Welcome!

Virtual tutorial starts at 15:00 BST

Python for HPC

ARCHER Virtual Tutorial, Wed 13th August 2014

Rupert Nash <rupert.nash@ed.ac.uk>

Python for HPC

- Scientific Python – slide 4
- Introduction to Python – slide 12
- NumPy – slide 40
- MatPlotLib – slide 59
- SciPy – slide 79
- Interfacing to Fortran / C : F2Py – slide 88

Scientific Python

Andy Turner, ARCHER CSE Team
a.turner@epcc.ed.ac.uk

Reusing this material

http://www.archer.ac.uk
support@archer.ac.uk

Scientific Computing Requirements

- Generate data
  - Usually from simulation on HPC facilities
  - (Also from experiment!)
- Process data
  - Generate appropriate results from simulation data
- Visualise results
  - To understand the significance of our work and gain scientific understanding
- Communicate results
  - Through publications, presentations, web, etc.

Why Python?

- Rich set of scientific computing functionality
  - Powerful numerical and scientific libraries
  - Rich plotting functionality
  - Excellent support for interfacing to existing Fortran/C/C++ code
  - Interactive and scripting interface
- Simple to learn and code is very readable
  - Scientists are usually self-taught programmers
  - Syntax enables clarity in algorithms (in a similar way to Fortran)
- Free and Open Source
  - Widely-available so code is portable

Useful packages

- IPython
  - Advanced Python shell
- Matplotlib
  - Rich featured plotting (2D and 3D)
- Numpy
  - Tools for manipulating numerical arrays efficiently
- Scipy
  - High-level scientific routines for common algorithms: optimisation, Fourier transform, linear algebra and others
- f2py
  - Interface external code with Python
- mpi4py
  - Message passing parallel programming

Python: Interactive and Programs

- Python can be used as an interactive tool
  - For example, when producing simple plots to quickly analyse data
  - The IPython shell adds additional useful functionality
- It can also be used for writing programs
  - These can range from quick-and-dirty single use scripts to full programs
  - Can interface to C/C++ and Fortran code

IPython Shell

- IPython extends the standard Python shell with a number of useful things, including:
  - Tab completion
  - Interactive help
  - Built-in debugging and profiling
  - Pasting of code snippets from websites
  - Saving of sessions
- `quickref` command gives a summary of capabilities

Introduction to Python

Arno Proeme, ARCHER CSE Team
aproeme@epcc.ed.ac.uk
Attributed in part to Jussi Enkovaara &
Martti Louhivuori, CSC Helsinki

Reusing this material

http://www.archer.ac.uk
support@archer.ac.uk

Python origins

- Created early 1990s (Guido van Rossum, CWI)
- Driven by desire to provide more programmer-friendly alternative to C to speed up application development
- Inspired by an earlier interactive programming environment and language (ABC)
- Not created specifically for scientific computing (unlike e.g. Fortran)

Python now

- Most popular first taught programming language at top 39 US computer science departments
- Used by Youtube, Dropbox, Google, Industrial Light & Magic, Quant Finance, …
- Version 3.x breaks backwards compatibility with 2.x
  - 2.x still most widely used, including in this course

In natural sciences & engineering?

- Used mainly:
  - As a multipurpose workflow environment for data analysis and visualisation
  - As "glue", i.e. interface code, to heavy numerical kernels written in a compiled language like C/C++ or Fortran (e.g. Fluidity, ASE)
  - For rapid prototyping of algorithms
  - For non-HPC simulations

- Though performance continues to improve and there are some 100% Python codes (e.g. GPAW), these are still not widely used for heavy numerics.

Python characteristics

- Python is a **high-level** language (compared e.g. to C),
  - Simple syntax, more easily readable code and shorter programs

  but
  - Sacrifice some performance due to abstraction overheads
  - Development time considered more valuable than compute time

- Python is a fully-featured general purpose programming language (like C, C++, Fortran, Java, etc.)

- Python supports (but does not enforce) different programming styles, e.g. object-oriented

- Python is open source

The Python interpreter

- Python code is not generally compiled into a standalone executable, but executed by the Python interpreter, `python`

- Python code contained in a script file (ending in .py) can be execute by the interpreter as follows:

```
aproeme$ cat hello.py
print("Hello World")
aproeme$ python hello.py
Hello World
```

Interactive Python

- If not supplied with an input script file, the Python interpreter runs as an interactive Python runtime environment (a Python shell session)

```
aproeme$ python
```

Interactive Python

- If not supplied with an input script file, the Python interpreter runs as an interactive Python runtime environment (a Python shell session)

```
aproeme$ python
Python 2.7.7 |Anaconda 2.0.1 (x86_64)| (default, Jun  2 2014,
12:48:16)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
Anaconda is brought to you by Continuum Analytics.
>>>
```

Interactive Python

- If not supplied with an input script file, the Python interpreter runs as an interactive Python runtime environment (a Python shell session)

```
aproeme$ python
Python 2.7.7 |Anaconda 2.0.1 (x86_64)| (default, Jun  2 2014,
12:48:16)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
Anaconda is brought to you by Continuum Analytics.
>>> print(“Hello World”)
```

Interactive Python

- If not supplied with an input script file, the Python interpreter runs as an interactive Python runtime environment (a Python shell session)

```
aproeme$ python
Python 2.7.7 |Anaconda 2.0.1 (x86_64)| (default, Jun  2 2014,
12:48:16)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
Anaconda is brought to you by Continuum Analytics.
>>> print(“Hello World”)
Hello World
```

Interactive Python

- Python shell lets you explore Python functionality directly without needing to compile your code
- This is useful for incremental / progressive code development and rapid prototyping
- In case of any errors, debugging (TraceBack) information is provided within the Python shell (which usually does not simply crash)
- Once you have worked out how to get Python to do what you want it to, save the code as a Python script (.py file)

Interactive Python vs Matlab *et al*

- The experience of using interactive Python to work, especially iPython, is similar to using other scripting languages e.g. Matlab, Mathematica, Maple, R, etc.
- As well as having a good range of scientific libraries Python is more easily extendable
- As popularity grows more and more packages become available, Python becomes the preferred workflow shell to tie everything together

Data types

- Variables in Python are dynamically typed
  - i.e. don't specify explicitly whether int, string, etc.
  - Type is determined based on format of assigned value or other variables involved in calculation

```
X = 1.0
my_name = Arno

Y = my_name + X
```

The slides that follow are attributed to:

Jussi Enkovaara & Martti Louhivuori, CSC Helsinki

Numerical data types

- Integers
- Floats
- Complex numbers
- Basic operations
  - + and –
  - *, / and **
  - Implicit type conversions
  - Be careful with integer division!

```
x = 4
y = 6.0
z = 1.4 + 4.2j
```

```
>>> 4.0 + 5 - 2
7.0
>>> 2.0**2 / 2.0*(4.2-2j)
(8.4-4j)
>>> 2/5
0
>>> 2./5
0.4
```

String

- Strings are enclosed by “ or '
- Multiline strings can be defined with three double quotes

s1 = “very simple string”
s2 = 'same simple string'
s3 = “this isn't so simple string”
s4 = 'is this “complex” string?’
s5 = “””This is a long string
expanding to multiple lines,
so it is enclosed by three “'s”””

+ and * operators with strings:

```
>>> "Strings can be " + "combined"
'Strings can be combined'
>>> "Repeat! " * 3
'Repeat! Repeat! Repeat!
```

Data structures

- Lists
- Tuples
- No arrays! (wait for NumPy)

Lists

- Python lists are dynamic arrays
- List items are indexed (index starts from 0)
- List item can be any Python object, items can be of different type
- New items can be added to any place in the list
- Items can be removed from any place in the list

Lists

- Defining lists

```
>>> l1 = [3, “egg”, 6.2, 7]
>>> l2 = [12, [4, 5], 13, 1]
```

- Accessing list elements

```
>>> l1[0]
3
>>> l2[1]
[4, 5]
>>> l1[-1]
7
```

- Modifying list items

```
>>> l[-2] = 4
>>> l1
[3, 'egg', 4, 7]
```

Lists

- Adding items to list

```
>>> l1 = [9, 8, 7, 6]
>>> l1.append(11)
>>> l1
[9, 8, 7, 6, 11]
>>> l1.insert(1,16)
>>> l1
[9, 16, 8, 7, 6, 11]
>>> l2 = [5, 4]
>>> l1.extend(l2)
>>> l1
[9, 16, 8, 7, 6, 11, 5, 4]
```

- + and * operators with lists:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> [1, 2, 3] * 2
[1, 2, 3, 1, 2, 3]
```

Lists

- It is possible to access slices of lists
- `>>> l1 = [0, 1, 2, 3, 4, 5]`
- ```
  >>> l1[0:2]
  [0, 1]
  >>> l1[:2]
  ```
- ```
  [0, 1]
  >>> l1[3:]
  [3, 4, 5]
  >>> l1[0:6:2]
  ```
- ```
  [0, 2, 4]
  >>> l1[::-1]
  [5, 4, 3, 2, 1, 0]
  ```
- Removing list items

```
>>> second = l1.pop(2)
>>> l1
[0, 1, 3, 4, 5]
>>> second
2
```
```

Tuples

- A tuple is number of comma-separated values, e.g.:
- `>>> t = ‘a’,2,3`
- `t[0]= bla`
- `Traceback (most recent call last):`
- `File “<stdin>”, line 1, in <module>`
- `TypeError: ‘tuple’ object does not support item assignment`

Variables

- Python variables are references

```
>>> l1 = [1,2,3,4]
>>> l2 = l1
```

- l1 and l2 are references to the same list
- Modifying l2 changes also l1!
  - `>>> l2[0] = 0`
  - `>>> l1`
    `[0, 2, 3, 4]`
- Copy can be made by slicing the whole list
  - >>> l3 = l1[:]
  - >>> l3[-1] = 66
  - >>> l1
    [0, 2, 3, 4]
  - >>> l3
    [0, 2, 3, 66]

Objects

- Object is a software bundle of data (=variables) and related methods
- Data can be accessed directly or only via the methods (=functions) of the object
- In Python, everything is an object
- Methods of object are called with the syntax
  - obj.method
- Methods can modify the data of object or return new objects

Standard Library

- Standard library includes:
  - OS interface
  - Basic Maths functions & random number generator
  - Performance measurement
  - Output formatting
  - Data compression
  - Internet access
  - Simple multithreading
  - Logging

Misc.

- Third party Python packages (modules) are loaded with
- `import modulename`
- Code blocks are indented
- Documentation:
  - https://docs.python.org/2.7/
  - http://scipy-lectures.github.io/

NumPy

Arno Proeme, ARCHER CSE Team
aproeme@epcc.ed.ac.uk
Attributed to Jussi Enkovaara &
Martti Louhivuori, CSC Helsinki

Reusing this material

http://www.archer.ac.uk
support@archer.ac.uk

NumPy

- Pure Python provides lists, but not arrays
  - Lists are slow for many numerical algorithms

- NumPy package provides:
  - a multidimensional array data type for Python
  - linear algebra operations and random number generators

- All elements of a NumPy array have the same type

Creating NumPy arrays

- From a list

```
>>> import numpy as np
>>> a = np.array((1, 2, 3, 4), float)
>>> a
array([ 1., 2., 3., 4.])
>>> list1 = [[1, 2, 3], [4,5,6]]
>>> mat = np.array(list1, complex)
>>> mat
array([[ 1.+0.j, 2.+0.j, 3.+0.j],
[ 4.+0.j, 5.+0.j, 6.+0.j]])
>>> mat.shape
(2, 3)
>>> mat.size
6
```

Creating NumPy arrays

- Using NumPy functions:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.linspace(-4.5, 4.5, 5)
>>> b
array([-4.5 , -2.25, 0. , 2.25, 4.5 ])
>>> c = np.zeros((4, 6), float)
>>> c.shape
(4, 6)
>>> d = np.ones((2, 4))
>>> d
array([[ 1., 1., 1., 1.],
[ 1., 1., 1., 1.]])
```

Indexing and slicing arrays

- Simple indexing

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat[0,2]
3
>>> mat[1,-2]
>>> 5
```

- Slicing is possible over all dimensions

```
>>> a = np.arange(10)
>>> a[1:7:2]
array([1, 3, 5])
>>> a = np.zeros((4, 4))
>>> a[1:3, 1:3] = 2.0
>>> a
array([[ 0., 0., 0., 0.],
[ 0., 2., 2., 0.],
[ 0., 2., 2., 0.],
[ 0., 0., 0., 0.]])
```

archer | epcc | THE UNIVERSITY OF EDINBURGH

Views and copies of arrays

- Simple assignment creates references to arrays
- Slicing creates “views” to the arrays
- Use `copy()` for real copying of arrays

```
a = np.arange(10)
b = a # reference, changing values in b changes a
b = a.copy() # true copy
c = a[1:4] # view, changing c changes elements [1:4] of a
c = a[1:4].copy() # true copy of subarray
```

Array manipulation

- `reshape` : change the shape of array

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat
array([[1, 2, 3],
[4, 5, 6]])
>>> mat.reshape(3,2)
array([[1, 2], [3, 4], [5, 6]])
```

- `ravel` : flatten array to 1-d

```
>>> mat.ravel()
array([1, 2, 3, 4, 5, 6])
```

Array manipulation

- concatenate : join arrays together

```
>>> mat1 = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat2 = np.array([[7, 8, 9], [10, 11, 12]])
>>> np.concatenate((mat1, mat2))
array([[ 1, 2, 3],
[ 4, 5, 6],
[ 7, 8, 9],
[10, 11, 12]])
>>> np.concatenate((mat1, mat2), axis=1)
array([[ 1, 2, 3, 7, 8, 9],
[ 4, 5, 6, 10, 11, 12]])
```

- `split` : split array to N pieces

```
>>> np.split(mat1, 3, axis=1)
[array([[1], [4]]), array([[2], [5]]), array([[3], [6]])]
```

Array operations

- Most operations for numpy arrays are done element-wise
- – +, -, *, /, **

- `>>> a = np.array([1.0, 2.0, 3.0])`
- `>>> b = 2.0`
- `>>> a * b array([ 2., 4., 6.])`
- `>>> a + b array([ 3., 4., 5.])`
- `>>> a * a array([ 1., 4., 9.])`

Array operations

- Numpy has special functions which can work with array arguments, e.g. `sin, cos, exp, sqrt, log,` ...

- ```
  >>> import numpy, math
  >>> a = numpy.linspace(-pi, pi, 8)
  >>> a
  array([-3.14159265, -2.24399475, -1.34639685, -
  0.44879895,0.44879895, 1.34639685, 2.24399475, 3.14159265])
  ```
- `>>> math.sin(a)`
- `Traceback (most recent call last): File "<stdin>", line 1, in ?`
- `TypeError: only length-1 arrays can be converted to Python scalars`
- ```
  >>> numpy.sin(a)
  array([ -1.22464680e-16, -7.81831482e-01, -9.74927912e-01,
  ```
- ```
  -4.33883739e-01, 4.33883739e-01, 9.74927912e-01, 7.81831482e-01,
  1.22464680e-16])
  ```
  ```

Vectorized operations

- for loops in Python are slow
- Use “vectorized” operations when possible
- Example: difference

```
arr = np.arange(1000)
dif = np.zeros(999, int)
for i in range(1, len(arr)):
        dif[i1] = arr[i]  arr[i1]
```

- VS

```
arr = np.arange(1000)
dif = arr[1:]  arr[:1]
```

- – for loop is ~80 times slower!

I/O with Numpy

- NumPy provides functions for reading data from file and for writing data into the files
- Simple text files
  - numpy.loadtxt
  - numpy.savetxt
  - Data in regular column layout
  - Can deal with comments and different column delimiters

Random numbers

- The module `numpy.random` provides several functions for constructing random arrays
  - `random`: uniform random numbers – normal: normal distribution
  - `poisson`: Poisson distribution
  - etc....

```
>>> import numpy.random as rnd
>>> rnd.random((2,2))
array([[ 0.02909142, 0.90848 ],
[ 0.9471314 , 0.31424393]])
>>> rnd.poisson(size=(2,2))
array([[0, 1],
       [2, 0]])
```

Polynomials

- Polynomial is defined by array of coefficients p p(x, N) = p[0] xN-1 + p[1] xN-2 + ... + p[N-1]
- Least square fitting: `numpy.polyfit`
- Evaluating polynomials: `numpy.polyval`
- Roots of polynomial: `numpy.roots`
- ...

```
>>>  x = np.linspace(-4, 4, 7)
>>>  y = x**2 + rnd.random(x.shape)
>>>  p = np.polyfit(x, y, 2)
>>>  p
array([ 0.96869003, -0.01157275, 0.69352514])
```

Linear algebra

- Numpy can calculate matrix and vector products efficiently
  - `dot, vdot, …`
- Eigenproblems
  - `linalg.eig, linalg.eigvals, …`
- Linear systems and matrix inversion
  - `linalg.solve, linalg.inv`

```
>>> A = np.array(((2, 1), (1, 3)))
>>> B = np.array(((-2, 4.2), (4.2, 6)))
>>> C = np.dot(A, B)
>>> b = np.array((1, 2))
>>> np.linalg.solve(C, b) # solve C x = b
array([ 0.04453441, 0.06882591])
```

NumPy performance

- Matrix multiplication (C=A*B), matrix dimension 200
  - pure python: 5.30s
  - naive C: 0.09s
  - numpy.dot: 0.01s

Summary

- NumPy provides a static array data structure
- Multidimensional arrays
- Fast mathematical operations for arrays
- Arrays can be broadcasted into same shapes
- Tools for linear algebra and random numbers

- To get performance, use high-level syntax!

matplotlib

Andy Turner, ARCHER CSE Team
a.turner@epcc.ed.ac.uk

Reusing this material

http://www.archer.ac.uk
support@archer.ac.uk

Synopsis

- What is matplotlib?
- Basic concepts
  - Figures and subplots
- Simple plots from plain text files
  - Replacing gnuplot in your workflow
- More complex plots
  - Different types of plots
  - Preparation for publication

What is matplotlib?

- matplotlib is a plotting library for Python
- Philosophy is to *"make the easy things easy and the hard things possible".*
- Designed for both:
  - Interactive plotting
  - Production of publication-quality figures
- Large amount of functionality:
  - Scientific and statistical plots
  - Heatmaps and contours
  - Surfaces
  - Geographical and map-based plotting
- Closely integrated with numpy

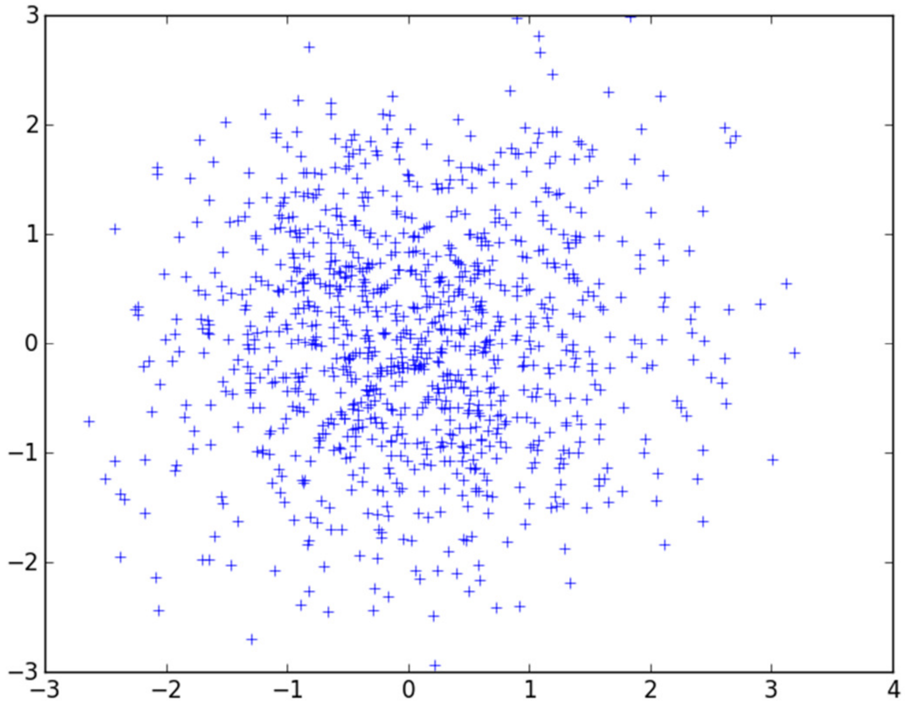
Basic Concepts

- Everything assembled by Python commands
  - Lines, points, axes
  - Titles, legends
  - Multiple plots
- Issue *show* command to display plot
- You use commands to set which *subplot* you are currently working on
- Default is to plot to screen but you can also save to image with single command

Example: random scatterplot

- Assuming we are using ipython –pylab:

```
x = randn(1000)
y = randn(1000)
plot(x, y, '+')
```



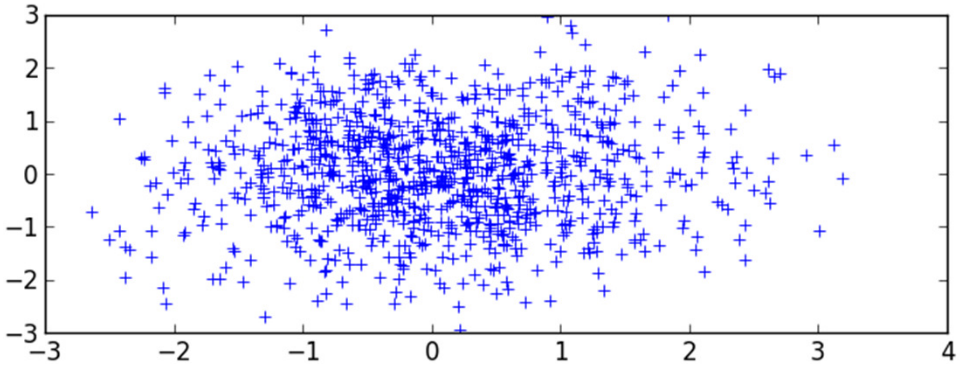
Figures and subplots

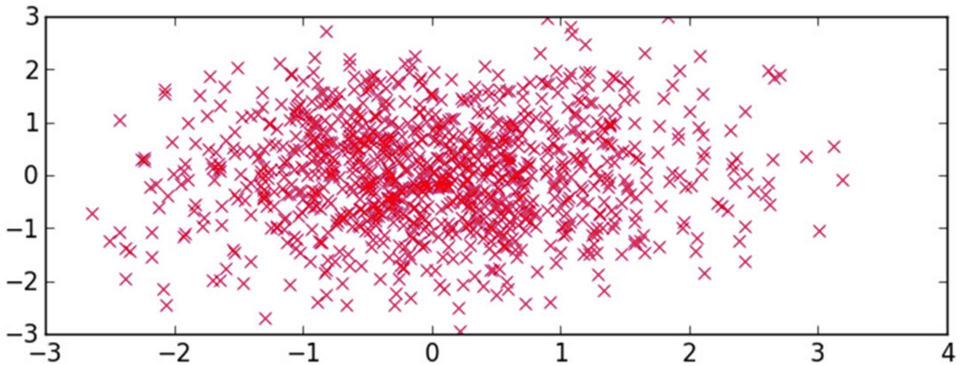
- The whole plotting area is known as the *figure*
- Within the figure there can be *subplots*
  - subplots are placed on a regular grid within the figure
  - (If you need more control over placement you can use *axes*)
  - For simple plots there is usually only one subplot (1, 1, 1)
- You use the `subplot` command to specify which subplot you are currently working on
  - `subplot(nrows, ncols, plot number)`

Example: random scatterplots

- Assuming we are using ipython –pylab:

```
x = randn(1000)
y = randn(1000)
fig = figure()
subplot(2, 1, 1)
plot(x, y, 'b+')
subplot(2, 1, 2)
plot(x, y, 'rx')
fig.show()
```





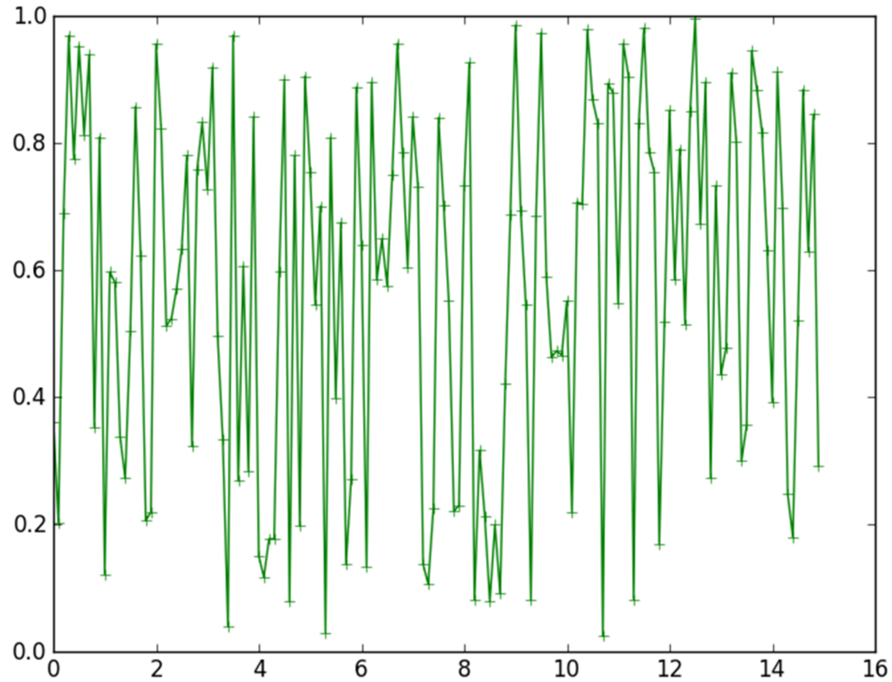
Simple plots from plain text

- People often want to have a quick look at data in a plain text file
  - Gnuplot/Excel often used for this but matplotlib can provide a simple, feature-rich replacement.
  - Manipulate the data interactively and replot
  - Can save the session to keep record of what you did if required
- Use numpy functions for reading data
  - Simple interface to complex reading if required
- As data is in numpy, matplotlib can plot it easily

Example: read and plot x, y data

- Assuming we are using ipython –pylab:

```
data = genfromtxt('random1.dat')
fig = figure()
subplot(1, 1, 1)
plot(data[:,0], data[:,1], 'g+-')
fig.show()
```



Setting axis labels, titles and legends

- Axis labels: use *xlabel* and *ylabel* (they act on the currently selected subplot)*:*
  - `xlabel(“Job Size”)`
- Title: use fig.suptitle:
  - `fig.suptitle(“Job Size Distribution on ARCHER”)`
- Legend: use legend (acts on the currently selected subplot):
  - Requires that label is set for plot:

```
plot(jobs[:,0], jobs[:,1], ‘r–’, label=“2014”)
legend()
```

Save to image file

- Saving to image file is simple using *fig.savefig*
  - File format is determined from the extension
  - e.g. to save to a PNG image:

```
fig.savefig(“archer_jobs.png”)
```

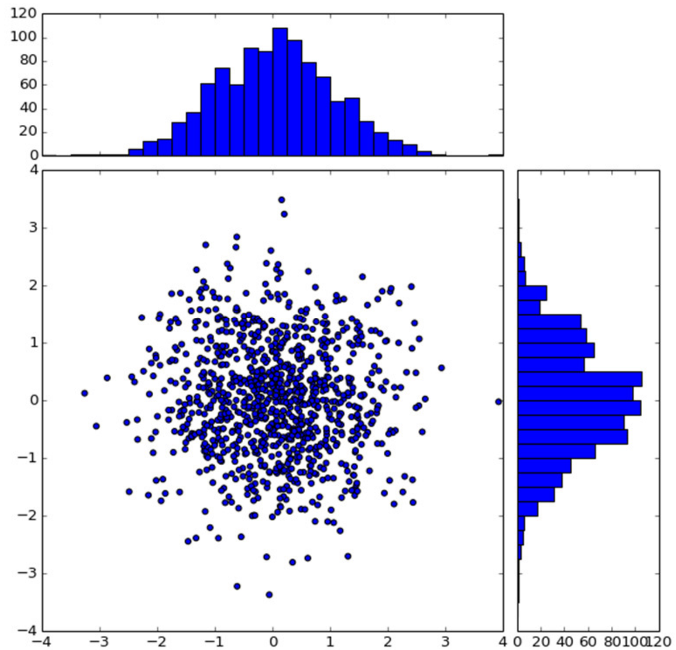
- Resolution set using *dpi* option:
  - e.g:

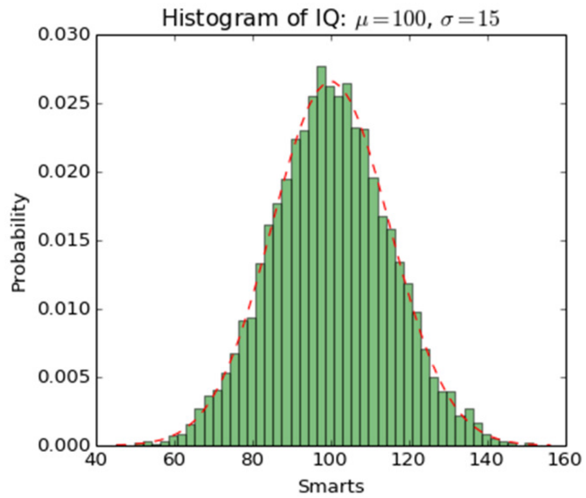
```
fig.savefig(“archer_jobs.png”, dpi=300)
```

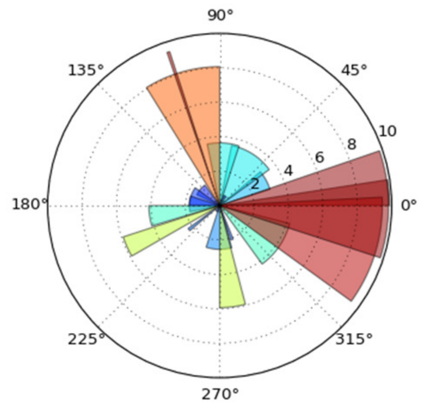
- Commonly supports: png, jpg, pdf, ps

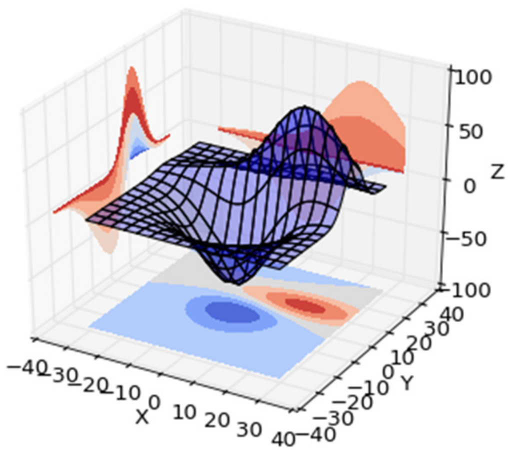
Other types of plots

- http://matplotlib.org/gallery.html









Preparing publication images

- You will probably want different settings for each journal
- matplotlib uses a settings file: *matplotlibrc*, to setup font sizes, types and plotting defaults
  - Useful to keep a different *matplotlibrc* file for each journal
- Import a particular settings file with:

```
from matplotlib import rc_file
rc_file('/path/to/my/matplotlibrc')
```

- From Damon McDougall: http://bit.ly/1jIuuU0

Useful matplotlibrc font settings

```
# Font sizes and types
axes.labelsize  : 9.0  # fontsize of the x any y labels
xtick.labelsize : 9.0  # fontsize of the tick labels
ytick.labelsize : 9.0  # fontsize of the tick labels
legend.fontsize : 9.0  # fontsize in legend
font.family     : serif
font.serif      : Computer Modern Roman

# Marker size
lines.markersize : 3

# Use TeX to format all text
text.usetex : True
```

Setting a nice figure ratio

```
WIDTH = 500.0  # Figure width in pt (usually from LaTeX)
FACTOR = 0.45  # Fraction of the width you'd like the figure to occupy
widthpt  = WIDTH * FACTOR

inperpt = 1.0 / 72.27
golden_ratio  = (np.sqrt(5) - 1.0) / 2.0  # because it looks good

widthin  = widthpt * inperpt
heightin = widthin * golden_ratio
figdims    = [widthin, heightin] # Dimensions as list

fig = plt.figure(figsize=figdims)
```

Setting a nice figure ratio (cont.)

- When you include in the LaTeX source you should specify the scale factor as the width:

```
\begin{figure}
\includegraphics[width=0.45\textwidth]{figure.pdf}
\end{figure}
```

Eliminate unnecessary whitespace

- Eliminate the whitespace with:

```
fig.tight_layout(pad=0.1)
```

- Finally, save your figure in a useful format:

```
fig.savefig('plot.pdf', dpi=600)
```

Summary

- Simple ,interactive plotting:
  - numpy allows you to easily read data
  - Plotting syntax is simple and concise
- Complex plotting types also available
  - Can start from code for simple plots
  - Many examples available online
- Producing publication-ready images is relatively simple
  - Easily customised for different scenarios
- The more you use matplotlib, the more you get out of it!

SciPy & other packages

Arno Proeme, ARCHER CSE Team
aproeme@epcc.ed.ac.uk
Attributed to Jussi Enkovaara &
Martti Louhivuori, CSC Helsinki

Reusing this material

http://www.archer.ac.uk
support@archer.ac.uk

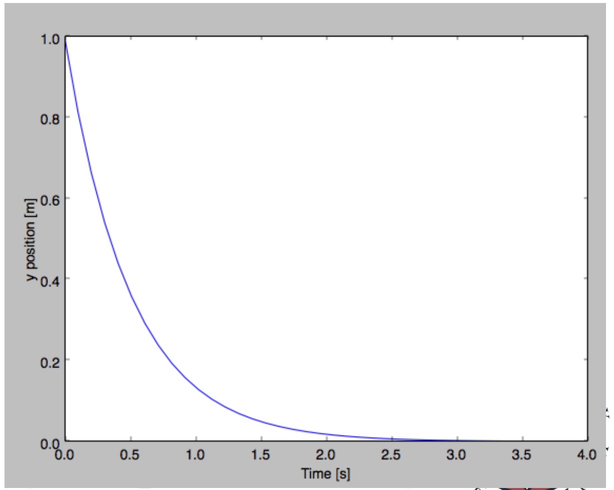
SciPy

- NumPy provides arrays, basic linear algebra, random number generation, and Fourier transforms
- SciPy builds on NumPy (e.g. by using arrays) and expands this with (additional) routines for:
  - Numerical integration
  - Interpolation
  - Linear algebra and wrappers to LAPACK & BLAS
  - Sparse linear algebra
  - Image processing
  - Optimisation
  - Signal processing
  - Statistical functions
  - Spatial data structures and algorithms
  - Airy functions
- Note: no PDE solvers (though other packages exist)

Integration

- Routines for numerical integration – single, double and triple integrals
- Function to integrate can be given by function object or by fixed samples
- e.g. solve the ODE
  - dy/dt = -2y between t = 0..4, with the initial condition y(t=0) = 1

```
import numpy as np
from scipy.integrate import odeint
def calc_derivative(ypos, time):
        return -2*ypos
time_vec = np.linspace(0, 4, 40)
yvec = odeint(calc_derivative, 1, time_vec)
pl.plot(time_vec, yvec)
```



Optimisation

- Several classical optimisation algorithms
  - Quasi-Newton type optimisations
  - Least squares fitting
  - Simulated annealing
  - General purpose root finding
- Rosenbrock function

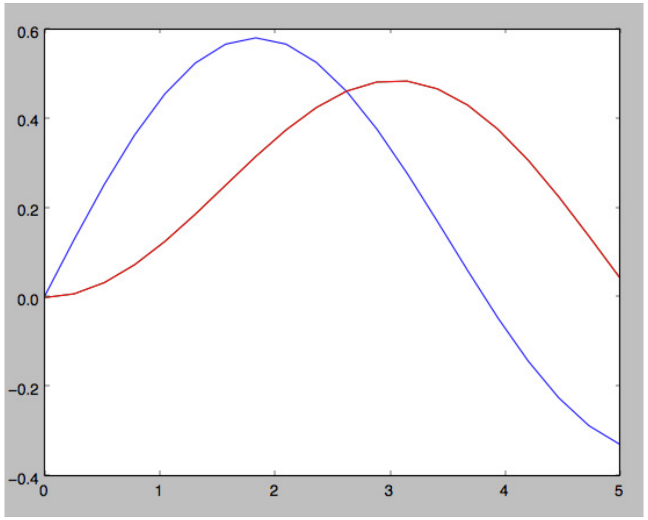
$$f(\mathbf{x}) = \sum_{i=1}^{N-1} (x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2$$

```
>>> from scipy.optimize import fmin
>>> def rosen(x):
... return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin(rosen, x0, xtol=1e-8)
 Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
```

Special functions

- SciPy contains huge set of special functions – Bessel functions
  - Legendre functions
  - Gamma functions
  - Bessel function →

```
>>> from scipy.special import *
>>> x = np.linspace(0, 5, 20)
>>> plot(x, jv(1, x))
>>> plot(x, jv(2, x))
```



Linear Algebra

- Wider set of linear algebra operations than in Numpy
  - decompositions,
  - matrix exponentials
- Routines also for sparse matrices
  - storage formats
  - iterative algorithms

```
>>> import numpy as np
>>> from scipy.sparse.linalg import LinearOperator, cg
>>> # Define "Sparse" matrix-vector product
>>> def mv(v):
>>>      return np.array([ 2*v[0], 3*v[1]])
>>> A = LinearOperator( (2,2), matvec=mv, dtype=float )
>>> b = np.array((4.0, 1.0))
>>> x = cg(A, b) # Solve linear equation Ax = b with conjugate gradient
>>> x
(array([ 2.        ,  0.33333333]), 0)
```

Other packages

- Pandas
  - Offers R-like statistical analysis of numerical tables and time series
- SymPy
  - Python library for symbolic computing
- scikit-image
  - Advanced image processing
- scikit-learn
  - Package for machine learning
- Sage
  - Open source replacement for Mathematica / Maple / Matlab (built using Python)

Fortran/C Interface: f2py

Andy Turner, ARCHER CSE Team
a.turner@epcc.ed.ac.uk

Reusing this material

http://www.archer.ac.uk
support@archer.ac.uk

Why interface to Fortran/C

- Provide glue to dynamically organise code
  - Complex software coordination provided by Python
- Performance of compiled codes with flexibility of Python
  - *e.g.* incorporate Python analysis and visualisation into existing codebase
  - Provide flexible way to extract results from code using Python
- Reuse code that you already have
  - Gradually introduce new functionality using Python

What is required?

- Name of external function
- Types of arguments to be passed from Python to external functions:
    - Integers, real numbers, arrays, characters?
- Sequence of arguments
- Are the arguments input parameters, output parameters or to be modified by the external function?
- Packaged in a way that can be imported by Python
- *f2py* provides a way to do this simply and quickly

f2py: Interfacing to Fortran

- Provides a way to describe external functions and their arguments
- Packages-up the external code in a way that can be imported and used by Python
- You need to provide:
  - The Fortran source code (to be compiled)
  - A file describing the external function and arguments (f2py can help you generate this)

Example: array_sqrt.f90

```
! Example Fortran: sqrt of array
subroutine array_sqrt(n, a_in, a_out)
       implicit none
       integer, intent(in) :: n
       real*8, dimension(n), intent(in) :: a_in
       real*8, dimension(n), intent(out) :: a_out
       integer :: i
       do i = 1, n
              a_out(i) = sqrt(a_in(i))
       end do
end subroutine array_sqrt
```

Create signature file

- f2py can try to create the signature file automatically:

```
f2py array_sqrt.f90 -m farray -h array_sqrt.pyf
```

- The Python module will be called: farray
- Signature in text file called: “array_sqrt.pyf”

Produce compiled library

- Once you have verified that the signature file is correct
- Use f2py to compile the library file that can be imported into Python:

```
f2py -c array_sqrt.pyf array_sqrt.f90
```

- Produces a library file called: farray.so

Calling from Python

```
>>> from farray import array_sqrt
>>> import numpy as np
>>> a = np.array([1.0,2.0,3.0,4.0])
>>> array_sqrt(a)
array([ 1.        ,  1.41421356,  1.73205081,  2.        ])
```

f2py: Interfacing to C

- f2py is the simplest way to interface C to Python
- Basic procedure is very similar to Fortran
- Differences:
  - You must write the signature file by hand
  - You must use the intent(c) attribute for all variables
  - You must define the function name with the intent(c) attribute
  - <u>Only 1D arrays can be handled by C, if you pass a multidimensional array you must compute the correct index</u>.
- Build in exactly the same way as Fortran example (but with different source code!)

Example: Signature file

```
python module farray
  interface
    subroutine array_sqrt(n,a_in,a_out)
      intent(c) :: array_sqrt
      intent(c)  ! Adds to all following definitions
      integer, optional,intent(in),check(len(a_in)>=n),depend(a_in) :: 
n=len(a_in)
      real*8 dimension(n),intent(in) :: a_in
      real*8 dimension(n),intent(out),depend(n) :: a_out
    end subroutine array_sqrt
  end interface
end python module farray
```

Other Options for C

- Native Python interface
  - Fully-flexible and portable
  - Complex and verbose
  - Best if you are interfacing a large amount of code and/or have a large software development project
- Cython
  - Standard C-like Python (or Python-like C)
  - (I have never had much success…)
- SWIG
  - Very generic and feature-rich
  - Supports multiple languages other than Python (e.g. Perl, Ruby)

Summary

- f2py is a simple way to call Fortran/C code from Python
  - Simpler for Fortran than for C
  - Care needed when using multidimensional arrays in C
- Calling sequence is converted to something more Pythonic:

  `array_sqrt(n, a_in, a_out)`, becomes:
  `a_out = array_sqrt(a_in)`

- Fortran/C can give better performance than Python

Goodbye!

Virtual tutorial has finished