Python

Clancy Rowley and Robert Lupton

7 October 2014

Outline

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

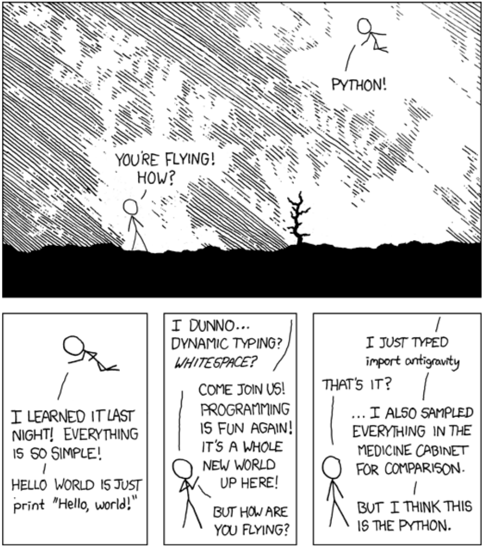
Official Tutorial and Manual

`https://docs.python.org/2/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 `2.7.8`) features were added from time to time. If you're very 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 `3.4`), but I'm not in a hurry. Neither is google.

XKCD


YOU'RE FLYING! HOW?
PYTHON!
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"; the ./ is in case your current directory, ., isn't in your $PATH)

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 ↑↓←→ 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:

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

These days we definitely recommend `ipython`. You can install it from `https://store.continuum.io/cshop/anaconda` along with lots of other useful-to-essential packages, some of which we'll discuss today.

On `nobel.princeton.edu` you can use `python 2.7` and `ipython 2.2.0` by saying `module load anaconda` (maybe in your **.bashrc** file). If you find other Princeton machines where this doesn't work please let CSES know (and notify us).

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]
```

Useful list methods: append, insert, pop, reverse, sort, index

- tuple: a list that is "frozen" and cannot be changed (immutable)

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

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

- set: a 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"""
```

(there's no difference between " and ', unlike the unix shells). 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'. Sometimes.

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 = {"cwrowley" : "Clancy", "jmstone" : "Jim", "rhl" : "Robert"}

>>> print di['rhl']
Robert
>>> print di.keys(), di.values()
['cwrowley', 'rhl', 'jmstone'] ['Clancy', 'Robert', 'Jim']

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

>>> di["president"] = "Eisgruber"
>>> di["provost"] = "Lee"
```

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

You can use dictionaries in conjunction with `%` formatting:

```
>>> foods = dict(a="Apple", b="Banana", c="Carrot")
>>> print "%(a)s %(b)s %(c)s" % foods
Apple Banana Carrot
```

This style `%` formatting is actually deprecated in python `>= 2.6`; you're supposed to say things like

```
"{0} {1} {c}".format("Apple", "Banana", c="Carrot")
```

but this seems pretty clunky to me. It seems unlikely that `%` formatting will ever go away.

Mix and Match

There is no restriction that the elements of any of these data types be simple.

```
>>> addressBook = {}
>>> addressBook["Clancy"] = ["cwrowley", "Rowley"]
>>> addressBook["Robert"] = ["rhl", "Lupton"]
>>> print addressBook["Clancy"][0]
cwrowley
```

```
>>> addressBook = {}
>>> addressBook["Clancy"] = dict(email = "cwrowley", surname = "Rowley")
>>> addressBook["Robert"] = {}
>>> addressBook["Robert"]["email"] = "rhl"
>>> addressBook["Robert"]["surname"] = "Lupton"
>>> print addressBook["Robert"]["email"]
rhl
```

You can't use a list as a key in a dict (as you might modify the list later), but you *can* use a tuple as it's immutable.

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. git) to ignore *.pyc* files, *e.g.* by adding `*.pyc` to your **.gitignore** 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. Examining `foo.__file__` can help diagnose the problem.

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 == 1) {
   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 == 1) {
   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.

Warning: Looping over strings can do the wrong thing

```
>>> for c in ("abc",): print c
abc
```

```
>>> for c in ("abc"): print c
a
b
c
```

That comma is essential. It isn't really the loop's fault, it's just that a string is treated as a list of characters.

Functions

```
def my_range(n):
   """Return (0...n)"""
   i = 0
   out = list()
   while i < n:
      out.append(i)
      i += 1
   return out

for i in my_range(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, the closest is list(xxx). You can always use import copy; copy.copy(xxx) (but it's slower). This is a *shallow copy*, but copy.deepcopy is also available.

Hello World

A better **hello.py** script is

```
$ cat hello.py
#!/usr/bin/env python
def greet(who="world")
   print "Hello %s" % (who)

if __name__ == "__main__":
   greet()
```

The advantage is that I can say either

```
$ ./hello.py
Hello world
```

or

```
>>> import hello
>>> hello.greet("class")
Hello class
```

(__name__ is "__main__" when run from the shell, and "hello" when imported).

Default arguments

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

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

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

   return out
```

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

List comprehensions

```
>>> print [10 + x for x in range(5)]
[10, 11, 12, 13, 14]
```

```
print [10 + x for x in range(5) if x%2 == 0]
[10, 12, 14]
```

This is surprisingly useful

If you write instead

```
>>> r = (10 + x for x in range(5))
```

you get a *generator* instead:

```
>>> print r
<generator object <genexpr> at 0x1005cde10>
```

```
>>> print [x for x in r]
[10, 11, 12, 13, 14]
>>> print list(r)
[10, 11, 12, 13, 14]
```

See the next slide for an explanation.

Generators and iterators

Consider

```
def fibonacci(n):
    prev = 0
    cur = 1
    for j in range(n):
        yield cur
        (cur, prev) = (cur + prev, cur)
```

```
>>> [x for x in fibonacci(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

fibonacci returns a *generator* which has a method next. The first call to next calls fibonacci and returns the value of the yield statement. When you call next again it miraculously resumes just after the yield and continues until it reaches yield again; when it returns (either explicitly or implicitly) a StopIteration exception is raised:

```
>>> f = fibonacci(1)
>>> f.next()
1
>>> f.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Exceptions

Don’t do this at home:

```
>>> my_range(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)my_range()
(Pdb) p i
-5184308
(Pdb)
```

We’re counting down to $-\infty$

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

Catching exceptions

An exception need not be fatal:

```
try:
   my_range(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 Oriented 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["Clancy"] = people.Person("cwrowley", "Rowley")
>>> addressBook["Robert"] = people.Person(surname="Lupton")
>>> print addressBook["Clancy"].email
cwrowley
```

Special Methods

Let's take a look at Clancy:

```
>>> print addressBook["Clancy"]
<people.Person object at 0x10056cd90>
```

That's annoying. The solution is to add a method __str__ to Person:

```
class Person(object):
    ...
    def __str__(self):
        return "(%s, %s)" % (self.email, self.surname)
```

After a reload and after rebuilding addressBook with our new version of Person, we get:

```
>>> print addressBook["Clancy"]
(cwrowley, Rowley)
```

Dynamic typing

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

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

That's it. *N.b.* templates provide exactly this sort of 'duck typing' for C++ (the code's valid if a and b support >)

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

>>> import people
>>> Clancy = people.Person("cwrowley", "Rowley")
>>> Robert = people.Person("rhl", "Lupton")
>>> print max(Clancy, Robert)
(cwrowley, Rowley)
```

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/2/library/index.html`

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

- re Regular expressions
- argparse Argument parsing
- matplotlib Plotting
- numpy Array operations

you can install matplotlib and numpy (and more) using anaconda; re and argparse are part of python's standard library.

Regular Expressions, re

Python supports all the standard regular expressions (^, $, ., [], (), \s, . . .)

Searching is simple

```
import re
s = "hello world"
if re.search(r"^h", s):
   print "Matches"
```

prints Matches

The object returned by re.search contains matched substrings:

```
mat = re.search(r"\s+(\S+)$", s)
if mat:
   print mat.group(1)
```

prints world

For efficiency, you can pre-compile strings:

```
>>> pat = re.compile(r"\s+(\S+)$")
>>> pat.search(s).group(1)
'world'
```

Python's re module provides two searching methods. I've been using search, but there is also match. I recommend that you **never** use match (because re.match(r"RE", ...) == re.search(r"^RE", ...)).

Command Line Parsing

One of the uses of python is to write utilities run from the command line. In C you'd use getopt, in C++ getopt or boost::program_options. In python you have (sigh) three options:

- getopt Deprecated since python 2.3
- optparse Deprecated in python 2.7
- argparse The new kid on the block; only in python `2.7` (and `3.?`), but back-ported to `2.6`.

Your best choice in new code

argparse

```
#!/usr/bin/env python
import argparse

parser = argparse.ArgumentParser(description='Say hello')

parser.add_argument('who', metavar='who', type=str, nargs='*',
                   help='List of people to greet', default=["world"])

parser.add_argument("-w", "--who",
                  dest="speaker", help="name of speaker", default="Robert")
parser.add_argument("-s", "--silent", action='store_true', default=False,
                  help="Refuse to say anything")

args = parser.parse_args()

if args.silent:
    print "I plead the fifth"
else:
    print "%s says \"Hello %s\"" % (args.speaker, " ".join(args.who))
```

```
$ hello.py
Robert says "Hello world"
$ hello.py -s TAs and the class
I plead the fifth
$ hello.py --who Clancy TAs and the class
Clancy says "Hello TAs and the class"
```

Plotting, matplotlib

While there are a number of plotting packages available for python, the most popular seems to be matplotlib; a list of other options may be found at `https://wiki.python.org/moin/NumericAndScientific/Plotting`.

We'll only discuss matplotlib here.

The package is available from anaconda or `http://matplotlib.sourceforge.net/index.html` if you want the bleeding edge version.

There's quite a nice tutorial from the EuroScipy 2012 conference at

`http://webloria.loria.fr/~rougier/teaching/matplotlib`

Using matplotlib from the python prompt

Using matplotlib from a python shell is a little tricky (see `http://matplotlib.org/users/shell.html`) for a discussion.
One way to get interactive plotting is to use `ipython -pylab` (or `ipython --pylab` in newer releases). See `http://ipython.org/ipython-doc/stable/interactive/reference.html#plotting-with-matplotlib`.
This is what we recommend.

If you insist on purity, you can use Qt4Agg by setting values in **$HOME/.matplotlib/matplotlibrc**:

```
backend      : Qt4Agg
interactive  : True
```

Using Qt4Agg requires that your version of python was built with Qt support (comes with Anaconda). matplotlib can use other backends (*e.g.* WXAgg) if you have the proper package installed (in this case wxPython)
The interactive: True means that matplotlib should *not* enter its wait-for-the-user interactive loop.

Plotting using matplotlib

There are two-and-a-half ways to use matplotlib

- Interactive:
  - uses matplotlib.pyplot package (or matplotlib.pylab)
  - good for quickly making single plots, hiding all the object-oriented aspects.
  - looks very similar to matlab
- Object-oriented (more *pythonic*), using pyplot.figure and axes
- Using the low-level objects directly:
  - 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.* Qt4Agg to use Qt)
  - A FigureCanvas to draw on
  - An Artist that knows how to use *renderers* to draw on *canvases*.

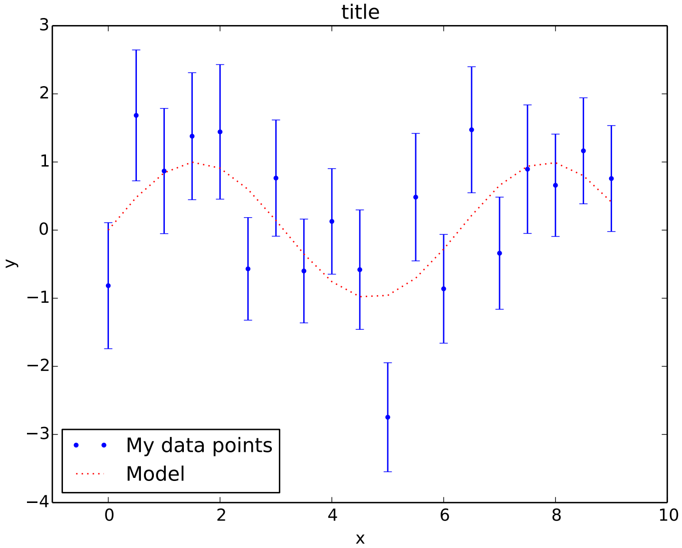
If you need fine control over your plots you need to know the classes and their methods, but you may not need to go far down that path.

Interactive plotting with matplotlib.pylab

For interactive use, probably the most convenient is to use pylab, which looks a lot like Matlab:

```
from matplotlib.pylab import *
from numpy import *

# make data
x = linspace(0, 9)
model = sin(x)
dy = random.uniform(0.75, 1, len(x))
y = model + random.normal(scale=dy)

# plot the data
plot(x, y, "b.", label="My data points")
errorbar(x, y, yerr=dy, fmt="none", color='b')
plot(x, model, "r:", label="Model")

# axis limits
xlim(-1, 10)

# labels
xlabel("x")
ylabel("y")
title("title")

# add a legend using the labels you gave to plot()
legend()
```

plot_sin.pdf



Using matplotlib.pyplot

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

# make data
x = np.linspace(0.0, 9.0, 19)
model = np.sin(x)
dy = np.random.uniform(0.75, 1, len(x))
y = model + np.random.normal(scale=dy)

# plot the data
plt.plot(x, y, "b.", label="My data points")
plt.errorbar(x, y, xerr=None, yerr=dy, fmt=None, color='b')
plt.plot(x, model, "r:", label="Model")

# axis limits
plt.xlim(-1, 10)

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

# add a legend using the labels you gave to plot()
plt.legend(loc="best", ncol=1).draggable()

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

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

Format characters

The format string is of the form `CM` (`C`olour`M`arker)

`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` (abbreviation: `ls`) keywords

Semi-*OO* plotting with matplotlib

We can do the same thing using a figure:

```
fig = plt.figure()

axes = fig.add_axes((0.1, 0.1, 0.85, 0.80))

# plot the data
axes.plot(x, y, "b.", label="My data points")
axes.errorbar(x, y, xerr=None, yerr=dy, fmt=None, color='b')
axes.plot(x, model, "r:", label="Model")

# axis limits
axes.set_xlim(-1, 10)

# labels
axes.set_xlabel("x")
axes.set_ylabel("y")
axes.set_title("Clever Title")

# add a legend using the labels you gave to plot()
fontProps = dict(size = "small")
axes.legend(loc="best", prop=fontProps, ncol=1).draggable()
```

iPython notebooks

We can run these commands in the browser with a command like:

```
$ ipython notebook --no-browser src/notebooks/sin.ipynb
```

This provides a nice way of documenting your work.

See `http://ipython.org/ipython-doc/1/interactive/notebook.html`

Multi-panel plots

Once again, I need a figure, and then the command to select the third sub-window out of a 2x2 set is

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

so I could say

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

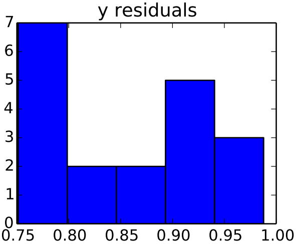
But I'm lazy and I don't like duplicating 2, 2

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

```
fig = plt.figure()

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(dy, bins=5)
axes.set_title("y residuals")
```

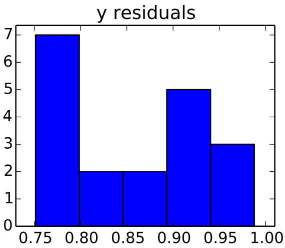


Hmm. Not a good choice for the axis limits.

Panel I: Histogram

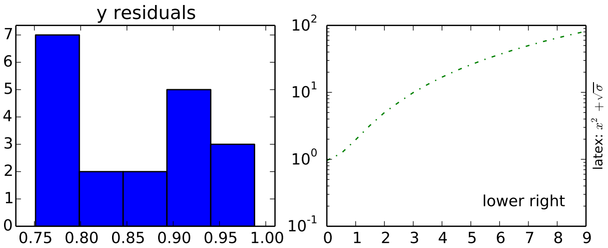
Let’s fix those limits:

```
fig = plt.figure()

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(dy, bins=5)
axes.set_title("y residuals")

axes.set_xlim(0.73, 1.01)
ymin, ymax = axes.get_ylim()
axes.set_ylim(ymin, 1.05*ymax)
```


y residuals
0
1
2
3
4
5
6
7
0.75
0.80
0.85
0.90
0.95
1.00

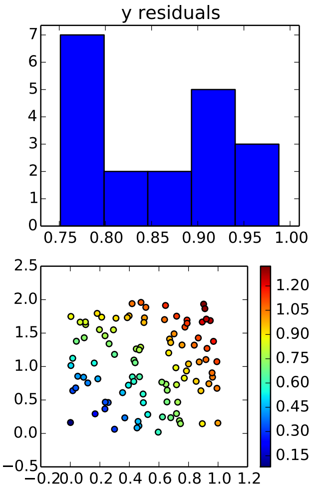
Panel II: Log-linear

```
# Initialize and make a log plot
z = x**2 + np.sqrt(dy)

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
0.75
0.80
0.85
0.90
0.95
1.00
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

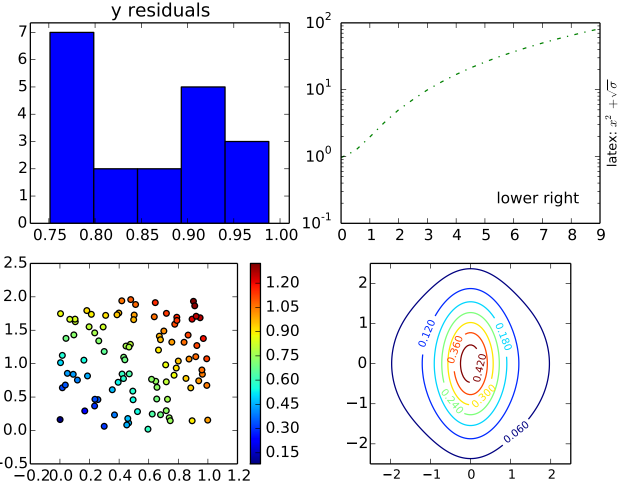
```
# Initialize and calculate points
axes = subplots.next()
xs = np.random.random(100)
ys = np.random.random(100)*2
zs = np.sqrt(xs**2 + ys**2/4.0)

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


y residuals
0.75 0.80 0.85 0.90 0.95 1.00
0 1 2 3 4 5 6 7
10^{-1} 10^{0} 10^{1} 10^{2}
0 1 2 3 4 5 6 7 8 9
lower right
latex: $x^2 + \sqrt{\sigma}$
−0.2 0.0 0.2 0.4 0.6 0.8 1.0 1.2
−0.5 0.0 0.5 1.0 1.5 2.0 2.5
0.15 0.30 0.45 0.60 0.75 0.90 1.05 1.20

Panel IV: Contours

```
# Initialize and calculate data
axes = subplots.next()
axis = np.linspace(-2.5, 2.5, 100)
X, Y = np.meshgrid(axis, axis)

sigma_x, sigma_y, f = 0.5, 1.0, 3
Z = 1/(2*np.pi*sigma_x*sigma_y)*np.exp(-0.5*((X/sigma_x)**2 + (Y/sigma_y)**2)) + \
    2/(2*np.pi*(f*sigma_x)**2)*np.exp(-0.5*(X**2 + Y**2)/(f*sigma_x)**2)

# Make a contour plot
CS = axes.contour(X,Y,Z)

# put labels on the contours
axes.clabel(CS, inline=1, fontsize=8)

# make circles circular
axes.set_aspect('equal')

# Change the ticklabel size
axes.tick_params(axis="x", labelsize="small")

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

plot_multi.pdf



Non-interactive plotting

What if you just want to make a plot, and not worry about interactive and Qt? One way is to use a canvas:

We needn't import pyplot: and the command to make the figure is a little different:

```
fig = matplotlib.figure.Figure()
```

The rest of the plotting is identical:

```
axes = fig.add_axes((0.1, 0.1, 0.85, 0.80))

# plot the data
axes.plot(x, y, "b.", label="My data points")
axes.errorbar(x, y, xerr=None, yerr=dy, fmt=None, color='b')
axes.plot(x, model, "r:", label="Model")

# axis limits
axes.set_xlim(-1, 10)

# labels
axes.set_xlabel("x")
axes.set_ylabel("y")
axes.set_title("Clever Title")
```

And finally we use that canvas:

```
from matplotlib.backends.backend_pdf import FigureCanvasPdf as FigCanvas

canvas = FigCanvas(fig)
canvas.print_figure("foo.png")          # a PNG file this time
```

Array operations, numpy

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

NumPy home (but it's easier to get it from *anaconda*)

http://numpy.scipy.org

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

```
import numpy as np

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

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

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

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

The import numpy as np is common enough that it's what the numpy documentation assumes.

numpy Arrays

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

```
>>> x = np.arange(5); print np.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 = np.arange(5)
>>> y = np.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, 0.90929742682568171),
 (3, 0.14112000805986721), (4, -0.7568024953079282)]
```

```
>>> print "\n".join(("%d %6.3f" % z for z 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

```
>>> np.random.seed(666)
>>> np.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 np.random.normal(loc=np.arange(5), scale=0.2)
[-0.2177586   0.88484585  1.66341985  3.04583705  3.64867496]
>>> print np.random.normal(np.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 = np.linspace(-2.0, 2.0, 5)
>>> X, Y = np.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 np.cos(X)*np.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 np.fft.fft(X)*np.sin(Y)
[[-0.00000000+0.j          2.27324357-3.12885135j  2.27324357-0.73862161j
   2.27324357+0.73862161j  2.27324357+3.12885135j]
 [-0.00000000+0.j          2.10367746-2.89546363j  2.10367746-0.68352624j
   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:

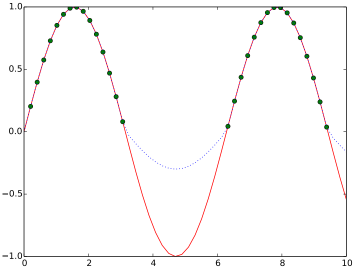
```
>>> x = np.arange(-4, 5); print x
[-4 -3 -2 -1  0  1  2  3  4]
>>> i = x**2 > 4
>>> print i
[ True  True False False False False False  True  True]
>>> print x[i]
[-4 -3  3  4]

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

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

Plotting and Extended Indexing

Here's another example

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

x = np.linspace(0, 10, 50); y = np.sin(x)

plt.plot(x, y, '-', color="red")
l = y > 0
plt.plot(x[l], y[l], 'o', color="green")

plt.plot(x, np.where(l, y, 0.3*y), color="blue", ls=':')
```

The np.where is like C/C++'s ?: operator.



numpy Linear Algebra

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

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

Traps await the unwary:

```
>>> M = np.zeros((n,n)); M[(i,i)] = i + 1
>>> U, s, Vt = np.linalg.svd(M)
>>> U * np.diag(s) * Vt
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

```
>>> U.dot(np.diag(s)).dot(Vt)
```

numpy Linear Algebra

Beware: vectors are treated differently from matrices. The vector x is the same as the vector x.T:

```
>>> x = np.array((1, 2))
>>> x
array([1, 2])
>>> x.T
array([1, 2])
>>> np.dot(x, x.T)
5
>>> np.dot(x.T, x)
5
```

If you want to distinguish between row vectors and column vectors, need to use a $1 \times n$ or $n \times 1$ matrix:

```
>>> x.resize(1,2)
>>> x
array([[1, 2]])
>>> x.T
array([[1],
       [2]])
>>> np.dot(x, x.T)
array([[5]])
>>> np.dot(x.T, x)
array([[1, 2],
       [2, 4]])
```

numpy Linear Algebra

If you use a matrix, you don't need to use dot:

```
>>> v = np.matrix((1, 2))
>>> v * v.T
matrix([[5]])
>>> v.T * v
matrix([[1, 2],
        [2, 4]])
```

A future version of python will support U @ np.diag(s) @ Vt with @ meaning, "matrix multiply". This does not remove the confusion between vectors and matrices, however: it is merely a shorthand for U.dot(np.diag(s)).dot(Vt).

Other numpy capabilities

numpy has lots of libraries:

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

I used the statistics package in analyzing the course questionnaire:

```
cov = np.corrcoef(data, rowvar=False)
for i in range(len(cov[0])):
   print "%6.3f" np.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 four years ago 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)
            mim = exposure.getMaskedImage()

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

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

            cmimg = mim.clone()
            convolve(cmimg, exposure.getMaskedImage(), smoothingKernel)
            exposure.setMaskedImage(cmimg)

            warpedExposure = makeExposure(mim.clone(), wcs0)
            warpExposure(warpedExposure, exposure, warpingKernel)
```

Every operation in red is written in C++.