Python

Robert Lupton

30 May 2011

Interpreted Languages

- Fast development (no compile-link-run cycle)
- Interactive development
- High level (no need to worry about pointers)

Python

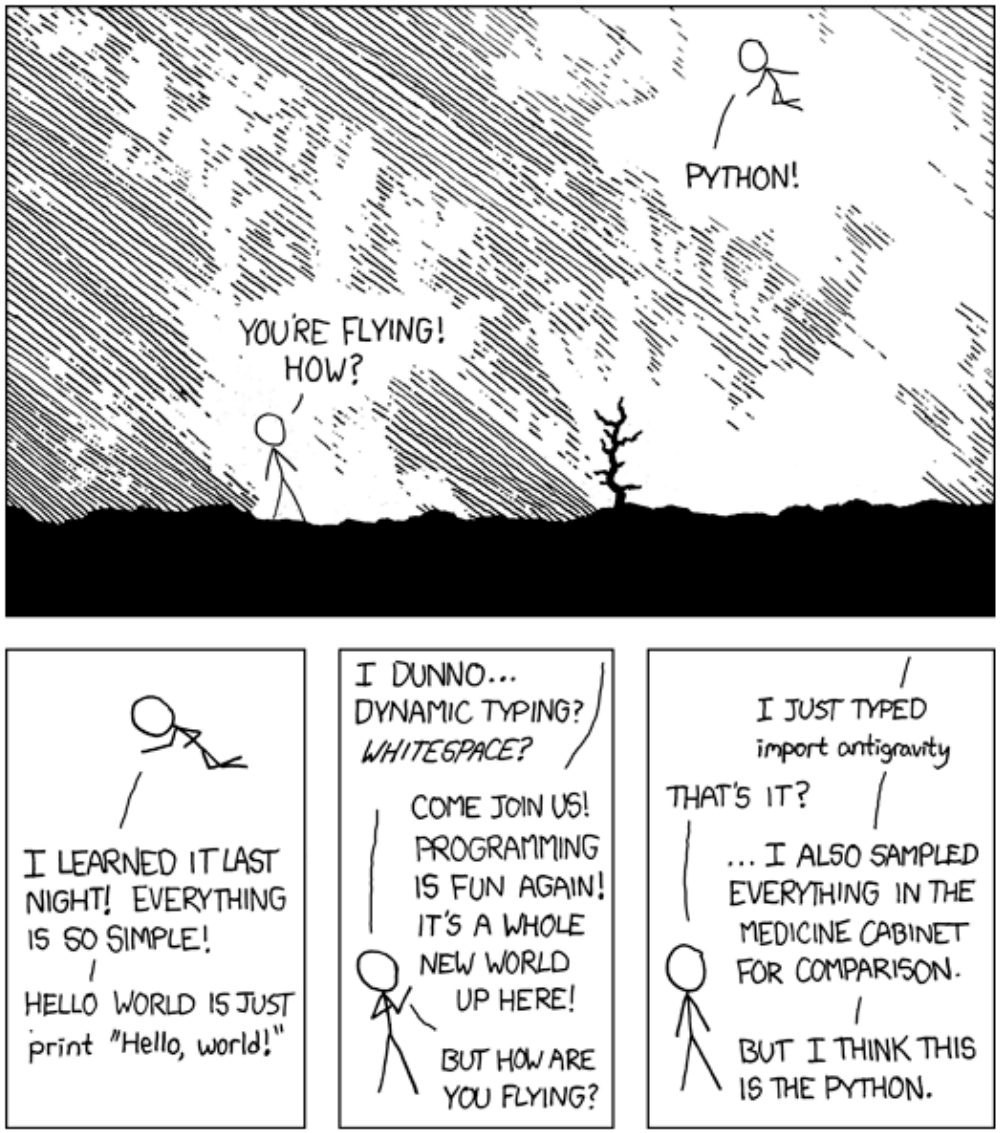
- Powerful builtins
- Object oriented
- Rich libraries
- dynamic typing

Official Tutorial and Manual

http://docs.python.org/tutorial/index.html

There are two slightly inconsistent versions of python in the wild, python `2.x` and python `3.x`
Within the `2.x` series (currently `2.7`) features were added from time to time. If you're concerned about portability you may want to avoid newer constructions (e.g. `X if LOGICAL else Y`, `with`) Eventually we'll all have to move to python 3 (currently at `3.2`), but I'm not in a hurry.

XKCD


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Hello World

Let us write “Hello world” in python:

```
print "Hello world"
```

You can run python scripts from the shell:

```
$ cat hello.py
#!/usr/bin/env python
print "Hello world"
$ ./hello.py
Hello world
```

(That `#!` line is standard unix magic for, “use python to run this script”)

Or interactively:

```
$ python
>>> print "Hello world"
Hello world
```

Interactive Usage

These days we are all spoilt by the unix shells. We expect:

- To be able to use $\uparrow\downarrow\leftarrow\rightarrow$ to save typing
- To be able to use `TAB` to complete command and file names
- That our history be saved between sessions

This is all available in python. Two solutions:

- Use `ipython` (http://ipython.scipy.org/moin/)
- Put cunning and cryptic commands in your python startup file (`$PYTHONSTARTUP`)

Primitive types

- `None`
- `bool` (`True`, `False`)
- `int`
- `long` (arbitrary precision)
- `float`

Lists and Tuples

Python supports two separate-but-almost-equal list types:

- **list**

```
>>> li = [100, 101, 102, 103]
>>> li[0]
100
>>> x = li[1:3]
>>> x
[101, 102]                                  # not [100, 101, 102]

>>> li[-1] = 666
>>> li
[100, 101, 102, 666]
```

- **tuple**

```
>>> tp = (100, 101, 102, 103)
>>> tp[0]
100
>>> x = tp[1:3]
>>> x
(101, 102)

>>> tp[-2] = 666
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- There is also **set**
  A sorted list with each element appearing only once.

Strings

Python strings can be delimited with ", ', """, or '''

```
>>> s = "Hello world"
>>> s2 = 'Goodbye, sweet life'
>>> s3 = """I really like
to split greetings over multiple lines"""
```

I recommend **not** randomly switching between " and ' strings (as it makes it hard to find them in your editor). I personally follow the C convention: `"Hello world"` but `'H'`. Strings have several useful methods:

```
>>> print s.upper()
HELLO WORLD

>>> s.find('w')
6

>>> print s[s.find('w'):]
world

>>> s.split()
['Hello', 'world']
```

You can't interpolate variable (`"$a $b $c"`), but you can say

```
>>> a, b, c = "A", "B", "C"
>>> print "%s %s %s" % (a, b, c)
A B C
```

Dictionaries

```
>>> di = {"lsb": "Luis", "suzanne.aigrain": "Suzanne", "rhl": "Robert"}

>>> print di['rhl']
Robert
>>> print di.keys(), di.values()
['lsb', 'rhl', 'suzanne.aigrain'] ['Luis', 'Robert', 'Jim']

>>> di = dict(president = "Obama")

>>> di["prime minister"] = "Berlusconi"
```

N.b. python supports *garbage collection*; when we said `di = dict(president = "Obama")` the memory for our email dictionary was returned to the system.

Loading source files

If you have a file **foo.py**, you can make it visible from python with `import foo`. If you modify **foo.py** and repeat the import, nothing happens. To see your changes, you have to say `reload(foo)`
Python searches for **foo.py** by searching the directories in `$PYTHONPATH` (a `:` separated list) in order.
When you first `import` a file it's compiled to a *.pyc* file (**foo.pyc**). You'll probably want to tell your source code manager (e.g. `hg` or `svn`) to ignore *.pyc* files, *e.g.* by adding `*.pyc` to your **.hgignore** file.
"Orphan" *.pyc* files can be very confusing. If you move **foo.py** to a directory later in `$PYTHONPATH`, but leave **foo.pyc** behind, python will happily import the *.pyc* file for you; this may not be what you intended.

Control structures

Python has the standard control structures: **if-elif-else**, **for**, **while** and logicals **and**, **or**, **not** ==, <, . . .

```
if x == 1:
    print "One"
elif x == 2 or x == 3:
    print "Two or Three"
else:
    print "Something else"
```

The block structure is *defined* by whitespace. This seems weird, but you soon get used to it. I believe that it was a very bad design decision, but it's not going to change. Because there isn't any information about a program's block structure except the white space, you have to be very careful.

Another issue is mixing tabs and spaces; it's probably better to instruct your editor to insert spaces even when you hit the tab key to avoid the problem.

Changing program logic

In C I can write

```
if (x == 0) {
    printf("One\n");
} else {
    printf("Not one\n");
}
```

If I need to change the indentation level I can modify this to

```
if (y == 10) {
if (x == 0) {
    printf("One\n");
} else {
    printf("Not one\n");
}
}
```

and get my editor to reindent to make it look pretty.
In python, things aren't so nice.

```
if y == 10:
if x == 1:
     print "One"
else:
     print "Not one"
```

I cannot tell whether the `else` belongs to the `x` or `y` test. My only hope is to rigidly reindent the block (use ^C> in emacs)

for and while loops

```
for r in ("Arrow", "Birdland", "Matinee"):
    print r

n = 10
for i in range(n):
    for j in range(i, n):
        print i, j
```

(note that **range(n)** counts from 0 to **n-1**, not up to **n**).

```
i = 0
while True:
    i += 10
    if i == 100:
        break
    print i
```

continue is also available. But **goto** isn't.

Functions

```
def myRange(n):
    """Return (0...n)"""
    i, out = 0, []
    while i < n:
        out.append(i)
        i += 1

    return out

for i in myRange(10):
    print i
```

Simple variables (`int`, `float`) are passed by *value*; everything else is passed by *reference*.

This mans that if you modify a list or dictionary passed to a function it'll be modified in the calling routine too; you may need to make a copy:

```
li = li[:]
di = di.copy()
```

It'd be nice if `list` also supported `copy`; you can always use `import copy; copy.copy(XXX)`

Default arguments

You can also specify default values for arguments (as well as variable numbers of arguments):

```
def myRange(n, end=None, dn=1):
    """Return a list of integers
Details ...
"""
    if end == None:
        i, end = 0, n
    else:
        i = n

    out = []
    while i < end:
        out.append(i)
        i += dn

    return out

>>> myRange(3)
(0, 1, 2)
>>> myRange(2, 4)
(2, 3)
>>> myRange(2, 10, 2)
(2, 4, 6, 8)
>>> myRange(10, dn=2)
(0, 2, 4, 6, 8)
```

Exceptions

Don't do this at home:

```
>>> myRange(0, 10, -2)
```

the program will appear to hang until you hit `^C` (or run out of memory — I should have used `yield`)

```
>>> ^C^C
>>> import pdb;  pdb.pm()
0
0
> <stdin>(13)myRange()
(Pdb) p i
-5184308
(Pdb)
```

We're counting down to $-\infty$

```
def myRange(n, end=None, dn=1):
    ...
    if dn <= 0:
        raise RuntimeError("Increment is negative: %g" % (dn))
```

Catching exceptions

An exception need not be fatal:

```
try:
    myRange(0, 10, -2)
except RuntimeError, e:
    print "Caught exception:", e
```

There are also more complicated and powerful forms of this `try except` pattern.

Classes

Python is an Object Orientated language. In **people.py** I wrote:

```
class Person(object):
    """Describe a person"""

    def __init__(self, email=None, surname=None):
        self.email = email
        self.surname = surname
```

Note that **self** plays the part of C++'s **this**, but you have to explicitly write it out. All member functions expect **self** as their first argument. Let's use our new **class**

```
>>> import people
>>> addressBook = {}
>>> addressBook["Luis"] = people.Person("lsb", "Barro")
>>> addressBook["Robert"] = people.Person(surname="Lupton")
>>> print addressBook["Luis"].email
lsb
```

Dynamic typing

Let's return to another old friend, `max`[1]

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

That's it.

```
>>> print max(1, 2)
2
>>> print max("a", "b")
'b'
>>> print max(["a", "b"], ["a", "c"])
['a', 'c']

>>> import people
>>> Luis = people.Person("lsb", "Barro")
>>> Robert = people.Person("rhl", "Lupton")
>>> print max(Luis, Robert)
(lsb, Barro)
```

The comparison is consistent-but-undefined. If we want to sort by the email address:

```
def __cmp__(self, rhs):
    return cmp(self.email, rhs.email)
```

and now `max` works as expected.

[1] actually, `max` is a builtin, but builtin names are not protected

Libraries

The Official Library

http://docs.python.org/library/index.html

Python has *many* libraries. I'll skim the surface of two:

- `matplotlib`
  Plotting
- `numpy`
  Array operations

Enthought Scientific Python

http://www.enthought.com/products/epd.php

Plotting, `matplotlib`

There are a number of plotting packages available for python; I'll concentrate on **`matplotlib`**.

The package is available from *Enthought* or

http://matplotlib.sourceforge.net/index.html

Defaults are set in **$HOME/.matplotlibrc**, *e.g.*

```
backend      : TkAgg
```

Using **`TkAgg`** (which is probably a good idea) requires that your version of python was built with **`tkinter`** support. **`matplotlib`** can use other backends (*e.g.* **`WXAgg`**) if you have the proper package installed (*e.g.* **`wxPython`**)

Plotting using `matplotlib`

There are two ways to use `matplotlib`

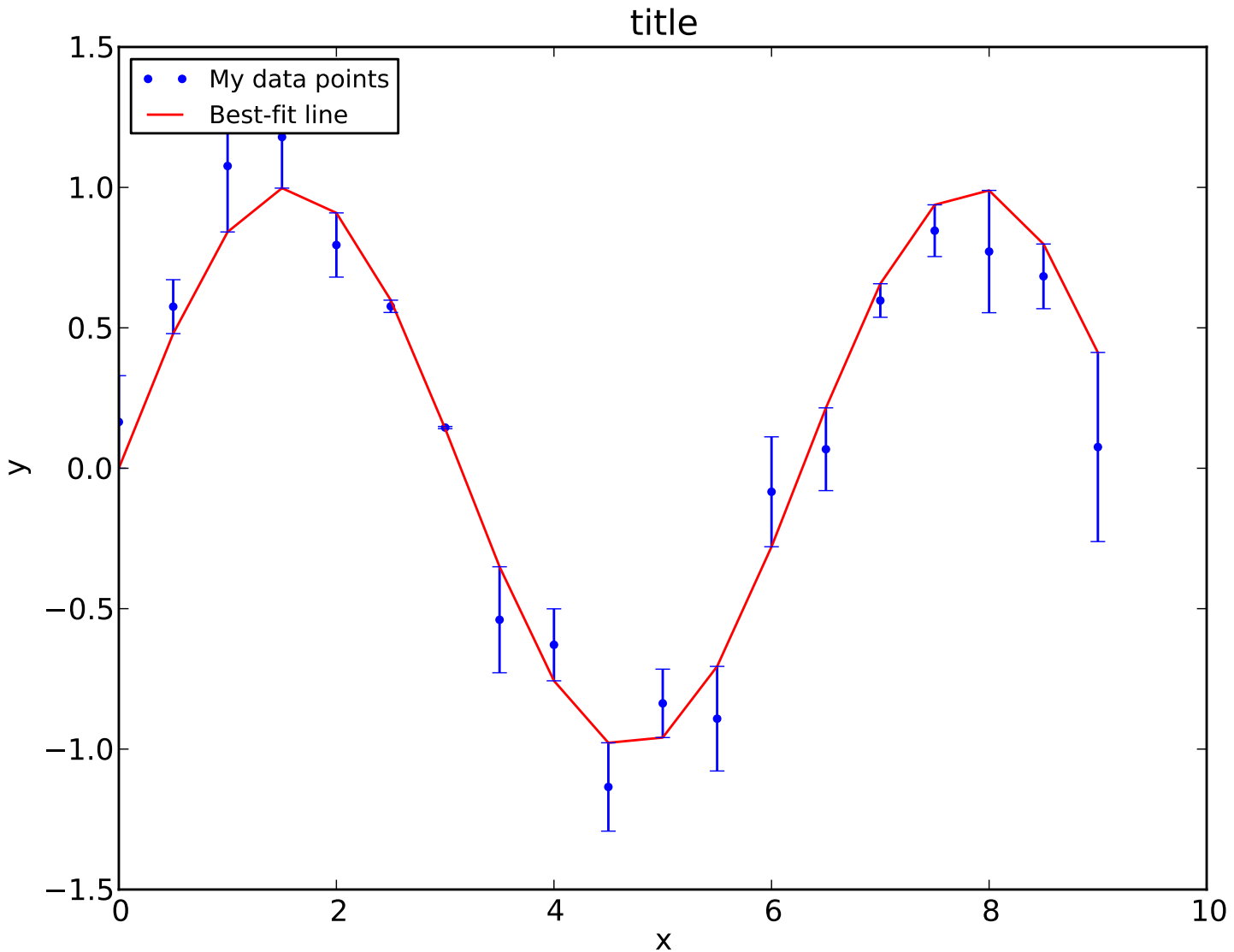
- Interactive:
  - uses `matplotlib.pyplot` package
  - good for quickly making single plots, hiding all the object-oriented aspects.
  - supposedly looks very similar to matlab
- Object-oriented (more *pythonic*):
  - Renderers which provide an abstract interface to drawing primitives (*e.g.* `draw_path`)
  - Backend objects which take care of how to actually draw the object (*e.g.* `TkAgg` to use `Tk`)
  - A `FigureCanvas` to draw on
  - An `Artist` that knows how to use *renderers* to draw on *canvases*.

If you need fine control over your plots you need to know the classes and their methods

Interactive plotting with matplotlib

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

# make data
x = numpy.linspace(0.0, 9.0, 19)
model = numpy.sin(x)
y = numpy.random.normal(loc=model, scale=0.2)
z = x**2
yerr = numpy.abs(y - model)

# plot the data
plt.plot(x, y, "b.", label="My data points")
plt.plot(x, model, "r-", label="Best-fit line")
plt.errorbar(x, y, xerr=None, yerr=yerr, fmt=None, color='b')

# Labels
plt.xlabel("x")
plt.ylabel("y")
plt.title("title")

# add a legend using the labels you gave to plot()
fontProps = dict(size = "small")
plt.legend(loc="upper left", prop=fontProps, ncol=1)

# Show the figure (should pop up a new window)
plt.show()
# Save the plot to a file
plt.savefig("figures/plot_sin.pdf", format="pdf")

# Clear the figure (so we can make a new one)
plt.clf()
```

plot_sin.pdf


title
My data points
Best-fit line
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0
2
4
6
8
10
x

Format characters

The format string is of the form `CM` (ColourMarker)

`b`	blue	`-`	solid line	`.`	point
`g`	green	`--`	dashed line	`,`	pixel
`r`	red	`:`	dotted line	`o`	circle
`c`	cyan	`-.`	dot-dash line	`v`	triangle_down
`m`	magenta			`^`	triangle_up
`y`	yellow			`<`	triangle_left
`k`	black			`>`	triangle_right
`w`	white				

There are more colours, but it's better to use the **`color`** keyword. For markers, it's really better to use the **`marker`** and **`linestyle`** keywords

OO plotting with matplotlib

The matplotlib command to select the third sub-window out of a 2x2 set is

```
figure.add_subplot(2, 2, 3)
```

so I could say

```
figure.add_subplot(2, 2, 1)
# make a plot
figure.add_subplot(2, 2, 2)
# make another plot
figure.add_subplot(2, 2, 3)
# keep plotting
figure.add_subplot(2, 2, 4)
# plot plot plot
```

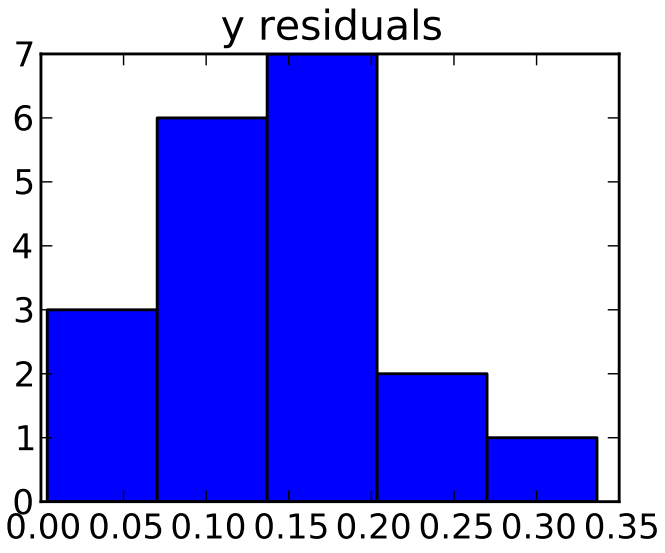
But I'm lazy and I don't like duplicating 2, 2

Instead, I'll use a generator

```
def makeSubplots(figure, nx=2, ny=2):
    """Return a generator of a set of subplots"""
    for window in range(nx*ny):
        yield figure.add_subplot(nx, ny, window + 1) # 1-indexed

subplots = makeSubplots(fig)
# Initialize
axes = subplots.next()
```

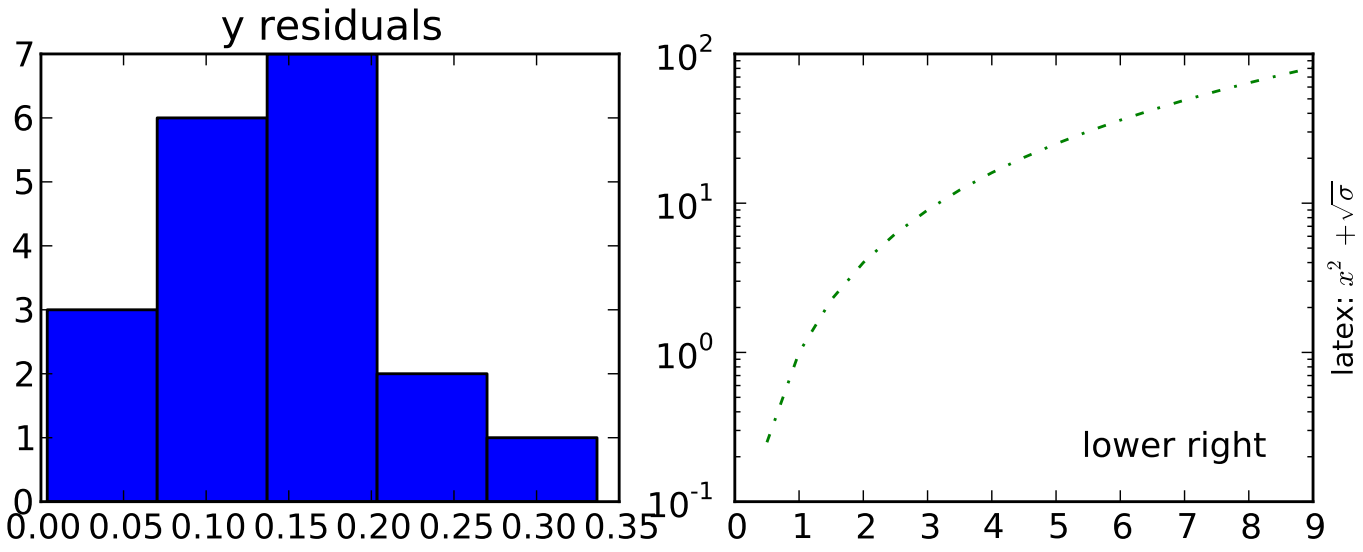
Panel I: Histogram

```
#make the figure (Artist object) that will draw the plot
fig = matplotlib.figure.Figure()

#make the canvas where the figure will be drawn
from matplotlib.backends.backend_pdf import FigureCanvasPdf as FigCanvas
canvas = FigCanvas(fig)

def makeSubplots(figure, nx=2, ny=2):
    """Return a generator of a set of subplots"""
    for window in range(nx*ny):
        yield figure.add_subplot(nx, ny, window + 1) # 1-indexed

subplots = makeSubplots(fig)
# Initialize
axes = subplots.next()

#make a histogram of residuals, returns bin delimiters and number/bin
myhist = axes.hist(yerr, bins=5)
axes.set_title("y residuals")
```


y residuals
7
6
5
4
3
2
1
0
0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35

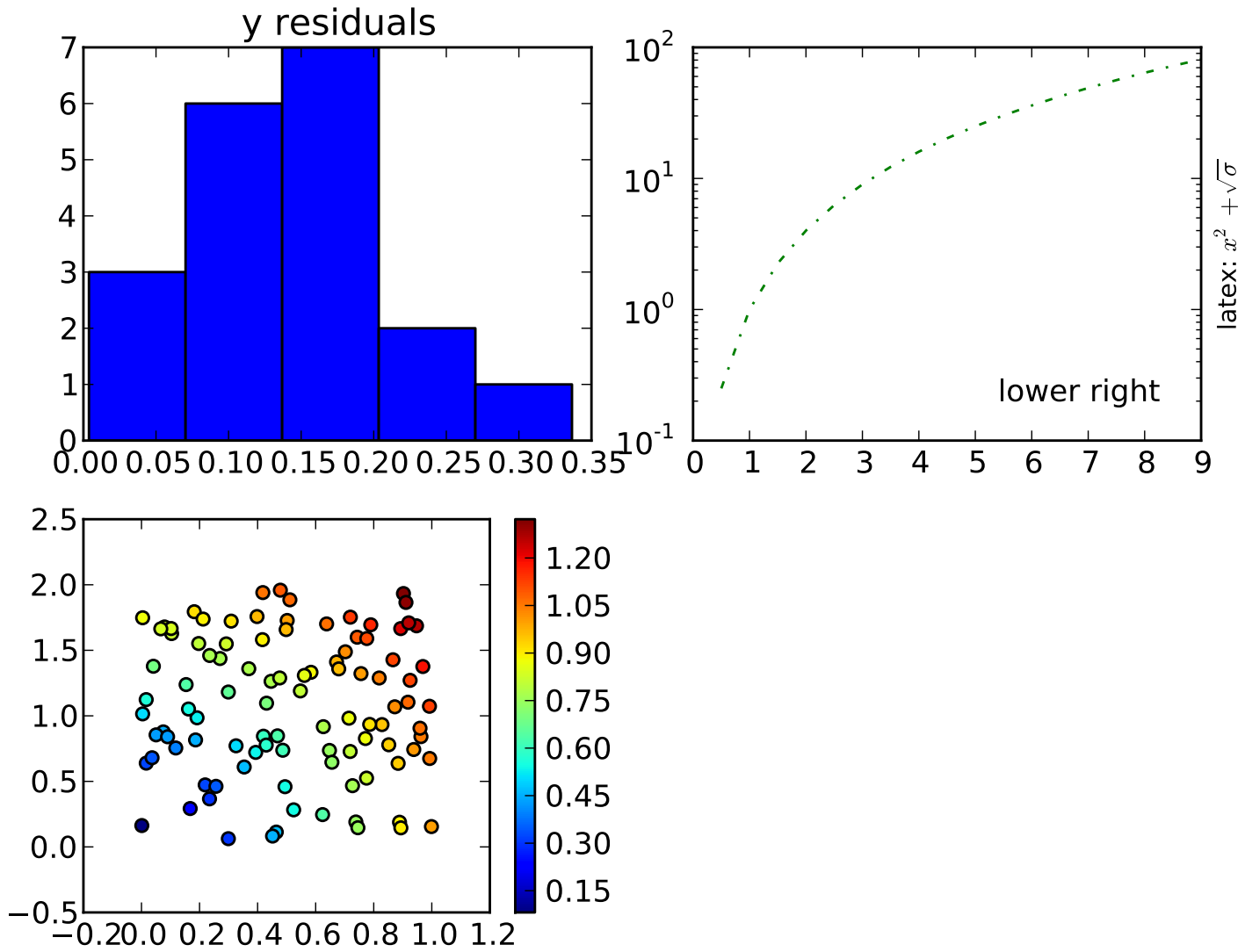
Panel II: Log-linear

```
# Initialize and make a log plot
axes = subplots.next()
axes.semilogy(x, z, "g-.")

# Move the axis label to the right hand size
axes.yaxis.set_label_position("right")
axes.set_ylabel(r"latex: $x^2+\sqrt{\sigma}$", size="small")

# can work in pixel, figure, or axes or plotting coordinates
# in this case put the text in 60%, 10% of the axes
axes.text(0.6, 0.1, "lower right", transform=axes.transAxes)
```


y residuals
0
1
2
3
4
5
6
7
0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35
10^{2}
10^{1}
10^{0}
10^{-1}
0 1 2 3 4 5 6 7 8 9
lower right
latex: $x^2 + \sqrt{\sigma}$

Panel III: Scatter Plot

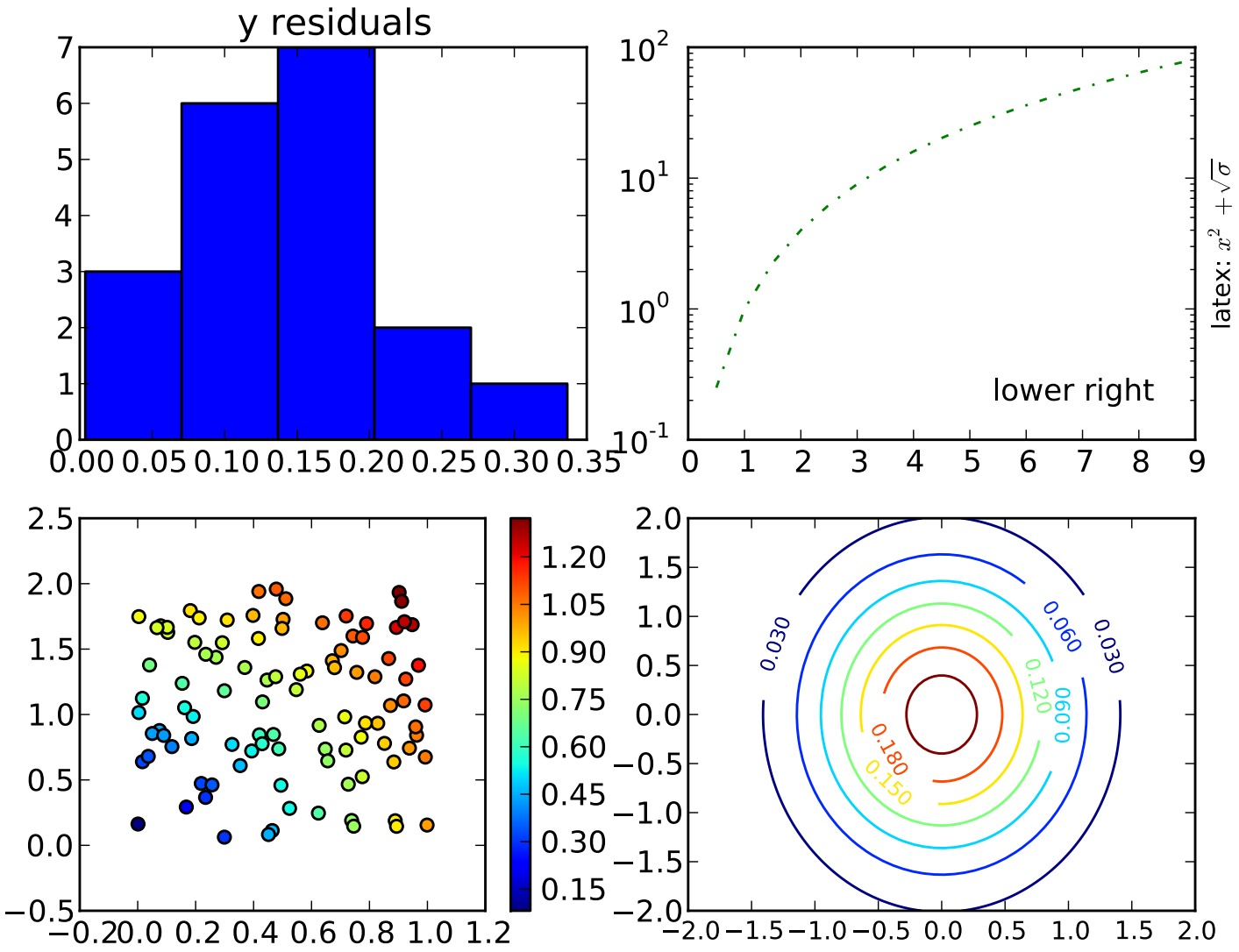
```
# Initialize and calculate points
axes = subplots.next()
xs = numpy.random.random(100)
ys = numpy.random.random(100)*2
zs = numpy.sqrt(xs**2 + ys**2/4.0)

# Make plot
sc = axes.scatter(xs, ys, c=zs)
fig.colorbar(sc)
```


y residuals
0 1 2 3 4 5 6 7
0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35
10^{2}
10^{1}
10^{0}
10^{-1}
0 1 2 3 4 5 6 7 8 9
lower right
latex: $x^2 + \sqrt{\sigma}$
2.5 2.0 1.5 1.0 0.5 0.0 −0.5
−0.2 0.0 0.2 0.4 0.6 0.8 1.0 1.2
1.20 1.05 0.90 0.75 0.60 0.45 0.30 0.15

Panel IV: Contours

```
# mlab has lots of matlab-like functions; we'll just fake some data
from matplotlib.mlab import bivariate_normal

# Initialize and calculate data
axes = subplots.next()
axis = numpy.linspace(-2.0, 2.0, 100)
X, Y = numpy.meshgrid(axis, axis)
Z = bivariate_normal(X, Y, 0.7, 1.0, 0.0, 0.0) # from matplotlib.mlab

# Make a contour plot
CS = axes.contour(X,Y,Z)
#put labels on the contous
axes.clabel(CS, inline=1, fontsize=8)

# Change the ticklabel size
try:
    axes.tick_params(axis="x", labelsize="small") # new in 1.0
except AttributeError:
    for l in axes.xaxis.get_ticklabels() + axes.yaxis.get_ticklabels():
        l.set_size("x-small")

# Save the plot to a file
fig.savefig("figures/plot_multi.pdf")
```

plot_multi.pdf



Array operations, `numpy`

While the array library, **`numpy`**, is not part of the python standard library it is widely available.

NumPy home (or get it from *Enthought*)

http://numpy.scipy.org

We used a few pieces of **`numpy`** in the **`matplotlib`** examples:

```
import numpy

x = numpy.linspace(0.0, 9.0, 19)
model = numpy.sin(x)

yerr = numpy.abs(y - model)
zs = numpy.sqrt(xs**2 + ys**2/4.0)

numpy.random.seed(666)
xs = numpy.random.random(100)
y = numpy.random.normal(loc=model, scale=0.2)

axis = numpy.linspace(-2.0, 2.0, 100)
X, Y = numpy.meshgrid(axis, axis)
```

numpy Arrays

```
>>> x = numpy.linspace(0.0, 5.0, 11); print x
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5. ]
```

We could have used **arange** (analogous to python's **range**):

```
>>> print numpy.arange(0.0, 5.1, 0.5)
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5. ]
```

There's also

```
>>> print numpy.zeros(4), numpy.ones(4), numpy.empty(4, dtype='i')
[0.  0.  0.  0.] [1.  1.  1.  1.] [9 0 18402543 1]

>>> x = numpy.arange(5); print numpy.multiply.outer(x, x)
[[ 0  0  0  0  0]
 [ 0  1  2  3  4]
 [ 0  2  4  6  8]
 [ 0  3  6  9 12]
 [ 0  4  8 12 16]]
```

numpy Mathematical functions

```
>>> x = numpy.arange(5)
>>> y = numpy.sin(x); print y
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025 ]
```

There are lots of other mathematical builtins (**sin**, **cos**, **tan**, **arcsin**, **arctan2**, **abs**, **sqrt**, . . .)

```
>>> print zip(x, y)
[(0, 0.0), (1, 0.8414709848078965), (2, 0.9092974268256817),
 (3, 0.1411200080598672), (4, -0.7568024953079282)]

>>> print "\n".join(["%g %6.3f" % (t, s) for t, s in zip(x, y)])
0  0.000
1  0.841
2  0.909
3  0.141
4 -0.757
```

(OK, so that's a python, not **numpy**, trick)

numpy Random Numbers

```
>>> numpy.random.seed(666)
>>> numpy.random.random(10)
array([ 0.70043712,  0.84418664,  0.67651434,  0.72785806,  0.95145796,
        0.0127032 ,  0.4135877 ,  0.04881279,  0.09992856,  0.50806631])
```

(*n.b.* I didn't say **print**, so I got the **repr** not the **str** value of the result)

```
>>> print numpy.random.normal(loc=numpy.arange(5), scale=0.2)
[-0.2177586   0.88484585  1.66341985  3.04583705  3.64867496]
>>> print numpy.random.normal(numpy.arange(5), 0.2)
[ 0.16892652  1.05544397  2.17058031  3.03891992  4.26212754]
```

The two calls are identical, but the random numbers are (of course) different.

numpy in n-D

```
>>> axis = numpy.linspace(-2.0, 2.0, 5)
>>> X, Y = numpy.meshgrid(axis, axis)
>>> print X
[[-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]
 [-2. -1.  0.  1.  2.]]
>>> print Y
[[-2. -2. -2. -2. -2.]
 [-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]
 [ 2.  2.  2.  2.  2.]]

>>> print numpy.cos(X)*numpy.sin(Y)
[[ 0.37840125 -0.4912955  -0.90929743 -0.4912955   0.37840125]
 [ 0.35017549 -0.45464871 -0.84147098 -0.45464871  0.35017549]
 [-0.          0.          0.          0.         -0.        ]
 [-0.35017549  0.45464871  0.84147098  0.45464871 -0.35017549]
 [-0.37840125  0.4912955   0.90929743  0.4912955  -0.37840125]]

>>> print numpy.fft.fft(X)*numpy.sin(Y)
[[-0.00000000+0.j          2.27324357-3.12885135j  2.27324357-0.73862161
   2.27324357+0.73862161j  2.27324357+3.12885135j]
 [-0.00000000+0.j          2.10367746-2.89546363j  2.10367746-0.68352624
   2.10367746+0.68352624j  2.10367746+2.89546363j]
 [ 0.00000000+0.j         -0.00000000+0.j         -0.00000000+0.j
   0.00000000-0.j          0.00000000-0.j        ]
...
```

numpy extended indexing

You aren't restricted to using scalars as array indexes:

```
>>> x = numpy.arange(-4, 5); print x
[-4 -3 -2 -1  0  1  2  3  4]
>>> i = x**2 > 4
>>> print i
[ True  True False False False False False  True  True]
>>> print x[i]
[-4 -3  3  4]

>>> x[i] = 10 + numpy.abs(x[i])
>>> print x
[14 13 -2 -1  0  1  2 13 14]

>>> I = numpy.array([2, 7])
>>> print x[I]
[-2 13]
```

numpy Linear Algebra

```
>>> n = 3; i = numpy.arange(n); M = numpy.zeros(n*n); M.resize(n, n)
>>> M[(i,i)] = i + 1; print M
[[ 1.  0.  0.]
 [ 0.  2.  0.]
 [ 0.  0.  3.]]
>>> numpy.linalg.inv(M)
array([[ 1.        ,  0.        ,  0.        ],
       [ 0.        ,  0.5       ,  0.        ],
       [ 0.        ,  0.        ,  0.33333333]])

>>> M = numpy.matrix(M)
>>> U, s, V = numpy.linalg.svd(M)
>>> U*numpy.diag(s)*V                    # should == M
matrix([[ 1.,  0.,  0.],
        [ 0.,  2.,  0.],
        [ 0.,  0.,  3.]])
```

Traps await the unwary:

```
>>> M = numpy.zeros(n*n); M.resize(n, n); M[(i,i)] = i + 1
>>> U, s, V = numpy.linalg.svd(M)
>>> U*numpy.diag(s)*V
array([[ 0.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  0.]])
```

Uh oh; that's an element-by-element product. An `array` is not a `matrix`; you have to say

```
>>> numpy.dot(U, numpy.dot(numpy.diag(s), V))
```

Other `numpy` capabilities

`numpy` has lots of libraries:

- FFTs
- Linear algebra
- Statistics
- *etc.*

I used the statistics package in analyzing the course questionnaire:

```
cov = numpy.cov(data, rowvar=False)
for i in range(len(cov[0])):
    print "%6.3f" numpy.mean(data[:, i]), \
          " ".join(["%6.3f" % x for x in cov[i]])
```

The `scipy` package adds many more:

- N-dimensional image convolution
- Interpolation
- Sparse linear algebra (*e.g.* `3M x 5k` least-squares problems)
- Optimization
- *etc.*

Embedding C/C++/Fortran in python

One extremely powerful technique is to wrap your own code in python, a topic that we'll cover later in the course. To whet your appetite, here's some analysis code that I wrote last week:

mosaic.py

```
smoothingKernel = AnalyticKernel(ksize, ksize,
                                 GaussianFunction2D(alpha, alpha))

for f in filters:
    imgList = vectorMaskedImageF()

    for run, camCol, (field0, field1) in inputs:
        camColImgList = vectorMaskedImageF()

        fields = []
        for field in range(field0, field1 + 1):
            exposure = getExposure(run, camCol, field, f)

            if subtractBackground:
                bkgd = makeBackground(mim, BackgroundControl(nx, ny))

                im = exposure.getMaskedImage().getImage()
                im -= bkgd.getImageF()
                del im

            cmimg = maskedImageFactory(exposure.width(),exposure.height())
            convolve(cmimg, exposure.getMaskedImage(), smoothingKernel)
            exposure.setMaskedImage(cmimg)

            img = maskedImageFactory(exposure.getDimensions())
            warpedExposure = makeExposure(img, wcs0)
            warpExposure(warpedExposure, exposure, warpingKernel)
```

Every operation in red is written in C++.