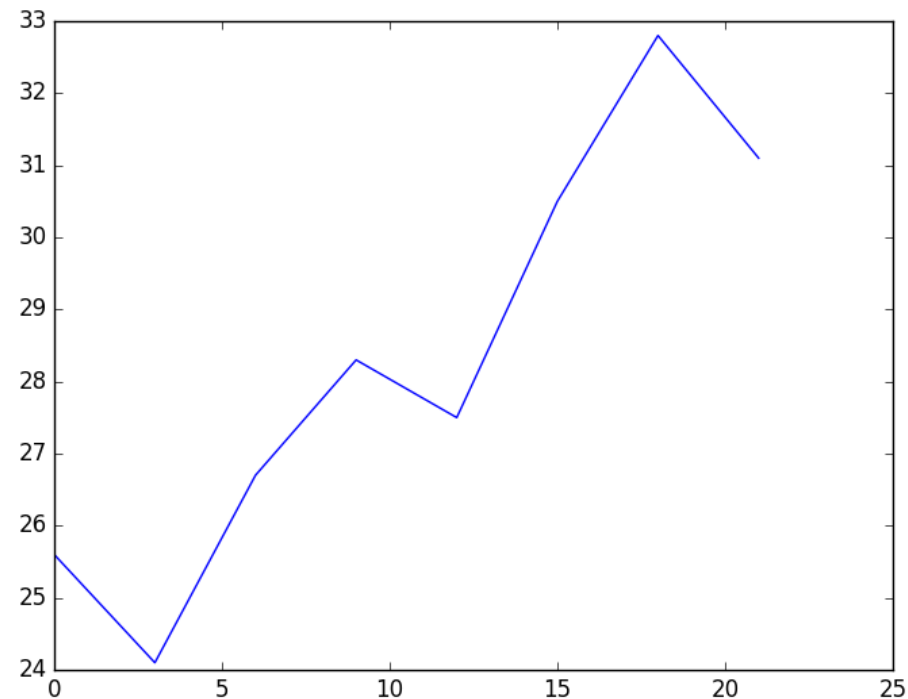
Zeichen	Beschreibung
-	Durchgezogene Linie
--	Gestrichelte Linie
-.	Strichpunkt-Linie
,	Pixelmarker
o	Kreismarker
x	X-Marker
*	Sternmarker
D	Rautenmarker
v	Dreiecksmarker
Zeichen	Farbe
b	Blau
g	Grün
r	Rot
k	Schwarz



Matplotlib

```
import matplotlib.pyplot as plt

>>> tage = list(range(0,22,3))
>>> celsius = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 31.1]
>>> plt.plot(tage, celsius)
>>> plt.show()
```



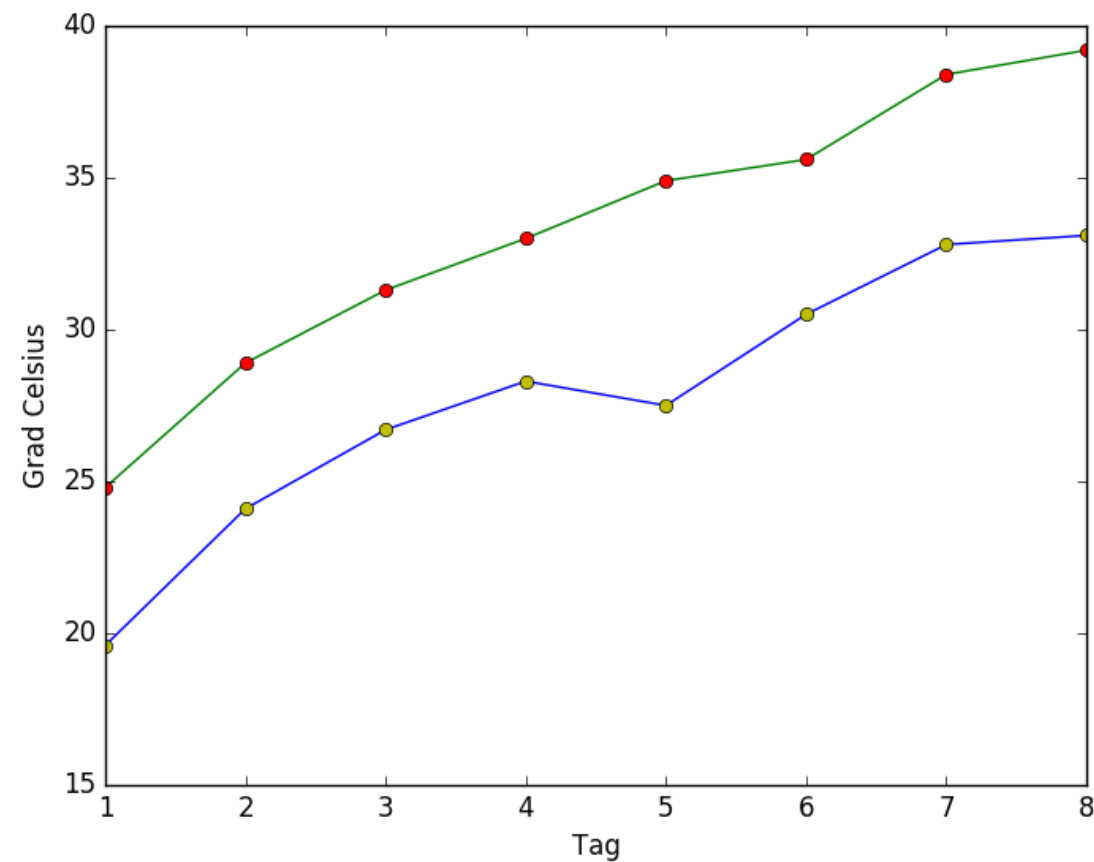
Matplotlib

```
import matplotlib.pyplot as plt

>>> tage = list(range(1,9))
>>> plt.ylabel('Grad Celsius')
>>> celsiusMin = [19.6, 24.1, 26.7, 28.3,
                  27.5, 30.5, 32.8, 33.1]
>>> celsiusMax = [24.8, 28.9, 31.3, 33.0,
                  34.9, 35.6, 38.4, 39.2]

>>> plt.xlabel('Tag')
>>> plt.ylabel('Grad Celsius')
>>> plt.plot(tage, celsiusMin, 'oy',
              tage, celsiusMax, 'or')

>>> plt.show()
```

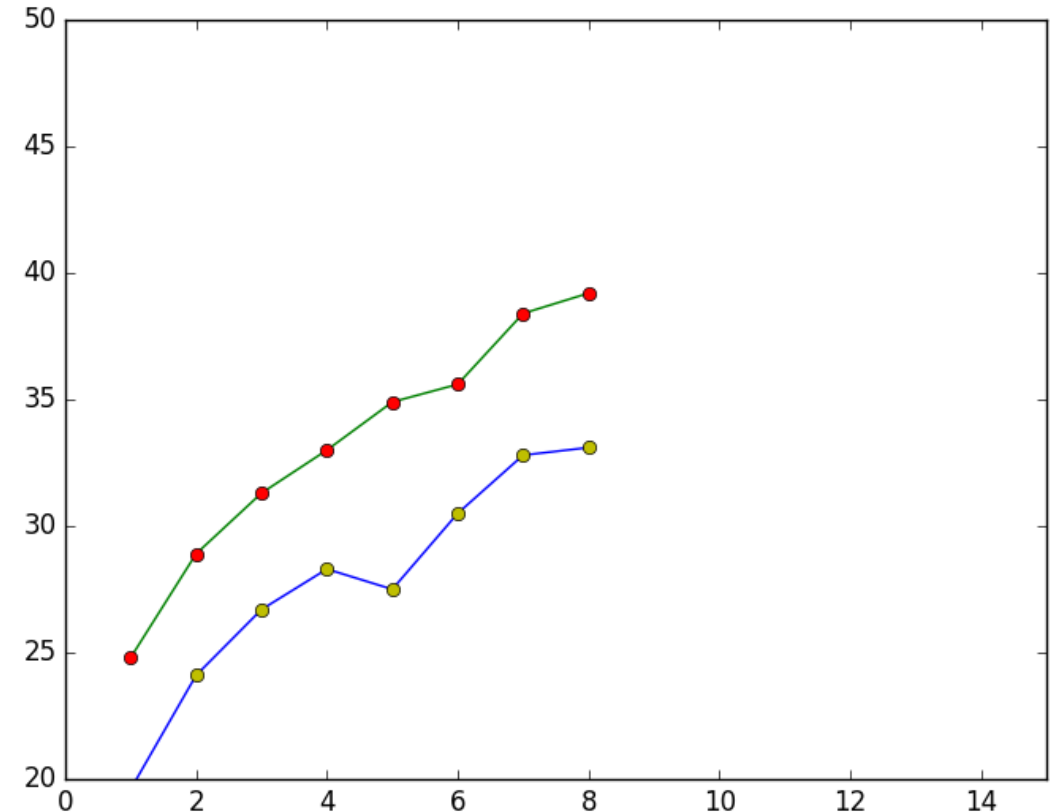


Matplotlib

- Mit der Funktion `axis()` lässt sich der Wertebereich eines Plots abfragen und ändern

```
import matplotlib.pyplot as plt

#Code wie vorher
>>> plt.axis()
(1.0, 8.0, 15.0, 40.0)
>>> plt.axis([0, 15, 20, 50])
>>> plt.show()
```

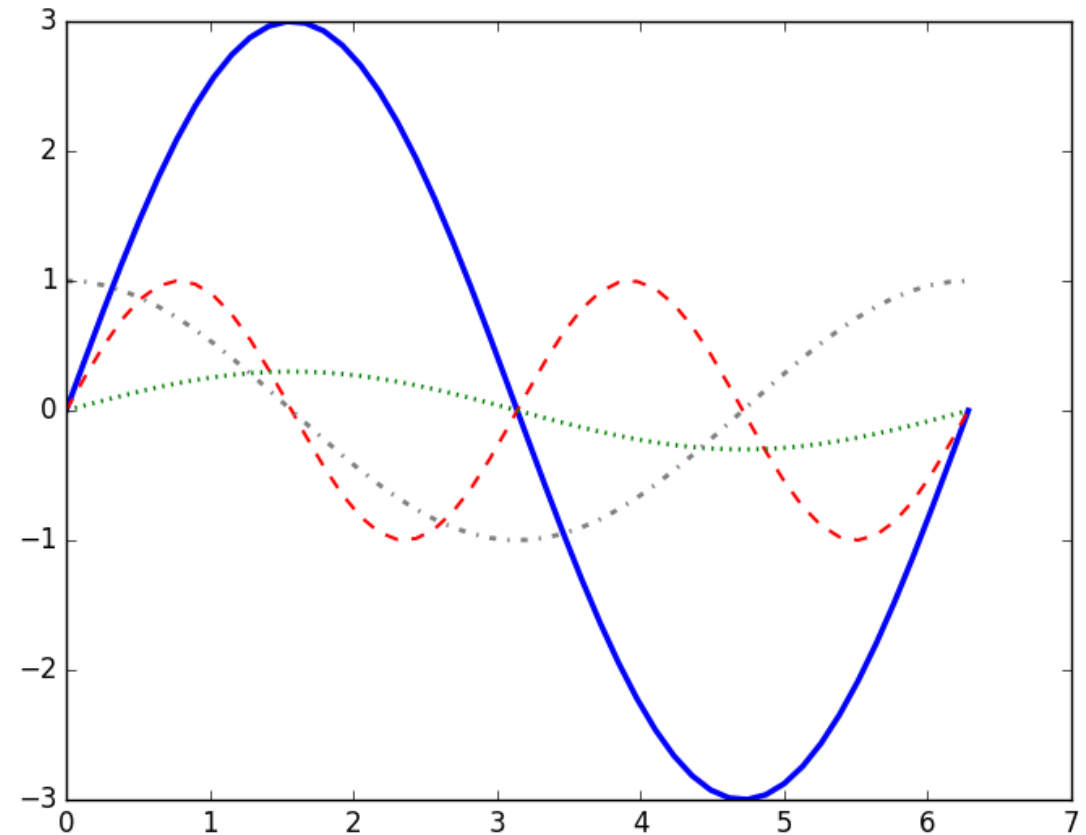


Matplotlib

```
import matplotlib.pyplot as plt

>>> x = np.linspace(0, 2 * np.pi, 50,
                    endpoint=True)

>>> F1 = 3 * np.sin(x)
>>> F2 = np.sin(2*x)
>>> F3 = 0.3 * np.sin(x)
>>> F4 = np.cos(x)
>>> plt.plot(x, F1, color="blue",
             linewidth=2.5, linestyle="-")
>>> plt.plot(x, F2, color="red",
             linewidth=1.5, linestyle="--")
>>> plt.plot(x, F3, color="green",
             linewidth=2, linestyle=":")
>>> plt.plot(x, F4, color="grey",
             linewidth=2, linestyle="-.")
>>> plt.show()
```

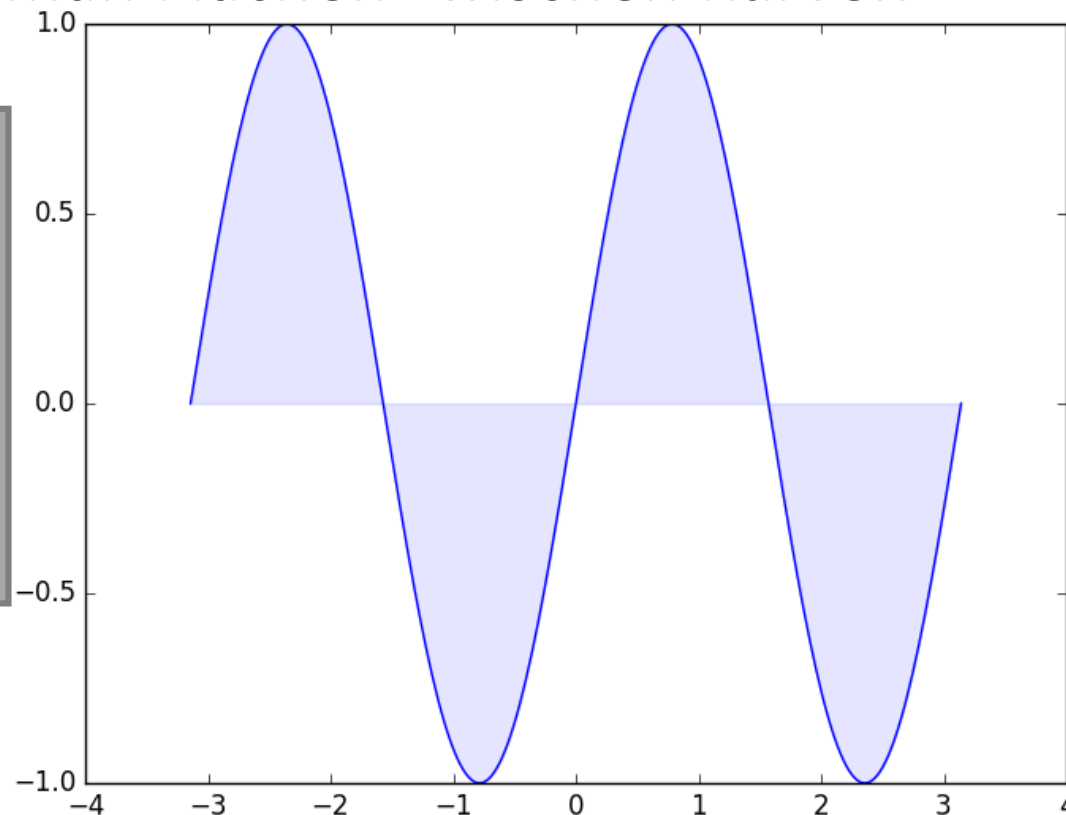


Matplotlib

- Mit der Funktion *fill_between()* kann man Flächen zwischen Kurven und Achsen einfärben

```
import matplotlib.pyplot as plt

>>> n = 256
>>> X = np.linspace(-np.pi, np.pi, n)
>>> Y = np.sin(2*X)
>>> plt.plot(X, Y, color='blue', alpha=1.00)
>>> plt.fill_between(X, 0, Y, color='blue',
>>>                  alpha=.1)
>>> plt.show()
```



fill_between(x, y1, y2=0, where=None, interpolate=False)

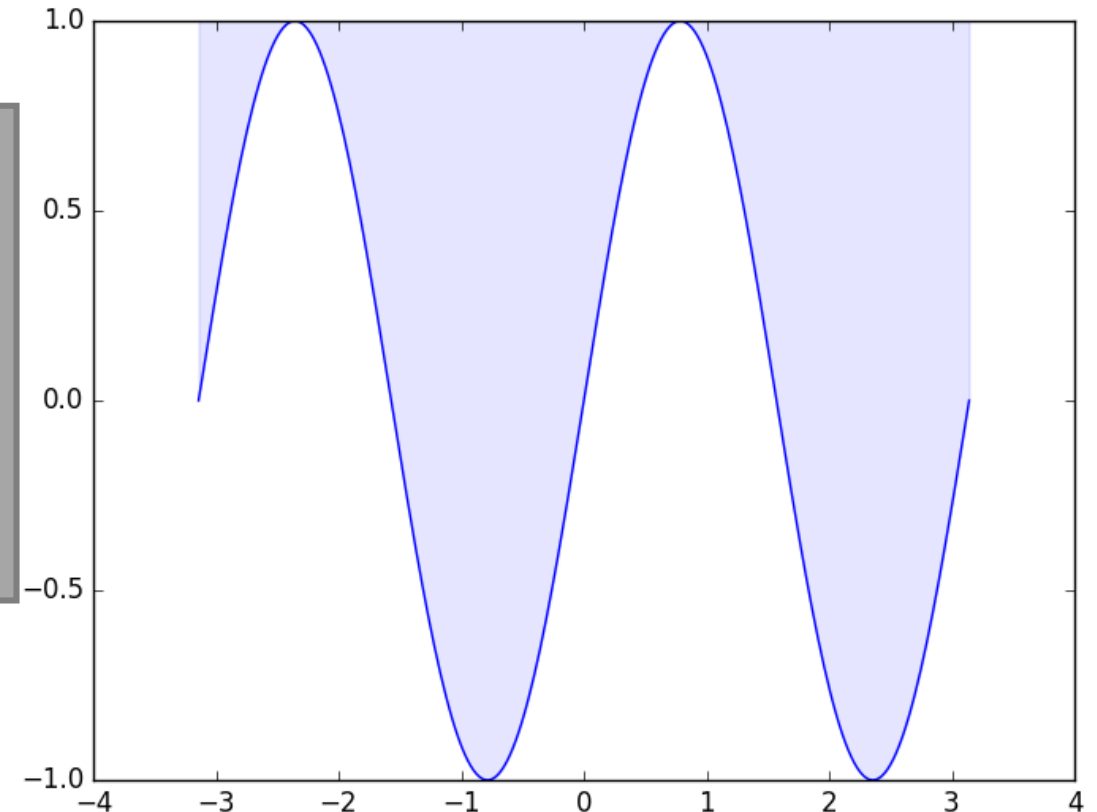


Matplotlib

- Mit der Funktion *fill_between()* kann man Flächen zwischen Kurven und Achsen einfärben

```
import matplotlib.pyplot as plt

>>> n = 256
>>> X = np.linspace(-np.pi, np.pi, n)
>>> Y = np.sin(2*X)
>>> plt.plot(X, Y, color='blue', alpha=1.00)
>>> plt.fill_between(X, Y, 1, color='blue',
>>>                  alpha=.1)
>>> plt.show()
```



fill_between(x, y1, y2=0, where=None, interpolate=False)



Matplotlib – Spines und Ticks

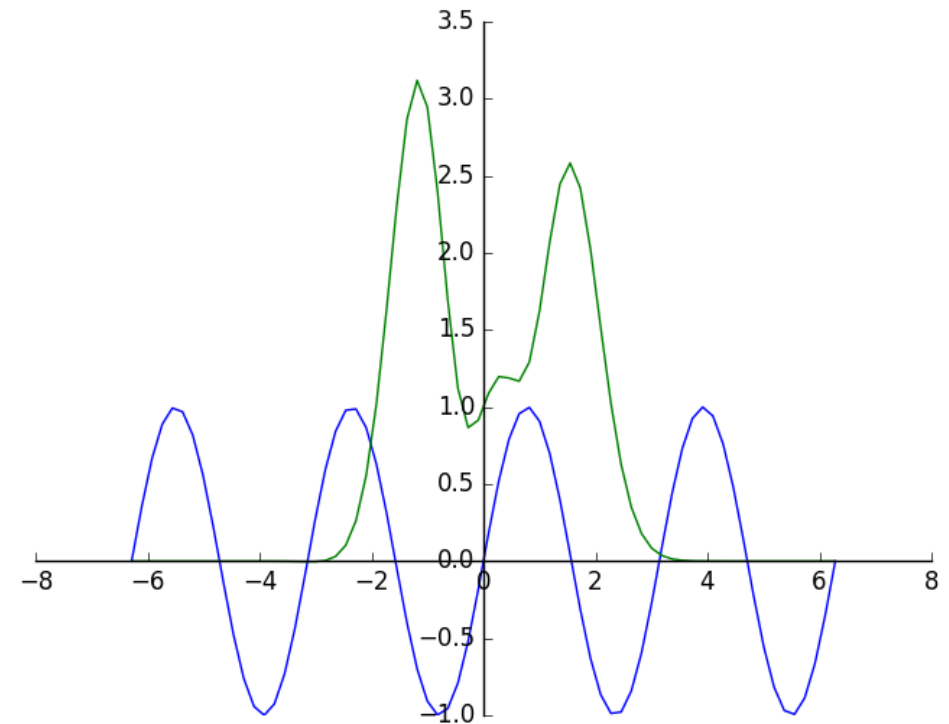
- Spines (Achsenmarkierungen) und Ticks (Abstandsmarkierungen) jeder Plotachse können beliebig angepasst werden
- `gca()` gibt die Referenzen der Achsenobjekte des Plots wieder

```
import matplotlib.pyplot as plt

>>> X = np.linspace(-2 * np.pi, 2 * np.pi,
                    70, endpoint=True)

>>> F1 = np.sin(2 * X)
>>> F2 = (2*X**5 + 4*X**4 - 4.8*X**3
          + 1.2*X**2 + X + 1)*np.exp(-X**2)

ax = plt.gca()
>>> ax.spines['top'].set_color('none')
>>> ax.spines['right'].set_color('none')
>>> ax.xaxis.set_ticks_position('bottom')
>>> ax.spines['bottom'].set_position(('data',0))
>>> ax.yaxis.set_ticks_position('left')
>>> ax.spines['left'].set_position(('data',0))
>>> plt.plot(X, F1)
>>> plt.plot(X, F2)
>>> plt.show()
```

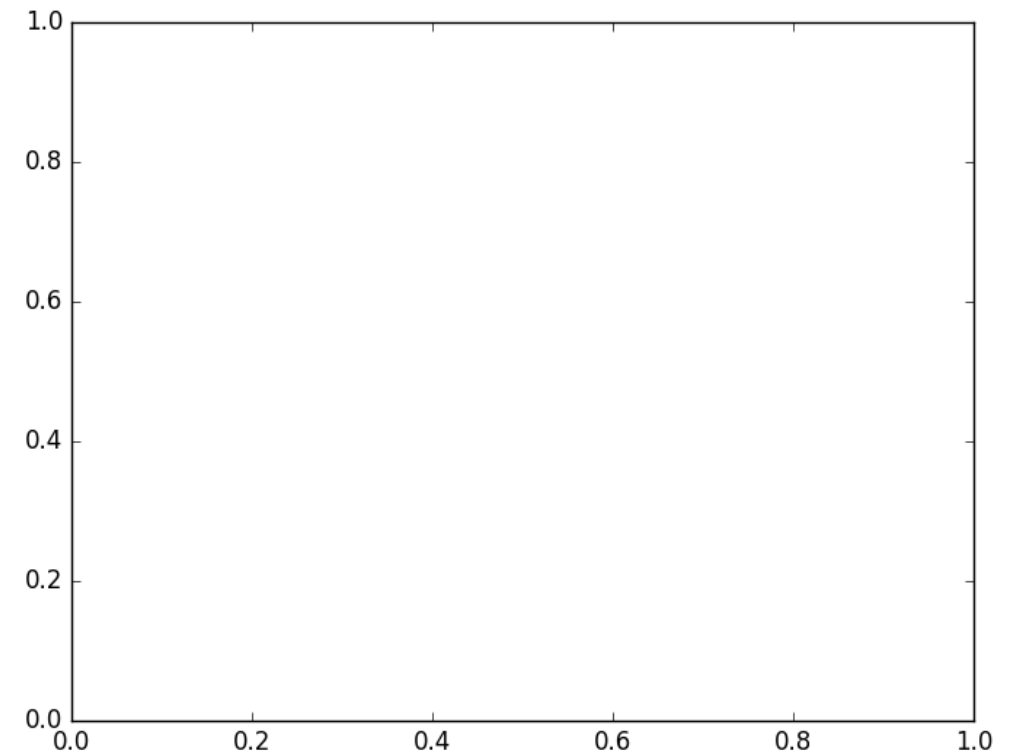


Matplotlib – Spines und Ticks

- Bis jetzt hat Matplotlib automatisch den Abstand der Punkte auf den Achsen festgelegt, aber mit den Methoden *xticks()/yticks()* lässt sich auch das individuell anpassen

```
import matplotlib.pyplot as plt

>>> ax = plt.gca()
>>> locs, labels = plt.xticks()
>>> print(locs, labels)
[ 0.  0.2  0.4  0.6  0.8  1. ]
<a list of 6 Text xticklabel objects>
>>> plt.show()
```

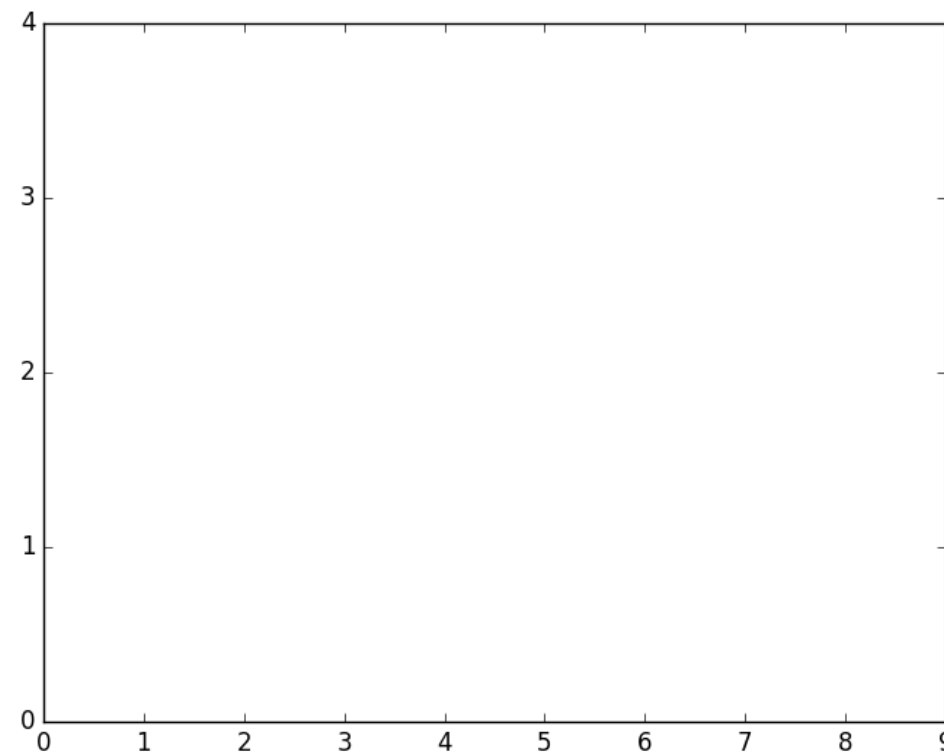


Matplotlib – Spines und Ticks

- Bis jetzt hat Matplotlib automatisch den Abstand der Punkte auf den Achsen festgelegt, aber mit den Methoden *xticks()/yticks()* lässt sich auch das individuell anpassen

```
import matplotlib.pyplot as plt

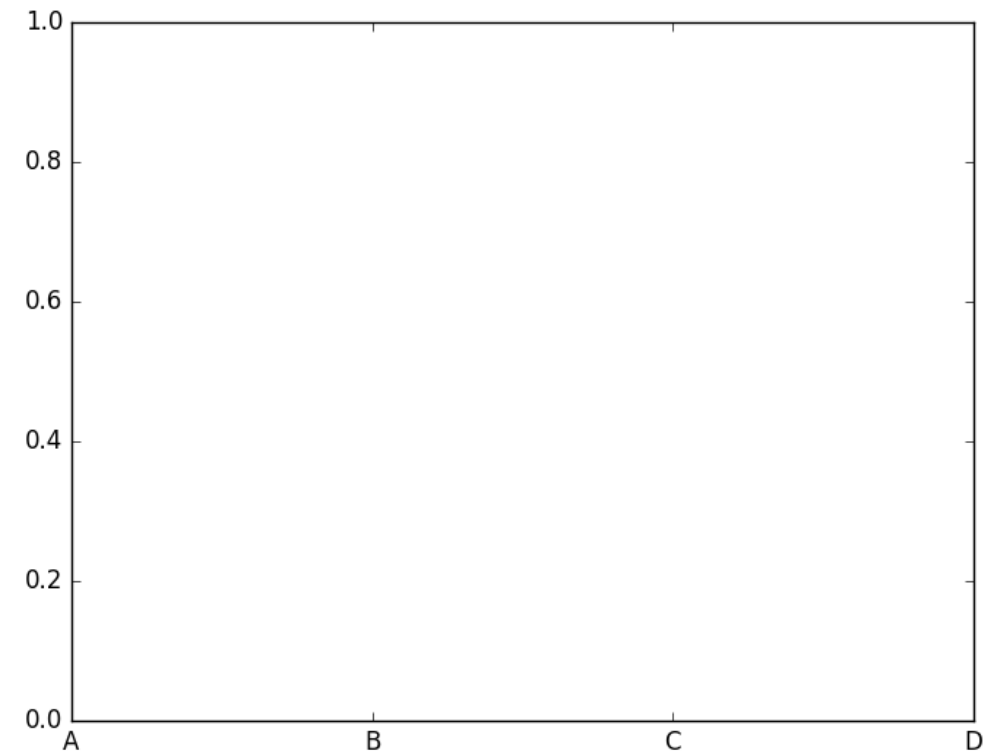
>>> ax = plt.gca()
>>> plt.xticks(np.arange(10))
>>> plt.yticks(np.arange(5))
>>> plt.show()
```



Matplotlib – Spines und Ticks

- Bis jetzt hat Matplotlib automatisch den Abstand der Punkte auf den Achsen festgelegt, aber mit den Methoden *xticks()/yticks()* lässt sich auch das individuell anpassen

```
import matplotlib.pyplot as plt  
  
>>> ax = plt.gca()  
>>> plt.xticks(np.arange(4), ('A', 'B', 'C', 'D'))  
>>> plt.show()
```

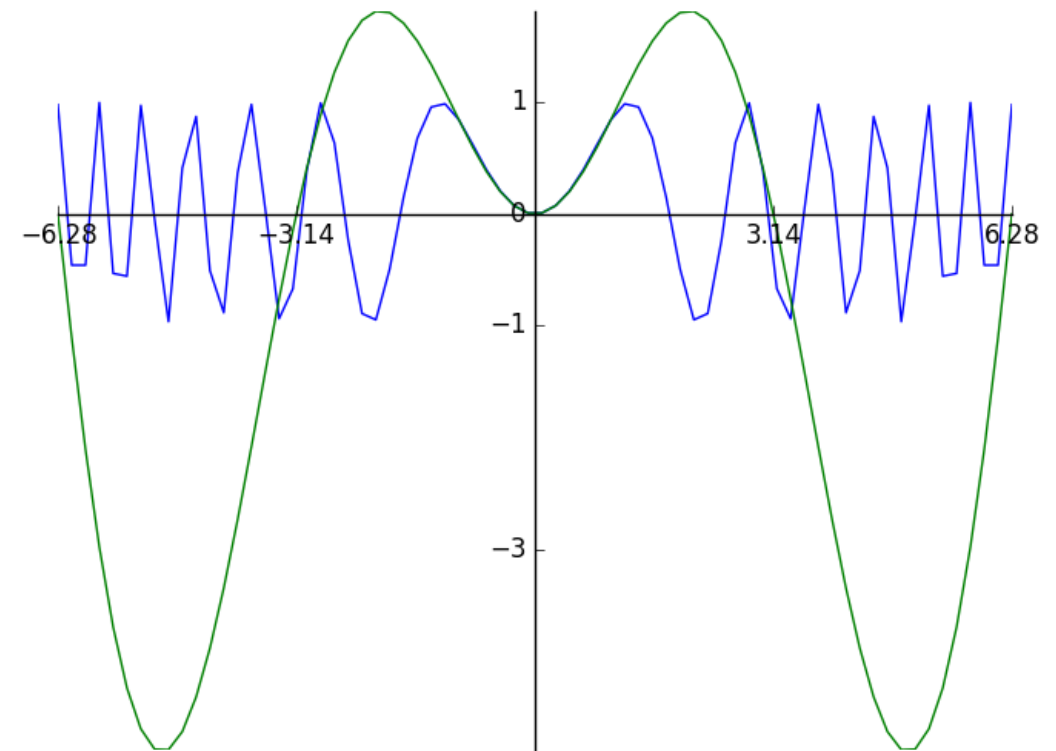


Matplotlib – Spines und Ticks

```
import matplotlib.pyplot as plt

>>> X = np.linspace(-2 * np.pi, 2 * np.pi, 70,
                    endpoint=True)

>>> F1 = np.sin(X**2)
>>> F2 = X * np.sin(X)
>>> ax = plt.gca()
>>> ax.spines['top'].set_color('none')
>>> ax.spines['right'].set_color('none')
>>> ax.xaxis.set_ticks_position('bottom')
>>> ax.spines['bottom'].set_position(('data',0))
>>> ax.yaxis.set_ticks_position('left')
>>> ax.spines['left'].set_position(('data',0))
>>> plt.xticks([-6.28, -3.14, 3.14, 6.28])
>>> plt.yticks([-3, -1, 0, +1, 3])
>>> plt.plot(X, F1)
>>> plt.plot(X, F2)
>>> plt.show()
```



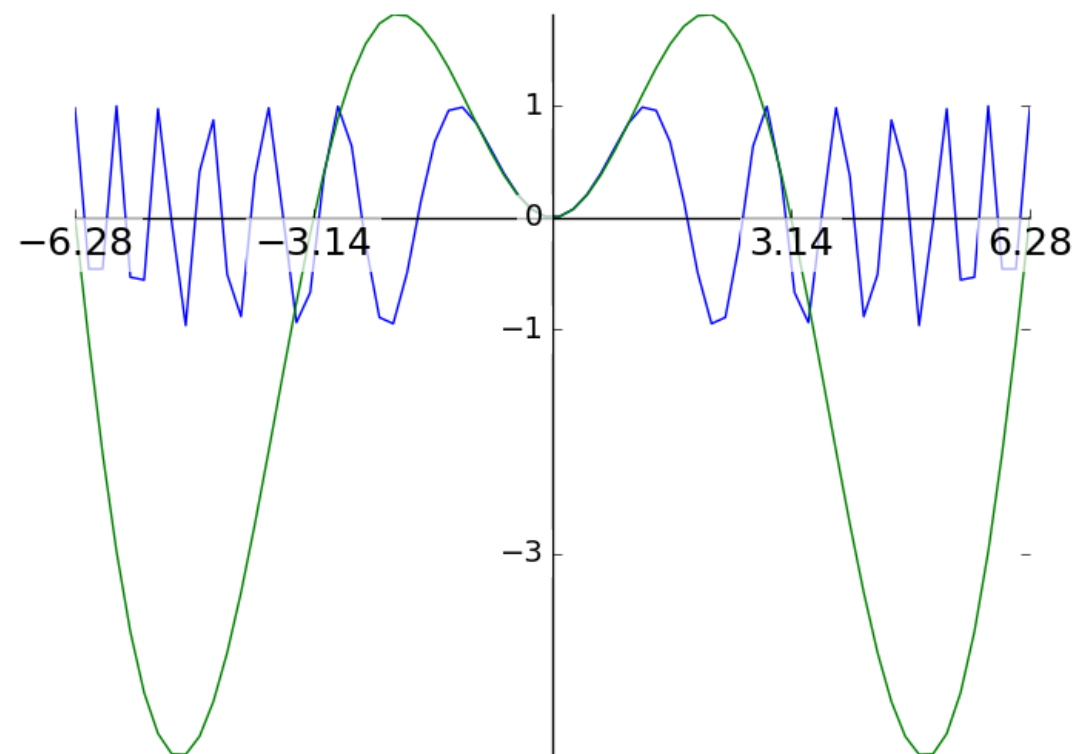
Matplotlib – Spines und Ticks

```
import matplotlib.pyplot as plt

#Code wie vorher
>>> for xtick in ax.get_xticklabels():
    xtick.set_fontsize(18)
    xtick.set_bbox(dict(facecolor='white',
                        edgecolor='None', alpha=0.7 ))

>>> for ytick in ax.get_yticklabels():
    ytick.set_fontsize(14)
    ytick.set_bbox(dict(facecolor='white',
                        edgecolor='None', alpha=0.7 ))

>>> plt.plot(X, F1)
>>> plt.plot(X, F2)
>>> plt.show()
```

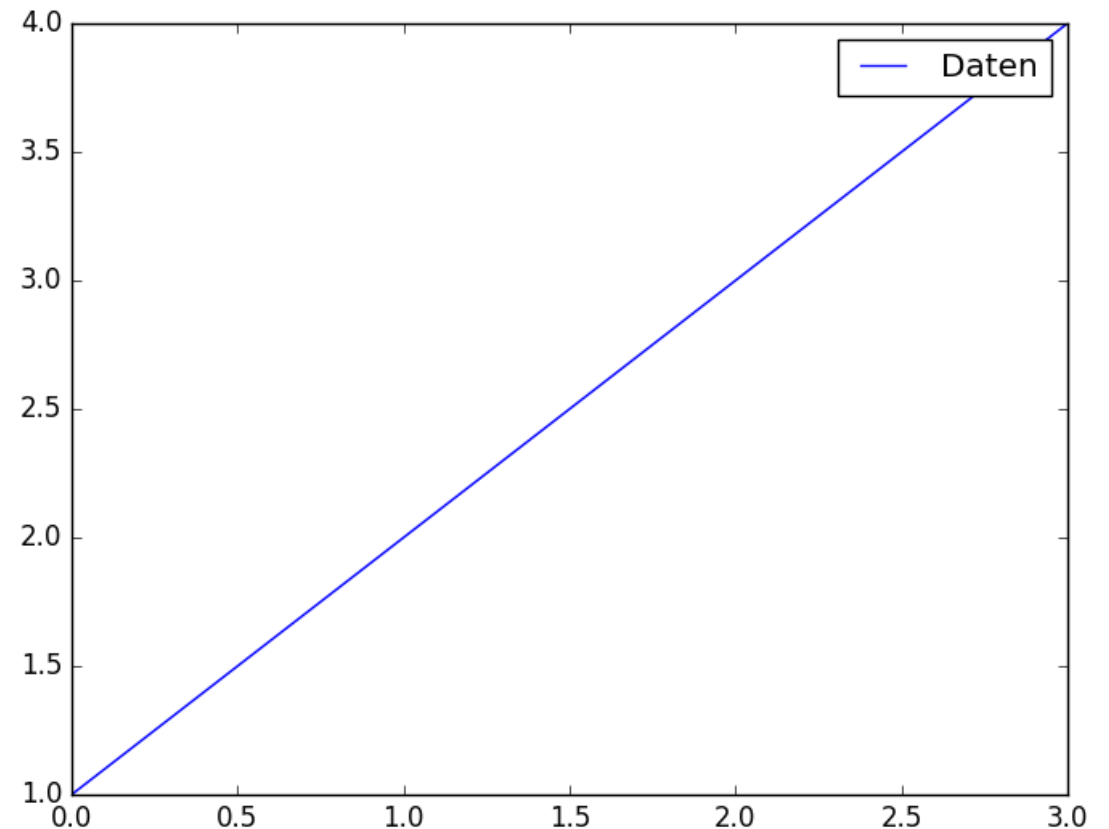


Matplotlib – Legenden

- Um einem Plot eine Legende hinzuzufügen gibt es die einfache Methode *legend()*

```
import matplotlib.pyplot as plt

>>> ax = plt.gca()
>>> ax.plot([1, 2, 3, 4])
>>> ax.legend(['Daten'])
>>> plt.show()
```

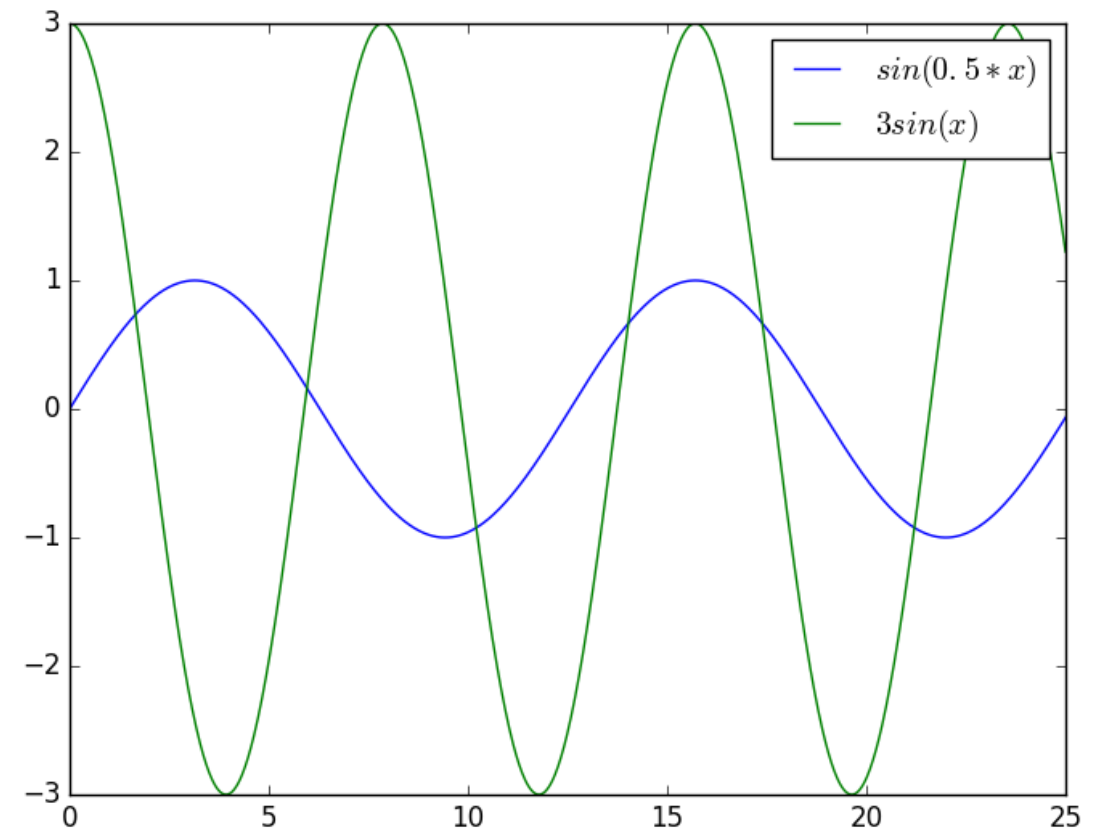


Matplotlib – Legenden

- Die Methode *legend()* besitzt viele Parameter um die Position und das Aussehen der zu zeichnenden Legende zu ändern

```
import matplotlib.pyplot as plt

>>> X = np.linspace(0, 25, 1000)
>>> F1 = np.sin(0.5 * X)
>>> F2 = 3 * np.cos(0.8*X)
>>> plt.plot(X, F1, label="$sin(0.5 * x)$")
>>> plt.plot(X, F2, label="$3 sin(x)$")
>>> plt.legend(loc='upper right')
>>> plt.show()
```

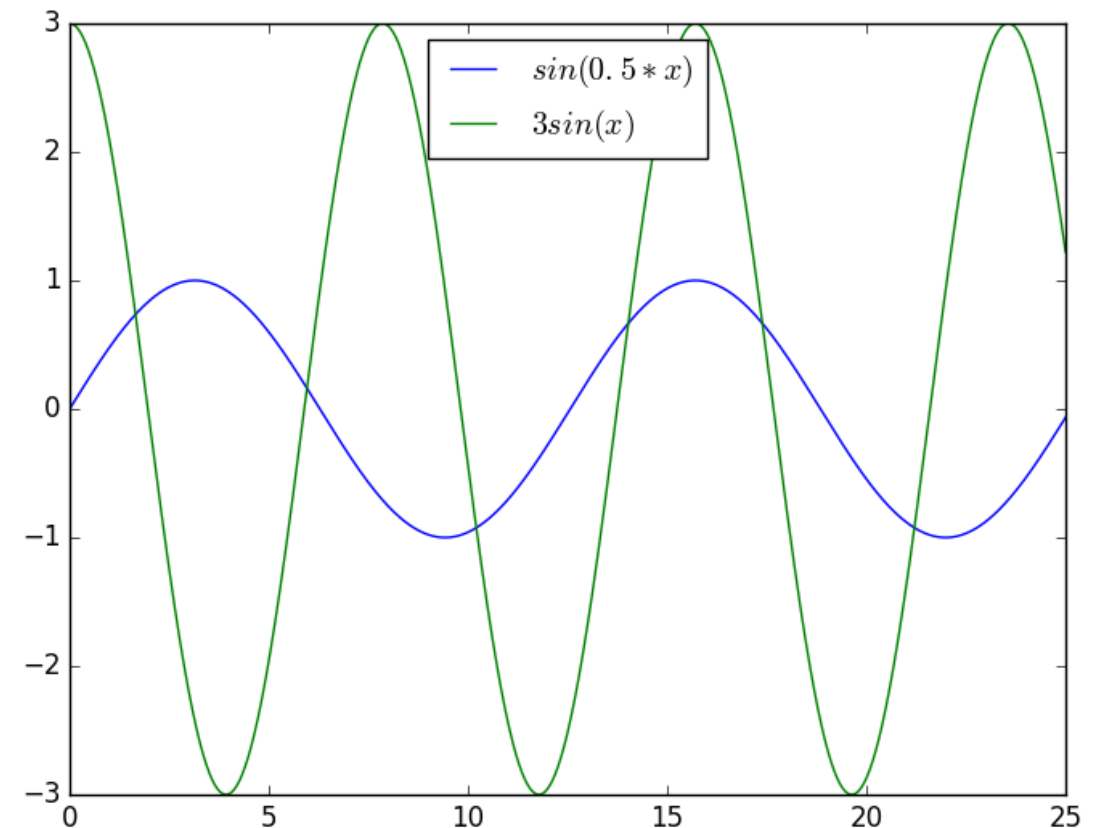


Matplotlib – Legenden

- Die Methode *legend()* besitzt viele Parameter um die Position und das Aussehen der zu zeichnenden Legende zu ändern

```
import matplotlib.pyplot as plt

>>> X = np.linspace(0, 25, 1000)
>>> F1 = np.sin(0.5 * X)
>>> F2 = 3 * np.cos(0.8*X)
>>> plt.plot(X, F1, label="$sin(0.5 * x)$")
>>> plt.plot(X, F2, label="$3 sin(x)$")
>>> plt.legend(loc='best')
>>> plt.show()
```

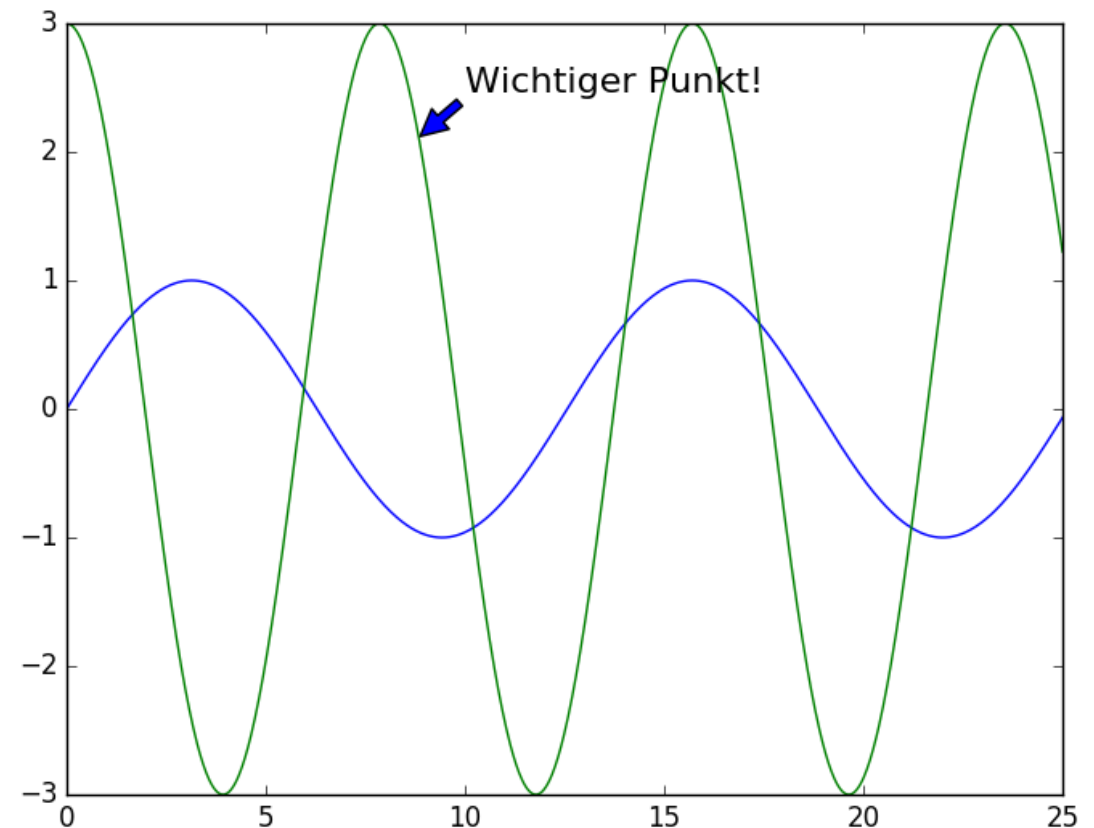


Matplotlib – Legenden

- Mit der Methode *annotate()* lassen sich Anmerkungen direkt in einen Graphen plotten

```
import matplotlib.pyplot as plt

>>> X = np.linspace(0, 25, 1000)
>>> F1 = np.sin(0.5 * X)
>>> F2 = 3 * np.cos(0.8*X)
>>> plt.plot(X, F1)
>>> plt.plot(X, F2)
>>> plt.annotate('Wichtiger Punkt!',
                xy=(11.3*np.pi/4, 3*np.sin(3*np.pi/4)),
                xycoords='data',
                xytext=(+20, +20),
                textcoords='offset points',
                fontsize=16,
                arrowprops=dict(facecolor='blue'))
>>> plt.show()
```



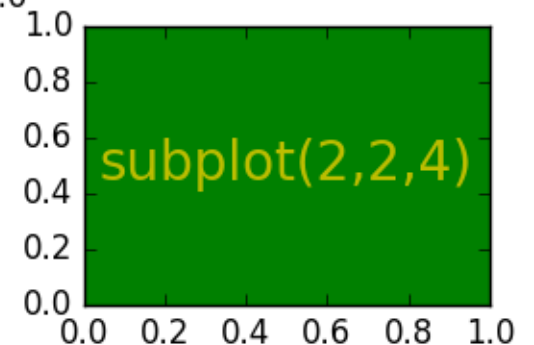
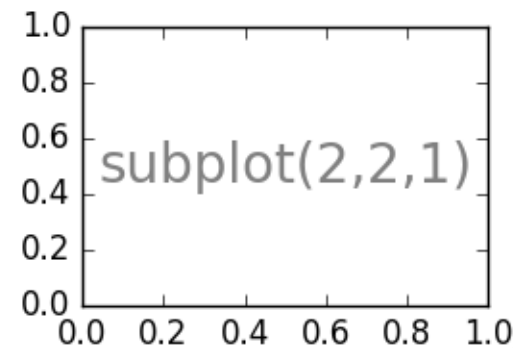
Matplotlib – Subplots

- Um mehrere Plots in einem Graphen unterzubringen gibt es u. A. die *subplots()*-Funktion um das Layout des Graphen zu verwalten

subplots(nrows, ncols, plotNumber)

```
import matplotlib.pyplot as plt

>>> plt.figure(figsize=(6, 4))
>>> plt.subplot(221)
>>> plt.text(0.5, 0.5, 'subplot(2,2,1)',
            horizontalalignment='center',
            verticalalignment='center',
            fontsize=20,
            alpha=0.5)
>>> plt.subplot(224, axisbg='g')
>>> plt.text(0.5, 0.5, 'subplot(2,2,4)',
            ha='center', va='center', font=20,
            alpha=0.5)
>>> plt.show()
```



Matplotlib – Subplots

- Um Matplotlib und im besonderen Subplots effektiv benutzen zu können sollte man eigentlich immer objektorientiert arbeiten
- Jede Matplotlib-Funktion die ein grafisches Element plottet, erzeugt im Hintergrund immer ein Objekt dieses grafischen Elements
- Über diese Objekte lässt sich dann einfach auf alle seine Attribute zugreifen (ähnlich wie bei den Widget-Objekten von Tkinter)



Matplotlib – Subplots

```
import matplotlib.pyplot as plt

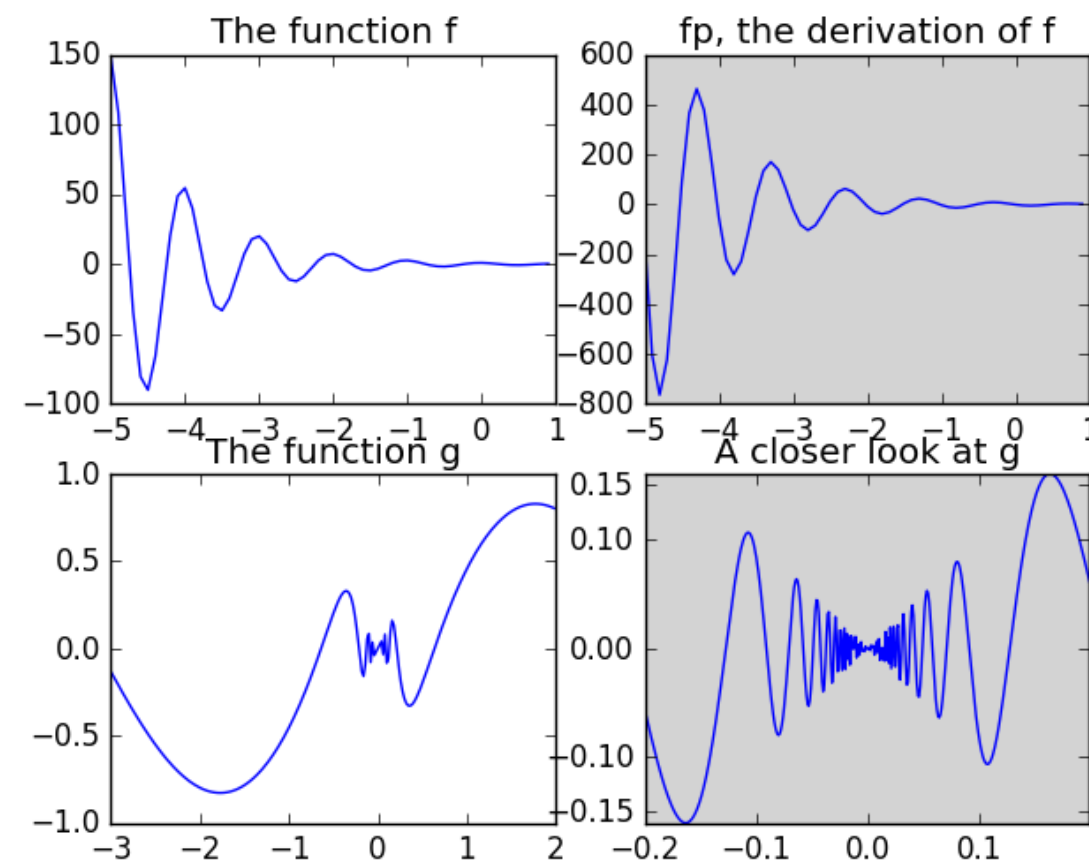
fig = plt.figure(figsize=(6, 4))
sub1 = fig.add_subplot(221)
sub1.set_title('The function f')
sub1.plot(t, f(np.arange(-5.0, 1.0, 0.1)))

sub2 = fig.add_subplot(222, axisbg="grey")
sub2.set_title('fp, the derivation of f')
sub2.plot(t, fp(t))

sub3 = fig.add_subplot(223)
sub3.set_title('The function g')
sub3.plot(t, g(np.arange(-3.0, 2.0, 0.02)))

sub4 = fig.add_subplot(224, axisbg="grey")
sub4.set_title('A closer look at g')
sub4.set_xticks([-0.2, -0.1, 0, 0.1, 0.2])
sub4.set_yticks([-0.15, -0.1, 0, 0.1, 0.15])
sub4.plot(t, g(np.arange(-0.2, 0.2, 0.001)))

plt.show()
```



Matplotlib – Subplots

```
import matplotlib.pyplot as plt

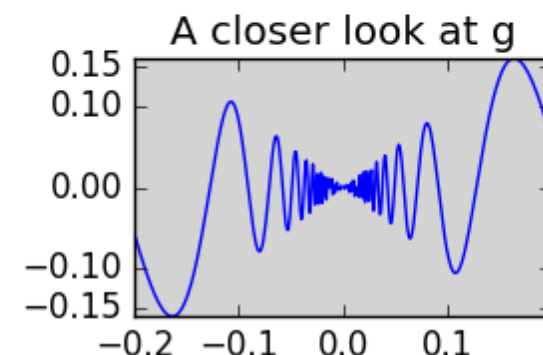
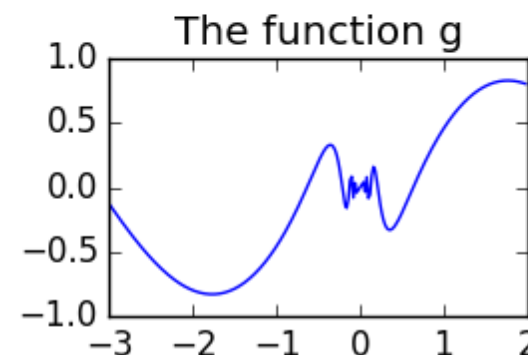
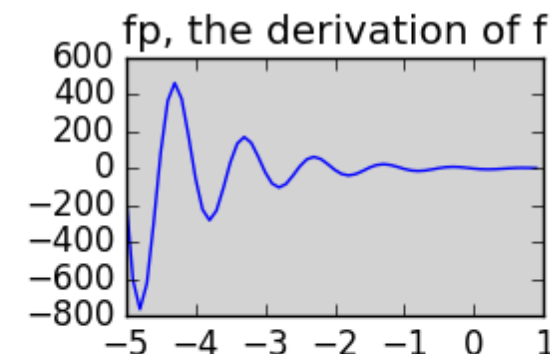
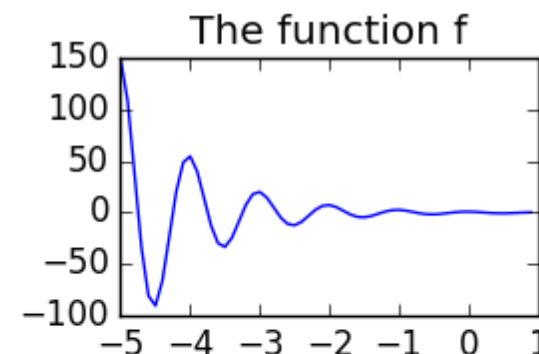
fig = plt.figure(figsize=(6, 4))
sub1 = fig.add_subplot(221)
sub1.set_title('The function f')
sub1.plot(t, f(np.arange(-5.0, 1.0, 0.1)))

sub2 = fig.add_subplot(222, axisbg="grey")
sub2.set_title('fp, the derivation of f')
sub2.plot(t, fp(t))

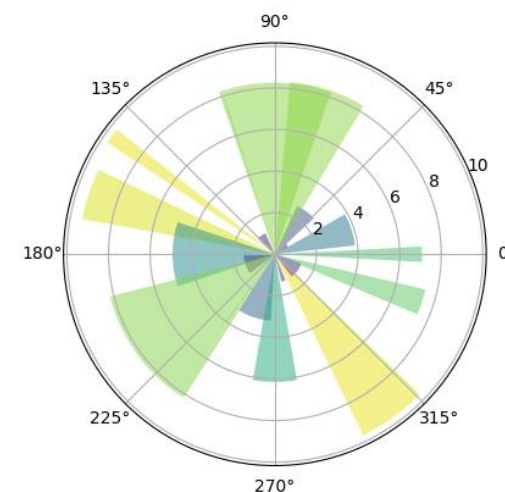
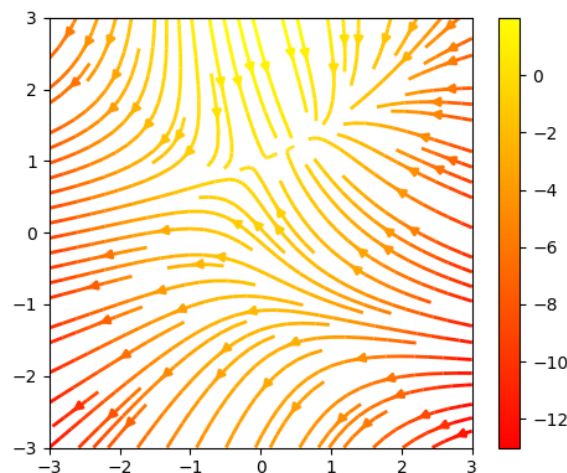
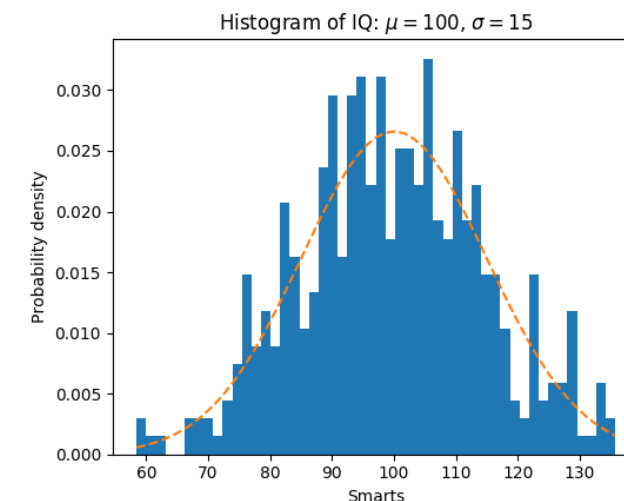
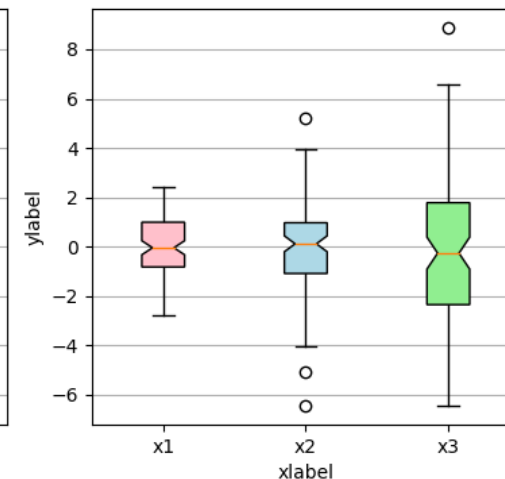
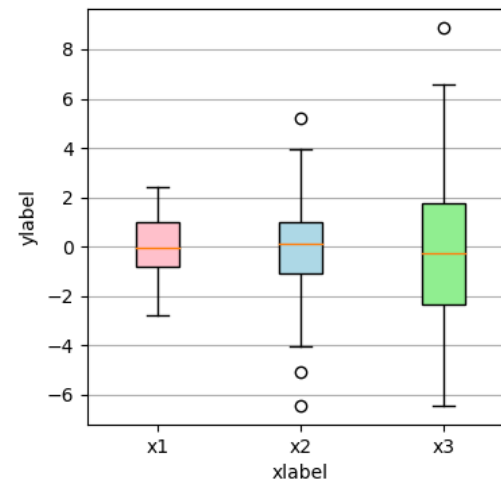
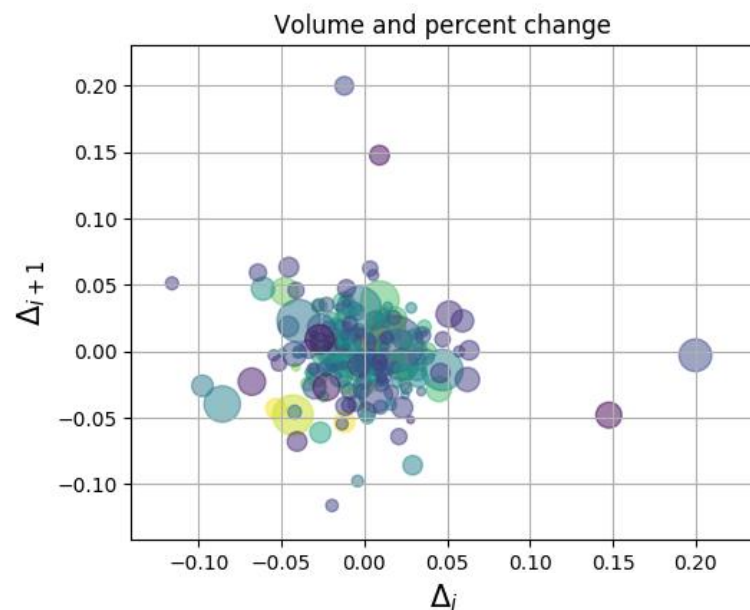
sub3 = fig.add_subplot(223)
sub3.set_title('The function g')
sub3.plot(t, g(np.arange(-3.0, 2.0, 0.02)))

sub4 = fig.add_subplot(224, axisbg="grey")
sub4.set_title('A closer look at g')
sub4.set_xticks([-0.2, -0.1, 0, 0.1, 0.2])
sub4.set_yticks([-0.15, -0.1, 0, 0.1, 0.15])
sub4.plot(t, g(np.arange(-0.2, 0.2, 0.001)))

plt.tight_layout()
plt.show()
```



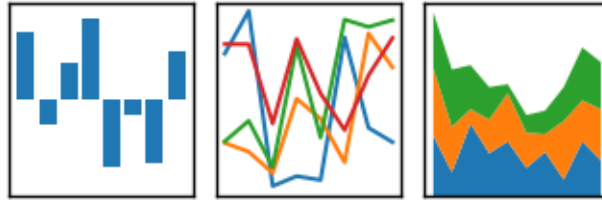
Matplotlib – Was sonst noch möglich ist



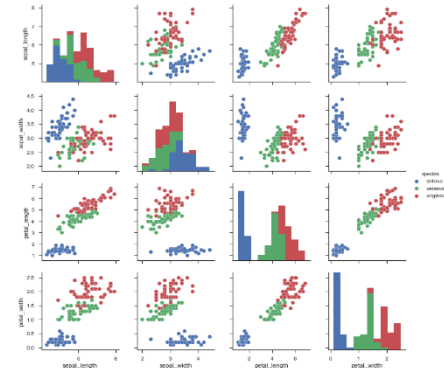
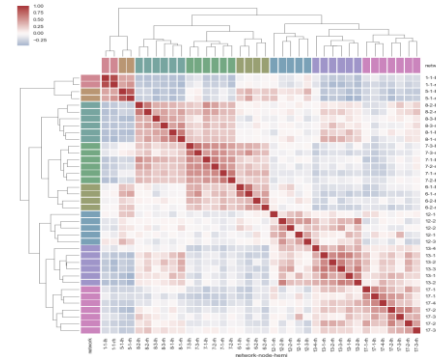
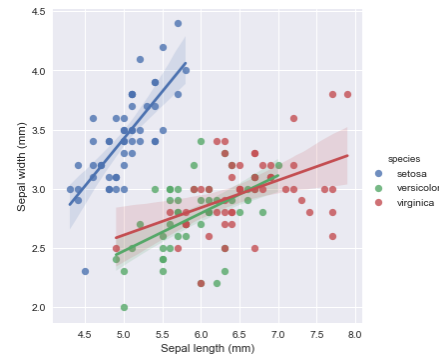
Noch mehr Module für das wissenschaftliche Arbeiten

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



seaborn



bokeh



