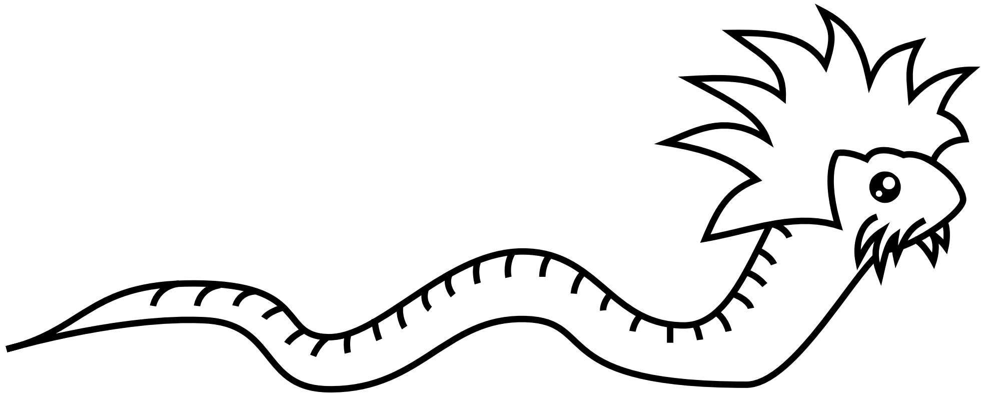
Python 3 for scientific computing

Lecture 6, 7.3.2018
The APIs: NumPy, SciPy, Matplotlib, SymPy

Juha Jeronen
juha.jeronen@tut.fi



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

TAMPERE UNIVERSITY OF TECHNOLOGY

NumPy

http://www.numpy.org/

- **Short recap**: **import** numpy **as** np
  - The core of high-performance numerics in Python.
  - Similar to MATLAB: vectorized array operations actually implemented as fast low-level loops, at the C (language) level.
  - Different from MATLAB: cartesian tensors.
  - *np.ndarray*: immutable size, mutable data. Looks almost like a Python list, but behaves quite differently. Views (to the same memory).
  - **Last time**, we talked about *how to allocate and index into arrays* in NumPy (lecture 5, slides 3–12).
- **Today**, we will cover a subset of the API, to show what NumPy offers.
- For the technical details of np.ndarray, universal functions and broadcasting, or other advanced topics – and the full API – consult the NumPy Reference.

NumPy

http://www.numpy.org/

- A subset of the NumPy API (as of v1.14):
  With headings as they appear in the reference.

☆

• Mathematical functions	– numerical, vectorized for *np.ndarray*s
• Linear algebra	– equation systems, eigenvalues
• Sorting, searching and counting	– *argmin*, *argmax*
• Logic functions	– *all*, *any*, et al. for *np.ndarray*s
• Random sampling	– many different distributions
• Statistics	– *average*, *median*, *percentile*, *corrcoef*
• Discrete Fourier Transform	
• Polynomials	
• Floating point error handling	– ***with*** *np.errstate(…):*
• Indexing routines	– raveling, indexing-like operations
• Masked array operations	
• NumPy-specific help functions	– keyword search: *np.lookfor('max')*

- ⚠ But:
  - Set routines – Not so useful, Python's *set* can be faster!

- Next, let's look at ☆ some of them in more detail.

NumPy

http://www.numpy.org/

- **Mathematical functions** [a sampling; see full reference]
  - Just import numpy; these live directly in the numpy namespace.
  - In the docs, the '/' in the param list means the end of anonymous, positional-only parameters. (Tech detail: functions that have them can only be defined via Python's C API. See PEP 436.)

  - **Trigonometrics, hyperbolics** – *sin*, *cos*, *tan*, *arcsin*, *arccos*, *arctan2*, *sinh*, *cosh*, *tanh*, *sinc*

```
import numpy as np
s = np.sin(np.pi / 4)          # angles in radians (as usual in scientific contexts)
x,y = (1, 3**(1/2))
alp = np.arctan2(y, x)         # quadrant-preserving two-argument atan
print(s, np.rad2deg(alp))
```

Listed under *Other special functions* in the reference.

  - **Rounding**: *floor*, *ceil*, *trunc*
  - **Sums, products, differences**: *prod*, *sum*, *cumprod*, *cumsum*, *diff*, *trapz*
  - **Exponents and logarithms**: *exp*, *log*, *log10*, *log2*, *log1p*
  - **Floating point routines**: *copysign*, *nextafter*, *spacing* (*ε = spacing(1.0)*)
  - **Complex numbers**: *angle*, *real*, *imag*, *conj*
  - **Arithmetic operations**: *add*, *subtract*, *multiply*, *divide*, *power*, *modf*
    - Mainly useful for their *out=…* kwarg.
  - **Miscellaneous**: *clip*; *sqrt*, *cbrt* (real!); *absolute* (complex); *fabs* (real only); *heaviside*; *interp*
    - *interp* is a basic form of piecewise linear interpolation; mainly useful if you need only that, and don't want your program to depend on SciPy (which offers more options).
    - NumPy defines some aliases: *abs* → *absolute*; *max* → *amax*; *min* → *amin*
    - See also *nanmin*, *nanmax* (convenient with *np.nan* as a “not applicable” placeholder)

NumPy

http://www.numpy.org/

- **Linear algebra** [a sampling; see full reference]

  - **Dense linear algebra**. For sparse linear algebra, see SciPy.
  - Many of the routines can compute results for several matrices with one call. To do this, stack the matrices into the same array, with the last two axes indexing the matrix elements.

  - **Matrix and vector products**: *dot*, *linalg.multi_dot*, *vdot*, *inner*, *outer*, *matmul*, *tensordot*, *einsum*, *matrix_power*, *kron*

    - Except *multi_dot*, these live directly in the numpy namespace.

    - *einsum_path* can be used to compute a performance-optimal contraction order for *einsum*; in some cases this can help by several orders of magnitude.

    - The various dot/inner products behave slightly differently:

      - *dot(a,b)* sums over the <u>last</u> axis of *a* and the <u>second-to-last</u> of *b* (historically used for matrix multiplication).
      - *linalg.multi_dot(a,b,c,d*) chains *dot*, optimizing evaluation order automatically.
      - *inner* is the standard symmetric bilinear form of vectors (**no** complex conjugation!).
      - *vdot* is the inner product of vectors; **complex-conjugates** <u>first</u> argument (if complex) so that vdot induces a norm. A.k.a. hermitian form; symmetric sesquilinear form.
      - *matmul* is the matrix product, same as the @ operator.
      - *tensordot* is a general tensor contraction, i.e. a sum product over arbitrary axes.
      - *einsum* is a general array operation using the Einstein summation convention.

NumPy

http://www.numpy.org/

- **Linear algebra** [a sampling; see full reference]   **import** numpy.linalg
  - These live in the *numpy.linalg* namespace, except *trace* directly in *numpy*.
  - **Matrix decompositions**: *cholesky*, *qr*, *svd*
  - **Eigenvalues and eigenvectors**: *eig*, *eigh*, *eigvals*, *eigvalsh*
    - As usual, right eigenvectors: $\boldsymbol{A}\,\boldsymbol{x} = \lambda\,\boldsymbol{x}$ (contrast left eigenvectors, $\boldsymbol{A}^{\mathrm{H}}\,\boldsymbol{y} = \bar{\lambda}\,\boldsymbol{y}$)
    - *-h*: version for hermitian (or symmetric real) $\boldsymbol{A}$
    - *-vals*: eigenvalues only (no eigenvectors), faster
    - See the _*geev* routines in LAPACK (dgeev for real, zgeev for complex; double precision)
  - **Linear equation systems**: *solve*, *tensorsolve*, *lstsq*, *inv*, *pinv*, *tensorinv*
    - *solve*: $\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$, standard linear equation system.
    - *tensorsolve*: all axes of $\boldsymbol{x}$, and the rightmost axes of $\boldsymbol{A}$, are summed over.
      - A solver to invert *tensordot* for a given RHS, in the same sense as *solve* inverts *dot*.
    - *lstsq*: least-squares solution, allowing under- or overdetermined equation systems.
    - *inv*: matrix inverse.
    - *pinv*: Moore–Penrose pseudoinverse.
    - *tensorinv*: related to *tensordot* in the same sense as *inv* is to *dot*.
  - **Norms and other numbers**: *norm*, *cond*, *det*, *matrix_rank*, *slogdet*, *trace*
    - *norm*: several different matrix and vector norms.
    - *matrix_rank*: uses SVD to determine the rank, in the sense of linear algebra.

NumPy

http://www.numpy.org/

- **Sorting, searching and counting**:
  - **Sorting**: *sort*, *lexsort*, *argsort, partition, argpartition*
  - **Searching**: *argmin*, *argmax*, *nanargmin*, *nanargmax*, argwhere, *nonzero*, *where*, *searchsorted*, *extract*
    - *where(condition, a, b)*: choose elements from *a* or *b* based on *condition*.
    - *nonzero(a)*: return multi-indices of nonzeros, in transposed format suitable for indexing.
      - output: tuple of arrays, where the *k*th element is an index sequence for the *k*th axis.
    - *argwhere(a)*: return multi-indices of nonzeros, in human-readable format.
      - output: rank-2 array, *k*th row contains the multi-index of the *k*th nonzero element.
  - **Counting**: *count_nonzeros*
  - Difference to MATLAB: NumPy's *max* and *sort* return <u>only the data</u>, no indices. Hence, use:

```
import numpy as np
x = np.random.randint(10, size=(5,))
idx = np.argsort(x)  # get indices that would sort x
x[idx]               # then get the sorted data by indexing the original
```

  - Searching for elements that satisfy a condition, we can also:

```
mask = x > 3    # create a boolean array, same shape as x...
x[mask]         # ...which can be used for indexing
```

NumPy

http://www.numpy.org/

- **Logic functions**:
  - **Truth value testing**: *any*, *all*
    - For *np.arrays*. Contrast Python's built-in **any**, **all**, which operate on iterables.
  - **Array contents**: *isfinite*, *isinf*, *isnan*
  - **Array type testing**: *iscomplex*, *isreal*
  - **Logical operations**: *logical_and*, *logical_or*, *logical_not*, *logical_xor*
    - Elementwise.
    - Sometimes seen in the wild: * instead of *logical_and*, + instead of *logical_or*.
    - Boolean algebra is essentially integer arithmetic in $\mathbb{Z}_2$; although strictly speaking [1] [2], + maps to *xor*. Usual software implementations of booleans, however, treat any number ≥ 1 as truthy, so + behaves like *or*.
  - **Comparison**: *allclose*, *isclose*, *array_equal*, *array_equiv*, *greater*, *greater_equal*, *less*, *less_equal*, *equal*, *not_equal*
    - Useful mostly for the *out=… kwarg*, and the fact that these accept also Python tuples.
    - With *np.arrays*, can instead use Python's operators >, >=, <, <=, ==, !=

NumPy

http://www.numpy.org/

- **Random sampling**: **import** numpy.random
  - **Simple random data**: *rand* (uniform), *randn* (normal), *randint* (0 to *n* – 1)
    - Also *random* (i.e. *numpy.random.random()*); almost the same as *rand*, but takes the size as a tuple, whereas *rand* takes separate positional arguments.
  - **Permutations**: *shuffle*, *permutation*
  - **Distributions**: *beta*, *binomial*, *chisquare*, *dirichlet*, …, *weibull*, *zipf*.
    - Most omitted here; e.g. NumPy v1.14 supports 45 different distributions.
  - **Random generator**: *seed*, *get_state*, *set_state*
    - Initializes automatically (like in MATLAB; unlike in C), so these low-level routines are usually not needed by Python programs.
    - Mainly useful if we want to produce always the same pseudorandom sequence, for unit testing purposes (by resetting the RNG with the same seed at the start of the test).
    - Algorithm used by NumPy: Mersenne Twister

NumPy

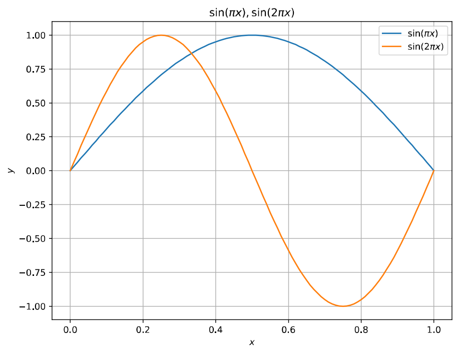
http://www.numpy.org/

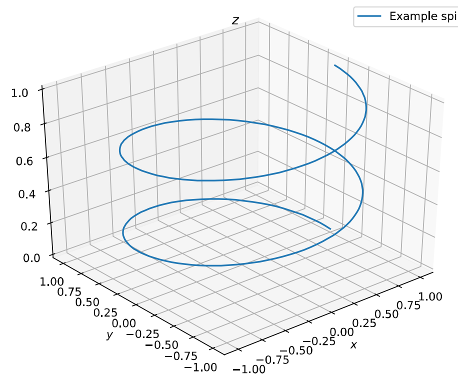
- **Statistics**:
  - **Order statistics**: *amin*, *amax*, *nanmin*, *nanmax*, *ptp*, *percentile*, *nanpercentile*
  - **Averages and variances**: *median*, *average*, *mean*, *std*, *var*
    - *average* can be weighted; *mean* is the arithmetic mean.
  - **Correlating**: *corrcoef*, *correlate*, *cov*
    - *corrcoef*: Pearson product-moment correlation coefficients – roughly, in the 2D case, a measure of linear dependence between x and y coordinates in a set of points.
    - *correlate*: cross-correlation of two 1D sequences (signal processing), a.k.a. sliding dot product. A measure of similarity between two time series.
    - *cov*: covariance matrix.
  - **Histograms**: *histogram* (1D or flattened), *histogram2d*, *histogramdd*, *bincount*, *digitize*

Matplotlib

http://matplotlib.org/

- **Short recap**: **import** matplotlib.pyplot **as** plt # mostly enough
  - Tool of choice for print-quality scientific graphics in Python.
  - Procedural and object-oriented APIs.
  - History as a 2D plotter; 3D was added later.
  - **Last time**, we talked about basic 2D plotting and 3D plotting (lecture 5, slides 13–15).





- Large library; the full PDF manual is ≈2200 pages. Online version.
- **Today**, let's go online and look at the gallery (too large to inline here).

Matplotlib

http://matplotlib.org/

- **Some interesting categories** in the gallery (as of v2.1.2):

  - Lines, bars and markers
  - Images, contours and fields
  - mplot3d – extensive set of examples on 3D plotting: surfaces, wireframes, contours, …
  - Statistical plots
  - Subplots, axes and figures
  - Color – demonstrating colormaps that come with matplotlib. (It's important.)
  - Axis scales [what's a logit?]
  - Text, labels and annotations
  - Ticks and spines
  - Pie and polar charts
    - See also geo_demo; radar_chart; custom curvilinear grid
  - API – some specific tricks
  - pylab examples – many useful specific examples, misleading title
    - pylab is, essentially, the following plus a few other imports, for convenient interactive use:

      ```
      from matplotlib.pyplot import *
      from numpy import *
      ```

    - In IPython (including the Spyder REPL), the command **pylab** loads it.
    - The examples in “pylab examples” don't actually need pylab; they import numpy and matplotlib separately, in the usual way.

  - axes_grid – heavy customization (advanced)

SymPy

http://www.sympy.org/

- **Short recap**: import sympy as sy # ...or as sp
  - For automating your pen-and-paper math: symbolic algebra, integration, differentiation.
  - Slower than a dedicated CAS, but as a Python library, superb integration into the workflow when writing a Python application (or solver).
  - As much a CAS construction kit as a ready-made CAS.
  - **Last time**, the very basics of symbols and symbolic matrices; some complete examples (lecture 5, slides 16–20).
- **Today**, a sampling of the API.
- Consult the extensive documentation as needed: Tutorial, Gotchas and pitfalls, User's guide, Modules reference.

SymPy

http://www.sympy.org/

- A subset of the SymPy API (as of v1.1.1):
  - Core – see esp. methods of Expr
  - Mathematical functions – **symbolic**; also unknown functions
  - Simplification
  - Term rewriting – expand, cse (common subexpression elimination)
  - Polynomials
  - Symbolic integrals
  - Solvers (also old API useful); Inequalities; ODEs; PDEs
  - Numerical evaluation – for converting a symbolic result into a number
  - Numerical computation – for creating NumPy-compatible Python **lambda**s
  - Assumptions – e.g. real, positive, commutative, … see *help(sy.Q)*
  - Matrices
  - Holonomic functions
  - Plane geometry
  - Lie algebras
  - Logic – boolean expressions, simplification, satisfiability
  - Printing – pretty-printing, LaTeX output
  - Utilities – esp. Codegen (if you Fortran, make SymPy code for you)

SymPy

http://www.sympy.org/

- **Mathematical functions**:
  - Trigonometric: *sin*, *cos*, *tan*, *cot*, *sec*, *csc*, *sinc, asin, acos, atan, acot, asec, acsc, atan2*
  - Hyperbolic: *sinh, cosh, tanh, coth, sech, csch, asinh, acosh, atanh, acoth, asech, acsch*
  - *exp*, *log, root, sqrt*
  - *Min*, *Max* (of a list of symbols, utilizing any assumptions)
  - See also combinatorial and special functions
- **Simplification**:
  - Methods of Expr:
    - *.expand()*, *.collect()*, *.factor()*, *.together()*
    - *.combsimp()*, *.radsimp()*, *.ratsimp()*, *.trigsimp()*
    - Some of these are equivalent to functions defined elsewhere in SymPy.
  - *cse()*
- **Calculus**:
  - *integrate*, *diff, idiff* (implicit)
  - The result of *diff* may sometimes need applying *.doit()*

SymPy

http://www.sympy.org/

- **Solving equations**:
  - *solve*: solve a set of equations (where RHS is zero; give only LHS) for given variables.
  - *solveset*: like solve, but returns a set of results, and behaves more consistently.
    - Relatively new; intended to eventually replace *solve*, but not yet at feature parity.
    - Example:
      ```
      import sympy as sy
      x = sy.symbols('x')
      sy.pprint(sy.solveset(sy.sin(x), x))    # {2·n·π | n ∊ ℤ} ∪ {2·n·π + π | n ∊ ℤ}
      ```
    - Inequalities (over ℝ) also allowed.
- **Printing**:
  - *pprint*: pretty-printer for the terminal / REPL session (see also *init_printing*)
  - *latex*: get LaTeX representation of given expr
  - For more, see tutorial.

SciPy

https://scipy.org/

- **Short recap**: **import** scipy.linalg # or some other specific module
  - Beside NumPy, the other main library for numerics.
  - Sparse matrices, numerical integration (quad), initial value problems for ODEs, special functions, signal processing, some basic optimization routines.
  - Especially in the case of (dense) linear algebra, may provide an advanced version (with more options) of the “same” routine NumPy already offers.
  - Still not a one-stop shop; add-ons sometimes useful (e.g. sparseqr).
  - **Last time**, we briefly talked about sparse matrices, and noted that SciPy expects the user to specify the storage format (more explicit this way).
- **Today**, we will look at a sampling of the API. For more and a tutorial, see the full reference.

SciPy

https://scipy.org/

- A subset of the SciPy API (as of v1.0.0):

☆
  - Linear algebra [tutorial] – advanced routines
  - Sparse matrices – e.g. COO, CSC, CSR, LIL [explanation at Wikipedia]
  - Sparse linear algebra – *spsolve()*, *factorized()*, *bicgstab()*, *gmres()*, *lsqr()*
  - Integration and ODEs [integration tutorial]
  - Special functions [tutorial] – Airy, Bessel, gamma, beta, hypergeometric, ...
  - Optimization and root finding [tutorial]
  - Interpolation [tutorial]
  - Signal processing [tutorial] – 1D and 2D
  - Discrete Fourier transforms [tutorial]
  - Spatial [tutorial] – Delaunay, Voronoi, convex hull, k-d-trees
  - Compressed sparse graphs – for example, for word ladders
  - Statistical functions [tutorial]
  - Orthogonal distance regression – a cousin of least-squares regression
  - Clustering – see esp. hierarchical
  - Input and output – esp. MATLAB .mat files
  - Constants

To find nearest neighbors quickly in a set of points.

SciPy

https://scipy.org/

- **Linear algebra** [a sampling, see full reference]: **import** scipy.linalg
  - **Dense** linear algebra. More routines, and advanced versions.
  - **Basics**: *solve*, *solve_banded*, *solveh_banded*, *solve_triangular*, *lstsq*, *orthogonal_procrustes*, *matrix_balance*, *subspace_angles*, *det*, *norm*, *inv*, *pinv*, *pinv2*, *pinvh*, *kron*, *tril*, *triu*
    - *-h*: version for hermitian matrix
    - *solve*: options for generic, symmetric, hermitian, positive definite
    - *lstsq*: option to select the LAPACK driver (default _GELSD; available _GELSS, _GELSY)
    - *pinv*: least-squares algorithm; *pinv2*: SVD algorithm, using all “large” singular values
    - *orthogonal_procrustes(A,B)*: find a rotation/reflection that most closely maps $\boldsymbol{A}$ to $\boldsymbol{B}$
    - *matrix_balance*: equalize the 1-norm of the columns/rows of a matrix (also used internally)
    - *subspace_angles*: principal angles between subspaces of two matrices
  - **Eigenvalue problems**: *eig*, *eigvals*, *eigh*, *eigvalsh*, *eig_banded*, *eigvals_banded*, *eigh_tridiagonal*, *eigvalsh_tridiagonal*
    - Both standard $\boldsymbol{A}\,\boldsymbol{x} = \lambda\,\boldsymbol{x}$ and generalized $\boldsymbol{A}\,\boldsymbol{x} = \lambda\,\boldsymbol{B}\,\boldsymbol{x}$ eigenvalue problems.
    - Also <u>left</u> eigenvectors, $\boldsymbol{A}^{\mathrm{H}}\,\boldsymbol{y} = \bar{\lambda}\,\boldsymbol{B}^{\mathrm{H}}\,\boldsymbol{y}$
    - *_banded*: hermitian or real symmetric banded matrix
    - *_tridiagonal*: real symmetric tridiagonal matrix

SciPy

https://scipy.org/

- **Linear algebra** [a sampling, see full reference]: **import** scipy.linalg
  - **Decompositions**: *lu*, *lu_factor*, *lu_solve*, *svd*, *svdvals*, *orth*, *cholesky*, *cholesky_banded*, *cho_factor*, *cho_solve*, *cho_solve_banded*, *polar*, *qr*, *rq*, *qz* (generalized Schur), *schur*, *hessenberg*
    - *_factor*, *_solve*: Separate factor/solve steps for quickly solving many linear equation systems with the same matrix but different RHS vector (amortized cost only O(n^2))
    - *lu, qr*, *rq*, *polar*, *schur*, *hessenberg*: general
    - *cholesky*: hermitian (or real symmetric) positive definite
      - *cho_factor*: **for use with *cho_solve***; unused matrix entries contain random data
      - *cholesky*: for use anywhere, unused matrix entries zeroed out
    - *qz*: QZ decomposition for generalized eigenvalues of a pair of matrices.
    - *svd*: compute ***U***, ***Σ***, ***V***
    - *svdvals*: compute only the singular values ***Σ***
    - *orth*: orthonormal basis for the range of the matrix, via SVD

SciPy

https://scipy.org/

- **Linear algebra** [a sampling, see full reference]: **import** scipy.linalg
  - **Matrix functions**: *expm*, *logm*, *cosm*, *sinm*, *tanm*, *coshm*, *sinhm*, *tanhm*, *signm*, *sqrtm*, *funm*, *expm_frechet*, *expm_cond*, *fractional_matrix_power*
    - *funm*: custom function specified as a callable
  - **Matrix equation solvers**: *solve_sylvester*, *solve_continuous_are*, *solve_discrete_are*, *solve_continuous_lyapunov*, *solve_discrete_lyapunov*
    - *sylvester*: $\boldsymbol{A}\,\boldsymbol{X} + \boldsymbol{X}\,\boldsymbol{B} = \boldsymbol{Q}$
    - *are*: algebraic Riccati equation
      - *continuous-time*: $\boldsymbol{X}\,\boldsymbol{A} + \boldsymbol{A}^{\mathrm{H}}\,\boldsymbol{X} - \boldsymbol{X}\,\boldsymbol{B}\,\boldsymbol{R}^{-1}\,\boldsymbol{B}^{\mathrm{H}}\,\boldsymbol{X} + \boldsymbol{Q} = 0$
      - *discrete-time*: $\boldsymbol{A}^{\mathrm{H}}\,\boldsymbol{X}\,\boldsymbol{A} - \boldsymbol{X} - (\boldsymbol{A}^{\mathrm{H}}\,\boldsymbol{X}\,\boldsymbol{B})\,(\boldsymbol{R} + \boldsymbol{B}^{\mathrm{H}}\,\boldsymbol{X}\,\boldsymbol{B})^{-1}\,(\boldsymbol{B}^{\mathrm{H}}\,\boldsymbol{X}\,\boldsymbol{A}) + \boldsymbol{Q} = 0$
    - *lyapunov*:
      - *continuous*: $\boldsymbol{A}\,\boldsymbol{X} + \boldsymbol{X}\,\boldsymbol{A}^{\mathrm{H}} = \boldsymbol{Q}$
      - *discrete*: $\boldsymbol{A}\,\boldsymbol{X}\,\boldsymbol{A}^{H} - \boldsymbol{X} + \boldsymbol{Q} = 0$
  - **Sketches and random projections**: *clarkson_woodruff_transform*
  - **Special matrices**: *block_diag*, *companion*, *hadamard*, *hankel*, *helmert*, *hilbert*, *invhilbert*, *leslie*, *pascal*, *invpascal*, *toeplitz*, *tri*
  - **Low-level routines**: *get_blas_funcs*, *get_lapack_funcs*, *find_best_blas_type*
    - The low-level routines are usually not needed.

SciPy

https://scipy.org/

- **Sparse matrices** [a sampling, see full reference]: **import** scipy.sparse

  - Sparse matrices support arithmetic; they also have methods such as *dot()* for sparse matrix multiplication, and conversion methods such as *todense()*. See the reference for each class.

  - **Sparse matrix classes**, i.e. supported storage formats:

• *bsr_matrix*: Block Sparse Row	– for sparse matrices with dense sub-blocks
• *coo_matrix*: COOrdinate	– for building, automatically sums duplicates
• *csc_matrix*: Compressed Sparse Column	– fast linalg if accessed by column
• *csr_matrix*: Compressed Sparse Row	– fast linalg if accessed by row
• *dia_matrix*: DIAgonal	– e.g. tridiagonal or pentadiagonal
• *dok_matrix*: Dictionary Of Keys	– for building, O(1) access, **no** duplicates
• *lil_matrix*: List of lists: row-based linked list	– for building row by row

  - **Functions** to operate on sparse matrices:
    - **Building** sparse matrices: *eye*, *kron*, *kronsum*, *diags*, *block_diag*, *tril*, *triu*, *bmat*, *hstack*, *vstack*, *random*
      - *kronsum*: $\mathrm{kron}(\boldsymbol{I}_n, \boldsymbol{A}) + \mathrm{kron}(\boldsymbol{B}, \boldsymbol{I}_m)$ where $\boldsymbol{A}$ is of size (m,m) and $\boldsymbol{B}$ of size (n,n).
      - *bmat*: make sparse matrix from sparse sub-blocks

    - **Other**: *save_npz*, *load_npz*, *find*, *isspmatrix*
      - *npz* is NumPy's archive format
      - *find*: return indices and values of nonzeros
      - *isspmatrix*: return whether a given matrix is stored as a sparse matrix.
        - Also *isspmatrix_bsr*, *isspmatrix_coo*, ...

SciPy

https://scipy.org/

- **Sparse linear algebra** [a sampling, see full reference]:   **import** scipy.sparse.linalg

  - **Matrix operations, norms**: *inv*, *expm*, *expm_multiply*, *norm*, *onenormest*
    - *inv*: if the inverse is expected to be dense, faster to convert and use *scipy.linalg.inv()*.
    - *expm*: *exp(**A**)* using Pade approximation.
    - *expm_multiply*: compute *exp(**A**) **B***
    - *norm*: some norms supported, others not
    - *onenormest*: lower bound for the 1-norm

  - **Linear equation systems**:

    ⚠ Name **spsolve** instead of **solve**!

    - **Direct methods**: *spsolve*, *spsolve_triangular*, *factorized*, *use_solver*
      - *spsolve*: direct solver using UMFPACK or SuperLU.
      - *factorized*: return a solver function that accepts the RHS and returns solution.
      - *use_solver*: by default, SciPy uses UMFPACK if *scikits.umfpack* is installed, otherwise SuperLU. This can be used to switch UMFPACK off even if available.

    - **Iterative methods**: *bicg*, *bicgstab*, *cg*, *cgs*, *gmres*, *lgmres*, *minres*, *qmr*, *gcrotmk*, *lsqr*, *lsmr*
      - *bicg*, *bicgstab*, *cgs*, *gmres*, *lgmres*, *qmr* (real only), *gcrotmk*: general real or complex
      - *cg*: hermitian (or real symmetric) positive definite; *minres*: real symmetric
      - *lsqr*, *lsmr*: solve least-squares problems.
        - In least-squares problems, row scaling affects error measure; cannot be equalized.
        - Usually *lsqr* is fine, but for badly scaled problems sparseqr is more robust.

SciPy

https://scipy.org/

- **Sparse linear algebra** [a sampling, see full reference]: **import** scipy.sparse.linalg
  - **Matrix factorizations**:
    - **Eigenvalue problems**: *eigs*, *eigsh*, *lobpcg*
      - *eigs*, *eigsh*: general matrix; standard or generalized eigenvalue problem; right eigenvectors.
        - ARPACK, routines _*NEUPD*, Implicitly Restarted Arnoldi method.
      - *lobpcg*: real symmetric positive definite matrix; Locally Optimal Block Preconditioned Conjugate Gradient method.
    - **Singular value problems**: *svds*
      - k largest singular values and corresponding vectors.
      - Uses ARPACK as an eigensolver on $\boldsymbol{A}^{\mathrm{H}} \boldsymbol{A}$ or $\boldsymbol{A} \boldsymbol{A}^{\mathrm{H}}$, whichever is more efficient.
    - **LU**: *splu*, *spilu*
      - Complete or incomplete LU decomposition for a sparse square matrix.
  - **Abstract linear operators** [advanced]: *LinearOperator*, *aslinearoperator*
    - May be useful e.g. with Krylov subspace methods, which only need to compute the action of a linear operator on a vector (no need for an explicitly stored matrix representation).
- Which solver? $\lessapprox 10^3$ unknowns, dense ok; $\lessapprox 10^6$, sparse direct; more, sparse iterative.
- Least-squares and eigenvalue problems may benefit from switching to sparse much earlier.

SciPy

https://scipy.org/

- **Integration and ODEs** [a sampling, see full reference]: **import** scipy.integrate
  - **Integration**:
    - Callables: *quad*, *dblquad*, *tplquad*, *nquad*, *fixed_quad*, *quadrature*, *romberg*, *newton_cotes*
      - *quad*: adaptive, using QUADPACK
      - *dblquad*: integrate over a plane region between two curves
      - *tplquad*: integrate over a volume between two surfaces
      - *nquad*: integrate over a hypercube (cartesian product of 1D ranges)
      - *fixed_quad*: fixed-**order** Gaussian quadrature
      - *quadrature*: fixed-**tolerance** Gaussian quadrature
      - *romberg*: Romberg's method [essentially, Richardson-accelerated trapz]
      - *newton_cotes*: get weights and error coefficient for Newton–Cotes integration
    - Fixed samples: *trapz*, *cumtrapz*, *simps*, *romb*
  - **Initial value problems for ODEs**:
    - The standard first-order IVP:

      $$d\underline{u}/dt = f(u, t)\ ,\qquad u|_{t=0} = u_0\ .$$

    - *solve_ivp*: the interface routine; methods RK45, RK23, Radau (IIA, order 5), BDF, LSODA
      - LSODA comes from ODEPACK.

For quick reference

Cheat sheets

- DataCamp: Data Analysis in Python
- Dataquest: Python for Data Science
- Julian Gaal: python-cheat-sheet

- DataCamp: Linear Algebra in Python

- DataCamp: Plotting in Python

- SymPy cheatsheet
- SymPy Cheat Sheet

- Top 28 Cheat Sheets for Machine Learning, Data Science, Probability, SQL & Big Data
  (of which 9 for Python)

Meta

Next time

- Parallel computing in Python, smaller scientific libraries, behavior of floating point numbers.
- See you next week!