Introduction to Python

Part 2: Scientific Computing and Plotting

Michael Kraus (michael.kraus@ipp.mpg.de)

Max-Planck-Institut für Plasmaphysik, Garching

24. November 2011

Important Tools for Scientific Computing with Python

- NumPy: N-dimensional array manipulation
  http://numpy.scipy.org/
- SciPy: numerical routines, data manipulation and analysis
  http://www.scipy.org/
- Matplotlib: 2D plotting
  http://matplotlib.sourceforge.net/
- MayaVi: 3D plotting and interactive scientific data visualisation
  http://code.enthought.com/projects/mayavi/
- HDF5/netCDF: self-describing and efficient storage of scientific data
  http://www.hdfgroup.org/HDF5/
  http://www.unidata.ucar.edu/software/netcdf/
  http://code.google.com/p/h5py/
  http://code.google.com/p/netcdf4-python/

NumPy

- NumPy: fundamental package for scientific computing with Python

→ contains a powerful N-dimensional array object (`ndarray`) and various derived objects (such as matrices) and an assortment of routines for fast operations on these arrays (mathematical, shape manipulation, sorting, selecting, I/O, ...)

→ defines universal functions for arrays (`ufunc`): allow to apply a function to each element of a `ndarray` in one call

→ extensive capabilities for linear algebra, statistics, Fourier transforms, random simulations, ...

→ it's fast: most routines are optimised, pre-compiled C code

- don't forget to import the `numpy` module before using it:

```
>>> import numpy as np
```

NumPy: Arrays

- attributes of `ndarray`:
  - `shape`: tuple that gives the number of points in each dimension
  - `ndim` : number of dimensions
  - `size` : number of elements
  - `dtype`: data type (int64, float64, complex128, ..., structured data types)
- arithmetic operations of `ndarray`:
  - `a + b` : element-wise addition
  - `a - b` : element-wise subtraction
  - `a * b` : element-wise multiplication
  - `a / b` : element-wise division
- important functions of `ndarray`:
  - `dot(a)`: vector/matrix multiplication
  - `max()` : largest element
  - `min()` : smallest element
  - `prod()`: product of all elements
  - `sum()` : sum of all elements

NumPy: Creating Arrays

- create a 1D array from a list:

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> a.shape
(4,)
```

- create a 2D array from a list of two lists:

```
>>> b = np.array([[0, 1, 2], [3, 4, 5]])
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
```

NumPy: Creating Arrays

- arrays with regularly incrementing values:

  `arange([start,] stop[, step, dtype])`

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(1, 9, 2)
array([1, 3, 5, 7])
```

- arrays with equally/logarithmic spaced elements between the specified beginning and end values:

  `linspace(start, stop[, num=50, endpoint=True])`

  `logspace(start, stop[, num=50, endpoint=True, base=10.0])`

```
>>> np.linspace(0, 1, 6)
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> np.linspace(0, 1, 5, endpoint=False)
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
>>> np.logspace(-2, 2, 5)
array([  1.0000000e-02,   1.0000000e-01,   1.0000000e+00,
         1.0000000e+01,   1.0000000e+02])
```

NumPy: Creating Arrays

- array filled with zeros: `zeros(shape[, dtype])`

```
>>> a = np.zeros((2, 2))
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
```

- array filled with ones: `ones(shape[, dtype])`

```
>>> b = np.ones((3, 3), dtype=int)
>>> b
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

- allocated but empty arrays: `empty(shape[, dtype])`

```
>>> c = np.empty((2, 2))
>>> c
array([[ -2.00000000e+000,   1.49166824e-154],
       [  1.00000000e+000,   2.78134373e-309]])
```

NumPy: Creating Arrays

- create arrays that look like another array (shape and type):

`zeros_like(a[, dtype])`

```
>>> b = np.ones((2, 2), dtype=int)
>>> b
array([[1, 1],
       [1, 1]])
>>> np.zeros_like(b)
array([[0, 0],
       [0, 0]])
```

`ones_like(a[, dtype])`

```
>>> a = np.zeros((2, 2))
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.ones_like(a)
array([[ 1.,  1.],
       [ 1.,  1.]])
```

NumPy: Creating Arrays

- identity matrix: `eye(dim[, dtype=float, ...])`

```
>>> i = np.eye(3)
>>> i
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]]
```

- diagonal matrix: `diag(vec)`

```
>>> d = np.diag(np.array([1, 2, 3, 4, 5]))
>>> d
array([[1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 4, 0],
       [0, 0, 0, 0, 5]])
```

NumPy: Creating Arrays

- upper triangular matrix: `triu(m, k=0)`

→ returns a copy of `m` with the elements below the k-th diagonal zeroed

```
>>> m = np.ones((3,3)); m
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.triu(m)
array([[ 1.,  1.,  1.],
       [ 0.,  1.,  1.],
       [ 0.,  0.,  1.]])
>>> np.triu(m,1)
array([[ 0.,  1.,  1.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
```

- lower triangular matrix: `tril(m, k=0)`

```
>>> np.tril(m)
array([[ 1.,  0.,  0.],
       [ 1.,  1.,  0.],
       [ 1.,  1.,  1.]])
```

NumPy: Importing and Exporting Text Files

population.txt

```
1900        30.0e3  4.0e3   51300
1901        47.2e3  6.1e3   48200
...
```

- reading data:

```
>>> data = np.loadtxt('population.txt')
>>> data
array([[  1900.,  30000.,   4000.,  51300.],
       [  1901.,  47200.,   6100.,  48200.],
...
```

- writing data:

```
>>> np.savetxt('population_out.txt', data)
```

population_out.txt

```
1.900000000000000000e+03 3.000000000000000000e+04 ...
1.901000000000000000e+03 4.720000000000000000e+04 ...
...
```

NumPy: Importing and Exporting Text Files

```
loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None,
        converters=None, skiprows=0, usecols=None, ...)
```

- `fname`: file object or filename to read
- `dtype`: data-type of the resulting array, in the case of a record type, the number of columns used must match the number of fields in the record
- `comments`: character used to indicate the start of a comment
- `delimiter`: string used to separate values (by default, this is any whitespace)
- `converters`: dictionary mapping column number to a function that will convert that column to a float (can also be used to provide a default value for missing data)
- `skiprows`: skip the first `skiprows` lines
- `usecols`: which columns to read, e.g. `usecols = (1,4,5)` will extract the 2nd, 5th and 6th columns, the default results in all columns being read

NumPy: Importing and Exporting Text Files

```
savetxt(fname, X, fmt='%.18e', delimiter='', newline='\n',
        header='', footer='', comments='# ')
```

- `fname`: file object or filename to write to
- `X`: data to be saved to a text file.
- `fmt`: a single format (`%10.5f`), a sequence of formats (`%d, %10.5f`), or a multi-format string, e.g. `'Iteration %d - %10.5f'`
- `delimiter`: character separating columns
- `newline`: string that marks a new line
- `header`: string that will be written at the beginning of the file
- `footer`: string that will be written at the end of the file
- `comments`: string that will be prepended to the header and footer strings, to mark them as comments

NumPy: Indexing Arrays

- items of an array can be accessed and assigned to the same way as other Python sequences

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(0, 2, 9)
>>> a[3:7]
array([3, 4, 5, 6])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[5]  = -1
>>> a[6:] = 0
>>> a
array([ 0,  1,  2,  3,  4, -1,  0,  0,  0,  0])
```

→ in 2D, the first dimension corresponds to rows, the second to columns

→ for multidimensional `a`, `a[0]` is interpreted by taking all elements in the unspecified dimensions, e.g. in 3D, `a[0]` corresponds to `a[0,:,:]`

NumPy: Copies and Views of Arrays

- a slicing operation creates a view on the original array, which is just a way of accessing array data (the original array is not copied in memory)

→ when modifying the view, the original array is modified as well:

```
>>> a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]; b
array([0, 2, 4, 6, 8])
>>> b[0] = 10
>>> b; a
array([10,  2,  4,  6,  8])
array([10,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

→ if you want a copy of the data, you have to tell Python explicitly:

```
>>> a = np.arange(10)
>>> b = a[::2].copy()
>>> b[0] = 10
>>> b; a
array([10,  2,  4,  6,  8])
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy: Advanced Indexing

- masks:

```
>>> a = np.arange(10)
>>> (a % 2 == 0)
array([ True,  False, True,  False, True,
        False, True,  False, True,  False], dtype=bool)
>>> a[a % 2 == 0]
array([0, 2, 4, 6, 8])
```

- indexing with an array of integers:

```
>>> a[[2, 4]]
array([2, 4])
>>> a[[2, 4]] = a[[4,2]]
>>> a
array([0, 1, 4, 3, 2, 5, 6, 7, 8, 9])
```

- `slice` represents the set of indexes `range(start, end, step)`:

```
>>> sl = slice(0,5,2)
>>> a[sl]
array([0, 2, 4])
```

NumPy: Array Shape Manipulation

- flattening: `ndarray.ravel()`

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.ravel()
array([1, 2, 3, 4, 5, 6])
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> a.T.ravel()
array([1, 4, 2, 5, 3, 6])
```

NumPy: Array Shape Manipulation

- reshaping: `ndarray.reshape(shape)`

```
>>> b = a.ravel()
>>> b.reshape((2, 3))
array([[1, 2, 3],
       [4, 5, 6]])
>>> b.reshape((3, 2))
array([[1, 2],
       [3, 4],
       [5, 6]]
>>> np.arange(18).reshape((3,6))
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17]])
```

NumPy: Element-wise Operations

- with scalars:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
>>> a * 2
array([2, 4, 6, 8])
>>> 2**a
array([ 2,  4,  8, 16])
```

- arithmetic:

```
>>> b = np.ones(4) + 1
>>> b
array([ 2.,  2.,  2.,  2.])
>>> a - b
array([-1.,  0.,  1.,  2.])
>>> a * b
array([ 2.,  4.,  6.,  8.])
>>> b**a
array([  2.,   4.,   8.,  16.])
```

NumPy: Element-wise Operations

- comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False,  True, False,  True], dtype=bool)
>>> a > b
array([False, False,  True, False], dtype=bool)
```

- logical:

```
>>> a = np.array([1, 1, 0, 0], dtype=bool)
>>> b = np.array([1, 0, 1, 0], dtype=bool)
>>> a | b
array([ True,  True,  True, False], dtype=bool)
>>> a & b
array([ True, False, False, False], dtype=bool)
```

NumPy: Basic Linear Algebra

- matrix multiplication:

```
>>> a = np.triu(np.ones((3, 3)), 1)
>>> a
array([[ 0.,  1.,  1.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
>>> b = np.diag([1, 2, 3]);
>>> b
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
>>> a.dot(b)
array([[ 0.,  2.,  3.],
       [ 0.,  0.,  3.],
       [ 0.,  0.,  0.]])
>>> np.dot(a,b)
array([[ 0.,  2.,  3.],
       [ 0.,  0.,  3.],
       [ 0.,  0.,  0.]])
```

NumPy: Basic Linear Algebra

- transpose:

```
>>> a.T
array([[ 0.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  1.,  0.]])
```

- inverses:

```
>>> A = a + b
>>> A
array([[ 1.,  1.,  1.],
       [ 0.,  2.,  1.],
       [ 0.,  0.,  3.]])
>>> B = np.linalg.inv(A)
>>> B.dot(A)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

NumPy: Basic Linear Algebra

- trace:

```
>>> np.trace(A)
6.0
```

- determinant:

```
>>> np.linalg.det(A)
6.0
```

- eigenvalues:

```
>>> np.linalg.eigvals(A)
array([ 1.,  2.,  3.])
```

- linear equation systems: `solve(A,b)` solves the equation $A \cdot \vec{x} = \vec{b}$ for $\vec{x}$

```
>>> x = np.linalg.solve(A, [1, 2, 3])
>>> x
array([-0.5,  0.5,  1. ])
>>> A.dot(x)
array([ 1.,  2.,  3.])
```

NumPy: Composite Data Types

- a `dtype` can be composed of other data types:

```
>>> samples = np.zeros(6, dtype=[('sensor_code', 'S4'), \
...                 ('position', float), ('value', float)] )
>>> samples.ndim
1
>>> samples.shape
(6,)
>>> samples.dtype.names
('sensor_code', 'position', 'value')
>>> samples[:] = [('ALFA', 1, 0.35), ('BETA', 1.0, 0.11),\
...                  ('TAU',  1, 0.39), ('ALFA', 1.5, 0.35),\
...                  ('ALFA', 2, 0.11), ('TAU',  1.2, 0.39)]
>>> samples['sensor_code']
array(['ALFA', 'BETA', 'TAU', 'ALFA', 'ALFA', 'TAU'],
      dtype='|S4')
>>> samples['value']
array([ 0.35,  0.11,  0.39,  0.35,  0.11,  0.39])
>>> samples[0]
('ALFA', 1.0, 0.35)
>>> samples[0]['sensor_code']
'ALFA'
```

NumPy: Universal Functions

- `ufunc` performs and element-wise operation on all elements of an array

```
>>> output = elementwise_function(input)
```

→ both `output` and `input` can be a single value or a `ndarray` of arbitrary size and dimension

- most of the functions in `numpy` are universal functions:
  - `add`, `subtract`, `multiply`, `divide`
  - `absolute`, `exp`, `log`, `log2`, `log10`, `power`, `sqrt`, `square`
  - `conj`, `negative`, `sign`
  - `ceil`, `floor`, `rint`, `trunc`
  - `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`

- the author of an `ufunc` only as to supply the element-wise operation, NumPy takes care of the rest

→ the operation needs to be implemented in C of Cython

SciPy

- the `scipy` package contains various toolboxes dedicated to common issues in scientific computing (similar to GSL)
- `scipy` operates on `numpy` arrays ($\rightarrow$ `numpy` is required to run `scipy`)

`fftpack`	fast Fourier transforms
`integrate`	integration routines (quadrature, Romberg, ODEs, ...)
`interpolate`	interpolation
`io`	data input and output (MatLab, ...)
`linalg`	linear algebra routines (SVD, QR, LU, Cholesky, Schur, ...)
`ndimage`	n-dimensional image package (rotation, filtering, ...)
`optimize`	optimisation (minimisation, curve fitting, root finding)
`signal`	signal processing
`sparse`	sparse matrices
`special`	special mathematical functions (Bessel, Gamma, Erf, ...)
`stats`	statistics, random processes
...	

Matplotlib

- provides a very quick way to visualise data
- allows generation of plots, histograms, power spectra, bar charts, error charts, scatte rplots, etc., with just a few lines of code
- full control of line styles, font properties, axes properties, etc., via an object oriented interface or via a set of functions similar to MATLAB
- works natively and transparently with numpy arrays
- the `pyplot` module provides high level plotting routines for scripting

```
>>> from matplotlib import pyplot
```

- the `pylab` module provides Matlab-like commands for convenient interactive usage

```
>>> from matplotlib import pylab
```

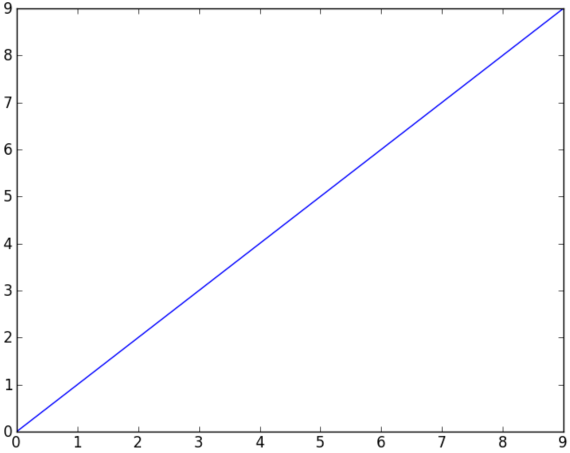
- for interactive plotting use `ipython` (enhanced, interactive Python shell)

```
> ipython -pylab
```

Matplotlib: Simple Plots

- plot the numbers from 0 to 9:

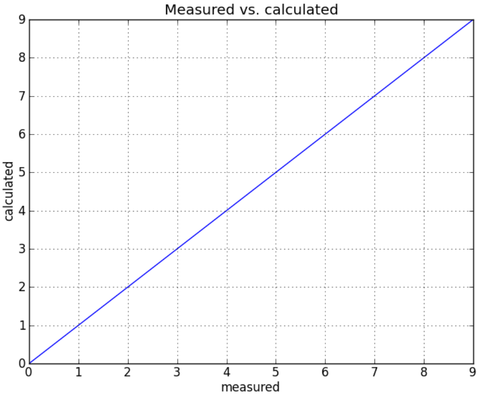
```
In [1]: plot(range(10))
```



Matplotlib: Simple Plots

- interactively add features to the plot:

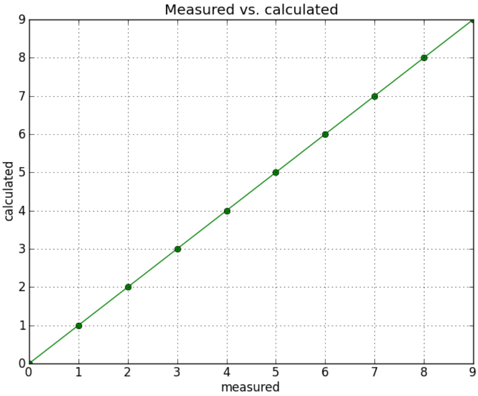
```
In [2]: xlabel('measured')
In [3]: ylabel('calculated')
In [4]: title('Measured vs. calculated')
In [5]: grid(True)
```



Matplotlib: Simple Plots

- with a reference to the plot an the line, we can set properties:

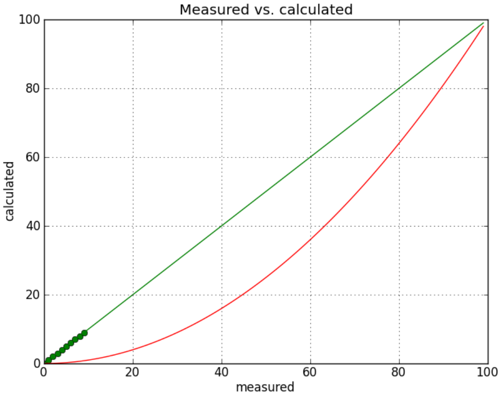
```
In [6]: my_plot = gca()
In [7]: line = my_plot.lines[0]
In [8]: line.set_marker('o')
In [9]: setp(line, color='g')
```



Matplotlib: Simple Plots

- add new lines to the plot:

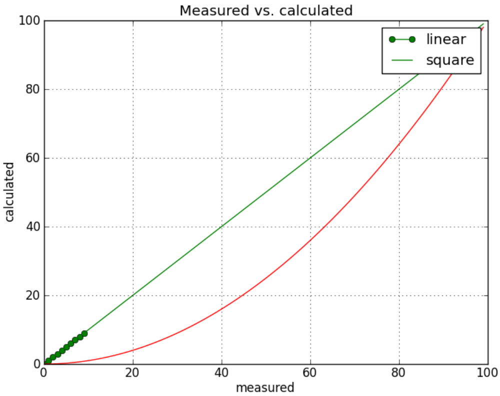
```
In [11]: x = arange(100)
In [12]: linear = arange(100)
In [13]: square = [v * v for v in arange(0, 10, 0.1)]
In [14]: lines = plot(x, linear, x, square)
```



Matplotlib: Simple Plots

- add a legend:

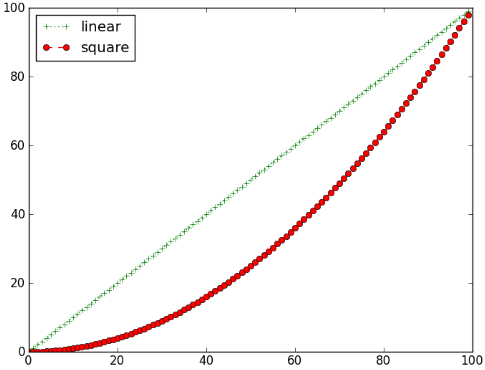
```
In [15]: legend(('linear', 'square'))
```



Matplotlib: Simple Plots

- that doesn't look nice → start over:

```
In [16]: clf()
In [17]: lines = plot(x, linear, 'g:+', x, square, 'r--o')
In [18]: l = legend(('linear', 'square'), loc='upper left')
```



Matplotlib: Properties

- properties of lines can be set via
  - keyword arguments at creation time: `plot(x, linear, 'g:+', x, square, 'r--o')`
  - the function `setp`: `setp(line, color='g')`
  - the `set_something` methods: `line.set_marker('o')`
- lines have several properties:

`alpha`	alpha transparency on $0-1$ scale
`antialiased`	`True` or `False` - use antialised rendering
`color`	matplotlib color arg
`label`	string optionally used for legend
`linestyle`	one of `-` `--` `:` `-.`
`linewidth`	`float`, the line width in points
`marker`	one of `+` `,` `o` `.` `s` `v` `x` `>` `<`, etc.
`markeredgewidth`	line width around the marker symbol
`markeredgecolor`	edge color if a marker is used
`markerfacecolor`	face color if a marker is used
`markersize`	size of the marker in points

Matplotlib: Properties

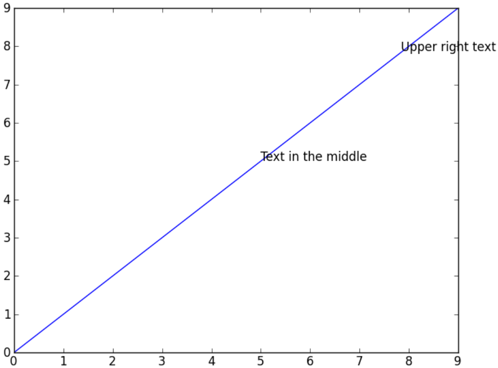
- colours can be given as
  - one-letter abbreviations
  - gray scale intensity from 0 to 1
  - RGB in hex and tuple format
  - any legal html color name
- the one-letter abbreviations are very handy for quick work:

Abbreviation	Color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Matplotlib: Text

- add text at a defined position:
  - → `text` adds the text with data coordinates
  - → `fixtext` adds the text with figure coordinates from 0 to 1

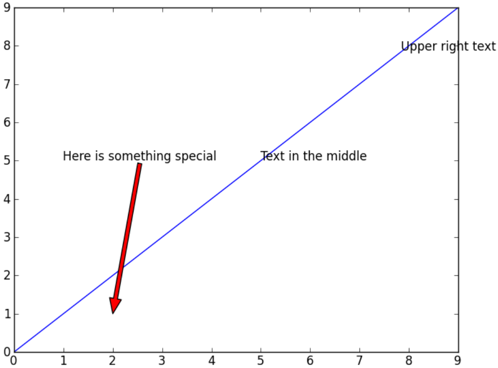
```
In [1]: plot(arange(10))
In [2]: t1 = text(5, 5, 'Text in the middle')
In [3]: t2 = figtext(0.8, 0.8, 'Upper right text')
```



Matplotlib: Text

- use arrows to highlight special details:

```
In [4]: ax = gca()
In [5]: ax.annotate('Here is something special', \
   ...:                 xy = (2, 1), xytext=(1,5),  \
   ...:                 arrowprops={'facecolor':'r'})
In [6]: draw()
```



Matplotlib: Text Properties

`alpha`	alpha transparency on $0-1$ scale
`color`	matplotlib color arg
`family`	set the font family, e.g. `sans-serif`, `cursive`, `fantasy`
`fontangle`	the font slant, one of `normal`, `italic`, `oblique`
`horizontalalignment`	`left`, `right` or `center`
`multialignment`	`left`, `right` or `center` only for multiline strings
`name`	font name, e.g. `Sans`, `Courier`, `Helvetica`
`position`	x,y location
`variant`	font variant, e.g. `normal`, `small-caps`
`rotation`	angle in degrees for rotated text
`size`	fontsize in points, e.g. 8, 10, 12
`style`	font style, one of `normal`, `italic`, `oblique`
`text`	set the text string itself
`verticalalignment`	`top`, `bottom` or `center`
`weight`	font weight, e.g. `normal`, `bold`, `heavy`, `light`

Matplotlib: Figures, Subplots, Axes

- a `figure` is the windows in the GUI that contains your plots

→ there are several parameters that determine how the `figure` looks like:

Argument	Default	Description
`num`	`1`	number of figure
`figsize`	`figure.figsize`	figure size in in inches (width, height)
`dpi`	`figure.dpi`	resolution in dots per inch
`facecolor`	`figure.facecolor`	colour of the drawing background
`edgecolor`	`figure.edgecolor`	colour of edge around the drawing bg
`frameon`	`True`	draw figure frame or not

→ the defaults can be specified in a config file

→ if you have several figures, you have to select the right one before calling plot commands with `figure(num)`

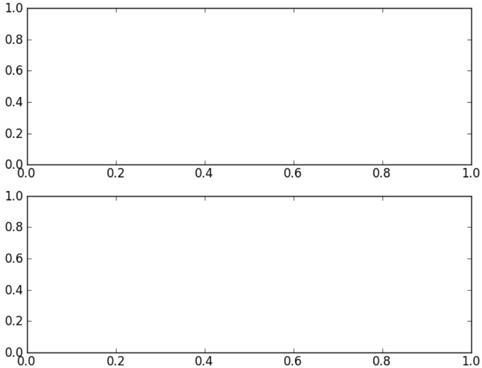
→ you can close a `figure` by calling `close(fig_num)`

Matplotlib: Figures, Subplots, Axes

- arrange plots in a regular grid with `subplot(nrows, ncols, nplot)`

→ a plot with two rows and one column is created with

```
In [1]: subplot(211)
In [2]: subplot(212)
```

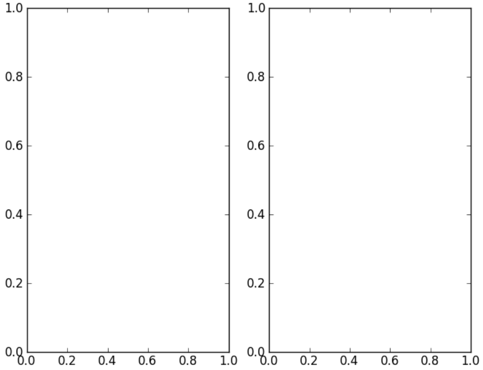


Matplotlib: Figures, Subplots, Axes

- arrange plots in a regular grid with `subplot(nrows, ncols, nplot)`

→ a plot with one row and two columns is created with

```
In [1]: subplot(121)
In [2]: subplot(122)
```

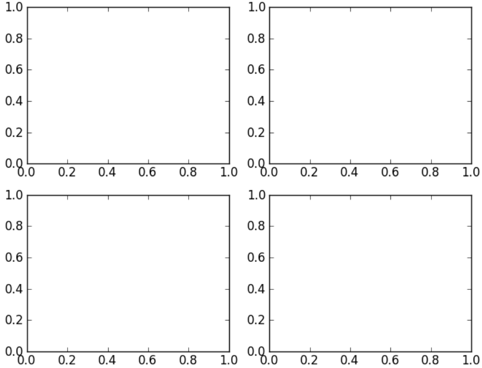


Matplotlib: Figures, Subplots, Axes

- arrange plots in a regular grid with `subplot(nrows, ncols, nplot)`

→ a two-by-two arrangement is created with

```
In [1]: subplot(221)
In [2]: subplot(222)
In [2]: subplot(223)
In [2]: subplot(224)
```

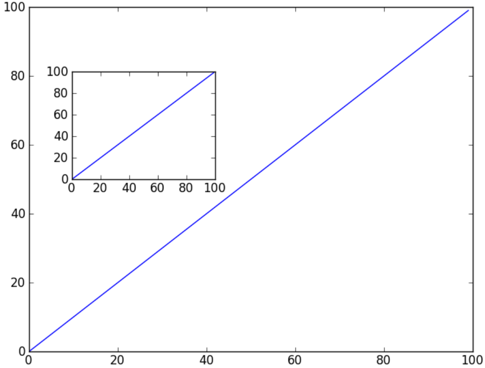


Matplotlib: Figures, Subplots, Axes

- axes allow placement of plots an any location in the figure

→ you can put a smaller plot inside a bigger one:

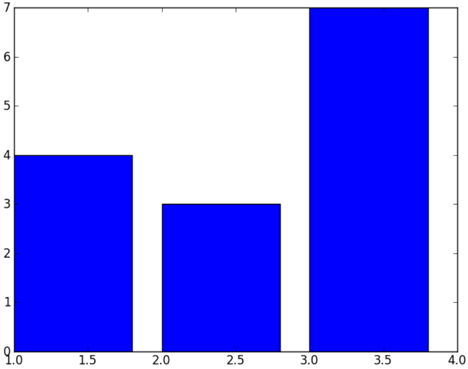
```
In [1]: x = range(100)
In [2]: plot(x)
In [3]: a = axes([0.2, 0.5, 0.25, 0.25])
In [4]: plot(x)
```



Matplotlib: Plot Types

- bar charts:

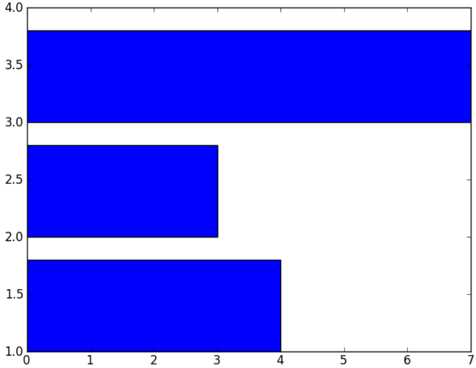
```
In [1]: bar([1, 2, 3], [4, 3, 7])
```



Matplotlib: Plot Types

- horizontal bar charts:

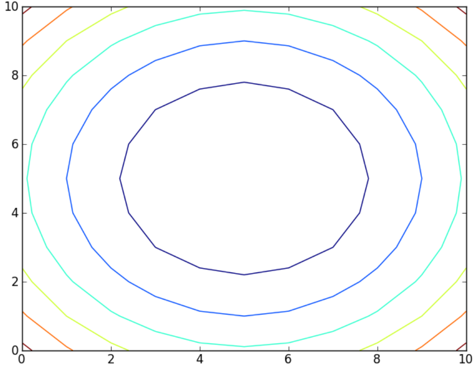
```
In [1]: barh([1, 2, 3], [4, 3, 7])
```



Matplotlib: Plot Types

- contour plots:

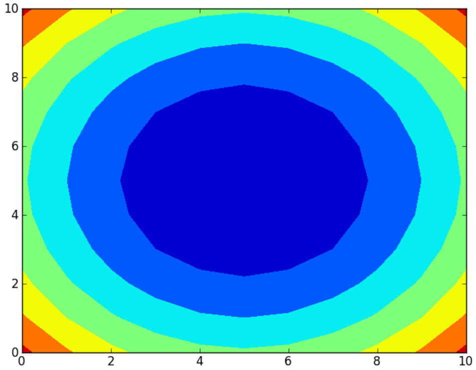
```
In [1]: x, y = np.mgrid[-5:5:11j, -5:5:11j]
In [2]: z = x**2 + y**2
In [3]: contour(z)
```



Matplotlib: Plot Types

- contour plots:

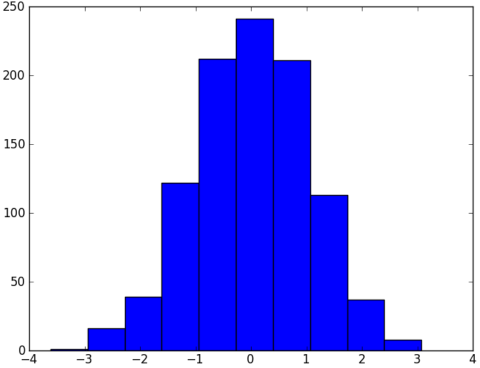
```
In [1]: x, y = np.mgrid[-5:5:11j, -5:5:11j]
In [2]: z = x**2 + y**2
In [3]: contourf(z)
```



Matplotlib: Plot Types

- histograms:

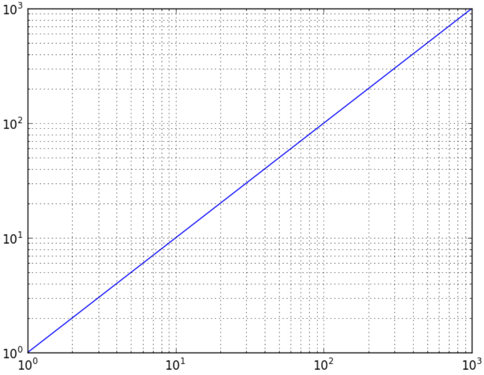
```
In [1]: import numpy as np
In [2]: rand_numbers = np.random.normal(size=1000)
In [3]: hist(rand_numbers)
```



Matplotlib: Plot Types

- loglog plots:

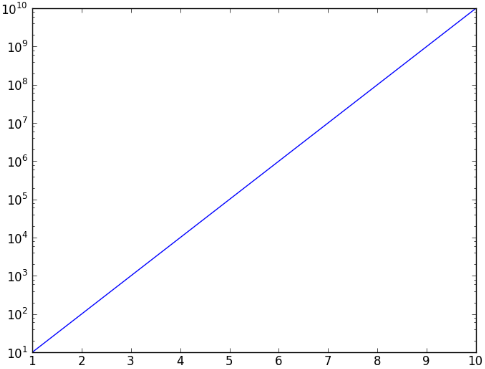
```
In [1]: loglog(arange(1000))
In [2]: grid(True)
In [3]: grid(True, which='minor')
```



Matplotlib: Plot Types

- semilog plots:

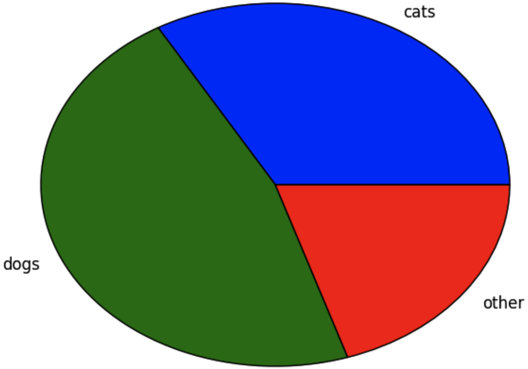
```
In [1]: x = np.linspace(1,10)
In [2]: y = np.logspace(1,10)
In [3]: semilogy(x,y)
```



Matplotlib: Plot Types

- pie charts:

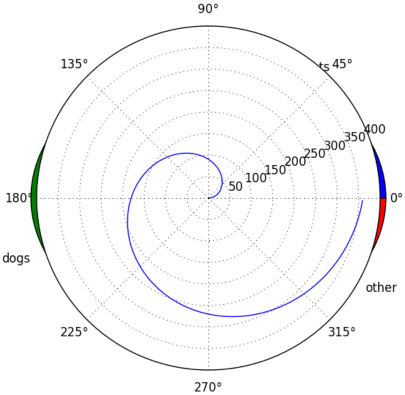
```
In [1]: data = [500, 700, 300]
In [2]: labels = ['cats', 'dogs', 'other']
In [3]: pie(data, labels=labels)
```



Matplotlib: Plot Types

- polar plots:

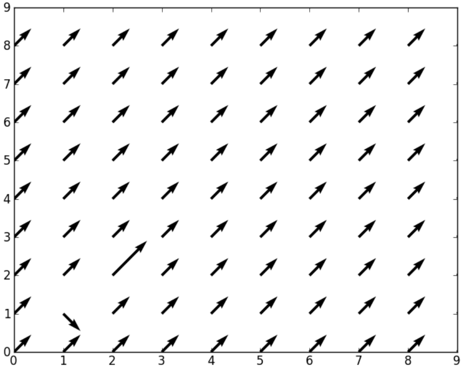
```
In [1]: r = arange(360)
In [2]: theta = r / (180/pi)
In [3]: polar(theta, r)
```



Matplotlib: Plot Types

- arrow plots:

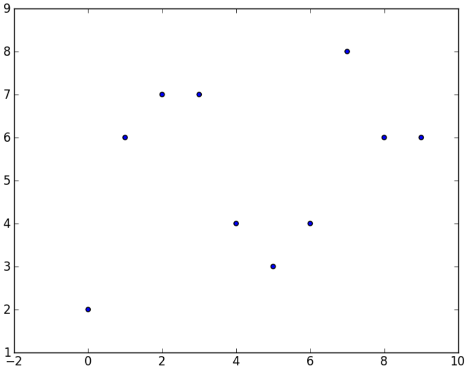
```
In [1]: x = y = arange(10)
In [2]: u = v = ones((10, 10))
In [3]: u[2, 2] = 2; v[2, 2] = 2
In [4]: v[1, 1] = -1
In [5]: quiver(x, y, u, v)
```



Matplotlib: Plot Types

- scatter plots:

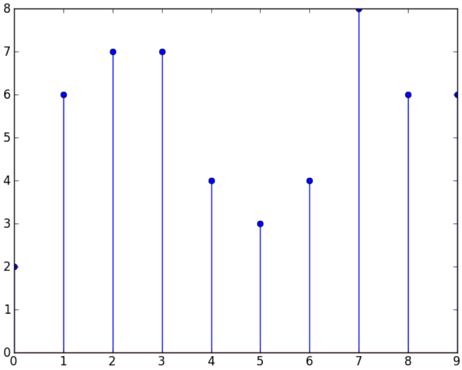
```
In [1]: import numpy as np
In [2]: x = arange(10)
In [3]: y = np.random.randint(0,10,10)
In [4]: scatter(x,y)
```



Matplotlib: Plot Types

- stem plots:

```
In [1]: import numpy as np
In [2]: x = arange(10)
In [3]: y = np.random.randint(0,10,10)
In [4]: stem(x,y)
```



MayaVi

- general purpose tool for 3D scientific data visualisation
- visualisation of scalar, vector and tensor data in 2D and 3D
- interactive use and easy scriptability
- works natively and transparently with NumPy arrays (just as matplotlib)
- don't forget to import the `mayavi` module

```
>>> import enthought.mayavi
```

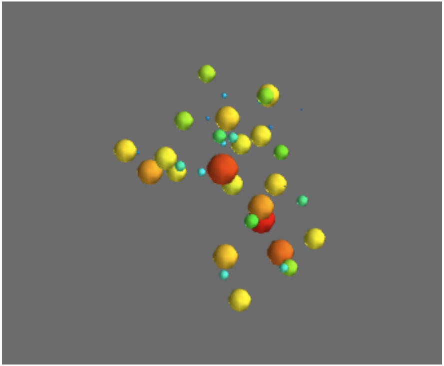
→ the `mlab` interface is very similar to matplotlib's `pylab` interface

```
>>> from enthought.mayavi import mlab
```

MayaVi: 3D Plotting Functions

- plot points at positions `x,y,z` with size and colour depending on `value` (which can be a `ndarray` like `(x,y,z)` or a function of `(x,y,z)`)

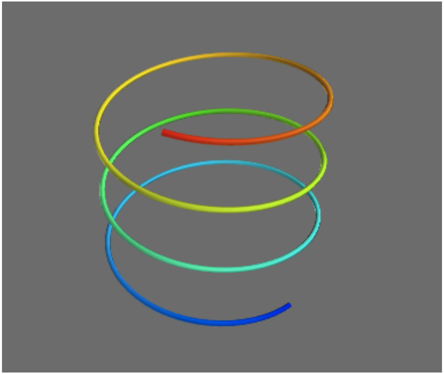
```
In [1]: import numpy as np
In [2]: from enthought.mayavi import mlab
In [3]: x, y, z, value = np.random.random((4, 40))
In [4]: mlab.points3d(x, y, z, value)
```



MayaVi: 3D Plotting Functions

- plot lines between the points described by `x,y,z` which can be 1D NumPy arrays of the same length or a set of parametric functions

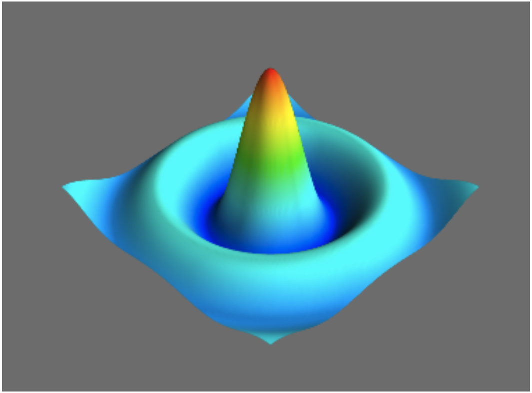
```
In [5]: mlab.clf()
In [6]: t = np.linspace(0, 20, 200)
In [7]: mlab.plot3d(np.sin(t), np.cos(t), 0.1*t, t)
```



MayaVi: 3D Plotting Functions

- plot a surface using regularly-spaced elevation data from a 2D array

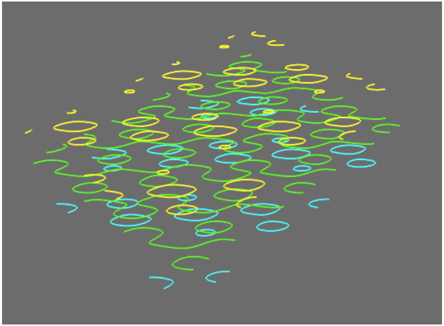
```
In [8]: mlab.clf()
In [9]: x, y = np.mgrid[-10:10:100j, -10:10:100j]
In [10]: r = np.sqrt(x**2 + y**2)
In [11]: z = np.sin(r)/r
In [12]: mlab.surf(z, warp_scale='auto')
```



MayaVi: 3D Plotting Functions

- plot the contours of a surface using grid-spaced elevation data from a 2D array (which in this case is supplied as a function)

```
In [13]: mlab.clf()
In [14]: def f(x, y):
    ...:     sin, cos = np.sin, np.cos
    ...:     return sin(x+y) + sin(2*x - y) + cos(3*x+4*y)
In [15]: x, y = np.mgrid[-7.:7.05:0.1, -5.:5.05:0.05]
In [16]: mlab.contour_surf(x, y, f)
```

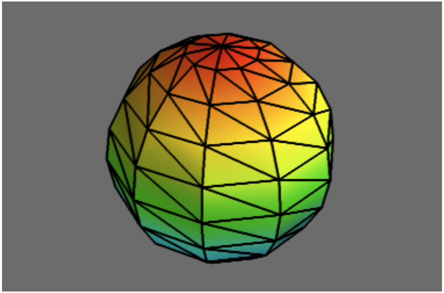


MayaVi: 3D Plotting Functions

- plot a surface described by three 2D arrays `x,y,z` giving the coordinates of the data points as a grid

→ unlike `surf()`, the surface is defined by its `x,y,z` coordinates

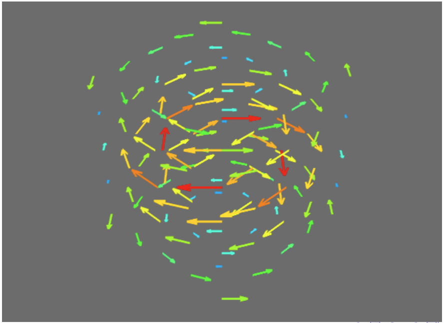
```
In [13]: mlab.clf()
In [14]: phi, theta = np.mgrid[0:np.pi:11j, 0:2*np.pi:11j]
In [15]: x = np.sin(phi) * np.cos(theta)
In [16]: y = np.sin(phi) * np.sin(theta)
In [17]: z = np.cos(phi)
In [18]: mlab.mesh(x, y, z, representation='wireframe',\
   ....:               color=(0, 0, 0))
```



MayaVi: 3D Plotting Functions

- plot arrows to represent vectors with components `u,v,w` at points `x,y,z`

```
In [19]: mlab.clf()
In [20]: x, y, z = np.mgrid[-2:3, -2:3, -2:3]
In [21]: r = np.sqrt(x**2 + y**2 + z**4)
In [22]: u =  y*np.sin(r)/(r+0.001)
In [23]: v = -x*np.sin(r)/(r+0.001)
In [24]: w = np.zeros_like(z)
In [25]: mlab.quiver3d(x, y, z, u, v, w, line_width=3,\
    ....:                         scale_factor=1)
```



HDF5 and netCDF

- HDF5: data model, library, and file format for storing numerical data
  - supports an unlimited variety of datatypes (whatever `dtype` you create)
  - designed for flexible and efficient I/O, high volume and complex data
  - no limit on the number or size of data objects in one file
  - supported by a lot of data analysis, math and visualisation tools (Mathematica, MatLab, Octave, VisIt, ParaView, IDL, HDFView, ...)
- netCDF: self-describing data format for array-oriented scientific data
  - from version 4, netCDF is implemented on top of HDF5
  - netCDF files include information about the data they contain (description, units, ...)

  → netCDF files are self-describing (main difference/advantage over HDF5)
- common features:
  - portable/machine-independent: data looks the same on computers with different ways of storing integers, characters, and floating-point numbers
  - scalable: a small subset of a large dataset may be accessed efficiently
  - libraries for C/C++, Fortran, Java, ..., and Python!

HDF5: h5py

- two kinds of objects are stored in HDF5 files:
  - datasets: homogeneous, regular arrays of data (just like NumPy arrays)
    - → scalar variables are 0D arrays
  - groups: containers that store datasets and other groups
- `h5py` is a simple Python interface to HDF5

→ interact with files, groups and datasets using Python and NumPy metaphors

- → groups behave like dictionaries
- → datasets have shape and dtype attributes
- → they can be sliced and indexed just like NumPy arrays

→ you don't need to know anything about the HDF5 library to use `h5py`, apart from the basic metaphors of files, groups and datasets

netCDF: netCDF4

- an additional kind of object is available in netCDF files:
  - attributes: used to store data about the data (metadata)

→ besides, netCDF and HDF5 file access works very similar

- `netCDF4` is the Python interface to netCDF
- `netCDF4.Dataset` is the main module that gives access to netCDF files

→ create and open a netCDF file: `Dataset(filename, mode[, format])`

→ close a netCDF file: `Dataset.close()`

```
>>> from netCDF4 import Dataset
>>> rootgrp = Dataset('test.nc', 'w', format='NETCDF4')
>>> rootgrp.close()
>>> rootgrp = Dataset('test.nc', 'a')
```

→ the file modes follow standard Python syntax (`w`: create/overwrite, `a`: append, `r`: read, ...)

→ with the `netCDF4.Dataset` module, you may read and write any type of data including dimensions, groups, variables and attributes

netCDF: Groups

- data can be organised in hierarchical groups, which are analogous to directories in a filesystem

→ groups serve as containers for variables, dimensions and attributes, as well as other groups

→ `netCDF4.Dataset` defines a special group, called the *root group*, which is similar to the root directory in a unix filesystem

→ create Group instances: `Dataset.createGroup(name)`

```
>>> fcstgrp = rootgrp.createGroup('forecasts')
>>> analgrp = rootgrp.createGroup('analyses')
```

→ all of the `Group` instances are stored in the dictionary `Dataset.groups`:

```
>>> print(rootgrp.groups)
OrderedDict([('forecasts', <netCDF4.Group object at 0x1b4b7b0
             ('analyses', <netCDF4.Group object at 0x1b4b970>
>>> for grpname, group in rootgrp.groups.iteritems():
...     print(grpname)
...     # do something to the group object
...
```

netCDF: Dimensions

- netCDF defines the sizes of all variables in terms of dimensions

→ before a variable can be created the dimensions it uses must be created

→ special case: scalar variables (have no dimensions)

→ create a dimension (`size=None` means unlimited size, growing):
`Dataset.createDimension(name, size)`,
`Group.createDimension(name, size)`

```
>>> level = rootgrp.createDimension('level', None)
>>> time  = rootgrp.createDimension('time',  None)
>>> lat = rootgrp.createDimension('lat', 73)
>>> lon = rootgrp.createDimension('lon', 144)
```

→ all of the `Dimension` instances are stored in the dictionary
`rootgrp.dimensions`

→ `len` returns the current size of a given dimension:

```
>>> print(len(lon))
144
```

netCDF: Variables

- netCDF variables behave much like `ndarray` objects

→ create a netCDF variable:

`Dataset.createVariable(name, dtype, dim),`

`Group.createVariable(name, dtype, dim)`

→ dimensions themselves are also defined as variables

```
>>> times   = rootgrp.createVariable('time', 'f8',('time',) )
>>> levels = rootgrp.createVariable('level','i4',('level',))
>>> latitudes   = rootgrp.createVariable('latitude', \
...                                 np.float32,('lat',))
>>> longitudes  = rootgrp.createVariable('longitude',\
...                                 np.float32,('lon',))
>>> temperature = rootgrp.createVariable('temp', \
...             np.float32,('time','level','lat','lon',))
```

→ the variables in a `Group` are also stored in a dictionary:

```
>>> longitudes2 = rootgrp.variables['longitudes']
```

netCDF: Variables

- write data to netCDF variables by assigning NumPy arrays:

```
>>> lats = np.arange(-90,91,2.5)
>>> lons = np.arange(-180,180,2.5)
>>> latitudes[:]  = lats
>>> longitudes[:] = lons
```

- retrieve data by accessing netCDF variables like NumPy arrays:

```
>>> print(latitudes[:])
[-90.   -87.5 -85.   -82.5 -80.   -77.5 -75.   -72.5 ...
 -60.   -57.5 -55.   -52.5 -50.   -47.5 -45.   -42.5 ...
 -30.   -27.5 -25.   -22.5 -20.   -17.5 -15.   -12.5 ...
 ...
>>> print(latitudes[::2])
[-90. -85. -80. -75. -70. -65. -60. -55. -50. ...
 -15. -10.  -5.   0.   5.  10.  15.  20.  25. ...
  60.  65.  70.  75.  80.  85.  90.]
```

netCDF: Attributes

- two types of attributes: global and variables

→ global attributes provide information about the dataset or a group

→ set by assigning values to `Dataset` or `Group` instance variables:

```
>>> import time
>>> rootgrp.description = 'bogus example script'
>>> rootgrp.history = 'Created ' + time.ctime(time.time())
>>> rootgrp.source  = 'netCDF4 python module tutorial'
```

→ variable attributes provide information about a single variable

→ set by assigning values to `Variable` instance variables:

```
>>> latitudes.units  = 'degrees north'
>>> longitudes.units = 'degrees east'
>>> levels.units     = 'hPa'
>>> temp.units       = 'K'
>>> times.units      = 'hours since 0001-01-01 00:00:00.0'
>>> times.calendar   = 'gregorian'
```

netCDF: Attributes

- retrieve the names of all the netCDF attributes:
  `Dataset.ncattrs()`, `Group.ncattrs()`, `Variable.ncattrs()`

```
>>> for name in rootgrp.ncattrs():
...     print(name)
description
history
source
```

- retrieve the value of attributes: `getattr(group, name)`

```
>>> for name in rootgrp.ncattrs():
>>>     print('Global attr' + name + '=' \
...                             + getattr(rootgrp,name))
Global attr description = bogus example script
Global attr history = Created Mon Nov  7 10.30:56 2005
Global attr source = netCDF4 python module tutorial
```

Outlook: Advanced Topics

- calling and embedding C and Fortran code:
  - Cython: extension of the Python language providing static typed functions and variables, generating efficient C code for fast computations
  - Weave: embed C code within your `.py` files
  - ctypes: wrap C code
  - f2py: wraps and compiles Fortran routines to behave like Python modules
- parallelisation:
  - threading
  - multiprocessing
  - parallel python (PP)
  - Python Remote Objects (PYRO)
  - pyMPI
- GUI programming:
  - PyQt
  - Traits
- Symbolic Calculations with SymPy and Sage