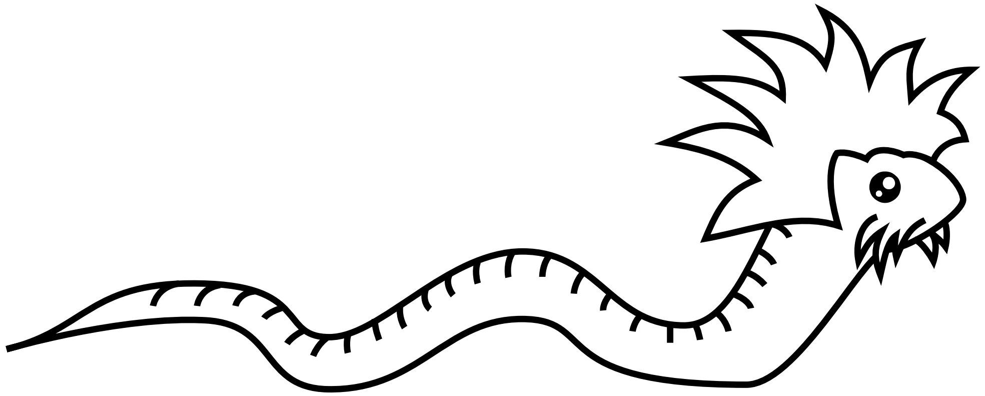
Python 3 for scientific computing

Lecture 5, 21.2.2018
Introduction and overview of scientific libraries

Juha Jeronen
juha.jeronen@tut.fi



Spring 2018, TUT, Tampere
RAK-19006 Various Topics of Civil Engineering

TAMPERE UNIVERSITY OF TECHNOLOGY

Introduction to scientific Python

The four main libraries

SciPy
https://scipy.org/

SymPy
http://www.sympy.org/

NumPy
http://www.numpy.org/

Matplotlib
http://matplotlib.org/

NumPy

http://www.numpy.org/

- MATLAB style API for *n*-dimensional array operations
  - Formalism: cartesian tensors (lecture 1)
  - **Use**: `np.array`; **do not use**: `np.matrix`
  - 0-based indexing, supports slicing, looks like a list; ***n*-dimensional**
  - C storage order (by default): the last index changes the fastest
  - `+`, `-`, `*`, `/`, `**` operate **elementwise** (like MATLAB's `.+`, `.-`, ...)
  - Matrix product @ (Python 3.5+), or `a.dot(b)`
  - Einstein summation notation: `np.einsum()`

- Under the hood: BLAS, LAPACK (like in MATLAB, Fortran)

- Python ≠ MATLAB:
  - Remember behavior of assignment, call-by-sharing (lectures 1–3)
  - Case sensitive; `Arr` and `arr` are different names.

- Single-threaded, by design (*explicit is better than implicit*)
- Matrices always generic (*simple is better than complex*)

NumPy

http://www.numpy.org/

- **Indexing**:

```
import numpy as np
A = np.arange(12).reshape(3,4)  # 2-dimensional (rank-2)

A[2,3]  # row 2, column 3     → one element, scalar
A[:,3]  # all rows, column 3  → vector (rank-1)
A[2,:]  # row 2, all columns  → vector (rank-1)
A[:,:]  # all
A[...]  # all; "..." means as many ":" as needed
A[:]    # all (special case)
```

- "A" **is still a rank-2 -tensor** (`A.ndim == 2`):

```
A[:] = np.random.random(12)  # ValueError
```

The shape on the RHS (12) is incompatible with the LHS shape (3,4).

```
A[:] = np.random.random(12).reshape(A.shape)  # OK
```

NumPy

http://www.numpy.org/

- ***C storage order*** …by default
  - *When accessing memory sequentially, the **last** index of the array changes the fastest (a.k.a. **row-major order**).*
  - *Sometimes important to remember, in order to maximize performance.*

```
import numpy as np
A = np.arange(12).reshape(3,4)
B = np.array(A, order='F')  # Fortran storage order!
```

Now, in “B”, the **first** index changes the fastest (***column-major order***).

But:

```
B[2,3]  # this still means row 2, column 3
```

In NumPy, the memory storage order of an array **does not** affect the order in which the indices are given. (Contrast C, Fortran.)

NumPy

http://www.numpy.org/

- Arrays are **created explicitly**:

```
import numpy as np

A = np.arange(10)  # [0, 1, ..., 9] → np.array
Z = np.zeros((5,5), dtype=np.float64)  # only zeros
O = np.ones((5,5),  dtype=np.float64)  # only ones
I = np.eye(5, dtype=np.float64)        # ident. matrix
D = np.diag(np.arange(10))             # diagonal mat.
M = np.array( [[1, 2, 3],   # Python lists → np.array
               [4, 5, 6],
               [7, 8, 9]], dtype=np.float64)
L = M.tolist()   # np.array → Python lists
K = np.array(M)  # copy (by calling constructor)
C = M.copy()     # copy, more explicit notation

V = M            # DANGER: new name, same instance!
W = M[:]         # DANGER: new view into the same memory!
```

NumPy

http://www.numpy.org/

- **Views** – new object instance, same memory:

```
import numpy as np

A = np.arange(12).reshape(3,4)
L = A.reshape(-1)  # Linear (1-index, raveled) view
R = A[0,:]         # Also a view (writing mutates “A”)
```

- **Coercion** into an array and back:

```
my_scalar = 42.0
B = np.atleast_1d(my_scalar)  # → array [42.0]
b = np.squeeze(B)             # → scalar 42.0
```

(Squeeze removes any length-1 axes, a.k.a. *singleton dimensions*.
Can be used to simplify code that needs to handle both scalars and arrays.
See also `atleast_2d`, `atleast_3d`.)

NumPy

http://www.numpy.org/

- **Subarrays and subsequences**:

```
import numpy as np

A = np.arange(12).reshape(3,4)

r = np.array((0,1,2), dtype=int)
c = np.array((1,2),   dtype=int)
A[np.ix_(r,c)]  # → subarray, rows r and columns c

r = np.array((0,2), dtype=int)
c = np.array((1,2), dtype=int)
A[r,c]          # → subsequence, [A[0,1], A[2,2]]
```

The latter is equivalent to (but vectorized):

```
[A[i,j] for i,j in zip(r,c)]
```

NumPy

http://www.numpy.org/

- **Index conversion**:
  - *np.ravel_multi_index*, *np.unravel_index*, *np.ravel*
  - https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

```
x = np.array([[1,2,3], [4,5,6]])
print(np.ravel(x))  # [1 2 3 4 5 6]
```

For example:

```
def genidx2D(nx, ny):
    """Generate index vectors to meshgrid and corresponding raveled array."""
    xx = range(nx)
    yy = range(ny)
    X,Y = np.meshgrid(xx,yy, indexing='ij')
    Xlin = np.reshape(X,-1)
    Ylin = np.reshape(Y,-1)
    XY = np.ravel_multi_index((Xlin,Ylin), (nx,ny))
    return Xlin, Ylin, XY
```

NumPy

http://www.numpy.org/

- **Creating a vector by slicing**:
  - In Python, slicing only allowed in index expressions (contrast MATLAB), so `np.r_`:

```
a = np.r_[0:100:2]  # → [0, 2, 4, …, 98]
```

  - More popular, and also **more pythonic**:

```
a = np.arange(0, 100, 2)
```

  - Sometimes also seen in the wild:

```
a = np.linspace(0, 98, 50)
```

NumPy

http://www.numpy.org/

Broadcasting:

- *[NumPy] Performing mathematical operations on arrays, which are differently shaped in a compatible way.*

- E.g. scalars broadcast to the whole array:

```
import numpy as np
A = np.arange(12).reshape(3,4)
B = A + 42
C = 2 * np.arange(4)
```

Arrays broadcast on the leading (first) axes, if the lengths of the trailing (last) axes match:

```
D = A + C  # [row + C for row in A], vectorized
```

- Online material:
  - https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html
  - https://eli.thegreenplace.net/2015/broadcasting-arrays-in-numpy/

NumPy

http://www.numpy.org/

- **Example**: creating a tridiagonal matrix:

```
# create a discretized 1D laplacian
#
n = 10
d = -2 * np.ones((n,), dtype=np.float64)
s = np.ones((n-1,), dtype=np.float64)
d = np.diag(d)       # main diagonal
u = np.diag(s, +1)  # upper subdiagonal
l = np.diag(s, -1)  # lower subdiagonal
T = l + d + u
```

- Slightly more on NumPy, see lecture notes, pp. 17–22; 37; 41–43

- User manual:
  https://docs.scipy.org/doc/numpy/index.html

- Coming from MATLAB? Slightly old, but mostly still good:
  https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html

Matplotlib

http://matplotlib.org/

- The main plotting library in the scientific Python ecosystem.
  - Has both procedural (matplotlib.pyplot) and object-oriented APIs.
  - The procedural API is very similar in spirit to MATLAB's.

- Main goal: high-quality scientific graphics for printing.

- Other visualization libraries:
  - ggplot (*Grammar of Graphics*)
  - VisPy (GPU, realtime) [YouTube] [paper]

- On the lecture, we have only some basic examples.

- The official Gallery contains many examples, and codes that produce them: https://matplotlib.org/gallery.html

Matplotlib

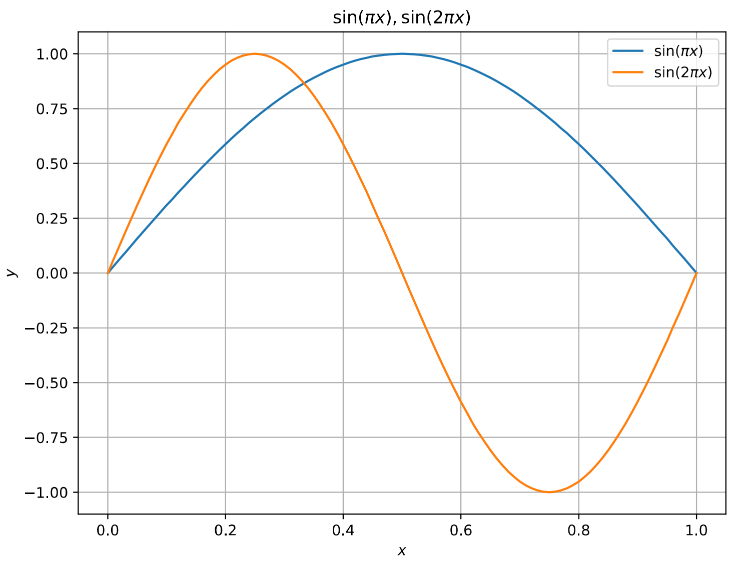
http://matplotlib.org/

- **Basic usage**:

```
import numpy as np
import matplotlib.pyplot as plt

xx = np.linspace(0, 1, 101)
yy1 = np.sin(np.pi * xx)
yy2 = np.sin(2 * np.pi * xx)

plt.figure(1, figsize=(8,6))  # size optional
plt.clf()  # clear (Spyder does not close)
plt.plot(xx, yy1, label=r'$\sin(\pi x)$')  # internal TeX interpreter
plt.plot(xx, yy2, label=r'$\sin(2 \pi x)$')  # hold enabled by default
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'$\sin(\pi x), \sin(2 \pi x)$')  # if you use plt.subplot, see also plt.suptitle
plt.grid(b=True, which='both')
plt.legend(loc='best')
plt.savefig('sin_x.svg')  # or .pdf, .png, etc.
```



Matplotlib

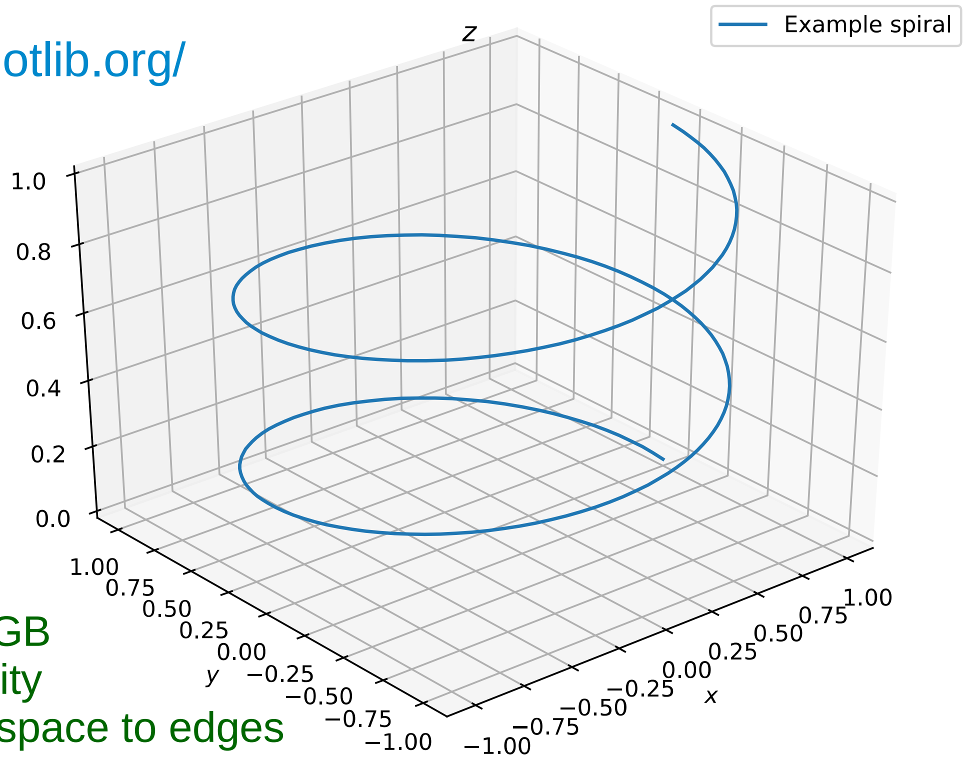
http://matplotlib.org/

- **3D plotting**:

```
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as axes3d

tt = np.linspace(0, 4*np.pi, 1001)
xx = np.cos(tt)
yy = np.sin(tt)
zz = np.linspace(0, 1, len(tt))

fig = plt.figure(1)
fig.patch.set_color((1,1,1))  # fig. background, RGB
fig.patch.set_alpha(1.0)  # fig. background, opacity
left_bot_w_h = [ 0.02, 0.02, 0.96, 0.96 ]  # more space to edges
ax = axes3d.Axes3D(fig, rect=left_bot_w_h)
# see also ax.plot_wireframe, ax.plot_surface, ax.plot_trisurf
ax.plot(xx, yy, zz, label='Example spiral')
ax.view_init(34, -130)  # elev, azim
ax.axis('tight')
ax.legend(loc='best')
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
ax.set_title(r'$z$')  # note! No “zlabel” due to Matplotlib's history as a 2D plotter.
plt.savefig('spiral.svg')
```



SymPy

http://www.sympy.org/

- CAS (computer algebra system) for Python
  - Slower than separate CAS software packages
  - But to Python programs, just a library like any other; excellent integration
    - Python equivalent for MATLAB's Symbolic Math Toolbox
  - Compare e.g. SageMath

- Symbolic algebra, differentiation, integration
  - As much a *CAS construction kit* as a CAS system: sometimes useful to define custom routines to process symbolic math expressions

- User manual: http://docs.sympy.org/latest/index.html

- Tutorial: http://docs.sympy.org/latest/tutorial/index.html

- On this lecture, just a few examples.

SymPy

http://www.sympy.org/

- **Symbols**:

  ```
  import sympy as sy
  ```

  ```
  x, y, z = sy.symbols('x, y, z')
  ```

- Can specify assumptions through kwargs, e.g. real=**True**

- Symbolic expressions in SymPy are Python expressions:

  ```
  p = x**2 + 2*y**3
  ```

- ...but careful with constants, to prevent Python from converting to float before SymPy gets access to the value:

  ```
  q = sy.S('5/3') * z
  ```

- Some predefind constants: sy.S.Zero, sy.S.One, sy.S.exp1, sy.S.Infinity, …

- An expression is a tree. If you manipulate with your own routines, preserve the invariant: for an expression expr, args is either empty, or expr == expr.func(*expr.args).

SymPy

http://www.sympy.org/

- **Matrices**:

```
εxx,εyy,εzz,εyz,εzx,εxy = sy.symbols('εxx, εyy, εzz, εyz, εzx, εxy')
exx,eyy,ezz,eyz,ezx,exy = sy.symbols('exx, eyy, ezz, eyz, ezx, exy')

ε = sy.Matrix( [[εxx, εxy, εzx],  # Cauchy strain
                [εxy, εyy, εyz],
                [εzx, εyz, εzz]] )
e = sy.Matrix( [[exx, exy, ezx],  # deviatoric strain
                [exy, eyy, eyz],
                [ezx, eyz, ezz]] )

εM_expr = sy.factor(sy.S('1/3') * ε.trace())  # mean volumetric strain
e_expr  = ε - εM_expr * sy.eye(3)

# check the symmetry
assert e_expr[1,0] == e_expr[0,1]  # exy
assert e_expr[2,0] == e_expr[0,2]  # ezx
assert e_expr[1,2] == e_expr[2,1]  # eyz
```

- **Chain rule** (lecture 1):

SymPy

http://www.sympy.org/

```
import sympy as sy
from sympy.core.function import UndefinedFunction

def nameof_as_symbol(sym):
    if hasattr(sym, 'name'):
        return sy.symbols(sym.name, **sym.assumptions0)
    else:  # an undefined function is anonymous, but its class has __name__
        return sy.symbols(sym.__class__.__name__, **sym.assumptions0)

def strip(expr):  # for printing: remove (maybe long) argument lists from unknown functions
    if isinstance(expr.__class__, UndefinedFunction):
        return nameof_as_symbol(expr)  # we strip args, no need to recurse into it
    elif expr.is_Atom:
        return expr
    else:  # compound other than an undefined function
        newargs = [strip(x) for x in expr.args]
        cls = type(expr)
        return cls(*newargs)

def main():
    x = sy.symbols('x')

    # Unknown function
    λf,λg = sy.symbols('f,g', cls=sy.Function)

    # Applied function
    g = λg(x)  # "g = g(x)"; the symbol name inside must be unique, so λg is single use only
    f = λf(g)  # f = f(g)

    # With the above definitions, SymPy automatically applies the chain rule:
    D = sy.diff(f, x).doit()
    sy.pprint(strip(D))

main()
```

Original:

```
  d                d
─────(f(g(x)))⋅──(g(x))
dg(x)              dx
```

strip(…):

```
d       d
──(f)⋅──(g)
dg      dx
```

SymPy
http://www.sympy.org/

- **Hermite polynomials**:

```
import sympy as sy

def hermite(k):  # Derive C**k continuous Hermite interpolation polynomials for the interval [0, 1]
    order = 2*k + 1
    *A,x = sy.symbols('a0:%d,x' % (order+1))

    w   = sum(a*x**i for i,a in enumerate(A))  # as a symbolic expression
    λw  = lambda x0: w.subs({x: x0})  # as a Python function; subs: symbolic substitution
    wp  = [sy.diff(w, x, i) for i in range(1,1+k)]  # diff: symbolic differentiation
    λwp = [(lambda expr: lambda x0: expr.subs({x: x0}))(expr) for expr in wp]  # why two lambdas: lecture notes sec. 5.8

    zero,one = sy.S.Zero, sy.S.One
    w0,w1 = sy.symbols('w0, w1')
    eqs  = [λw(zero) - w0, λw(one)  - w1]  # eqs. in form LHS = 0; see sy.solve()
    dofs = [w0, w1]

    for i,f in enumerate(λwp):
        d0_name = 'w%s0' % ((i+1) * 'p')  # p = 'prime', to denote differentiation
        d1_name = 'w%s1' % ((i+1) * 'p')
        d0,d1 = sy.symbols('%s, %s' % (d0_name, d1_name))
        eqs.extend([f(zero) - d0, f(one)  - d1])
        dofs.extend([d0, d1])

    coeffs = sy.solve(eqs, A)
    solution = sy.collect(sy.expand(w.subs(coeffs)), dofs)

    N = [solution.coeff(dof) for dof in dofs]  # result

    return tuple(zip(dofs, N))  # pairs (dof, interpolating function)
```

```
hermite(0)  # linear interpolation
((w0, -x + 1), (w1, x))

hermite(1)  # beam element
((w0, 2*x**3 - 3*x**2 + 1),
 (w1, -2*x**3 + 3*x**2),
 (wp0, x**3 - 2*x**2 + x),
 (wp1, x**3 - x**2))

hermite(2)  # 2nd derivative also continuous
((w0, -6*x**5 + 15*x**4 - 10*x**3 + 1),
 (w1, 6*x**5 - 15*x**4 + 10*x**3),
 (wp0, -3*x**5 + 8*x**4 - 6*x**3 + x),
 (wp1, -3*x**5 + 7*x**4 - 4*x**3),
 (wpp0, -x**5/2 + 3*x**4/2 - 3*x**3/2 + x**2/2),
 (wpp1, x**5/2 - x**4 + x**3/2))
```

SciPy

https://scipy.org/

- Python/NumPy style API to the libraries under the hood
  - LAPACK, ARPACK, SuperLU, UMFPACK

- **Sparse matrices**
  - SciPy rather good as-is
  - Need more algorithms? Add-on libraries:
    - Sparse Cholesky (CHOLMOD) → scikit.sparse
      - **Note!** There is also an old, obsolete *scikit**s**.sparse*
      - The current one called ***scikit.sparse***; module: **import** sksparse
    - SPQR (SuiteSparse QR) → sparseqr
      - Highly robust (works also for horribly scaled input)
      - Also works for overdetermined (least-squares) problems
      - Used also by MATLAB, `x = A \ b` when A is sparse

- SciPy also provides numerical integration (`quad`), initial value problems (ODE), special functions, signal processing, some basic optimization
  - Convex optimization? → CVXOPT

SciPy

https://scipy.org/

- User manual:
  - https://docs.scipy.org/doc/scipy/reference/index.html
- In the API reference, see especially:
  - https://docs.scipy.org/doc/scipy/reference/linalg.html
  - https://docs.scipy.org/doc/scipy/reference/sparse.html
  - https://docs.scipy.org/doc/scipy/reference/sparse.linalg.html
- Tutorials included in the manual, e.g.
  - https://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html
- Sometimes both NumPy and SciPy provide the “same” routine
  - NumPy provides a basic API, SciPy an advanced one (more options)

SciPy

https://scipy.org/

- **Example**: creating a sparse matrix:

```
import scipy.sparse

data = (1, 2, 3)  # matrix elements
drow = (0, 1, 4)  # their rows
dcol = (3, 2, 4)  # their columns
S = scipy.sparse.coo_matrix((data, (drow,dcol)),
                            shape=(5,5),
                            dtype=np.float64)
S = S.tocsr()   # → Compressed Sparse Row format
print(type(S))  # scipy.sparse.csr.csr_matrix
print(S)
```

- For more, see e.g. http://www.scipy-lectures.org/advanced/scipy_sparse/index.html

- Note that here, too, *explicit is better than implicit*: SciPy requires the user to specify the storage format (contrast MATLAB).

Meta

Next time

- More NumPy, SciPy, Matplotlib, SymPy.
- The third set of exercises.
- See you after the spring break, in two weeks!