Python as your next Matlab?

aka Python for Matlab Engineers
Training Session

Who is this for?

- You use Matlab for engineering tasks
- You use Windows
- You have little or no knowledge of Python
- You'd like to use Python

- We will focus on interactive use and iterative development, as in Matlab

Plan of the session

1. Context
2. Introduction to Python
3. Journey towards a Matlab-like Python
   - environment
   - numpy
   - matplotlib
   - I/O
4. Applications
   - data analysis
   - user interface

1. Context

As you know, Python is...

- a general-purpose language
- easy to write, read and maintain (generally)
- interpreted (no compilation)
- garbage collected (no memory management)
- weakly typed (duck typing)
- object-oriented if you want it to
- cross-platform: Linux, Mac OS X, Windows
    - this session will use Windows
    - most of what we present works on other platforms

Science & Engineering

- Python has been around for quite a while in the scientific and engineering communities, thanks to its ease of use as a "glue" language
- During the past 10 years, Python became a viable end solution for scientific computing, data analysis, plotting...
- This is mostly thanks to lots of efforts from the Open Source community, leading to the availability of mature 3rd-party tools
  e.g. numpy, scipy, matplotlib, ipython...
- Lots of momentum right now

Why Python?

	Matlab	Python
development model	closed	open
price	$$$	free
learning curve	easy	used to be hard

IMO: Python's advantage is its extreme flexibility makes coding fun and rewarding

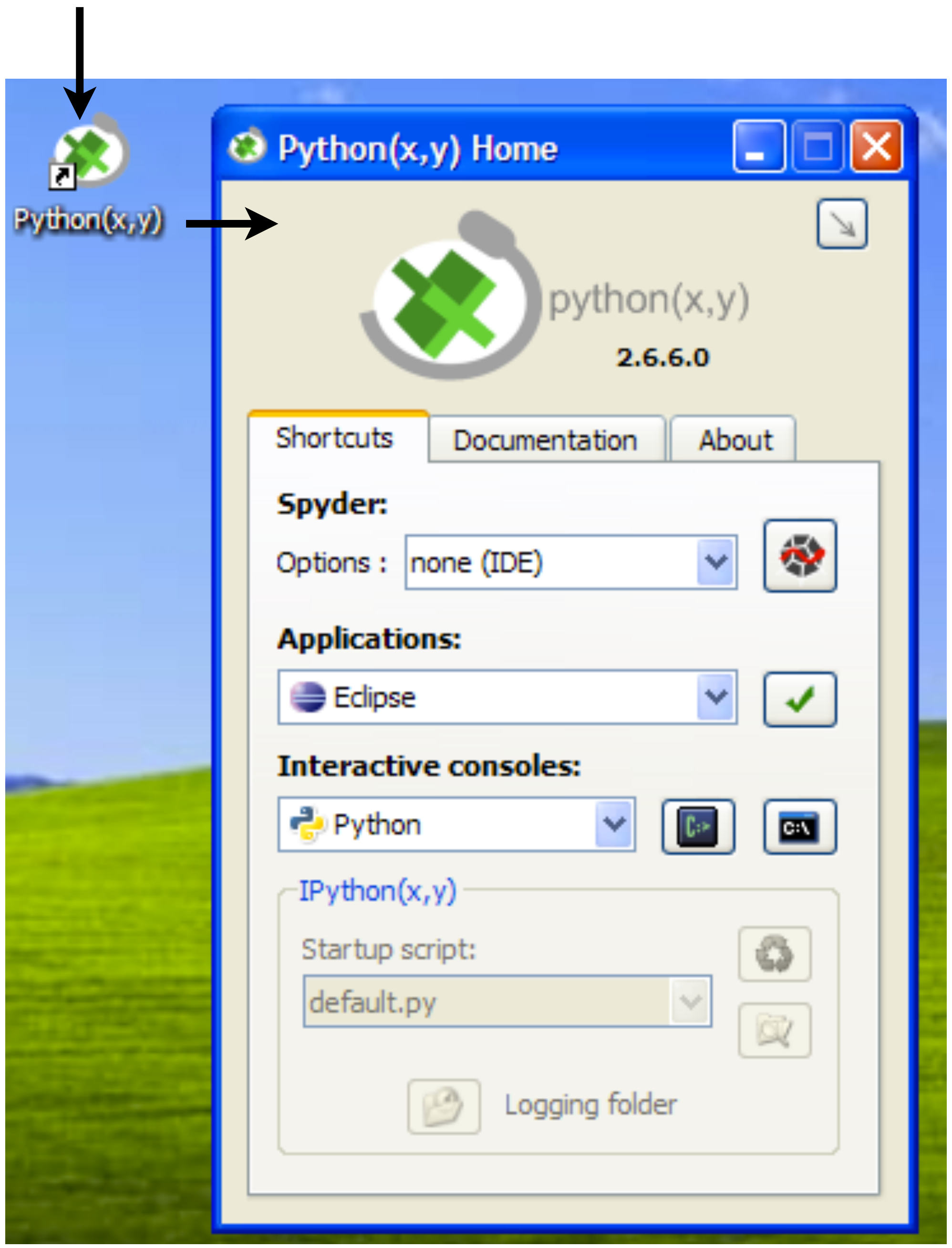
Fragmentation

- Problem:
  - fast Python development = lots of different versions
  - lots of packages to install with lots of versions
  - dependencies on each other
    - sometimes on 3rd-party libraries
  - installing the whole stack can be tricky
- Solution: Python distributions
  - Everything in the box
  - Python-CDAT, SAGE, EPD...
  - Today we use Python(x,y) (Windows-only so far)

Python(x,y)

- http://www.pythonxy.org
- If you haven't downloaded and installed it yet, please do it now (full install)
- Also download the data.zip archive linked to in the Session abstract page and unpack the data files
- Python(x,y) provides
  - A recent version of Python (2.6.6, 2.7 coming up)
  - pre-configured packages and modules for engineering in Python with dependencies
  - Visualization tools
  - improved consoles
  - Spyder (Matlab-like IDE)

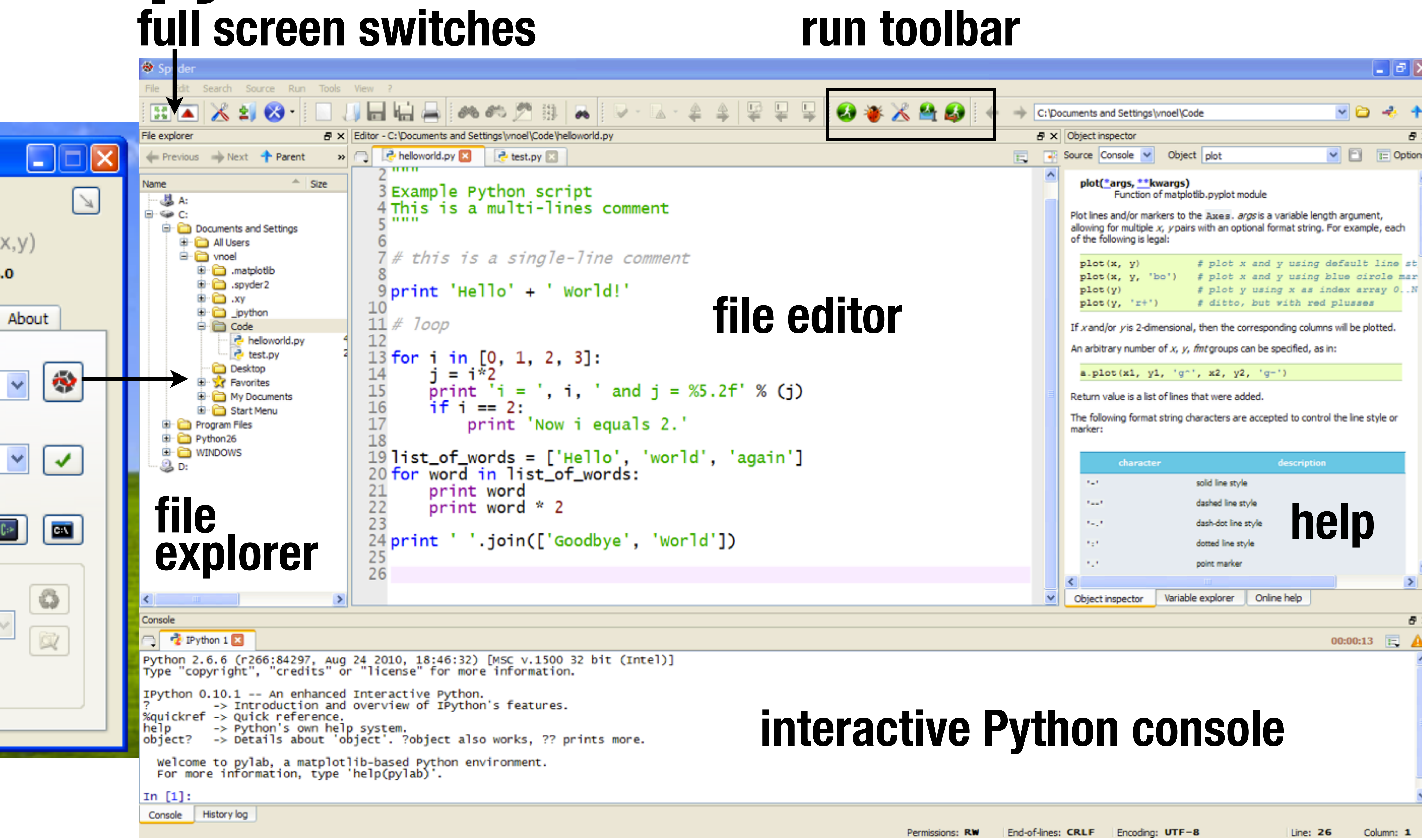
Python(x,y) launcher



Gives access to

- interactive consoles
- Python-related tools
- Documentation
- Spyder IDE
- can hide in the tray

Spyder


full screen switches
run toolbar
file editor
file explorer
help
interactive Python console

2. Python

Introduction to Python

- Interactive mode for now
  - please put the console in full screen

- Variables
- Functions and flow control
- Exceptions
- Modules
- Scripts

Variables | 1

Integer	`x = 1` `x = int('1')`	`x / 2 ?`
Float	`x = 1.` `x = float('1')`	`x / 2 ?`
String	`x = 'blablabla'` `x = '%d:%5.2f' % (1, 3.14)`	single or double quotes Q: 'a' + 'b' ?

Variables | 2

Arrays	`x = array([1,2,3,4])  # int32`
	`y = array([1.1, 2])   # float64`
	`z = array([])         # empty`
		fast, vectorized operations
		~ matrices

List	`x = [1, 2, 3]`
	`y = ['euro', 'Python', 2011]`
	`z = []`
		multiple-type arrays
		~ cells
		more flexible

Dictionaries	`p = {1:'a', 'b':23,'c':{53:'x', 12:0}}` `p.keys()`	named fields (e.g. parameters) ~ struct
Tuples	`x1 = (1, 2, 3)`	~ read-only lists

Arrays | creation

```
> x = array([1, 2.0, 3])
> x = arange([0, 10, 0.1])
> x = zeros([10, 10])
> x = ones([10, 10, 3])
```

Arrays | slicing

```
> x = array([0, 1, 2, 3, 4, 5, 6, 7])
> x[start:stop:step]
```

- First element index 0 (not 1)
- last element omitted: `x[n1:n1+n]` returns n elements

  ```
  > x[5:6] -> array([5])
  > x[5] -> 5
  > x[0:5:1] -> array([0, 1, 2, 3, 4])
  ```

- defaults: start=0, end=last element, step=1

  ```
  > x[0:5:1] == x[:5:] == x[:5]
  > x[::2]
  ```

Arrays | more slicing

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

- Negative indices: from the end

```
> x[-1] -> 7
> x[-2] -> 6
> x[-4:-2] -> array([4, 5])
```

- Negative stepping

```
> x[-2:-4:-1] -> array([6,5])
> x[::-1]
```

Arrays | more slicing

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

- Negative indices: from the end
  - `x[-1] -> 7`
  - `x[-2] -> 6`
  - `x[-4:-2] -> array([4, 5])`
- Negative stepping
  - `x[-2:-4:-1] -> array([6,5])`
  - `x[::-1]`

All this slicing gets confusing pretty fast. Remember :

`x[0]  x[-1]  x[:n]  x[-n:] x[::-1]`

Arrays | views

- A sliced array is a <u>view</u> of the original array

```
> x = ones([4,4])
> y = x[0:2,0:2]
> x[0,0] = 5
> y
array([[ 5.,  1.],
       [ 1.,  1.]])
```

- To get a separate array

```
y = x[0:2,0:2].copy()
```

- not needed most of the time

Arrays | r_[]

- `r_[]`
- can replace `array()`
  > `x = r_[1, 2.0, 3] = array([1, 2.0, 3])`
- can create vectors based on slice notation, with floats
  > `x = r_[1:10:0.1]  # start, stop, step`
  > `x = r_[1:10:100j] # start, stop, npoints`

Arrays | operations

- Inspection
  - shape(x)   # size in matlab
  - ndim(x)
  - size(x)
- Manipulation
  - x.T   # x' in matlab
  - x*y   # x.*y in matlab
  - reshape(x, [4, 2])
  - x = append(x, 8)
    - x = append(x, r_[8:20])
  - z = concatenate(x, y)
  - xx, yy = meshgrid(x, y)

Arrays | basic math

- math operations on arrays are vectorized, fast (~matlab)
  - `sin, cos, tan...`
    `mean, std, median, max, min, argmax...`
    `sum, diff, log, exp, floor, bitwise_and...`
  - `xsum = sum(x)`
- can be applied on single dimensions

```
> x = ones([4,2])
> sum(x, 1)
array([ 2.,  2.,  2.,  2.])
```

Lists | 1

- contain items of any type (incl. lists)
- strings are lists

```
> x = [0, 1, 2, 3.0, 'baba']
> y = [4, 5, x]
```

- indexed like 1-dimensional arrays

```
> y[0] ?
> y[2][4] ?
> y[2][4][3] ?
```

- very flexible, generic container

Lists | 2

- Deeply ingrained in Python
- Often you can loose the brackets
  > x = [1, 2, 3]
  > a, b, c = x # works with numeric arrays also
  > a, b = b, a
- numeric arrays and lists are converted when needed
- very useful with functions returning lists
- use them

Lists | functions

- lots of functions to deal with lists, try those
  - `x = [5,2,8,5,3]`
  - `len(x)`
  - `x.append('ab')`
  - `x.extend(['ba', 'abc'])`
  - `sort(x) # == x.sort(), sorts in-place`
  - `x.count('a')`
  - `x.index('a')`
  - `x.remove('a')`
  - `x.remove(x[3])`
- array functions work on lists, not the opposite

Dictionaries

- key-value structures ~ matlab struct
- keys can be any hashable - string, int, float

> x = {key1:value1, key2:value2}
>
> > x = {0:12, 'a':37, 'b':{'c':12, 2:23}}
> >
> > x['b'][2] -> 23

- convenient for parameters, named data

> > params = {'min':3., 'max':10.}
> >
> > data = {'dates':dates, 'temperature':temp}
> >
> > process_data(data, params)

Variables | one last thing

- Python tracks the **content of variables**, not their names. The same content can have several names.
    - In practice, this may or may not be a problem
    - but you need to know about it
- If content has no more names, it is deleted from memory

x=[0,1,2,3] y = x	x ↘ [0,1,2,3] y ↗
y[1]=5	x ↘ [0,5,2,3] y ↗

x=[0,1,2,3] y = x[:]	x → [0,1,2,3] y → [0,1,2,3]
y[1]=5	x → [0,1,2,3] y → [0,5,2,3]

the [:] syntax requests a copy

Flow control | for

- **followed by a colon :** and an **indented block**
- indentation directs logic
- no "end" keyword
- `for` <u>does not</u> iterate on number
- `for` iterates on elements from "iterables"
  - default iterables: arrays, lists, tuples, dictionaries, strings

```
xlist = [0,1,2,3]
for x in xlist:
    print x*2
for x in 1,2,3:
    print x*2
```

Flow control | for

- `for i in (0:12):` won't work
- Solutions
  - `for i in range(0,12):`
  - `for i in r_[0:12]:`
- Useful function 1: enumerate()

```
items = ['aa', 'bb', 'cc']
for i, item in enumerate(items):
  print i, item
```

- Useful function 2: zip()

```
xlist = r_[0:4]
ylist = ['a', 'b', 'c', 'd']
for x, y in zip(xlist, ylist):
  print x, y
```

list comprehensions

- An example of Python flexibility
- list manipulation

```
xlist = [0, 1, 2, 3]
ylist = []
for x in xlist:
   ylist.append(x*2)
```

→

```
xlist = [0, 1, 2, 3]
ylist = [x*2 for x in xlist]
```

- and filtering

```
ylist = [x*2 for x in xlist if x>2]
```

list comprehensions | 2

- Almost like vectorization

```
coslist = [cos(x) for x in r_[0:3.14:0.01]]
coslist = cos(r_[0:3:14:0.01])
```

- Works with any variable type

```
males = ['John', 'Albert', Bernard']
names = ['Mr. '+male for male in males]
```

Flow control | while and if

- while

```
x = 3
while x < 10:
    print 'x = ', x
    x = x + 1
    # x += 1
```

- if, elif, else

```
if x < 2:
    print 'x is small'
elif x < 4:
    print 'x is not so small'
else:
    print 'x is large'
```

- conditions

```
if x:             # = if x is True:
if not x:
if x==y and (x!=z or x==k):
if x<y<z:         # if (x<y) and (y<z):
if x in xlist:
if x not in xlist:
('a' in 'abc') == True
```

- no switch statement
- break exits a loop
- continue loops around
- pass does nothing

Scripts!

```
# this is a single-line comment

print 'Hello' + ' World!'

# loop

for i in [0, 1, 2, 3]:
    j = i*2
    print 'i = ', i, ' and j = %5.2f' % (j)
    if i == 2:
        print 'Now i equals 2.'

list_of_words = ['Hello', 'world', 'again']
for word in list_of_words:
    print word
    print word * 2
```

- script extension .py
  - convention
  - please follow it
- Indentation directs logic
- no «end» in blocks
- no end-of-line character ;
- no output without print

Scripts!

```
# this is a single-line comment

print 'Hello' + ' World!'

# loop

for i in [0, 1, 2, 3]:
    j = i*2
    print 'i = ', i, ' and j = %5.2f' % (j)
    if i == 2:
        print 'Now i equals 2.'

list_of_words = ['Hello', 'world', 'again']
for word in list_of_words:
    print word
    print word * 2
```

comment lines begin with a hash #

indentation can be any number of spaces or tabs, just stay consistent within a same script

linebreaks inside containers () [] {}... no symbol

Otherwise \

Modules

- Functions not in core Python are provided by modules
  - module = regular Python .py file
  - contains definitions of functions, variables, objects
  - Any Python file can be a module
  - modules ~ .m function files
- Python standard library: lots of modules
  - file manipulation, network, web, strings...
- Python(x,y): lots of 3rd-party modules
- A folder of related modules = a package
  - packages ~ Matlab toolboxes
  - scipy, numpy, matplotlib = packages

Modules | import

- A module must be **imported**

```
> import mymodule
```

- then use any function or object defined in the module, e.g.

```
> mymodule.fun(x,y) # call the fun function
```

- You can import modules or packages

Modules | import variants

import command	object access	
`import mymodule`	`mymodule.fun()`	function calls are hard to write
`from mymodule import fun`	`fun()`	function imports are hard to write
`from mymodule import *`	`fun()`	convenient but name collisions possible - **avoid if possible**
`import mymodule as my`	`my.fun()`	just right (trust us)

Modules | $PYTHONPATH

- Python looks for modules
  - in the same path as the running script
  - in the system-wide modules folder site-packages/
  - In paths specified in `$PYTHONPATH`
- Put a .py file in a folder, add the folder to the `$PYTHONPATH`: you can import the module from anywhere
  - ~ adding `addpath` commands to `startup.m`
- In Spyder, change `$PYTHONPATH` from the Tools menu

Functions

```
def multiply_by_two(x, and_add=0):
     xnew = x*2.
     xnew += and_add
     return xnew
```

- Functions are defined by `def`
- they `return` one variable of any type
- `and_add` is a keyword argument with a default value
  - `sum(x, 1) == sum(x, axis=1)`
  - `y = multiply_by_two(3) ?`
  - `y = multiply_by_two(3, and_add=7) ?`
  - `y = multiply_by_two(3, 7) ?`

Exceptions

- Errors trigger exceptions

```
> 'a' + 24
```

- An exception stops execution
- Most of the time, exception = bug - but not always.

Exceptions handling

- Exceptions can be caught

```
try:
  f = open('file.txt', 'r')
except IOError:
  print 'Cannot open file'
```

This is the valid way to handle IO error - the exception is not a bug

- ~ `try... catch` in Matlab
- A plain `except` catches all exceptions - use with extreme caution

Misc

- del var1, var2 # clear in matlab
- reset # clear all
- x = nan
  - isnan(x)
  - isinf(x)
  - isfinite(x)

End of Section #2

- There is more to Python
  - Generators
  - Decorators
  - string operations
  - Lambda functions
  - useful modules
    - datetime, csv, json
  - etc.
  - Some Python fun at the end
- http://www.python.org

Section #2 - Lab

Write a Python script that:

- reads command-line parameters
- outputs their total number
- lists them one by one with their position
- prints out each parameter multiplied by two
- try it with a few parameters
- Hints:
  - Command-line parameters are in the `argv` list from the `sys` module
    - `import, print, for, enumerate`

3. Journey towards a Matlab-like Python

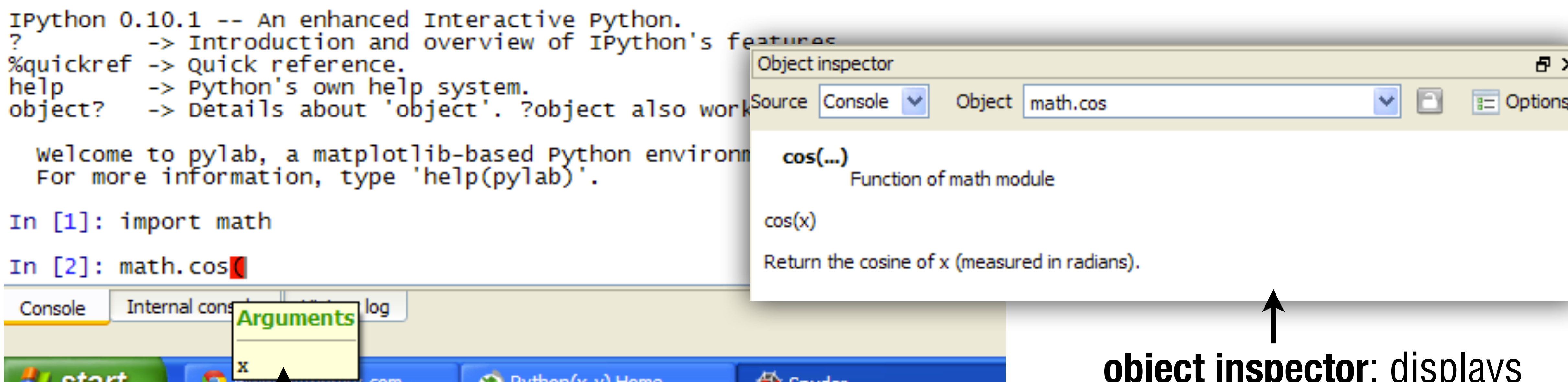
3. Journey towards a Matlab-like Python

1. Environment

Spyder

- Spyder provides a programming environment similar to recent Matlab
    - file explorer
    - code editor with autocompletion, suggestions
    - improved IPython console
    - contextual help linked to code editor, console
    - variable explorer (editable)
    - continuous code analysis

Spyder | help system

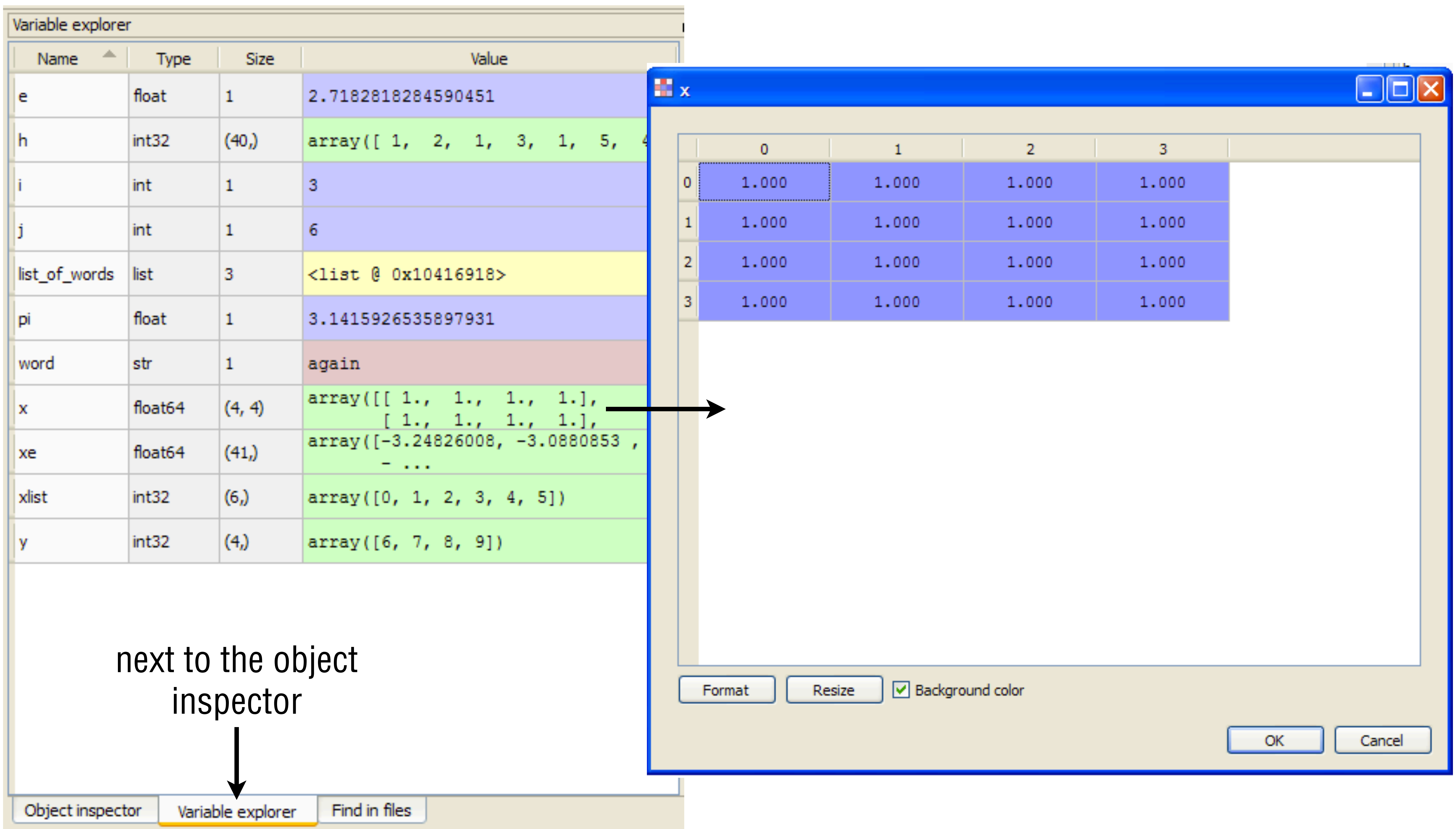


argument suggestion in the console and the editor

object inspector: displays documentation for what you type (console/editor)

- in the console
  - help function
  - function?
  - source function

Spyder | variable explorer



IPython

- In Spyder, the default Python console is **IPython**
  - you've been using it
  - much better than the standard Python console
  - tab completion for functions, modules, variables, files
  - filesystem navigation (`cd`, `ls`, `pwd`)
  - syntax highlighting
  - "magic" commands
    - `%whos`, `%hist`, `%reset`
    - `%run script.py`
    - type `%`⇥ to see them all
    - you can drop the `%` for most of them (e.g `whos`, `run`, `reset`)
  - works fine with matplotlib

Projects

- Spyder, IPython suited for interactive use and iterative development
- Project-oriented development (e.g. full applications): Python(x,y) includes Eclipse and Pydev (Python plug-in for Eclipse)
    - we won't cover that today

3. Journey towards a Matlab-like Python

2. Numpy

numpy

- provides the <u>array</u> variable type
- and all associated functions
- developed since 1995
  - child of Numeric and numarray
  - now stable and mature, v.1.6 released May 2011
  - Python 3 coming up
  - basis for scipy, matplotlib, and lots of others

numpy | importing

- you've been using numpy
- IPython import all of numpy automatically
  - > `from numpy import *`
  - numpy functions can be called without prefix
  - convenient for interactive use
- In scripts, favor `from numpy import np`
  - Prefix all numpy functions with np
  - No namespace collision
  - Easier to read and understand your code later
  - Official convention (examples, etc.)

numpy | boolean indexing

- arrays can be indexed through slicing
- ...or through **boolean indexing**

```
> x = np.r_[0:10:0.1]
> idx = (x > 2.) & (x < 8.) # boolean array
> np.mean(x[idx])
> x[x<2] = 0.
```

  - replaces matlab `find`
- Parenthesis are mandatory

```
> idx = x > 2 & x < 8      # won't work
```

- Lots of numpy functions return boolean indexes
  - e.g. `np.isfinite`, `np.isnan`
- Boolean indexing returns a new array (not a view)

numpy | boolean indexing

- `np.all(x > 3)`
- `np.any(x > 3)`
- `np.sum(x > 3)`
- `np.in1d(x, y)`
  - checks if x elements are in y
- `np.argmin(x > 3)`
  - index of first element > 3

numpy | other stuff

- structured arrays
- Some I/O (very limited on purpose)
- Matrix computations : `np.matrix()`
- Interpolation: `np.interp()` (more in scipy)
- Histograms at 1, 2, n dimensions: `np.histogram()`, `np.histogram2d()`, `np.histogramdd()`
- Modules within numpy
  - Random generators: `np.random`
  - Masked arrays: `np.ma`
- more at http://docs.scipy.org/doc/numpy/reference

numpy | Lab

- Write a script to
  - Draw 1000 random elements
  - finds elements > 0.5 and < 0.8
  - compute the mean and standard deviation of the population of these elements
  - Hints:
    - `np.random.rand()`

3. Journey towards a Matlab-like Python

3. Matplotlib

Matplotlib

- Lots of Python plotting modules/packages
  - PyNGL, Chaco, Veusz, gnuplot, Rpy, Pychart...
  - Some in python(x,y)
- Matplotlib emerging as a "standard"
  - all-purpose plot package
  - interactive or publication-ready EPS/PDF, PNG, TIFF
  - based on numpy
  - extensible, popular, stable and mature (v. 1.0.1)
  - Python 3 coming up

Matplotlib & Matlab

- Matplotlib can be used as an object-oriented framework
- can also follow a "matlab-like" imperative approach, through its pyplot module

```
> import matplotlib.pyplot as plt
> plt.figure()
> x = np.r_[0:2*pi:0.01]
> plt.plot(x, np.sin(x))
```

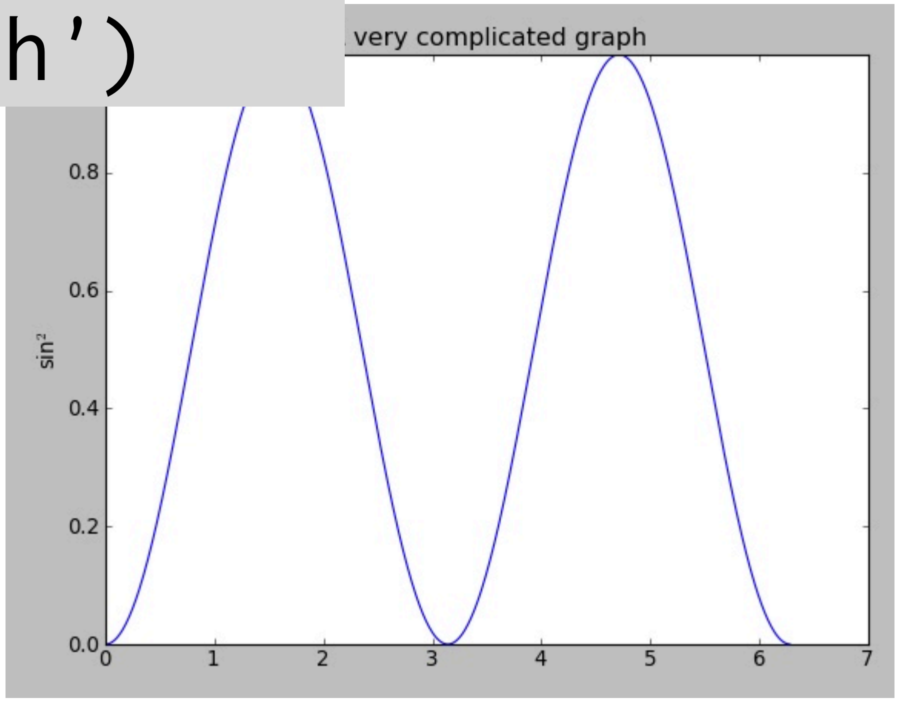
- `pyplot` functions follow a matlab-like syntax

```
plt.plot, semilogx/y, pcolor, pcolormesh,
   x/ylabel, title, legend, hist, figure, axis,
   subplot, contour, contourf, colorbar, quiver,
   axes, x/ylim, x/yticks...
```

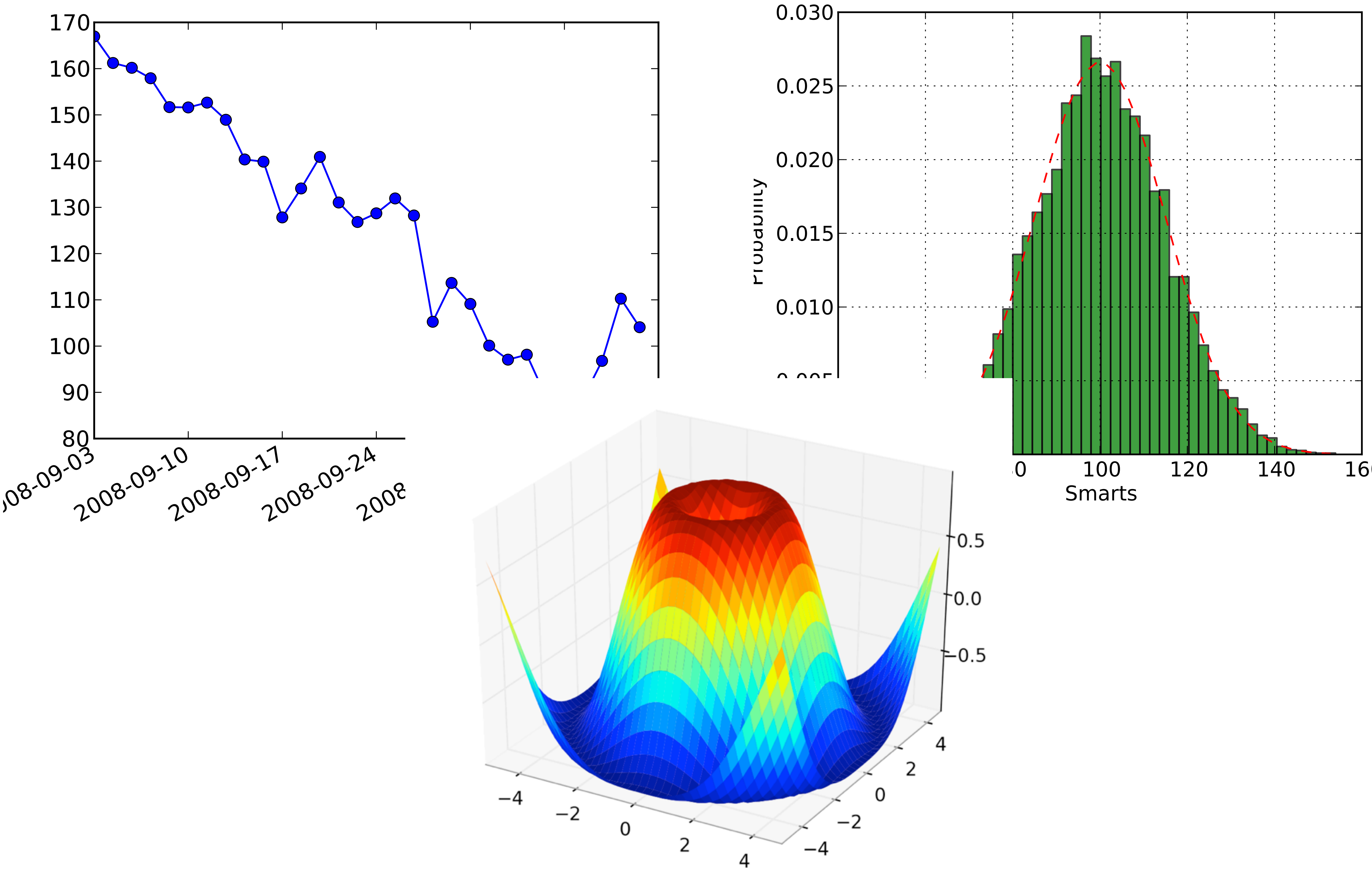
Matplotlib in python(x,y)

- IPython import all `matplotlib.pyplot`
- You can drop the `plt.` prefix for interactive use

```
figure()
x = r_[0:2*pi:0.01]
plot(x, sin(x)**2)
grid()
ylabel('sin$^2$')    # latex allowed
title('A very complicated graph')
```

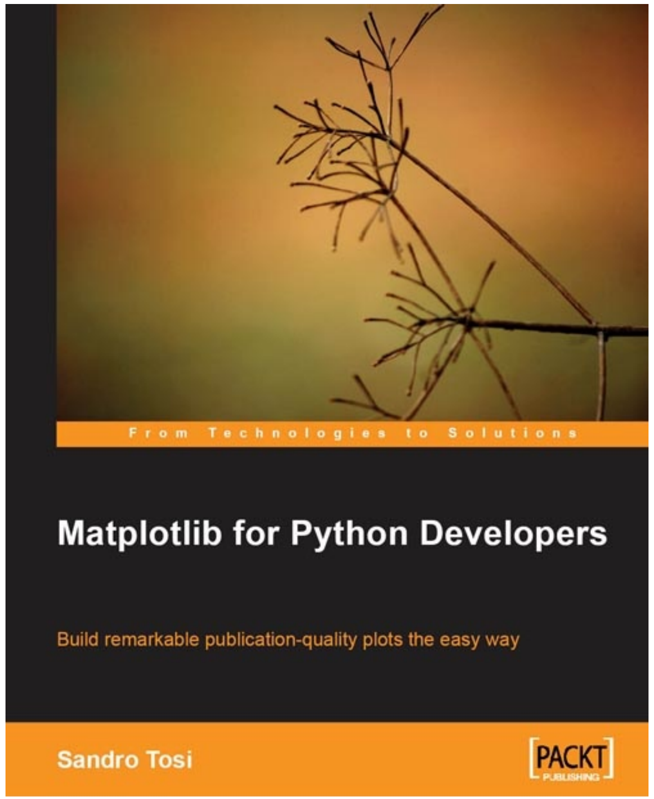


Examples


170
160
150
140
130
120
110
100
90
80
2008-09-10
2008-09-17
2008-09-24
0.030
0.025
0.020
0.015
0.010
100
120
140
160
Smarts
0.5
0.0
−0.5
−4
−2
0
2
4

Matplotlib | documentation

- Plotting commands are documented on the matplotlib homepage
  http://matplotlib.sourceforge.net
- The matplotlib gallery is a good starting point to find how to do a specific graph
  http://matplotlib.sourceforge.net/gallery.html



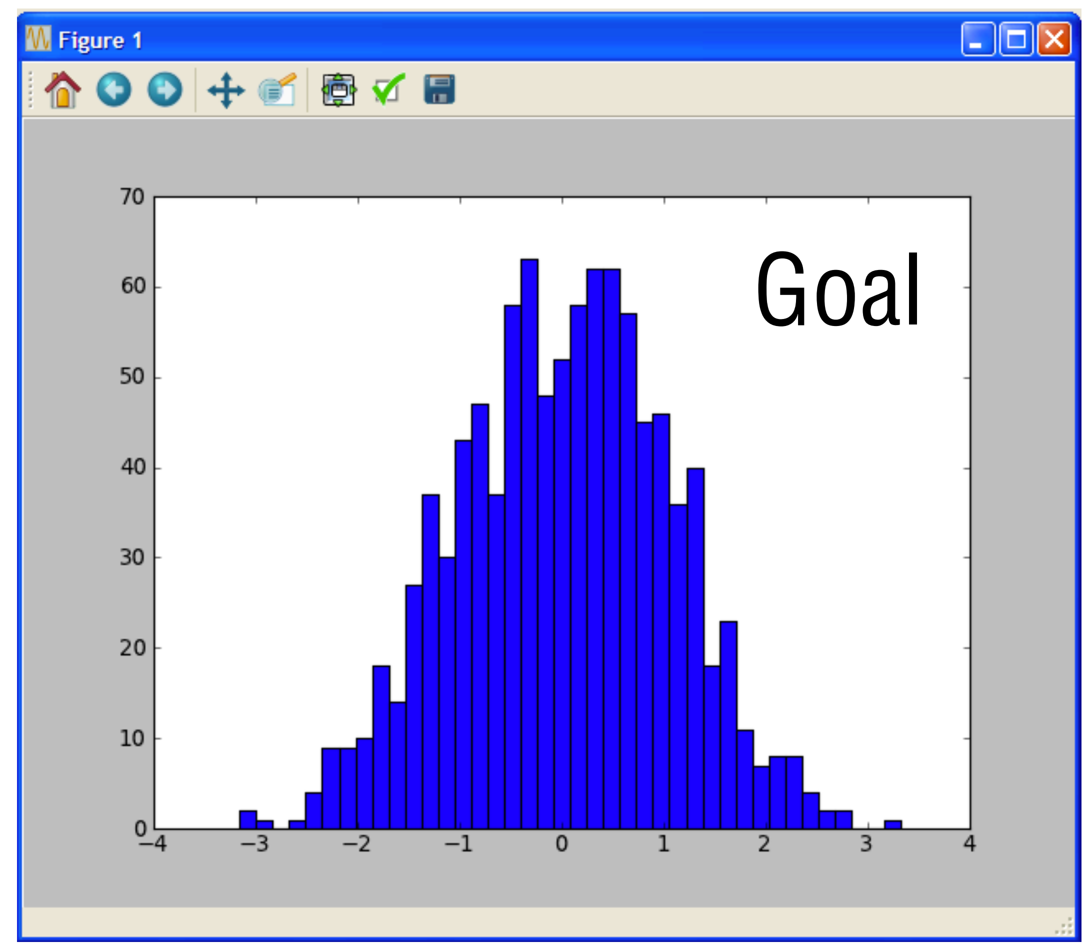
Matplotlib | Lab 1

- Write a Python script to
  - draws 1000 samples from a normal distribution
  - show a histogram of the sample distribution (40 bins)

- Hints:
  > `np.random.normal()`

  > `plt.hist()`

  - in a script you must call `plt.show()` when you want the figures to appear



3. Journey towards a Matlab-like Python

4. I/O

I/O | ASCII

- Read/write ASCII in pure Python
  - > `f = open('test.txt') # 'r' is default`
  - > `whole_file_as_a_string = f.read()`
  - > `f.close()`
  - > `f.readline()    # read a single line`
  - > `f.readlines()   # returns a list of all lines`

```
f = open('test.txt')
for line in f:
  print line
f.close()
```

```
# saving
f = open('test.txt', 'w')
f.write('bla=%d' % (2))
f.close()
```

I/O | ASCII

- Read data arrays from ASCII files
    - `data = np.loadtxt('data.txt')`
    - `returns a numpy array`
    - keyword parameters and defaults
        - `comments` - default `'#'`
        - `delimiter` - default `' '`
        - `converters={0:datestr2num}`
        - `skiprows -` default `0`
- `np.savetxt()`

I/O | npz

- easy way to read/save numpy arrays
  np.savez and np.load
- not very standard, confined to numpy use
- very useful for temporary storage
- ~ 'mat-files'

```
x = np.ones([100, 10])
y = x * 4.
np.savez('vars.npz',
xvar=x, yvar=y)
```

```
# later
npz = np.load('vars.npz')
x = npz['xvar']
y = npz['yvar']
```

I/O | Matlab files

- Python can read Matlab files
- `matlab` module of the `scipy.io` package
  > `from scipy.io import matlab`
- `loadmat` returns a dictionary
  > `mat = matlab.loadmat('file.mat')`
  > `mat.keys() -> names of variables`
  > `mat['longitude'] -> longitude array`
- Saving
  > `matlab.savemat('file.mat', {'longitude':lon})`

I/O | Excel files

- modules `xlrd`, `xlwt`

```
book = xlrd.open_workbook('simple.xls')
print book.sheet_names()
for sheet in book.sheets():
    print sheet.name, sheet.nrows, sheet.ncols
    for irow in range(sheet.nrows):
        for icol in range(sheet.ncols):
            print irow, icol, sheet.cell(irow, icol).value
```

```
    sheet = book.sheet_by_index(2)
```

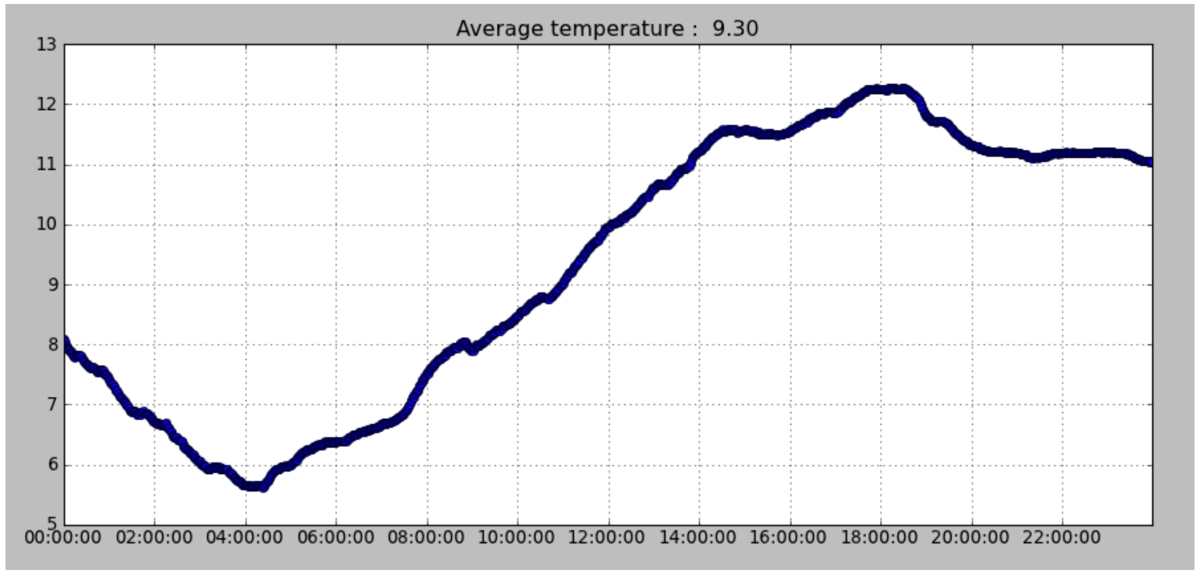
```
    sheet = book.sheet_by_name('sheet 1')
```

I/O | scientific datasets

- You might need to read and write datasets in structured and autodocumented file formats such as HDF or netCDF
- `netcdf4-python`
  - read/write netCDF3/4 files as Python dictionaries
  - supports data compression and packing
- `pyhdf`, `pyh5`, `pytables` : HDF4 and 5 datasets
- `pyNIO` : GRIB1, GRIB2, HDF-EOS
- in Python(x,y)
- Very good online documentation

I/O | Lab

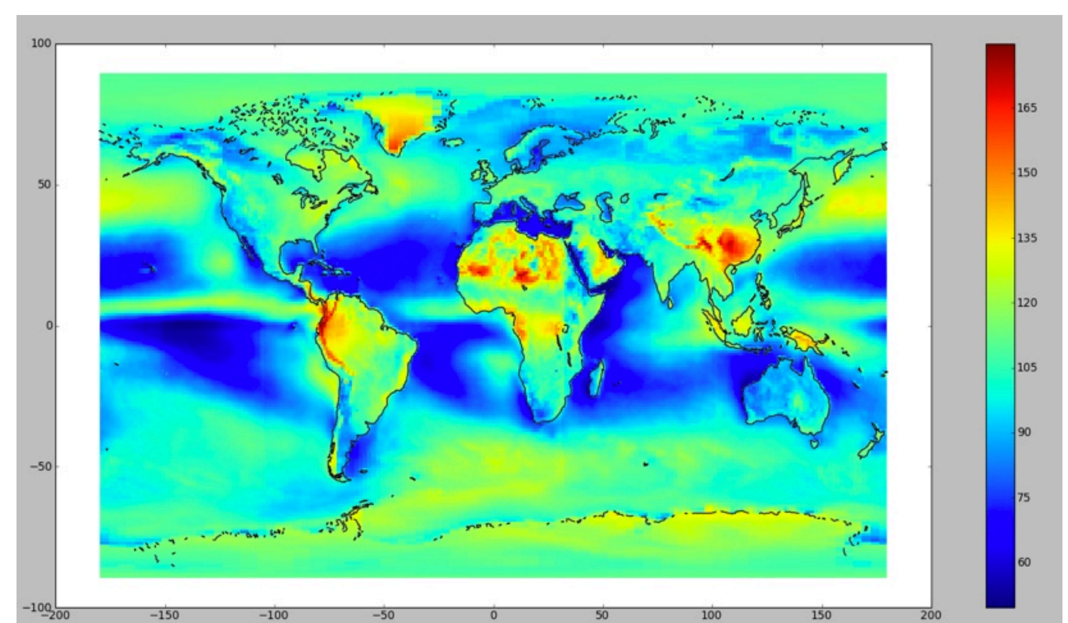
- Write a Python script to:
  - read the contents of the file `meteoz.asc`
  - plot the air temperature as a function of time when the air temperature quality flag is ok (=0)
  - display the temperature mean and standard deviation in the title
  - save the temperature in a npz file



- Hints:
  - > help `np.loadtxt`
  - > `datestr2num` is in `matplotlib.dates`
  - > `plt.plot_date()`

End of Section #3 | Lab

- Write a script to
  - read the variables `swup`, `lon`, `lat` from the matlab file `CERES_EBAF_TOA_Terra_Edition1A_200003-200510.mat`
  - average `swup` over the time dimension
  - plot the averaged `swup` as a map
  - plot the continent coastlines over the map in black
    - these are in `coastlines.mat`
  - add a `colorbar()`
- Hints
  > `plt.pcolormesh()`



4. Applications

4. Applications

1. Data Analysis

Scipy

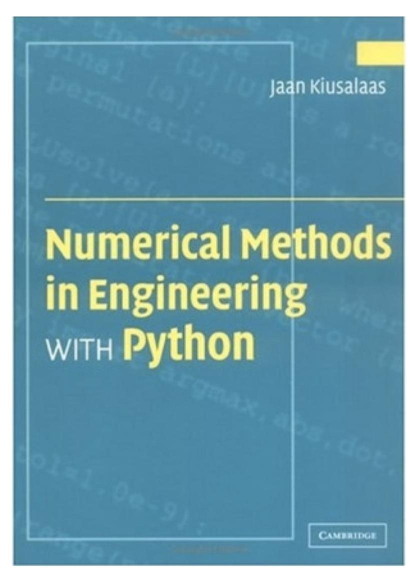
- Scipy is choke full of data analysis functions
- Functions are grouped in sub-packages
  - scipy.ndimage - image processing, morphology
  - scipy.stats
  - scipy.signal - signal processing
  - scipy.interpolate
  - scipy.linsolve, scipy.odeint
  - scipy.fftpack - Fourier transforms (1d, 2d, etc)
  - scipy.integrate...

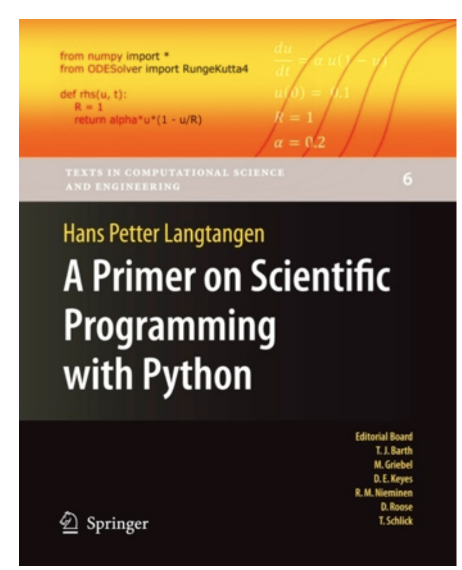
Scipy scikits

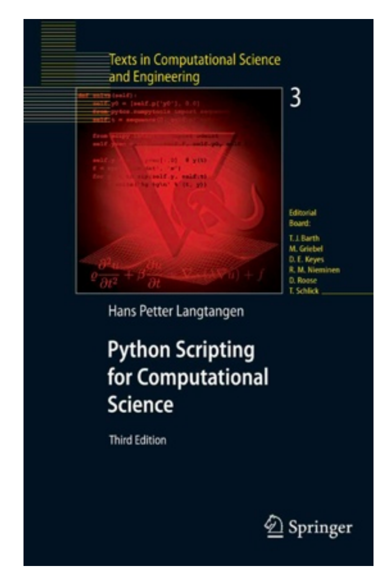
- SciKits are add-on packages for Scipy
- not in Scipy for various reasons
- http://scikits.appspot.com
  - datasmooth
  - odes - equation solvers
  - optimization
  - sound creation and analysis
  - learn - machine learning and data mining
  - cuda - Python interface to GPU libraries
  - ...

Scipy

- Too much to cover everything
- Scipy packages and modules are tailored for specific users
    - you don't even want to cover everything
- Best ways to find the function you need
    - google
    - tab exploration in IPython
    - `lookfor`
        - e.g. `lookfor("gaussian", module="scipy")`







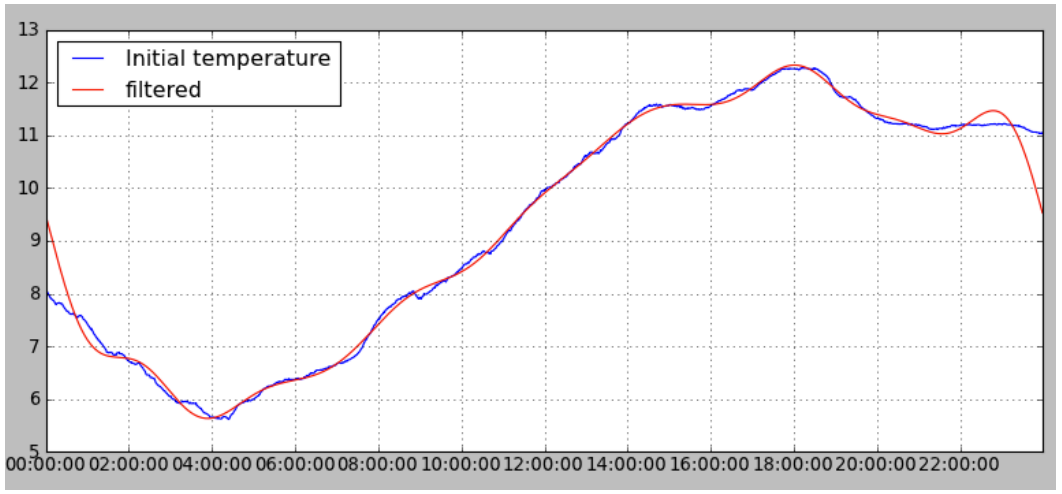
one example: Scipy.stats

- contains a lot of useful functions
- `nanmean/std`, `nanmin/max`, etc.
- pdf/cdf for ~100 distribution families
  - generic syntax: `scipy.stats.`<distribution>.<function>

```
> from scipy.stats import gamma
> x = np.r_[0:10:0.1]
> plt.plot(x, gamma.pdf(x,2))
> plt.plot(x, gamma.pdf(x,2,3))
```

  - catch them all with IPython autocomplete

scipy | Lab



- write a script to
  - plot air temperature data from the `meteoz.dat` file
  - compute the Fourier transform of the temperature
  - keep only the lowest 10 frequencies
  - compute filtered temperature using inverse Fourier transform
  - plot the initial temperature and the filtered temperature in different colors
  - add a legend (why not)
  - Hints

```
> from scipy.fftpack import fft, ifft
```

4. Applications

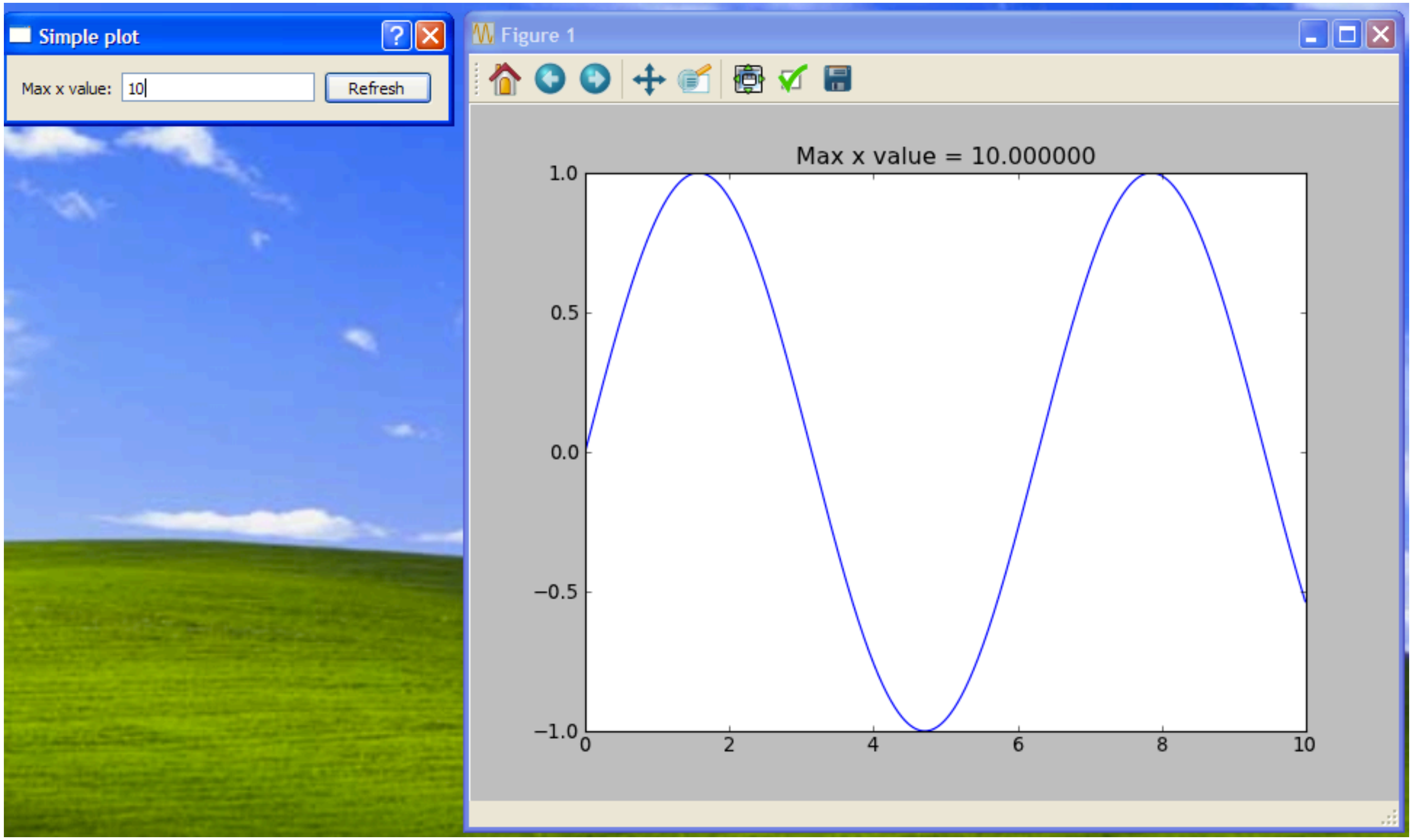
2. User interface

lots of options

- wxwindows, tk, gtk, matplotlib widgets, etc.
- Qt4
  - The Good
    - well-maintained, developed and documented
    - cross-platform, looks native on every platform
    - Complete Python bindings: PyQt4 (installed by Python(x,y))
  - The Bad: Some OOP required
  - The Ugly
    - Qt and PySide were supported by Nokia for phone interface research, after Microsoft takeover: future unclear...
    - for the near future one of the best choices

PyQt4

- You can mix Qt4 for the GUI and Matplotlib for plotting
- The best of both word



A bit of OOP

```
class MyObject(object):
  def __init__(self):
    self.x = 3

  def print_x(self):
    print self.x

  def add_to_x(self, y):
    self.x += y

myobj = MyObject()
myobj.print_x()
myobj.add_to_x(5)
myobj.print_x()
```

class keyword defines an object

__init__() method called at object instantiation

1st argument of class methods is always self

variables inside the object are accessed with self.variable

myobj is an instance of MyObject

object methods are called with `<object>.<method>`

the full script : ~40 LOC

```
# -*- coding: utf-8 -*-

import sys

from PyQt4.QtCore import *
from PyQt4.QtGui import *

import matplotlib.pyplot as plt
import numpy as np

class Dialog(QDialog):
    def __init__(self, parent=None):
        super(Dialog, self).__init__(parent)
        self.lineedit = QLineEdit('Max x value')
        layout = QHBoxLayout()
        self.button = QPushButton('Refresh')
        layout.addWidget(QLabel('Max x value:'))
        layout.addWidget(self.lineedit)
        layout.addWidget(self.button)
        self.setLayout(layout)
        self.lineedit.setFocus()
        self.connect(self.lineedit, SIGNAL('returnPressed()'), self.refresh)
        self.connect(self.button, SIGNAL('clicked()'), self.refresh)
        self.setWindowTitle('Simple plot')
        self.max = 2.*np.pi
```

```
    def refresh_values(self):
        self.x = np.r_[0:self.max:0.01]
        self.y = np.sin(self.x)

    def refresh(self):
        try:
            self.max = np.float(self.lineedit.text())
        except ValueError:
            print 'Please enter a numeric value'
            return
        self.refresh_values()
        plt.figure(1)
        plt.show()
        plt.clf()
        plt.subplot(111)
        plt.plot(self.x, self.y)
        plt.title('Max x value = %f' % (self.max))
        plt.draw()

app = QApplication(sys.argv)
dialog = Dialog()
dialog.show()
app.exec_()
```

```
# -*- coding: utf-8 -*-

import sys

from PyQt4.QtCore import *
from PyQt4.QtGui import *

import matplotlib.pyplot as plt
import numpy as np

class Dialog(QDialog):
    def __init__(self, parent=None):
        super(Dialog, self).__init__(parent)
        self.lineedit = QLineEdit('Max x value')
        layout = QHBoxLayout()
        self.button = QPushButton('Refresh')
        layout.addWidget(QLabel('Max x value:'))
        layout.addWidget(self.lineedit)
        layout.addWidget(self.button)
        self.setLayout(layout)
        self.lineedit.setFocus()
        self.connect(self.lineedit, SIGNAL('returnPressed()'), self.refresh)
        self.connect(self.button, SIGNAL('clicked()'), self.refresh)
        self.setWindowTitle('Simple plot')
        self.max = 2.*np.pi
```

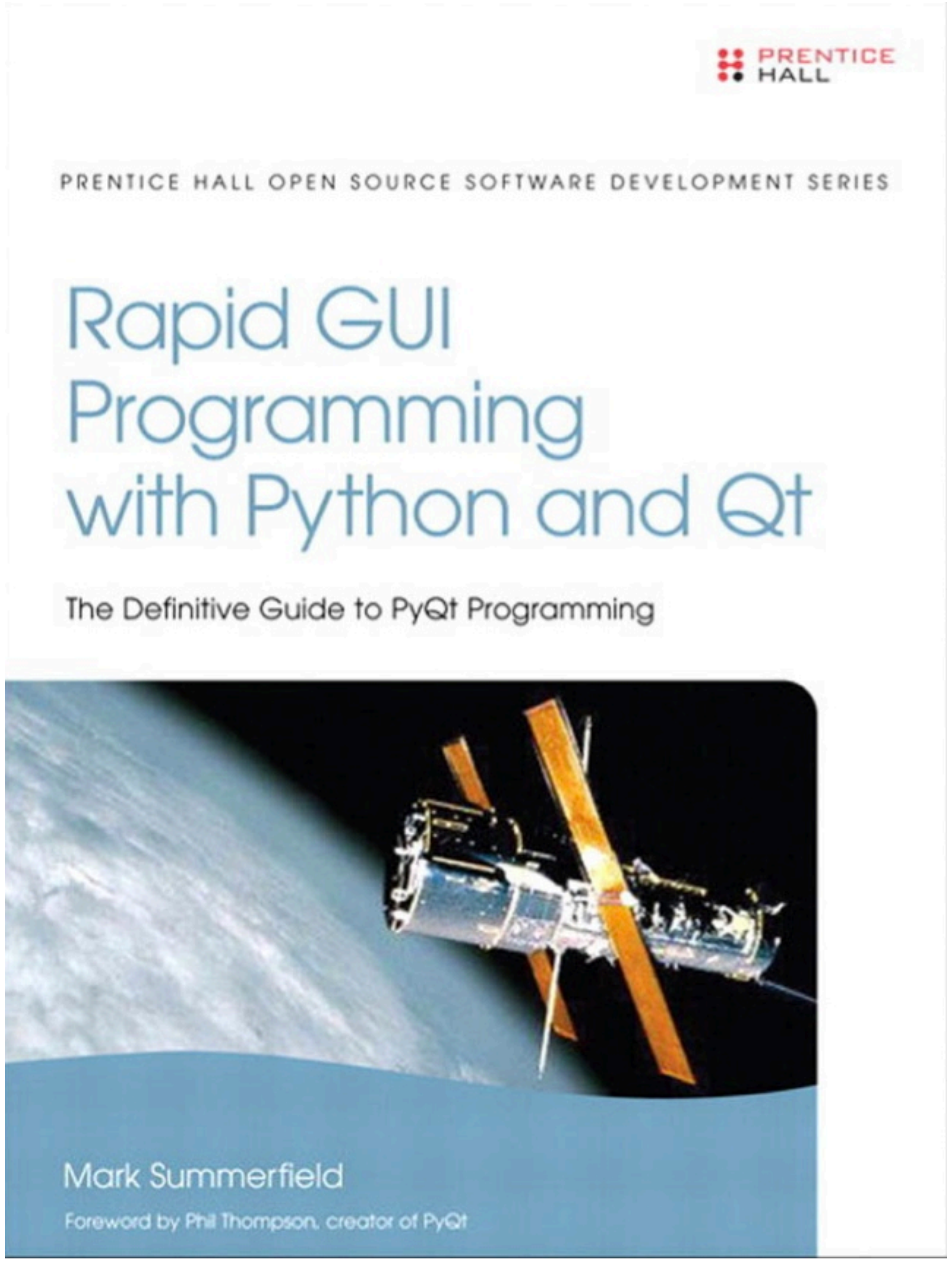
```
    def refresh_values(self):
        self.x = np.r_[0:self.max:0.01]
        self.y = np.sin(self.x)

    def refresh(self):
        try:
            self.max = np.float(self.lineedit.text())
        except ValueError:
            print 'Please enter a numeric value'
        self.refresh_values()
        if self.fig is None:
            self.fig = plt.figure()
            plt.show()
        plt.clf()
        plt.subplot(111)
        plt.plot(self.x, self.y)
        plt.title('Max x value = %f' % (self.max))
        plt.draw()

app = QApplication(sys.argv)
dialog = Dialog()
dialog.show()
app.exec_()
```

GUI | Lab

- Modify this script so the interface shows two buttons
  - one for plotting cos(x)
  - one for plotting sin(x)



A few last things

Fun with functions

- Like any Python object, a function can have several names

```
def f1(x):                  f2 = f1
    return x*2              y = f2(3)
```

- They can be inserted in lists, dictionaries

```
def f1(x):
    return x*2
def f2(x)
    return x*3
```

```
flist = [f1, f2]
y = [f(3) for f in flist]
```

```
fdict = {'day':f1, 'night':f2}
period = 'night'
y = fdict[period](3)
```

Dictionaries & json

- Dictionaries are great to store parameters

```
def process_data(data, params={'min':0, 'max':100}):
      idx = (data>params['min']) & (data<params['max'])
      valid_data = data[idx]
      processed_data = valid_data * 2.
      return processed_data

process_data(data)
params={'min':3., 'max':10.}
process_data(data,params=params)
```

- dictionaries can be easily stored and read in ASCII files using the json module

```
import json
params = {'xmin':3., 'xmax':10.}
f = open('file.txt', 'w')
json.dump(params, f)
```

→

```
file.txt
{"max": 3.0, "min": 0.0}
```

↓

```
params=json.load(open('file.txt'))
```

The datetime module

```
from datetime import date, datetime,
     timedelta
```

```
day1 = date(2006, 12, 1)
day2 = date(2007, 3, 23)
delta = day2 - day1  # timedelta object
print delta.days
```

```
# dec 1st 2006, 5:15 pm
day1 = datetime(2006, 12, 1, 17, 15)
delta = timedelta(days=3, seconds=3700)
day2 = day1 + delta

print day2
print day2.year, day2.month, day2.day
print day2.isoformat()
print day2.toordinal()
```

numpy arrays can contain datetime objects

```
arr = np.array([day1, day2])
```

matplotlib plot them as dates

matplotlib has lots of date formatting functions

```
matplotlib.dates
```

import gotchas

```
# script1.py
def times2(x):
   return x*2
# stuff
print 'bla'
```

```
# script2.py

import script1
x = script1.times2(3)
print x
```

```
> run script2.py
bla
6
```

commands from script1 are executed

not really what we wanted

the solution

```
# script1.py
def times2(x):
   return x*2
if __name__=='__main__':
   print 'bla'
```

double-underscored object are supposed to be 'private'

merely a convention, not enforced by the language

please follow the convention

why this is nice

```python
# day.py
def process_day(day):
    # ...
    return result
if __name__=='__main__':
    day = 3
    result = process_day(day)
    print result
```

```python
# month.py
from day import process_day
days = np.r_[1:32]
for day in days:
    res = process_day(day)
```

thanks