Intro_Python

July 3, 2020

Contents

Chapter 1

Introduction into Python for geotechnical engineers

Outline for today

1. Python - What is it and how to get it?
2. Programming paradigms
3. Basic programming tasks
4. Object oriented programming
5. Scientific programming
6. Special tasks

1.1 Python - What is it and how to get it?

- original developed by Guido van Rossum (1991)
- high-level programming language
- general-purpose programming
- ***easy to learn & easy to use***

1.1.1 Python versions

Two major versions in the wild:

- Python 2.7.x ***!!! not longer supported !!!***
- Python 3.6/7/8.x

Not fully compatible and different standard library.
All supported versions: `https://devguide.python.org/devcycle/`

1.1.2 Python implementations

Python language is interpreted (contrary to compiled languages as C, C++, Fortran, ...).
The interpreter itself for:

- Linux-Systems: `python2` or `python3`
- Windows-Systems: `python.exe`

Reference implementation:
CPython
Others:

- Jython
- IronPython
- PyPy

1.1.3 Anaconda - A Python-Distribution

Various ways to get a running Python-Interpreter on Windows:

- Download install package from `https://www.python.org/downloads/`
- Download via Windows Store (e.g. `https://www.microsoft.com/store/productId/9MSSZTT1N39L`)
- Download a Linux-System and run it via WSL (Windows Subsystem for Linux, e.g. `https://www.microsoft.com/store/productId/9NBLGGH4MSV6`)
- ***Install a Python-Distribution (like Anaconda)*** `https://www.anaconda.com/distribution/`

1.1.4 Anaconda - A Python-Distribution

- package manager for Python and R
- includes Python-Interpreter
- includes all necessary packages for data-analysis and scientific data processing
- easy to set-up and doesn't require admin rights

  Highlights:

- Data processing: numpy, scipy, pandas
- Visualization: matplotlib, seaborn, vtk
- Cloud: jupyter
- Documentation: pandoc
- HPC: cython, mpi4py
- GUIs: pyqt, tkinter
- IDE: Spyder

1.2 Programming paradigms

Several available / supported:

- imperative
- structural
- procedural
- modular
- functional
- object-oriented

1.2.1 Imperative programming

- a set of statements will modify a programs state

In [1]:
```
# comments start with '#'
# python does not now special floating types like C, only float
# python does not now special integer types like C, only int
a = 3.0  # dynamic typing --> float
b = 4.5  # dynamic typing --> float

c = a*b  # --> float
d = 2  # dynamic typing --> int
```

```
e = c**d  # power ** not ^

print('%.1f * %.1f = %.1f' % (a, b, c))  # my favorite type of output
print('%.1f ** %d = %.1f' % (c, d, e))
```

```
3.0 * 4.5 = 13.5
13.5 ** 2 = 182.2
```

1.2.2 Structural programming

- control the flow of the program
- if … elif … else
- while & for

Alignment of statements:

```
a = 4 # int
if a < 10: # : is necessary
    # intentiation is necessary and should be everywhere the same
    print('OK ') # utf-8 is natively supported
else: # else is optional
    print('Not OK ')
```

```
In [2]: a = 4
        if a < 10:
            print('OK ')
        elif a > 10 and a < 20:
            print('Not OK ')
        else:
            print('either the special number 10  or greater equal 20')
```

```
OK
```

1.2.3 Structural programming

Two loop types:

- for
- while

```
In [3]: # for loop
        for i in range(5): # numbers from 0 to 4, max. numbers of iterations known
            print(i, end=' ')

        print('')

        # while loop 1
        i = 0
        while i < 5: # pre-test-loop
            print(i, end=' ')
            i+=1

        print('')
```

```
# while loop 2
i = 0
while True: # pre-test-loop, infinite loop
    print(i, end=' ')
    i+=1

    if i > 4:
        break

print('')
```

```
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
```

1.2.4 Procedural programming

Functions solving special problems

```
In [4]: import numpy as np # numpy module for large arrays

        # see: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
        def sieve_eratosthenes(N):
            # sieve of eratosthenes for calculating prime numbers

            crossed = np.zeros((N+1,), dtype=bool)

            for i in range(2, int(np.sqrt(N))):
                if not crossed[i]:
                    print('%d ' % i, end='')

                    for j in range(i*i, N, i):
                        crossed[j] = True

            for i in range(int(np.sqrt(N)) + 1, N):
                if not crossed[i]:
                    print('%d ' % i, end='')

            print('')

        # call function
        sieve_eratosthenes(100)
        sieve_eratosthenes(500)
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137
```

1.2.5 Modular programming

Special designed modules solving problems:

- Standard-Library: `import math`, `import sys`, `import os`, `import socket`, `import zipfile`, ...
- Special libraries: `import numpy`, `import scipy`, `import matplotlib.pyplot`

Modules need to be **imported** before usage!

- plain import `import math`
- named import `from math import sqrt, sin, cos`
- alias import `import numpy as np`, `import matplotlib.pyplot as plt`, `import scipy.signal as sig`

Writing own module:

1. write code that should be inside the module in a seperate file (e.g. functions, classes, definitions, ...)
2. save the file as `<modulename>.py`
3. import the module into your code as `import <modulename>`
4. You're done!

1.2.6 Functional programming

Programming is done not with statements, it's done with the declaration of functions!
Variables == Functions which return constant value ...

```
In [5]: import math

        x = lambda: 4  # constant variable as functional code

        Abase = lambda r: math.pi * r**2  # base area cylinder
        AM = lambda r, h: 2*math.pi * r * h # shell area cylinder

        Vz = lambda r, h: Abase(r) * h # cylindrical volume

        print('Base area :', Abase(x()))
        print('Shell area:', AM(x(), 12))
        print('Volume    :', Vz(x(), 12))
```

```
Base area : 50.26548245743669
Shell area: 301.59289474462014
Volume    : 603.1857894892403
```

1.2.7 Object-oriented programming

The user writes special classes, which solve special problems.
Can also be a complex class hirachy:

```
In [6]: # simple class is defined with

        class Vehicle:
            # no real private data
            # visibility as in C++ is determined with member declaration
            # __ private members
            # _ protected members
            # public members
            def __init__(self, mass): # class constructor
                self._mass = mass # protected class value
                self._vel = 0.0

            @property # read-only property mass
```

```
        def Mass(self):
            return self._mass

        @property # property as decorators
        def Velocity(self):
            return self._vel

        def accel(self):
            self._vel += 1

        def deccel(self):
            self._vel -= 1

        def stop(self):
            self._vel = 0.0
```

In [7]:
```
# usage of the class

vec = Vehicle(1000)
vec.accel()
print(vec)
print('Accel:', vec.Velocity)
vec.stop()
print('Stop :', vec.Velocity)
```

```
<__main__.Vehicle object at 0x7f8f514a95f8>
Accel: 1.0
Stop : 0.0
```

In [8]:
```
# derived classes

class Car(Vehicle):
    def __init__(self, mass, a=1.5):
        Vehicle.__init__(self, mass)
        self.__a = a # private class member, not accessible in derived classes

    def accel(self):
        self._vel += self.__a

    def deccel(self):
        self._vel -= self.__a

    def __get_a(self):
        return self.__a

    def __set_a(self, a):
        self.__a = a

    Acceleration = property(__get_a, __set_a) # getter, setter, deleter, doc

# usage:

car = Car(1500, 1.75)
car.accel()
```

```
        print('Accel:', car.Velocity)
        car.stop()
        print('Stop :', car.Velocity)
        print('A    :', car.Acceleration)
```

```
Accel: 1.75
Stop : 0.0
A    : 1.75
```

1.2.8 Where do I get help?

- Language documentation: https://docs.python.org/3/
- Python tutorial: https://docs.python.org/3/tutorial/
- Books (available online at our university library):
  - Joshi, P., Hearty, J., Sjardin, B., Massaron, L., & Boschetti, A. (2016). Python: Real World Machine Learning. Packt Publishing Ltd.
  - Linge, S., & Langtangen, H. P. (2020). Programming for Computations-Python: A Gentle Introduction to Numerical Simulations with Python 3.6 (p. 332). Springer.
  - Stephenson, B. (2016). The Python Workbook. SPRINGER INTERNATIONAL PU.
- Documentation for several projects / packages
  - Numpy reference: https://docs.scipy.org/doc/numpy/reference/
  - Scipy reference: https://docs.scipy.org/doc/scipy/reference/
  - Matplotlib: https://matplotlib.org/contents.html
  - Pandas: https://pandas.pydata.org/pandas-docs/stable/

1.3 Basic programming tasks

1.3.1 Screen-Output

```
In [9]: print('Hello World!')
```

```
Hello World!
```

Output data and variables:

```
In [10]: a = 4.0
         b = 3
         c = 'Test'
         d = [a, b, c] # list, mutable
         d1 = (a, b, c) # tuple, non-mutable

         print(a, b, c, d, d1) # very simple
         print("%.2f; %d; %s" % (a, b, c)) # c-formatted output
```

```
4.0 3 Test [4.0, 3, 'Test'] (4.0, 3, 'Test')
4.00; 3; Test
```

1.3.2 Theory: Mutable and Non-Mutable datatypes

Python is **interpreted** language:

- compiler / interpreter determining datatype at runtime
- datatype can be changed at runtime for one variable (e.g. from float to int, float to class object, ...)
- memory allocation is done automatically
- Garbage Collector (GC) `import gc`

In [11]:
```
a = 3.0
print(id(a)) # id gets the pointer to the memory block of one variable
b = a
print(id(b))
```

```
140253499851352
140253499851352
```

In [12]:
```
b = 4.9
print(id(a))
print(id(b))
```

```
140253499851352
140253499851784
```

In [13]:
```
# lists and tuples
lis = [3, 4, 6] # lists are mutable
tup = (3, 4, 6) # tuples are non-mutable

# lists and tuples can usually be used the same way
def print_arr(arr):
    for i in range(len(arr)): # just loop over the array elements
        print(arr[i], end=' ')
    print(' id(%d)' % id(arr))

print_arr(lis)
print_arr(tup)

# change list and tuple
lis.append(8)
# will not work: tup.append(8)
tup = tup + (8,) # instead we need to do something like this

print_arr(lis)
print_arr(tup)
```

```
3 4 6  id(140253530084616)
3 4 6  id(140253520823280)
3 4 6 8  id(140253530084616)
3 4 6 8  id(140253499329656)
```

1.3.3 Reading and writing files

In [14]:
```
# write plain ASCII file
fid = open('file.txt', 'w')
```

```
fid.write('Hello World!\n')
fid.close() # do not forget to close the file
```

In [15]:
```
# pythonic way: using context managers
with open('file.txt', 'w') as fid:
    fid.write('Hello Context!\n') # context managers automatically closes the file when exitin
```

In [16]:
```
# reading in:
with open('file.txt') as fid:
    for line in fid.readlines():
        print(line)
```

```
Hello Context!
```

In [17]:
```
# check if the files exist:
import os.path as op

file_to_check = 'file1.txt'
if op.exists(file_to_check):
    # file exists, append
    print('Exists!')
    with open(file_to_check, 'w+') as fid:
        fid.write('Hello Exists!\n')
else:
    # file does not exist, create them
    print('Not Exists!')
    with open(file_to_check, 'w') as fid:
        fid.write('Hello Not Exists!\n')
```

```
Exists!
```

In [18]:
```
# use different encoding when saving files
with open('encoding.txt', 'w', encoding='utf-8') as fid:
    fid.write('\n')
try:
    with open('encoding_no.txt', 'w', encoding='ascii') as fid:
        fid.write('\n')
except Exception as ex:
    print(ex)
```

```
'ascii' codec can't encode characters in position 0-2: ordinal not in range(128)
```

In [19]:
```
# reading and writing binary data
data = bytearray([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
with open('binfile.bin', 'wb') as fid:
    fid.write(data)

with open('binfile.bin', 'rb') as fid:
    print(list(fid.read()))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

1.3.4 Implemented data-types

In [20]:
```
# implemented data-types:
# int, float, bool, str
# list [], tuple (), dict {}

sample_for_dict = {'test1': {'Name': 'Morgenstern','GivenName': 'Roy', 'points': [20, 10 , 5],
                   'test2': {'Name': 'Morgenstern', 'GivenName': 'Alexandra', 'points': [18, 1
                   'test3': {'Name': 'Konietzky', 'GivenName': 'Heinz', 'points': [20, 12 , 5]
                   'test4': {'Name': 'Herbst', 'GivenName': 'Martin', 'points': [15, 12 , 5],

d = 0.0
for key, value in sample_for_dict.items():
    # loop over items
    print('%s %s has written %s with mark %.1f' % (value['GivenName'], value['Name'], key, valu
    d += value['mark']

print('average: %.1f' % (d/len(sample_for_dict)))
```

```
Roy Morgenstern has written test1 with mark 1.3
Alexandra Morgenstern has written test2 with mark 1.3
Heinz Konietzky has written test3 with mark 1.0
Martin Herbst has written test4 with mark 1.7
average: 1.3
```

1.4 Object oriented programming

!!! everything in Python is an Object !!!
including:

- basic datatypes (int, float, bool, string, . . .)
- lists, tuples, dictionaries, maps, filter, reductions, zips, enumerations, ranges, . . .
- functions
- lambdas
- . . .

In [21]:
```
print(int)
print(float)
print(range)
```

```
<class 'int'>
<class 'float'>
<class 'range'>
```

In [22]:
```
def func(a, b):
    return a + b

help(func) # python integrated help-system,
           # prints documentation of each element
           # stored in __doc__ attribute
print('~'*20)

print(dir(func)) # list all elements / attributes of func-object
```

```
         func.__doc__ = 'returns sum of a and b' # overwrite __doc__ attribute

         print('~'*20)
         help(func)
```

```
Help on function func in module __main__:

func(a, b)

~~~~~~~~~~~~~~~~~~~~
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__',
~~~~~~~~~~~~~~~~~~~~
Help on function func in module __main__:

func(a, b)
    returns sum of a and b
```

1.4.1 functions vs. func-objects

```
In [23]: print(func(3, 5)) # calling the function and printing the output
         print(func) # using the defined function func as function-objects

         # magic:
         def adder(a, b):
             return a + b

         def subber(a, b):
             return a - b

         def work(x, y, worker):
             return worker(x, y)

         print(work(4, 5, adder))
         print(work(4, 5, subber))
```

```
8
<function func at 0x7f8f50027400>
9
-1
```

1.4.2 Using objects

When using Python we are already working with a huge object-system

```
In [24]: with open('file1.txt', 'w') as fid: # with: ContextManager-class,
                                             # fid: python-file-object
             fid.write('Hello Object oriented programming ... ! ')
             # accessing public member function write of python
             # file object to write unicode string-object

In [25]: import numpy as np # module numpy alias-import (module is also a class)
         import matplotlib.pyplot as plt # sub-module pyplot alias-import
```

```
t = np.linspace(0, 2*np.pi, 1000) # t class numpy.array
a = np.sin(t) # a class numpy.array
plt.figure(figsize=(4, 3)) # access figure-member of module plt
plt.plot(t, a)
plt.show();
```

```
<matplotlib.figure.Figure at 0x7f8f514a9400>
```

1.4.3 Writing new objects - writing classes

Working with an example:

Geotechnical FOS Manager for Room and Pillar Mining

Factor of Safety: $FOS = \frac{\sigma_G}{\sigma_L}$

Definitions for σ_L ($p = \rho \cdot g \cdot H$):

- Long rooms: $\sigma_L = \left(1 + \frac{b_K}{2a}\right) \cdot p$
- Quadratic pillars: $\sigma_L = \left(\frac{2a+b_K}{2a}\right)^2 \cdot p$
- Rectangular pillars: $\sigma_L = \left(\frac{(2a+b_K)\cdot(2b+b_D)}{4ab}\right) \cdot p$

```
In [26]: class FOSManagerLR: # Base class
             # FOS estimation for long room pillars
             def __init__(self, over_dens, depth, a, bk, grav=9.81): # constructor
                 # ref to obj , required parameters    , optional parameters
                 # protected members to store class data
                 self._over_dens = over_dens
                 self._depth = depth
                 self._a = a
                 self._bk = bk
                 self._grav = grav
                 self._p = over_dens*depth*grav
                 self._L = 1 + bk/(2*a)

             @property # decorators, read-only property
             def SigmaL(self):
                 return self._L*self._p

             @property # decorators, read-only property
             def P(self):
                 return self._p

             @property # decorators, read-only property
             def L(self):
                 return self._L

             @property # decorators, read-only property
             def V(self):
                 return 1.0/self._L * 100

             def estimate_failure(self, sigma_g):
                 # public member function
                 return sigma_g / (self._L*self._p)
```

```
In [27]: class FOSManagerQuad(FOSManagerLR): # derived class
             def __init__(self, over_dens, depth, a, bk, grav=9.81): # constructor
                 FOSManagerLR.__init__(self, over_dens, depth, a, bk, grav) # call base class construct

                 self._L = ((2*self._a + self._bk)/(2*self._a))**2

         class FOSManagerRect(FOSManagerLR): # derived class
             def __init__(self, over_dens, depth, a, bk, b, bd, grav=9.81): # constructor
                 FOSManagerLR.__init__(self, over_dens, depth, a, bk, grav) # call base class construct

                 self._b = b
                 self._bd = bd

                 self._L = ((2*self._a + self._bk)*
                            (2*self._b + self._bd))/\
                           (4*self._a*self._b)
```

Examining load of pillar
Empirical approaches:

- Hardy & Agapito (1977): $\sigma_G = UCS \cdot \left(\frac{V_s}{V_p}\right)^{0.188} \cdot \left(\frac{V_p H_s}{H_p W_s}\right)^{0.833}$
- Bieniawski (1983): $\sigma_G = UCS \cdot \left(0.64 + 0.36 \frac{W_p}{H_p}\right)$
- Obert & Duvall (1967): $\sigma_G = UCS \cdot \left(0.778 + 0.22 \frac{W_p}{H_p}\right)$
- Salomon & Munro (1967): $\sigma_G = UCS \cdot \frac{W_p^{0.46}}{H_p^{0.66}}$
- Esterhuizen et al. (2008): $\sigma_G = 0.65 \cdot UCS \cdot \frac{W_p^{0.30}}{H_p^{0.59}}$

```
In [28]: bienawski = lambda UCS, Wp, Hp: UCS * (0.64 + 0.36*(Wp/Hp))
         obert_duvall = lambda UCS, Wp, Hp: UCS * (0.788 + 0.22*(Wp/Hp))
         salomon_munro = lambda UCS, Wp, Hp: UCS * Wp**0.46 / Hp**0.66
         esterhuizen = lambda UCS, Wp, Hp: UCS * Wp**0.30 / Hp**0.59

         approaches = {'Bienawski (1983)': bienawski,
                       'Obert & Duvall (1967)': obert_duvall,
                       'Salomon & Munro (1967)': salomon_munro,
                       'Esterhuizen et al. (2008)': esterhuizen}

         UCS = 80.0e6 # MPa
         H = 500.0 # m
         rho = 2700.0 # kg /mş
         a = Wp = 4.0
         bk = Hp = 3.0

         fom = FOSManagerQuad(rho, H, a, bk) # usage of FOS Manager derived class

         for app, func in approaches.items():
             sg = func(UCS, Wp, Hp)
             print('%s: %.3f' % (app, fom.estimate_failure(sg)))

Bienawski (1983): 3.578
Obert & Duvall (1967): 3.455
Salomon & Munro (1967): 2.928
Esterhuizen et al. (2008): 2.533
```

1.5 Scientific programming

1.5.1 Using Python as Computer-Algebra-System

- special package available `import sympy`
- defining mathematical functions or algebraic systems in code
- using **Symbols** for calculation
- varios operations possible (e.g. integrate, differentiate, limit, solve, ...)

Documentation: https://docs.sympy.org/latest/index.html

In [29]:
```
import sympy as sp # import necessary module
sp.init_printing()
# defining symbols
x, a, b = sp.symbols('x, a, b', real=True) # with assumtions of real-valued symbols
f = a*x + b*x/(x - a) - b # defining simple function
# sp.pprint(f) # pretty-printing f(x)
# in Jupyter also Latex-Output is possible:
f
```

Out[29]:

$$ax + \frac{bx}{-a + x} - b$$

In [30]:
```
# solve for roots
sp.solve(f, x)
```

Out[30]:

$$\left[\frac{a}{2} - \frac{1}{2}\sqrt{a^2 - 4b}, \quad \frac{a}{2} + \frac{1}{2}\sqrt{a^2 - 4b}\right]$$

In [31]:
```
# differentiate
f1 = f.diff(x)
f1
```

Out[31]:

$$a - \frac{bx}{(-a + x)^2} + \frac{b}{-a + x}$$

In [32]:
```
# and integrate again
F = f.integrate(x)
F
```

Out[32]:

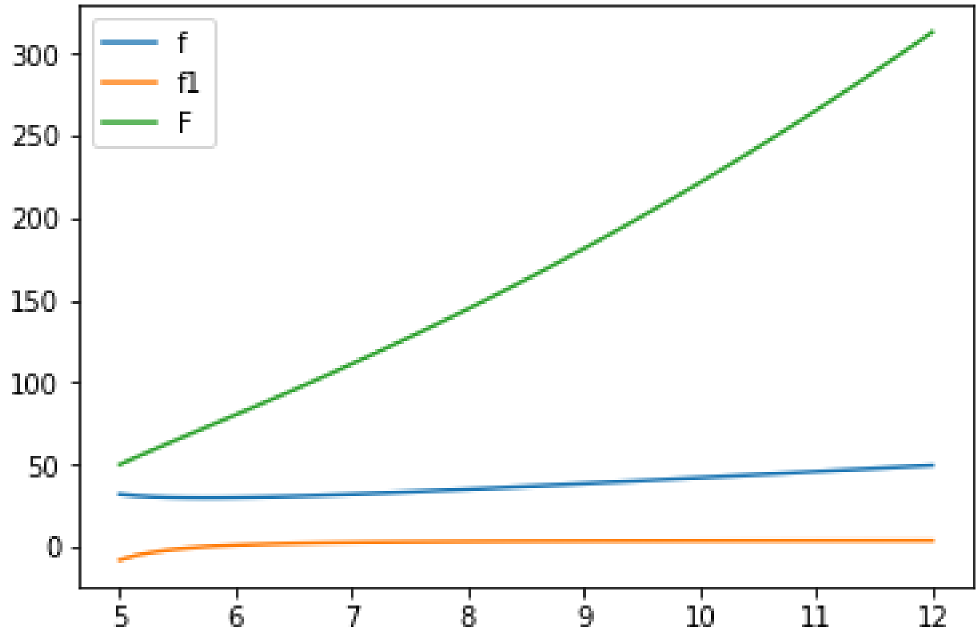
$$ab\log(-a + x) + \frac{ax^2}{2}$$

In [33]:
```
Fabs = f.integrate(x, (x, 0, 2))
Fabs
```

Out[33]:

$$a^2 b\log(-a) - a^2 b\log(-a + 2) + 2ab\log(-a + 2) - 2ab + \frac{4a}{3}$$

In [34]:
```
# make callable python-functions from sympy-definitions
f_func = sp.lambdify((x, a, b), f)
f1_func = sp.lambdify((x, a, b), f1)
F_func = sp.lambdify((x, a, b), F)

a, b = 4.0, 3.0
x = np.linspace(a + 1, a*b)
plt.plot(x, f_func(x, a, b), label='f')
plt.plot(x, f1_func(x, a, b), label='f1')
plt.plot(x, F_func(x, a, b), label='F')
plt.legend(); # ; supress output in jupyter
```

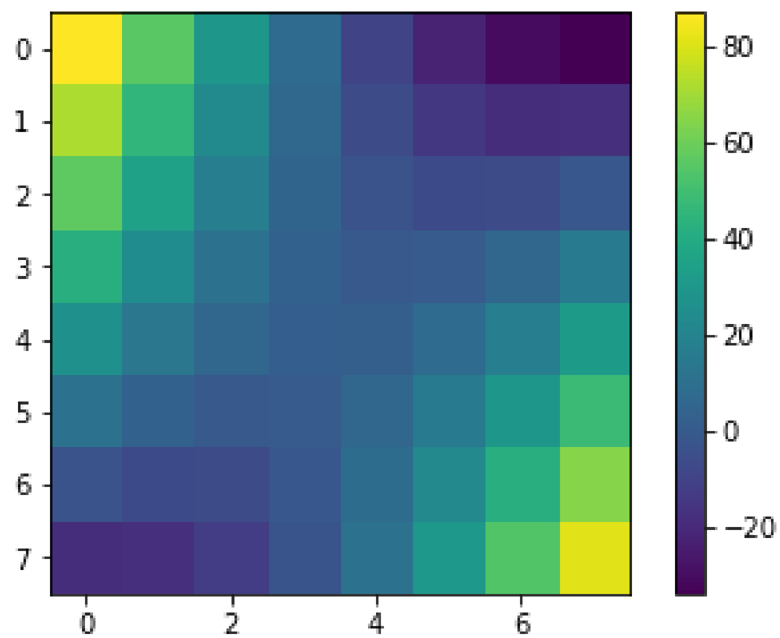


1.5.2 Using Python for Numerics

- special packages available for numerical data processing:
  - Base: numpy - basic linear algebra processing, implemented in C and Fortran
  - Extension: scipy - based on numpy and supports various features (e.g. clustering, FFT, ODEs, interpolation, advanced IO, optimization, image processing, signal procesing, spatial analysis, statistics, ...), implemented in C and Fortran
  - Pandas: works with numpy, special library for data analysis (especially for large datasets, big data), implemented in C and Fortran
- usally not needed to write numerical algorithms by hand
- use the implemented ones, well tested and optimized

In [35]:
```
# "Hello World" example
import numpy as np # usually the most common imports
import scipy as sc
import matplotlib.pyplot as plt
```

```
step0 = 1.5
x = np.arange(-5, 5+step0, step0) # evenly spaced vector in interval [-5, 5]
y = np.arange(-5, 5+step0, step0)
xx, yy = np.meshgrid(x, y) # matrices storing edge-points for x and y grids

# a 2d function
zz = xx**2 + xx*2*yy - 2*xx + 2.0

plt.figure()
plt.imshow(zz, interpolation=None)
plt.colorbar();
```

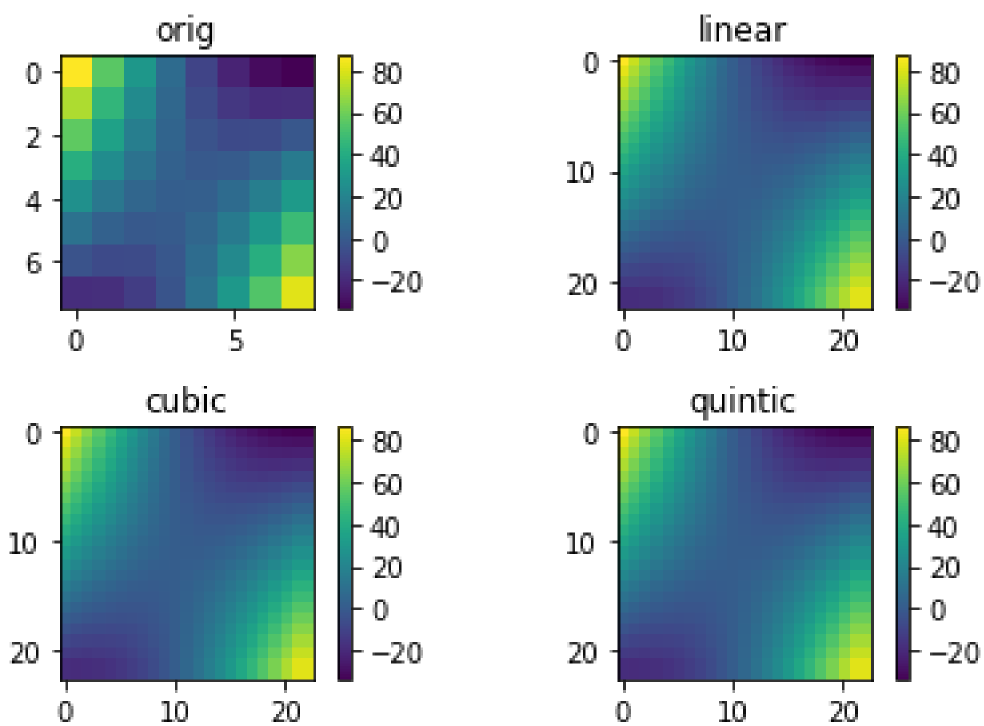



```
In [36]: # interpolate to finer grid
         from scipy.interpolate import interp2d
         step1 = 0.5
         xi = np.arange(-5, 5+step0, step1) # evenly spaced vector in interval [-5, 5]
         yi = np.arange(-5, 5+step0, step1)
         xxi, yyi = np.meshgrid(xi, yi)
         zzi = xxi**2 + xxi*2*yyi - 2*xxi + 2.0 # the real value

         intdi = {}
         for interp in ['linear', 'cubic', 'quintic']:
             # interpolating functions of different kinds
             fi = interp2d(x, y, zz, kind=interp)
             intdi.update({interp: fi})
             err = np.linalg.norm(zzi - fi(xi, yi), ord=-2) # smallest singular value
             print('Interpolating routine "%s": %.5e' % (interp, err))
```

```
Interpolating routine "linear": 1.75611e-31
Interpolating routine "cubic": 8.24443e-17
```

```
Interpolating routine "quintic": 2.41225e-16
```

In [37]:
```python
plt.figure()
plt.subplot(221)
plt.title('orig')
plt.imshow(zz, interpolation=None)
plt.colorbar()

i = 2
for k, v in intdi.items():
    plt.subplot(2, 2, i)
    plt.title(k)
    plt.imshow(v(xi, yi), interpolation=None)
    plt.colorbar()
    i+=1
plt.tight_layout()
```



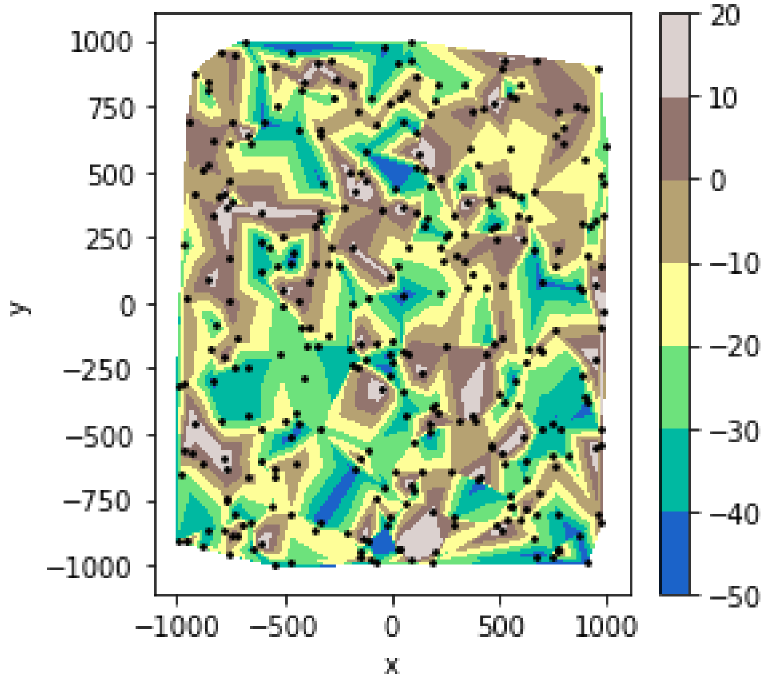
1.5.3 Real-World problem: Gridding data

- usually data is not equal sampled
- algorithms usually grid-based (matrix-based)
- gridding data is requested

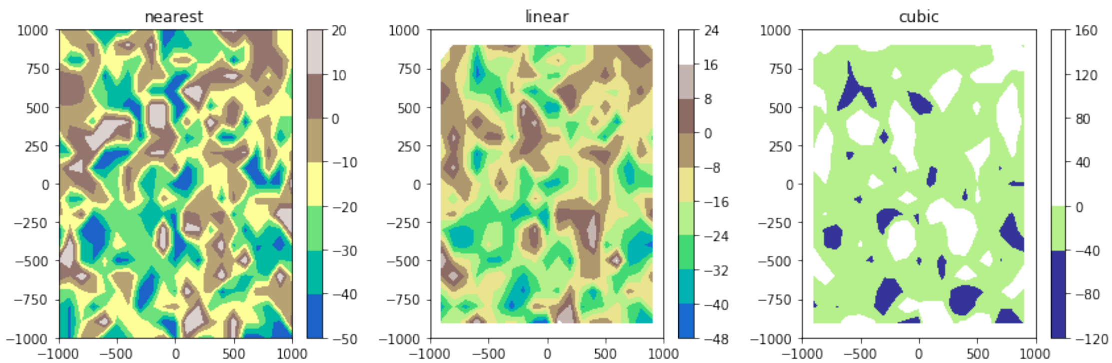
In [38]:
```python
# create random dataset
size = 1000
x = np.random.randint(-size, size, size=size//3) # random x and y coordinates
```

```
y = np.random.randint(-size, size, size=size//3)
data = np.random.randint(-50, 20, size=size//3) # random hydraulic heads
```

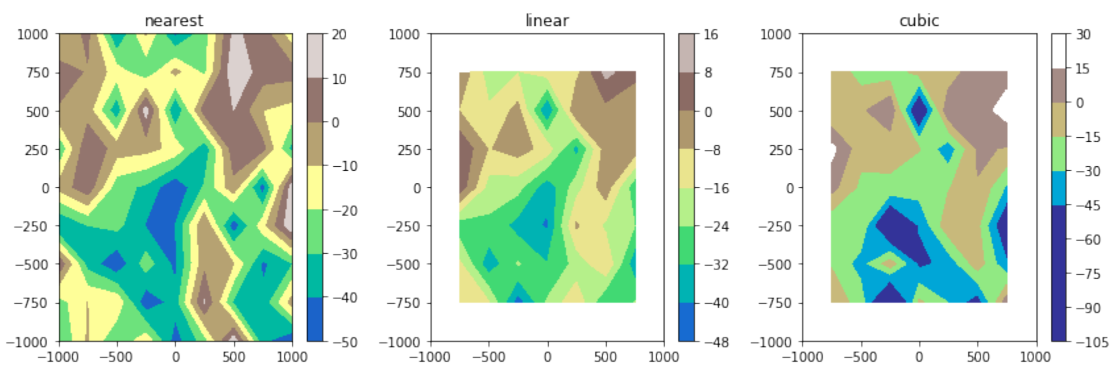
In [39]:
```
plt.figure(figsize=(4, 4))
plt.tricontourf(x, y, data, cmap='terrain', vmin=-50, vmax=20)
plt.colorbar()
plt.scatter(x, y, s=3, c='k')
plt.xlabel('x')
plt.ylabel('y');
```



In [40]:
```
dxy = 100 # step size
xi = np.arange(-size, size+dxy, dxy) # gridding positions xi and yi
yi = np.arange(-size, size+dxy, dxy)
xxi, yyi = np.meshgrid(xi, yi)

from scipy.interpolate import griddata

plt.figure(figsize=(12, 4))
for i, k in enumerate(['nearest', 'linear', 'cubic']):
    plt.subplot(1, 3, i+1)
    plt.contourf(xxi, yyi, griddata((x, y), data, (xxi, yyi), method=k), cmap='terrain', vmin=-
    plt.colorbar()
    plt.title(k)
plt.tight_layout()
```




```
In [41]: dxy = 250 # step size
         xi = np.arange(-size, size+dxy, dxy) # gridding positions xi and yi
         yi = np.arange(-size, size+dxy, dxy)
         xxi, yyi = np.meshgrid(xi, yi)

         from scipy.interpolate import griddata

         plt.figure(figsize=(12, 4))
         for i, k in enumerate(['nearest', 'linear', 'cubic']):
             plt.subplot(1, 3, i+1)
             plt.contourf(xxi, yyi, griddata((x, y), data, (xxi, yyi), method=k), cmap='terrain', vmin=
             plt.colorbar()
             plt.title(k)
         plt.tight_layout()
```



1.5.4 Real-World problem: Optimization of Room-and-Pillar layout

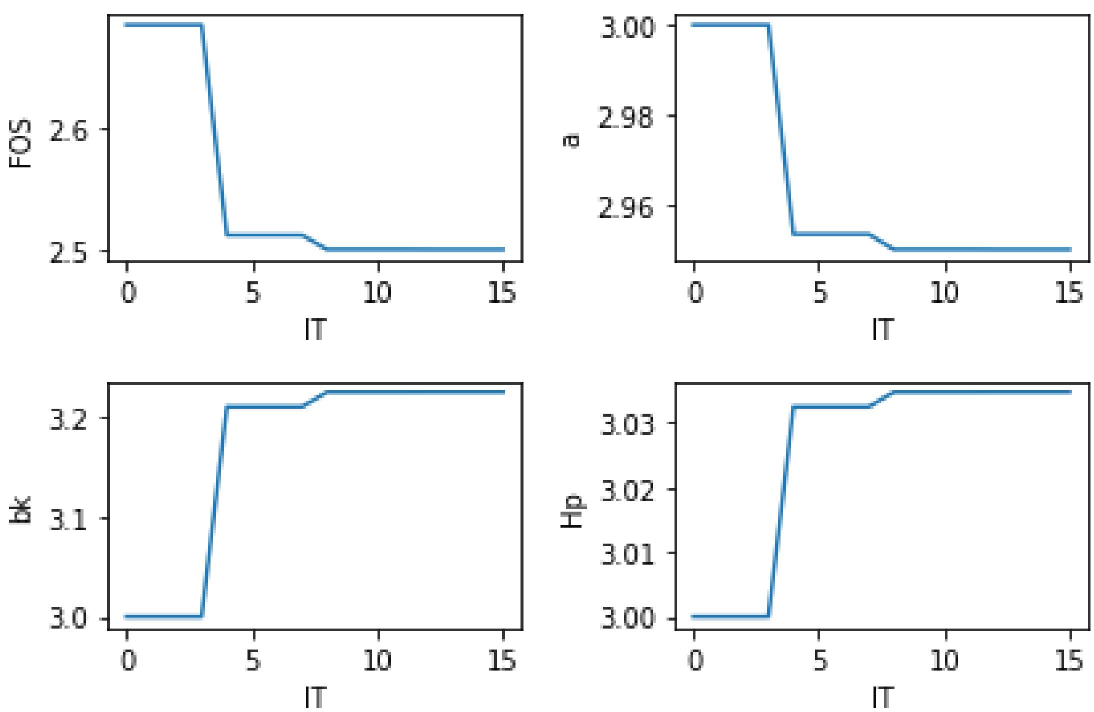
- problems for finding a suitable layout for a room-and-pillar mining
- Factor-of-Safety (FOS) ~ 2.5
- just as an approximation with empirical relations
- using our previously defined classes `FOSManager`

See for optimization methods: https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html

```
In [42]: from scipy.optimize import least_squares, minimize # simple optimization routine that minimize
         UCS = 80e6 # UCS value of the material
         H = 500.0 # m
         rho = 2700.0 # kg /mş
         FOS_target = 2.5 # targeted FOS value
         re = [] # empty list to store results of each optimization step

         def opti_fun(x, FOS_target, approach): # optimization function
             a, bk, Hp = x # unpack values
             # optimization parameters: pillar width a, chamber with bk, pillar height Hp, target FOS F
             fom = FOSManagerQuad(rho, H, a, bk) # usage of FOS Manager derived class, global vars rho
             if approach in approaches:
                 func = approaches[approach]
             else:
                 func = bienawski
             Wp = a # access local var a
             sg = func(UCS, Wp, Hp)
             fos = fom.estimate_failure(sg)
             err = abs(fos - FOS_target)
             re.append([a, bk, Hp, fos, err])
             return err # return error estimate, single number

In [43]: # assuming rectangular shaped pillar
         # starting estimates
         aWp0 = 3.0
         bk0 = 3.0
         Hp0 = 3.0

         # subjected to bounds
         aWpb = (2, 10) # pillar and room lengths
         bkb = (2, 10)
         Hpb = (3, 5) # pillar and romm height subjected to machine layout

         res = least_squares(opti_fun, x0=(aWp0, bk0, Hp0),
                             args=(FOS_target, 'bienawski'),
                             bounds=([aWpb[0], bkb[0], Hpb[0]], [aWpb[1], bkb[1], Hpb[1]]),
                             verbose=1)

`gtol` termination condition is satisfied.
Function evaluations 4, initial cost 1.7067e-02, final cost 8.5590e-19, first-order optimality 1.12e-09


In [44]: res

Out[44]:  active_mask: array([0, 0, 0])
                 cost: 8.5590363957478317e-19
                  fun: array([  1.30836053e-09])
                 grad: array([  1.17562424e-09,   6.45602668e-10,   3.05318356e-10])
                  jac: array([[ 0.89854762,  0.49344401,  0.2333595 ]])
              message: '`gtol` termination condition is satisfied.'
                 nfev: 4
                 njev: 4
```

```
    optimality: 1.1171621764319125e-09
        status: 1
       success: True
             x: array([ 2.95027147,  3.22540224,  3.03477515])
```

In [45]:
```python
re = np.array(re)
plt.figure()
for idx, k, m in zip((-2, 0, 1, 2), ('FOS', 'a', 'bk', 'Hp'), (1, 2, 3, 4)):
    plt.subplot(2, 2, m)
    plt.plot(re[:, idx])
    plt.xlabel('IT')
    plt.ylabel(k)

plt.tight_layout()
```

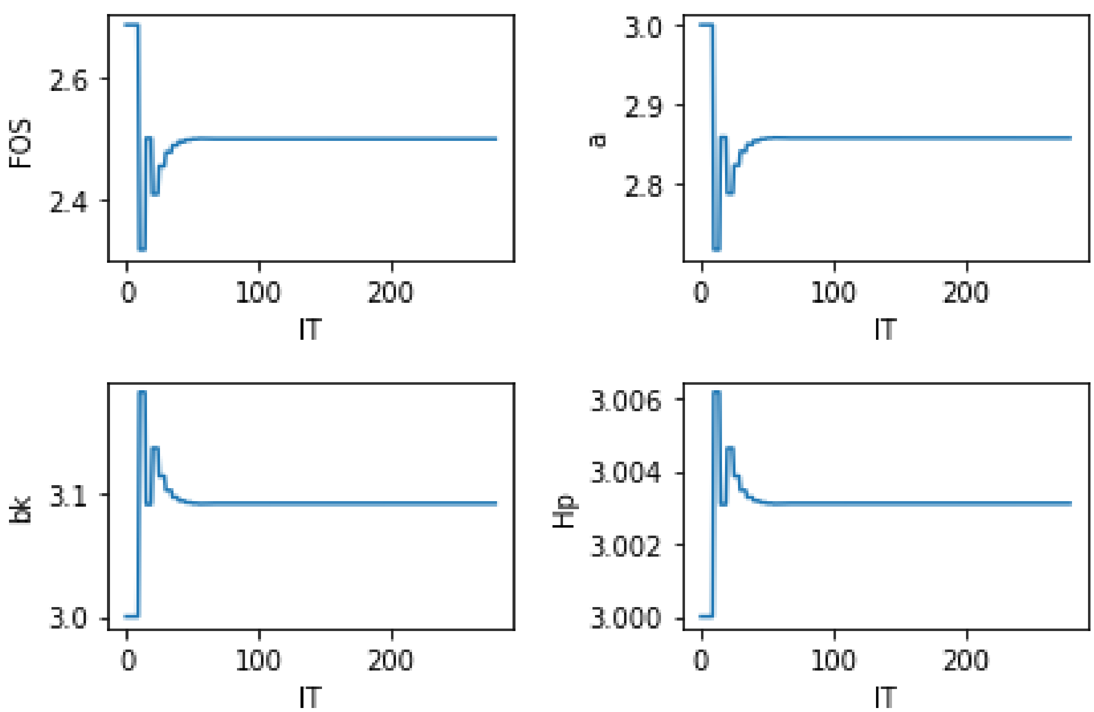


In [46]:
```python
# simple minimize function
re = []
res = minimize(opti_fun, x0=(aWp0, bk0, Hp0),
               args=(FOS_target, 'bienawski'),
               method='TNC', tol=1e-16,
               bounds=(aWpb, bkb, Hpb))
res
```

Out[46]:
```
     fun: 2.5950441795430379e-09
     jac: array([ 0.40001993,  0.56762741,  0.29027714])
 message: 'Converged (|x_n-x_(n-1)| ~= 0)'
    nfev: 55
     nit: 2
```

```
    status: 2
   success: True
         x: array([ 2.85828766,  3.092021   ,  3.00310571])
```

In [47]:
```python
re = np.array(re)
plt.figure()

for idx, k, m in zip((-2, 0, 1, 2), ('FOS', 'a', 'bk', 'Hp'), (1, 2, 3, 4)):
    plt.subplot(2, 2, m)
    plt.plot(re[:, idx])
    plt.xlabel('IT')
    plt.ylabel(k)

plt.tight_layout()
```



In [48]:
```python
# global optimization
from scipy.optimize import differential_evolution
re = []
res = differential_evolution(opti_fun, args=(FOS_target, 'bienawski'),
                             tol=1e-16, bounds=(aWpb, bkb, Hpb),
                             disp=True)
```

```
differential_evolution step 1: f(x)= 0.0371029
differential_evolution step 2: f(x)= 0.00919213
differential_evolution step 3: f(x)= 0.00919213
differential_evolution step 4: f(x)= 0.00919213
differential_evolution step 5: f(x)= 0.00919213
differential_evolution step 6: f(x)= 0.00548227
```

```
differential_evolution step 7: f(x)= 0.00548227
differential_evolution step 8: f(x)= 0.00548227
differential_evolution step 9: f(x)= 0.00548227
differential_evolution step 10: f(x)= 0.00548227
differential_evolution step 11: f(x)= 6.47249e-05
differential_evolution step 12: f(x)= 6.47249e-05
differential_evolution step 13: f(x)= 6.47249e-05
differential_evolution step 14: f(x)= 6.47249e-05
differential_evolution step 15: f(x)= 6.47249e-05
differential_evolution step 16: f(x)= 6.47249e-05
differential_evolution step 17: f(x)= 6.47249e-05
differential_evolution step 18: f(x)= 6.47249e-05
differential_evolution step 19: f(x)= 6.47249e-05
differential_evolution step 20: f(x)= 6.47249e-05
differential_evolution step 21: f(x)= 6.47249e-05
differential_evolution step 22: f(x)= 6.47249e-05
differential_evolution step 23: f(x)= 6.47249e-05
differential_evolution step 24: f(x)= 6.47249e-05
differential_evolution step 25: f(x)= 6.47249e-05
differential_evolution step 26: f(x)= 6.47249e-05
differential_evolution step 27: f(x)= 5.96018e-05
differential_evolution step 28: f(x)= 5.64435e-06
differential_evolution step 29: f(x)= 5.64435e-06
differential_evolution step 30: f(x)= 5.64435e-06
differential_evolution step 31: f(x)= 5.64435e-06
differential_evolution step 32: f(x)= 5.64435e-06
differential_evolution step 33: f(x)= 5.64435e-06
differential_evolution step 34: f(x)= 3.49682e-06
differential_evolution step 35: f(x)= 3.49682e-06
differential_evolution step 36: f(x)= 3.49682e-06
differential_evolution step 37: f(x)= 3.49682e-06
differential_evolution step 38: f(x)= 3.49682e-06
differential_evolution step 39: f(x)= 9.73542e-07
differential_evolution step 40: f(x)= 4.96403e-07
differential_evolution step 41: f(x)= 4.96403e-07
differential_evolution step 42: f(x)= 4.96403e-07
differential_evolution step 43: f(x)= 4.03011e-07
differential_evolution step 44: f(x)= 8.55159e-08
differential_evolution step 45: f(x)= 8.55159e-08
differential_evolution step 46: f(x)= 8.55159e-08
differential_evolution step 47: f(x)= 8.55159e-08
differential_evolution step 48: f(x)= 8.55159e-08
differential_evolution step 49: f(x)= 8.55159e-08
differential_evolution step 50: f(x)= 8.55159e-08
differential_evolution step 51: f(x)= 8.55159e-08
differential_evolution step 52: f(x)= 8.55159e-08
differential_evolution step 53: f(x)= 2.14184e-09
differential_evolution step 54: f(x)= 2.14184e-09
differential_evolution step 55: f(x)= 2.14184e-09
differential_evolution step 56: f(x)= 2.14184e-09
differential_evolution step 57: f(x)= 2.14184e-09
differential_evolution step 58: f(x)= 2.14184e-09
differential_evolution step 59: f(x)= 2.14184e-09
differential_evolution step 60: f(x)= 2.14184e-09
```

```
differential_evolution step 61: f(x)= 2.14184e-09
differential_evolution step 62: f(x)= 2.14184e-09
differential_evolution step 63: f(x)= 2.14184e-09
differential_evolution step 64: f(x)= 2.14184e-09
differential_evolution step 65: f(x)= 2.14184e-09
differential_evolution step 66: f(x)= 2.14184e-09
differential_evolution step 67: f(x)= 2.14184e-09
differential_evolution step 68: f(x)= 1.41339e-09
differential_evolution step 69: f(x)= 1.41339e-09
differential_evolution step 70: f(x)= 1.41339e-09
differential_evolution step 71: f(x)= 1.39217e-09
differential_evolution step 72: f(x)= 1.39217e-09
differential_evolution step 73: f(x)= 4.29841e-10
differential_evolution step 74: f(x)= 3.26687e-10
differential_evolution step 75: f(x)= 3.26687e-10
differential_evolution step 76: f(x)= 3.26687e-10
differential_evolution step 77: f(x)= 3.26687e-10
differential_evolution step 78: f(x)= 3.26687e-10
differential_evolution step 79: f(x)= 3.26687e-10
differential_evolution step 80: f(x)= 3.26687e-10
differential_evolution step 81: f(x)= 6.01408e-11
differential_evolution step 82: f(x)= 6.01408e-11
differential_evolution step 83: f(x)= 6.01408e-11
differential_evolution step 84: f(x)= 6.01408e-11
differential_evolution step 85: f(x)= 4.51204e-11
differential_evolution step 86: f(x)= 4.51204e-11
differential_evolution step 87: f(x)= 2.47393e-11
differential_evolution step 88: f(x)= 7.45892e-12
differential_evolution step 89: f(x)= 7.45892e-12
differential_evolution step 90: f(x)= 7.45892e-12
differential_evolution step 91: f(x)= 7.45892e-12
differential_evolution step 92: f(x)= 7.45892e-12
differential_evolution step 93: f(x)= 7.45892e-12
differential_evolution step 94: f(x)= 7.45892e-12
differential_evolution step 95: f(x)= 2.3439e-12
differential_evolution step 96: f(x)= 2.3439e-12
differential_evolution step 97: f(x)= 2.3439e-12
differential_evolution step 98: f(x)= 2.3439e-12
differential_evolution step 99: f(x)= 2.3439e-12
differential_evolution step 100: f(x)= 2.3439e-12
differential_evolution step 101: f(x)= 2.3439e-12
differential_evolution step 102: f(x)= 2.3439e-12
differential_evolution step 103: f(x)= 1.93534e-12
differential_evolution step 104: f(x)= 1.93534e-12
differential_evolution step 105: f(x)= 1.93534e-12
differential_evolution step 106: f(x)= 1.93534e-12
differential_evolution step 107: f(x)= 1.93534e-12
differential_evolution step 108: f(x)= 8.54872e-13
differential_evolution step 109: f(x)= 8.1668e-13
differential_evolution step 110: f(x)= 3.83693e-13
differential_evolution step 111: f(x)= 8.34888e-14
differential_evolution step 112: f(x)= 8.34888e-14
differential_evolution step 113: f(x)= 8.34888e-14
differential_evolution step 114: f(x)= 8.34888e-14
```

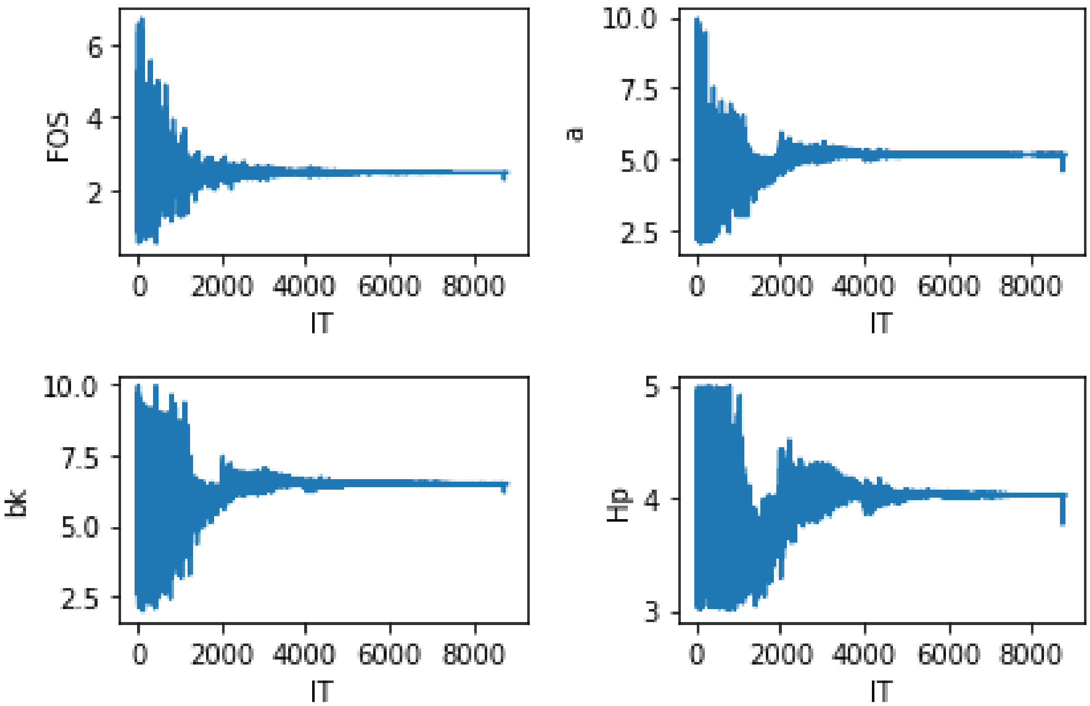
```
differential_evolution step 115: f(x)= 8.34888e-14
differential_evolution step 116: f(x)= 5.19584e-14
differential_evolution step 117: f(x)= 5.19584e-14
differential_evolution step 118: f(x)= 5.19584e-14
differential_evolution step 119: f(x)= 5.19584e-14
differential_evolution step 120: f(x)= 1.19904e-14
differential_evolution step 121: f(x)= 1.19904e-14
differential_evolution step 122: f(x)= 1.19904e-14
differential_evolution step 123: f(x)= 1.19904e-14
differential_evolution step 124: f(x)= 1.19904e-14
differential_evolution step 125: f(x)= 1.19904e-14
differential_evolution step 126: f(x)= 1.19904e-14
differential_evolution step 127: f(x)= 1.19904e-14
differential_evolution step 128: f(x)= 1.19904e-14
differential_evolution step 129: f(x)= 4.44089e-16
differential_evolution step 130: f(x)= 4.44089e-16
differential_evolution step 131: f(x)= 4.44089e-16
differential_evolution step 132: f(x)= 4.44089e-16
differential_evolution step 133: f(x)= 4.44089e-16
differential_evolution step 134: f(x)= 4.44089e-16
differential_evolution step 135: f(x)= 4.44089e-16
differential_evolution step 136: f(x)= 4.44089e-16
differential_evolution step 137: f(x)= 4.44089e-16
differential_evolution step 138: f(x)= 4.44089e-16
differential_evolution step 139: f(x)= 4.44089e-16
differential_evolution step 140: f(x)= 4.44089e-16
differential_evolution step 141: f(x)= 4.44089e-16
differential_evolution step 142: f(x)= 4.44089e-16
differential_evolution step 143: f(x)= 4.44089e-16
differential_evolution step 144: f(x)= 4.44089e-16
differential_evolution step 145: f(x)= 4.44089e-16
differential_evolution step 146: f(x)= 4.44089e-16
differential_evolution step 147: f(x)= 4.44089e-16
differential_evolution step 148: f(x)= 4.44089e-16
differential_evolution step 149: f(x)= 4.44089e-16
differential_evolution step 150: f(x)= 4.44089e-16
differential_evolution step 151: f(x)= 4.44089e-16
differential_evolution step 152: f(x)= 4.44089e-16
differential_evolution step 153: f(x)= 4.44089e-16
differential_evolution step 154: f(x)= 4.44089e-16
differential_evolution step 155: f(x)= 0
differential_evolution step 156: f(x)= 0
differential_evolution step 157: f(x)= 0
differential_evolution step 158: f(x)= 0
differential_evolution step 159: f(x)= 0
differential_evolution step 160: f(x)= 0
differential_evolution step 161: f(x)= 0
differential_evolution step 162: f(x)= 0
differential_evolution step 163: f(x)= 0
differential_evolution step 164: f(x)= 0
differential_evolution step 165: f(x)= 0
differential_evolution step 166: f(x)= 0
differential_evolution step 167: f(x)= 0
differential_evolution step 168: f(x)= 0
```

```
differential_evolution step 169: f(x)= 0
differential_evolution step 170: f(x)= 0
differential_evolution step 171: f(x)= 0
differential_evolution step 172: f(x)= 0
differential_evolution step 173: f(x)= 0
differential_evolution step 174: f(x)= 0
differential_evolution step 175: f(x)= 0
differential_evolution step 176: f(x)= 0
differential_evolution step 177: f(x)= 0
differential_evolution step 178: f(x)= 0
differential_evolution step 179: f(x)= 0
differential_evolution step 180: f(x)= 0
differential_evolution step 181: f(x)= 0
differential_evolution step 182: f(x)= 0
differential_evolution step 183: f(x)= 0
differential_evolution step 184: f(x)= 0
differential_evolution step 185: f(x)= 0
differential_evolution step 186: f(x)= 0
differential_evolution step 187: f(x)= 0
differential_evolution step 188: f(x)= 0
differential_evolution step 189: f(x)= 0
differential_evolution step 190: f(x)= 0
differential_evolution step 191: f(x)= 0
differential_evolution step 192: f(x)= 0
differential_evolution step 193: f(x)= 0
```

```
In [49]: res
```

```
Out[49]:      fun: 0.0
         message: 'Optimization terminated successfully.'
            nfev: 8814
             nit: 193
         success: True
               x: array([ 5.14000183,  6.47734573,  4.02517577])
```

```
In [50]: re = np.array(re)
         plt.figure()

         for idx, k, m in zip((-2, 0, 1, 2), ('FOS', 'a', 'bk', 'Hp'), (1, 2, 3, 4)):
             plt.subplot(2, 2, m)
             plt.plot(re[:, idx])
             plt.xlabel('IT')
             plt.ylabel(k)

         plt.tight_layout()
```



1.5.5 Theory: Selecting and iterating data

- iteration is possible over various Python objects (e.g. list, tuple, dict-(key- and/or value-)views, numpy arrays, strings, ...
- creating iteratable objects is possible:
  - need to implement `__iter__`-attribute
  - need to implement `__next__`-attribute
- selecting of subset of data is possible - **slicing**

```
In [51]: l1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

         # selecting data
         print(l1[0:4]) # selecting elements from index 0 to 4-1 from list l2
         print(l1[4:-1]) # selecting elements from index 4 to end-1 from list l2
         print(l1[::2]) # selecting every second element
         print(l1[::-1]) # reverse list

         for c in 'Hello Iteration!': # iterating over strings
             print(c, end=' ')
```

```
[1, 2, 3, 4]
[5, 6, 7, 8, 9]
[1, 3, 5, 7, 9]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
H e l l o   I t e r a t i o n !
```

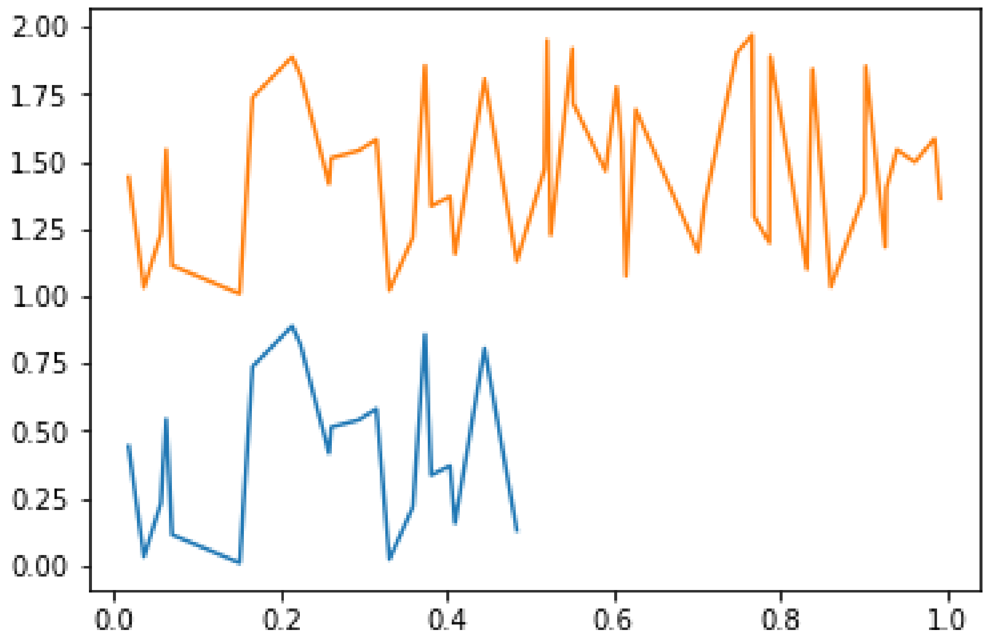
```
In [52]: # iterating over iterables
         for elem in l1:
             print(elem, end='; ')
         print()
         # iteration and selection
         for elem in l1[:-3:2]:
             print(elem, end='/ ')
         print()
         import numpy as np
         x = np.linspace(1, 5, 10, dtype=np.float16)
         for xi in x:
             print(xi, end='| ')
```

```
1; 2; 3; 4; 5; 6; 7; 8; 9; 10;
1/ 3/ 5/ 7/
1.0| 1.4443| 1.8887| 2.334| 2.7773| 3.2227| 3.666| 4.1094| 4.5547| 5.0|
```

```
In [53]: # iteration and references
         l2 = [1, 2, 3, 4, 5]
         for item in l2:
             item *= 2
         print(l2)

         x = np.linspace(1, 5, 10, dtype=np.float16)
         for xi in x:
             xi *= 2
         print(x)

         # different:
         for i in range(len(x)):
             x[i]*=2
         print(x)
```

```
[1, 2, 3, 4, 5]
[ 1.          1.44433594  1.88867188  2.33398438  2.77734375  3.22265625
  3.66601562  4.109375    4.5546875   5.        ]
[  2.           2.88867188   3.77734375   4.66796875   5.5546875
   6.4453125    7.33203125   8.21875      9.109375    10.        ]
```

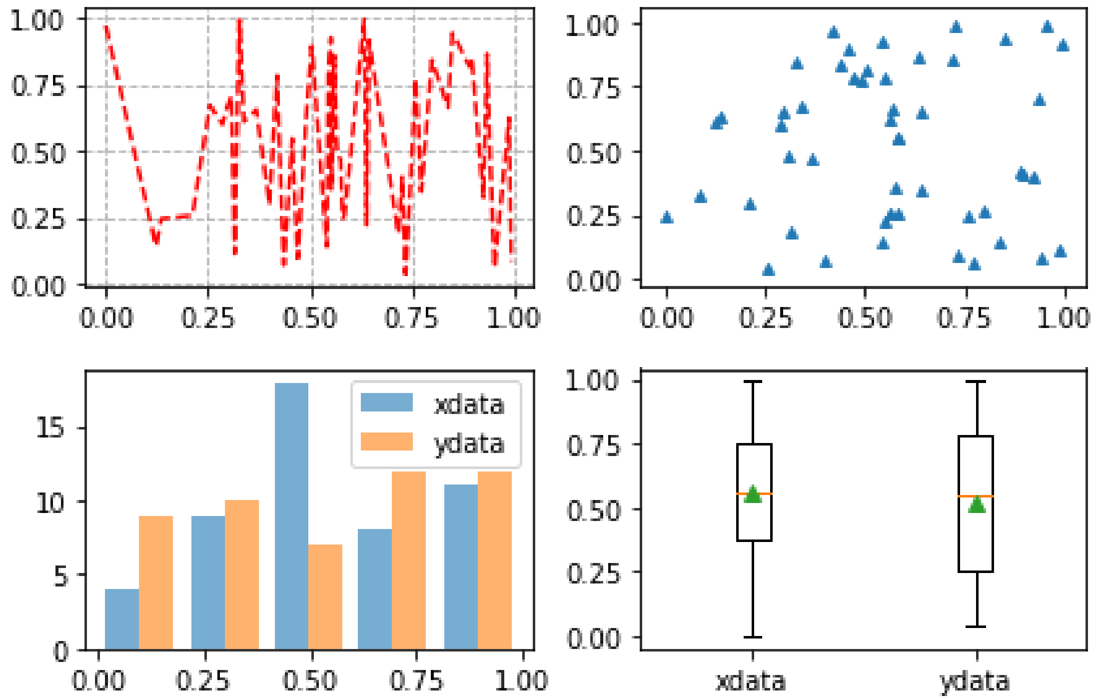
```
In [54]: # selectively plotting data
         import numpy as np
         import matplotlib.pyplot as plt

         x = np.sort(np.random.uniform(0, 1, 50))
         y = np.random.uniform(0, 1, 50)
         idx = np.argwhere(x < 0.5) # returns boolean array
         plt.figure()
         plt.plot(x[idx], y[idx])
         plt.plot(x, y+1);
```



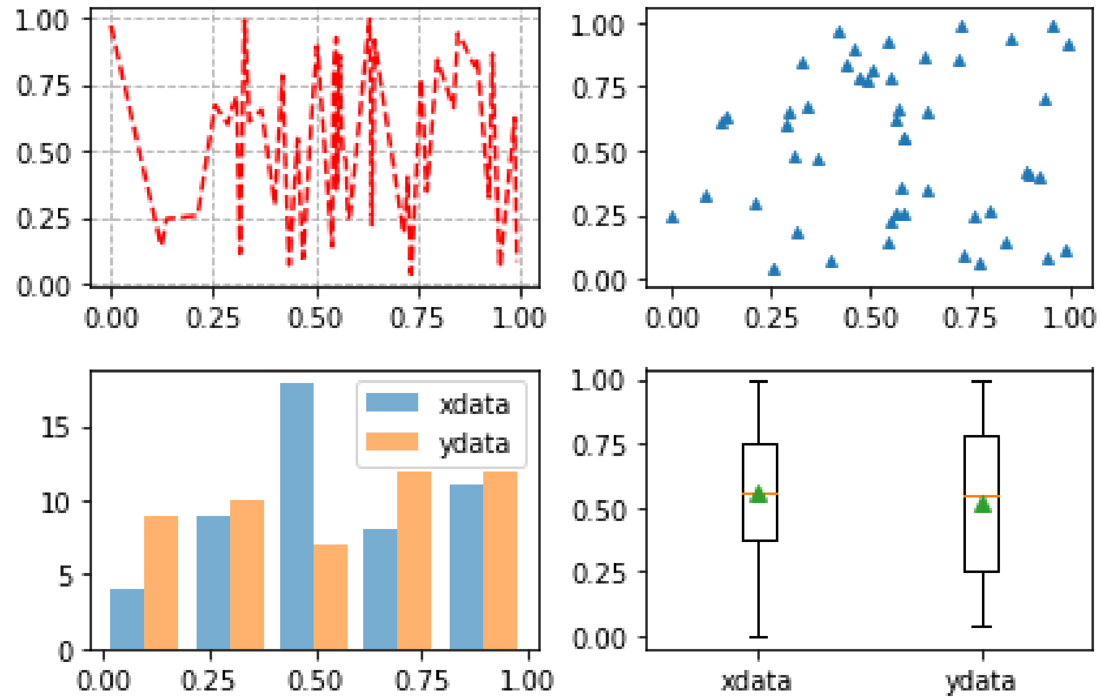
1.5.6 Scientific visualization

- usually done with `matplotlib`-package
- various possibilities
- line, contour, image, histogram, scatter and statistical plots and many more
- semi-3d plots using 2d-backend
- others: seaborn (statistical), pyvista (3D), vtk (3D)

In [55]:
```
import numpy as np
import matplotlib.pyplot as plt
xdat = np.random.rand(50)
ydat = np.random.rand(50)
plt.figure() # open new figure MATLAB-like syntax
plt.subplot(221) # open new subplot with 2x2 layout and subplot number 1 (upper left corner)
plt.plot(np.sort(xdat), ydat, ls='--', c='r') # plot x and y data with dashed linestyle and re
plt.grid(ls='--') # plot grid
plt.subplot(222) # upper right corner
plt.scatter(xdat, ydat, s=15, marker='^') # scatter plot of x and y data with 'hat' markers in
plt.subplot(223)
plt.hist((xdat, ydat), label=('xdata', 'ydata'), alpha=.6, bins=5) # histogram plot of two his
plt.legend() # plot automatic legend, containing label information
plt.subplot(224)
plt.boxplot((xdat, ydat), showmeans=True, labels=('xdata', 'ydata'))
plt.tight_layout() # automatically adjust subplots
```



In [56]:
```
# the same with axes objects: pythonic way
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
ax1.plot(np.sort(xdat), ydat, ls='--', c='r')
ax1.grid(ls='--')
ax2.scatter(xdat, ydat, s=15, marker='^')
ax3.hist((xdat, ydat), label=('xdata', 'ydata'), alpha=.6, bins=5)
ax3.legend()
ax4.boxplot((xdat, ydat), showmeans=True, labels=('xdata', 'ydata'))
fig.tight_layout()
```



1.5.7 Real-World problem: Visualizing laboratory / measurement data

- usually stored in CSV, ASCII or Excel files (easy, structure human-readable)
- some data in binary format (bit more difficult, structure and datatypes must be known)
- CSV, Text and Excel files can be read via:
  - `numpy.fromfile`, `numpy.loadtxt`, `numpy.genfromtxt`, ...
  - `pandas.read_csv`, `pandas.read_excel`, ... (also other possible formats: JSON, HDF, SQL, ...)
- Binary data can be read via:
  - `numpy.load` (numpy-Format, *.npy, *.npz), `numpy.fromfile`
  - `scipy.loadmat` (MATLAB-files *.mat)

Scenario 1: Reading UCS-data from Excel sheets

... using Pandas

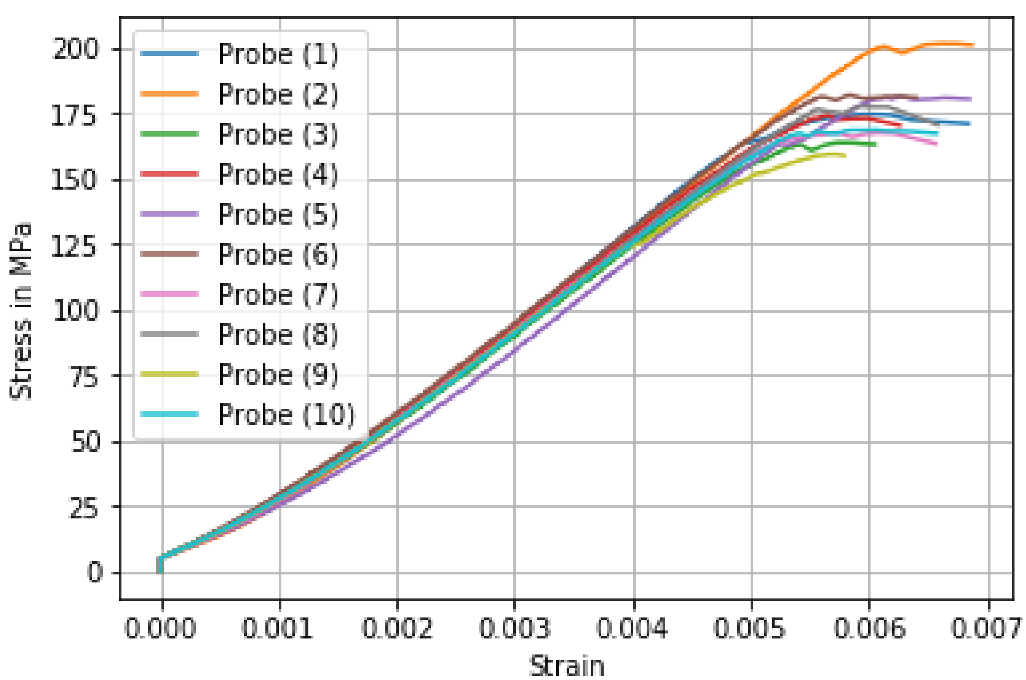
In [57]:
```
import pandas as pd

excf = './data/gneis_lab.xlsx' # filename
# the most flexible approach
excelfile = pd.ExcelFile(excf)
print(excelfile.sheet_names)
```

```
['Probe (1)', 'Probe (2)', 'Probe (3)', 'Probe (4)', 'Probe (5)', 'Probe (6)', 'Probe (7)', 'Probe (8)'
```

In [58]:
```
# parse first sheet into a pandas DataFrame
df = excelfile.parse(excelfile.sheet_names[0], skiprows=1, header=[0,1])
# lets have a look at the data
df.head()
```

Out[58]:
```
Zeit Kraft   Weg L- Dehnung L- Dehnung A L- Dehnung B L- Dehnung C  \
s       kN    mm         mm           mm           mm           mm
0.0     0.0  0.00       0.000        0.000        0.000        0.000
0.1    10.0  0.00       0.000        0.000        0.000        0.000
0.8    11.2  0.01       0.002        0.002        0.002        0.002
0.9    11.4  0.01       0.002        0.002        0.002        0.003
1.0    11.9  0.01       0.002        0.002        0.002        0.003

Zeit Ax-Spannung L-Dehnung
s            MPa         -
0.0         0.00  0.000000
0.1         5.09  0.000000
0.8         5.70  0.000039
0.9         5.81  0.000039
1.0         6.06  0.000039
```

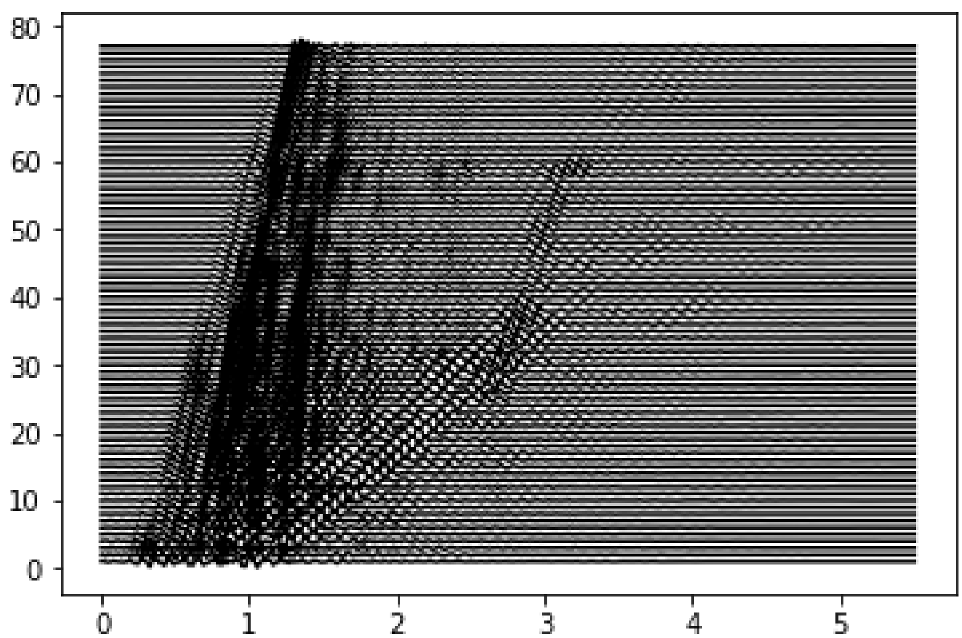
In [59]:
```
# plot everything into one plot
import matplotlib.pyplot as plt
plt.figure() # open new figure
for sheet in excelfile.sheet_names:
    df = excelfile.parse(sheet, skiprows=1, header=[0,1])
    plt.plot(df['L-Dehnung'], df['Ax-Spannung'], label=sheet)
plt.legend(loc=2)
plt.xlabel('Strain')
plt.ylabel('Stress in MPa')
plt.grid()
```



Scenario 2: Reading seismic data from binary (SU)-data

- SU format: 240 byte header + trace
- number of samples per trace ns in header at byte 115, sampling rate Δt at byte 117
- SU-Reference: https://github.com/JohnWStockwellJr/SeisUnix/wiki/Seismic-Unix-data-format

```
In [60]: import numpy as np
         import matplotlib.pyplot as plt
         sudata = './data/ozdata.10.su' # file is encoded in big-endian
         plt.figure()
         with open(sudata, 'rb') as f: # using python file objects
             f.seek(-1, 2) ; eof = f.tell() ; f.seek(0,0) # get file size
             while(f.tell() < eof-240):
                 tracl = int.from_bytes(f.read(4), byteorder='big') # read 4 bytes --> int; trace numbe
                 f.seek(115 - 5, 1) # seek to byte 115 in header omitting previously read bytes
                 ns = int.from_bytes(f.read(2), byteorder='big') # bytes 115 - 116; samples
                 dt = int.from_bytes(f.read(2), byteorder='big') / 1e6 # bytes 117 - 118; mu-sec
                 f.seek(240-118, 1) # seek to start of trace
                 trace = np.fromfile(f, dtype='>f4', count=ns)
                 if len(trace) == ns:
                     plt.plot(np.linspace(0, dt*ns, ns), tracl+trace/trace.max(), c='k')
```



1.6 Special tasks

What we have not spoken about:

- Multithreading
- Runtime analysis
- Code optimization

1.6.1 Multithreading

- `import threading`
- very complex
- programmer need to take care about:
  - synchronization of threads
  - locking of globally available elements (variable access-on-write, file-identifiers, sockets, streams, …)
- !!! Errors ***WILL*** occur !!!

1.6.2 Runtime analysis (Profiling)

- various possibilities:
  - simple: time measurement of one task using `import time`, `import timeit`
  - profiling of functions using `import cProfile`: detailed analysis
- finding of time-critical tasks
- when found:
  - ***doing nothing!***
  - consider replacement of already available library functions
  - using numpy and scipy-functions
  - rewriting code in C or Fortran and call them via `import ctypes` or `import numpy.f2py`
  - rewriting code to ***Cython*** != CPython

1.6.3 Code Optimization using cython

Cython is:

- a special language similar to C for writing Python-like code
- compiled into processor-specific code and linked against Python (not interpreted!)
- using Cython-functions like normal Python-functions

What you need:

- a working compatible C-Compiler
  - MSVC in the correct version (Windows)
  - gcc in the correct version on Linux
  - clang in the correct version on MacOS
- Cython-package

1.6.4 Example: Monte-Carlo-Approximation of Pi

Approximation of π can be done with following algorithm:

- randomly placing N dots in a unit square
- count dots, which fall into a quater of a unit circle
- will lead to $\frac{\pi}{4}$
- for $N \to \infty$

```
In [61]: # python-implementation
         import time
         import timeit
         import random

         def mc_pi(N: int = 1000000):
             in_ = 0
             for i in range(N):
                 x = random.random()
                 y = random.random()
                 if x*x + y*y < 1:
                     in_ += 1
             return float(4*in_)/N

         N = 1000000
         t1 = time.perf_counter()
         pi = mc_pi(N)
         print('Pi with %d iterations: %.8f (%.6fsec)' % (N, pi, time.perf_counter() - t1))
         print('Timeit with 50 runs  : %.6fsec' %
               (timeit.timeit('mc_pi(N)', setup='from __main__ import mc_pi, N', number=50)/50))
```

```
Pi with 1000000 iterations: 3.14064000 (0.208509sec)
Timeit with 50 runs  : 0.198341sec
```

1.6.5 Setup Cython

- various build-related files need to be set up correctly:
  - *.pyxbld with pyximport
  - setup.py on plain cython
- compiler need to be specified:
  - usually MSCV on Windows
  - gcc on Linux
  - clang on MacOS
- versions shoud be the same

```
In [3]: import sys
        import subprocess
        import platform
        import shlex
        print(sys.version)
        # compilers need to be in search path
        compilers = {'Linux': 'gcc --version', 'Windows': 'cl.exe', 'Darwin': 'clang --version'}
        try:
            print(subprocess.check_output(shlex.split(compilers[platform.system()])).decode('ascii'))
        except:
            pass
        # In Jupyter you need to load the Cython-Magics and you're done!
        %load_ext Cython
```

```
3.7.7 (default, Mar 23 2020, 23:19:08) [MSC v.1916 64 bit (AMD64)]
```

```
In [4]: %%cython -a
        # output an annotated version of the code
        import numpy as np
        cimport numpy as np # import c-bindings of numpy
        def mc_pi1(long N = 1000000): # cython code 1
            cdef long i, in_=0
            cdef double x, y
            for i in range(N):
                x = np.random.random()
                y = np.random.random()
                if x*x + y*y < 1.0:
                    in_+=1
            return (4.0*in_)/N

Out[4]: <IPython.core.display.HTML object>

In [64]: N = 1000000
         t1 = time.perf_counter()
         pi = mc_pi1(N)
         print('Pi with %d iterations: %.8f (%.6fsec)' % (N, pi, time.perf_counter() - t1))
         print('Timeit with 50 runs  : %.6fsec' %
               (timeit.timeit('mc_pi1(N)', setup='from __main__ import mc_pi1, N', number=50)/50))

Pi with 1000000 iterations: 3.14155600 (0.436344sec)
Timeit with 50 runs  : 0.434828sec


In [80]: %%cython -a
         from libc.stdlib cimport rand, RAND_MAX
         def mc_pi2(long N = 10000000): # cython code 2
             cdef long in_ = 0
             cdef long i
             cdef double x, y
             for i in range(N):
                 x = rand() / RAND_MAX
                 y = rand() / RAND_MAX
                 if x*x + y*y < 1.0:
                     in_+=1
             return (4.0*in_)/N

Out[80]: <IPython.core.display.HTML object>

In [66]: N = 1000000
         t1 = time.perf_counter()
         pi = mc_pi2(N)
         print('Pi with %d iterations: %.8f (%.6fsec)' % (N, pi, time.perf_counter() - t1))
         print('Timeit with 50 runs  : %.6fsec' %
               (timeit.timeit('mc_pi2(N)', setup='from __main__ import mc_pi2, N', number=50)/50))

Pi with 1000000 iterations: 3.14166800 (0.030010sec)
Timeit with 50 runs  : 0.030105sec


In [81]: %%cython -a
         import cython
         from libc.stdlib cimport rand, RAND_MAX
```

```
@cython.boundscheck(False)
def mc_pi3(long N = 10000000): # cython code 3
    cdef long in_ = 0
    cdef long i
    cdef double x, y
    for i in range(N):
        x = rand() / RAND_MAX
        y = rand() / RAND_MAX
        if x*x + y*y < 1.0:
            in_+=1
    return (4.0*in_)/N
```

```
Out[81]: <IPython.core.display.HTML object>
```

```
In [68]: N = 1000000
         t1 = time.perf_counter()
         pi = mc_pi3(N)
         print('Pi with %d iterations: %.8f (%.6fsec)' % (N, pi, time.perf_counter() - t1))
         print('Timeit with 50 runs  : %.6fsec' %
               (timeit.timeit('mc_pi3(N)', setup='from __main__ import mc_pi3, N', number=50)/50))
```

```
Pi with 1000000 iterations: 3.14131200 (0.031021sec)
Timeit with 50 runs  : 0.030042sec
```

1.6.6 Conclusion

- Code optimizations using cython very easy
- You need to know what you are doing
- Knowlegde in system architecture is needed (C datatypes vs. Python datatypes)
- Parallelization using OpenMP possible, but not platform-independent
- Setup of Cython is easier in Linux (gcc already installed and configured), on Windows correct compiler version needed
- usually a performance increase of factor 10 is possible

```
In [ ]:
```