Practical Scientific Computing in Python

Editors:
John D. Hunter
Fernando Pérez

With contributions from:
Perry Greenfield
Andrew Straw
Stéfan van der Walt
Jeff Whitaker

Contents

Introduction

This book is currently a work in progress, and ultimately we hope it will evolve into an open, community-driven document developed in tandem with the underlying tools, by the same scientists who have written them.

The book is aimed at practicing scientists, students and in general anyone who is looking for a modern, high-level and open platform for scientific computing. The Python language is in the opinion of the authors the leading candidate today for this role.

The book is broadly divided in two parts: the first is a general discussion of the Python tools used for scientific work, with an explanatory approach. It is not a complete Python reference book, as there are many excellent resources for the base language, both in print and online. But beyond the basic language and the NumPy book, it should serve as reasonably self-contained description of the core libraries for common numerical tasks.

The second part is meant as a practical workbook, and the build system used to produce the document will in the future allow users to create custom versions with only the examples that they deem practical for any given audience. This workbook approach grew out of a sequence of workshops taught by the editors at a number of research institutions and universities, and we've found it to be extremely convenient.

The workbook is structured as a collection of problems, meant to be solved by the reader as programming exercises. The entire book can be compiled in one of two forms: either with the examples in 'skeleton' form, where they contain incomplete code meant to be filled in, or with the full solution code. This should enable instructors to hand out the skeleton workbook at courses and workshops, with the solutions being available as well for after the teaching is over.

We hope that the community will continue to contribute many more examples, so that ultimately the projects allows for the easy construction of custom workbooks tailored to the needs of different audiences.

John D. Hunter and Fernando Pérez, editors.

Part 1

General Discussion

CHAPTER 1

Python for Scientific Computing

With material contributed by Perry Greenfield, Robert Jedrzejewski, Vicki Laidler and John Hunter

1. Who is using Python?

The use of Python in scientific computing is as wide as the field itself. A sampling of current work is provided here to indicate the breadth of disciplines represented and the scale of the problems addressed. The NASA Jet Propulsion Laboratory (JPL) uses Python as an interface language to FORTRAN and C++ libraries which form a suite of tools for plotting and visualization of spacecraft trajectory parameters in mission design and navigation. The Space Telescope Science Institute (STScI) uses Python in many phases of their pipeline: scheduling Hubble data acquisitions, managing volumes of data, and analyzing astronomical images [7]. The National Oceanic Atmospheric Administration (NOAA) uses Python for a wide variety of scientific computing tasks including simple scripts to parse and translate data files, prototyping of computational algorithms, writing user interfaces, web front ends, and the development of models [26, 6, 28]. At the Fundamental Symmetries Lab at Princeton University, Python is used to efficiently analyze large data sets from an experiment that searches for CPT and Lorentz Violation using an atomic magnetometer [22, 21]. The Pediatric Clinical Electrophysiology unit at The University of Chicago, which collects approximately 100 GB of data per week, uses Python to explore novel approaches to the localization and detection of epileptic seizures [19]. The Enthought Corporation is using Python to build customized applications for oil exploration for the petroleum industry. At the world's largest radio telescopes, e.g., Arecibo and the Green Bank Telescope, Python is used for data processing, modelling, and scripting high-performance computing jobs in order to search for and monitor binary and millisecond pulsars in terabyte datasets [32, 31]. At the Computational Genomics Laboratory at the Australian National University, researchers are using Python to build a toolkit which enables the specification of novel statistical models of sequence evolution on parallel hardware [20, 12]. Michel Sanner's group at the Scripps Research Institute uses Python extensively to build a suite of applications for molecular visualization and exploration of drug/molecule interactions using virtual reality and 3D printing technology [35, 36]. Engineers at Google use Python in automation, control and tuning of their computational grid, and use `SWIG` generated Python of their in-house C++ libraries in virtually all facets of their work [9, 38]. Many other use cases – ranging from animation at Industrial Light and Magic, to space shuttle mission control, to grid monitoring and control at Rackspace, to drug discovery, meteorology and air traffic control – are detailed in O'Reilly's two volumes of *Python Success Stories* [1, 2].

2. Advantages of Python

> *The canonical, "Python is a great first language", elicited, "Python is a great last language!"* – Noah Spurrier

This quotation summarizes an important reason scientists migrate to Python as a programming language. As a "great first language" Python has a simple, expressive syntax that is accessible to the newcomer. "Python as executable pseudocode" reflects the fact that Python syntax mirrors the obvious and intuitive pseudo-code syntax used in many journals [39]. As a great first language, it does not impose a single programming paradigm on scientists, as Java does with object oriented programming, but rather allows one to code at many levels of sophistication, including

BASIC/FORTRAN/Matlab style procedural programming familiar to many scientists. Here is the canonical first program "hello world" in Python:

```
# Python
print 'hello world'
```

world" in Java

```
// java
class myfirstjavaprog
{
  public static void main(String args[])
  {
    System.out.println("Hello World!");
  }
}
```

In addition to being accessible to new programmers and scientists, Python is powerful enough to manage the complexity of large applications, supporting functional programming, object orienting programming, generic programming and metaprogramming. That Python supports these paradigms suggests why it is also a "great last language": as one increases their programming sophistication, the language scales naturally. By contrast, commercial languages like Matlab and IDL, which also support a simple syntax for simple programs do not scale well to complex programming tasks.

The built-in Python data-types and standard library provide a powerful platform in every distribution [34, ?]. The standard data types encompass regular and arbitrary length integers, complex numbers, floating point numbers, strings, lists, associative arrays, sets and more. In the standard library included with every Python distribution are modules for regular expressions, data encodings, multimedia formats, math, networking protocols, binary arrays and files, and much more. Thus one can open a file on a remote web server and work with it as easily as with a local file

```
# this 3 line script downloads and prints the yahoo web site
from urllib import urlopen
for line in urlopen('http://yahoo.com').readlines():
   print line
```

Complementing these built-in features, Python is also readily extensible, giving it a wealth of libraries for scientific computing that have been in development for many years [13, 14]. `NumPy` supports large array manipulations, math, optimized linear algebra, efficient Fourier transforms and random numbers. `scipy` is a collection of Python wrappers of high performance FORTRAN code (eg LAPACK, ODEPACK) for numerical analysis [3]. `IPython` is a command shell ala Mathematica, Matlab and IDL for interactive programming, data exploration and visualization with support for command history, completion, debugging and more. `Matplotlib` is a 2D graphics package for making publication quality graphics with a Matlab compatible syntax that is also embeddable in applications. `f2py`, `SWIG`, `weave`, and `pyrex` are tools for rapidly building Python interfaces to high performance compiled code, `MayaVi` is a user friendly graphical user interface for 3D visualizations built on top of the state-of-the-art Visualization Toolkit [37]. `pympi`, `pypar`, `pyro`, `scipy.cow`, and `pyxg` are tools for cluster building and doing parallel, remote and distributed computations. This is a sampling of general purpose libraries for scientific computing in Python, and does not begin to address the many high quality, domain specific libraries that are also available.

All of the infrastructure described above is open source software that is freely distributable for academic and commercial use. In both the educational and scientific arenas, this is a critical point. For education, this platform provides students with tools that they can take with them outside the classroom to their homes and jobs and careers beyond. By contrast, the use commercial tools such as Matlab and IDL limits access to major institutions. For scientists, the use of open source tools is consistent with the scientific principle that all of the steps in an analysis or simulation should be open for review, and with the principle of reproducible research [11].

3. Mixed Language Programming

The programming languages of each generation evolve in part to fix the problems of those that came before [10]. FORTRAN, the original high level language of scientific computing [33], was designed to allow scientists to express code at a level closer to the language of the problem domain. ALGOL and its successor Pascal, widely used in education in the 1970s, were designed to alleviate some of the perceived problems with FORTRAN and to create a language with a simpler and more expressive syntax [5, 25].

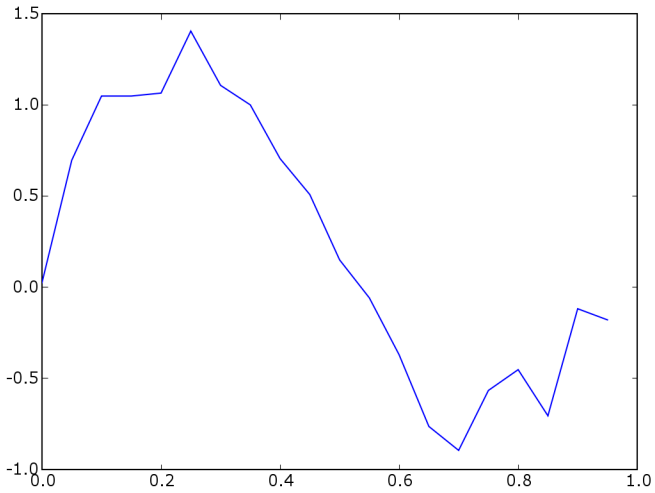

FIGURE 1. Loading ASCII data and displaying with plot

Object oriented programming languages evolved to allow a closer correspondence between the code and the physical system it models [16], and C++ provided a relatively high performance object orientated implementation compatible with the popular C programming language [41, 40]. But implementing object orientation efficiently requires programmers stay close to the machine, managing memory and pointers, and this created a lot of complexity in programs while limiting portability. Interpreted languages such as Tcl, Perl, Python, and Java evolved to manage some of the low-level and platform specific details, making programs easier to write and maintain, but with a performance penalty [27, 4]. For many scientists, however, pure object oriented systems like Java are unfamiliar, and languages like Matlab and Python provide the safety, portability and ease of use of an interpreted language without imposing an object oriented approach to coding [15, 17].

The result of these several decades is that there are many platforms for scientific computing in use today. The number of man hours invested in numerical methods in FORTRAN, visualization libraries in C++, bioinformatics toolkits in Perl, object frameworks in Java, domain specific toolkits in Matlab, etc...requires an approach that integrates this work. Python is the language that provides maximal integration with other languages, with tools for transparently and semi-automatically interfacing with FORTRAN, C, C++, Java, .NET, Matlab, and Mathematica code [18, 9]. In our view, the ability to work seamlessly with code from many languages is the present and the future of scientific computing, and Python effectively integrates these languages into a single environment.

4. Getting started

We'll get started with python by introducing arrays and plotting by working with a simple ASCII text file `mydata.dat` of two columns; the first column contains the times that some measurement was acquired, and the second column are the sampled voltages at that time. The file looks like

```
0.0000 0.4911
0.0500 0.5012
0.1000 0.7236
0.1500 1.1756
... and so on
```

While it would be easy enough to process this file by writing a python function to do it, there is no need to, since the matplotlib pylab module has a matlab-compatible `load` function for loading ASCII array data (Figure 1). To complete these exercises, you should have ipython and matplotlib installed, and start ipython in pylab mode with

```
> ipython -pylab
```

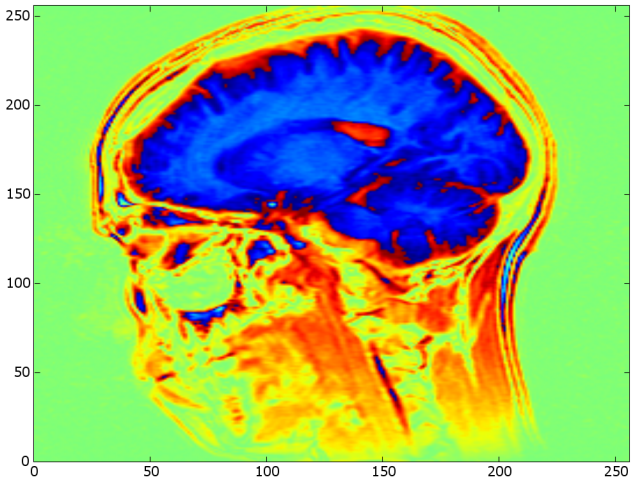

FIGURE 2. Loading binary image data and displaying with `imshow`

LISTING 1.1. Loading an ASCII text file and plotting the columns

```
In [1]: X = load('data/ascii_data.dat') # X is an array
In [2]: t = X[:,0]  # extract the first column
In [3]: v = X[:,1]  # extract the second column
In [4]: len(t)
Out[4]: 20
In [5]: len(v)
Out[5]: 20
In [6]: plot(t,v) # plot the data
Out[6]: [<matplotlib.lines.Line2D instance at 0xb65921ac>]
```

It is also easy to load data from binary files. In the example below, we have some image data in raw binary string format. The image is 256x256 pixels, and each pixel is a 2 byte integer. We read this into a string using python's `file` function – the 'rb' flag says to open the file in `read/binary` mode. We can then use the numpy `fromstring` method to convert this to an array, passing the type of the data (`int16`) as an argument. We reshape the array by changing the array shape attribute to 256 by 256, and pass this off to the matplotlib pylab command `imshow` for plotting. matplotlib has a number of colormaps, and the default one is jet; the data are automatically normalized and colormaps producing the image in Figure 2

LISTING 1.2. Loading an binary image data and plotting it in matplotlib

```
# open a file as "read binary" and read it into a string
In [1]: s = file('data/images/r1025.ima', 'rb').read()
# the string is length 256*256*2 = 131072
In [2]: len(s)
Out[2]: 131072
# the data are 2 byte / 16 bit integers
# fromstring converts them to array
In [3]: im = nx.fromstring(s, nx.Int16)
# reshape the array to 256x256
In [4]: im.shape = 256,256
# and plot it with matplotlib's imshow function
In [5]: imshow(im)
Out[5]: <matplotlib.image.AxesImage instance at 0xb659230c>
```

5. An Introduction to Arrays

5.1. Creating arrays. There are a few different ways to create arrays besides modules that obtain arrays from data files such

```
>>> x = zeros((20,30))
```

creates a 20x30 array of zeros (default integer type; details on how to specify other types will follow). Note that the dimensions ("shape" in numpy parlance) are specified by giving the dimensions as a comma-separated list within parentheses. The parentheses aren't necessary for a single dimension. As an aside, the parentheses used this way are being used to specify a Python tuple; more will be said about those in a later tutorial. For now you only need to imitate this usage.

Likewise one can create an array of 1's using the `ones()` function.

The `arange()` function can be used to create arrays with sequential values. E.g.,

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

Note that that the array defaults to starting with a 0 value and does not include the value specified (though the array does have a length that corresponds to the argument)

Other variants:

```
>>> arange(10.)
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9])
>>> arange(3,10)
array([3, 4, 5, 6, 7, 8, 9])
>>> arange(1., 10., 1.1) # note trickiness
array([1. , 2.1, 3.2, 4.3, 5.4, 6.5, 7.6, 8.7, 9.8])
```

Finally, one can create arrays from literal arguments:

```
>>> print array([3,1,7])
[3 1 7]
>>> print array([[2,3],[4,4]])
[[2 3]
 [4 4]]
```

The brackets, like the parentheses in the zeros example above have a special meaning in Python which will be covered later (Python lists). For now, just mimic the syntax used here.

5.2. Array numeric types. numpy supports all standard numeric types. The default integer matches what Python uses for integers, usually 32 bit integers or what numpy calls `int32`. The same is true for floats, i.e., generally 64-bit doubles called `float64` in numpy. The default complex type is `complex64`. Many of the functions accept a type argument. For example

```
>>> zeros(3, int8) # Signed byte
>>> zeros(3, dtype=uint8) # Unsigned byte
>>> array([2,3], dtype=float32)
>>> arange(4, dtype=complex64)
```

The possible types are `int8, uint8, int16, uint16, int32, uint32, int64, uint64, float32, float64, complex32, complex64.` To find out the type of an array use the .dtype() method. E.g.,

```
>>> arr.dtype() dtype('float32')
```

To convert an array to a different type use the `astype()` method, e.g,

```
>>> a = arr.astype(float64)
```

5.3. Printing arrays. Interactively, there are two common ways to see the value of an array. Like many Python objects, just typing the name of the variable itself will print its contents (this only works in interactive mode). You can also explicitly print it. The following illustrates both approaches:

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

By default the array module limits the amount of an array that is printed out (to spare you the effects of printing out millions of values). For example:

```
>>> x = arange(1000000)
print x
[    0     1     2 ..., 999997 999998 999999]
```

5.4. Indexing 1-D arrays. As with IDL and Matlab, there are many options for indexing arrays.

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

Simple indexing:

```
>>> x[2] # 3rd element
2
```

Indexing is 0-based. The first value in the array is `x[0]`

Indexing from end:

```
>>> x[-2] # -1 represents the last element, -2 next to last...
8
```

Slices

To select a subset of an array:

```
>>> x[2:5]
array([2, 3, 4])
```

Note that the upper limit of the slice is not included as part of the subset! This is viewed as unexpected by newcomers and a defect. Most find this behavior very useful after getting used to it (the reasons won't be given here). Also important to understand is that slices are views into the original array in the same sense that references view the same array. The following demonstrates:

```
>>> y = x[2:5]
>>> y[0] = 99
>>> y
array([99, 3, 4])
>>> x
array([0, 1, 99, 3, 4, 5, 6, 7, 8, 9])
```

Changes to a slice will show up in the original. If a copy is needed use `x[2:5].copy()`

Short hand notation

```
>>> x[:5] # presumes start from beginning
array([ 0, 1, 99, 3, 4])
>>> x[2:] # presumes goes until end
array([99, 3, 4, 5, 6, 7, 8, 9])
>>> x[:] # selects whole dimension
array([0, 1, 99, 3, 4, 5, 6, 7, 8, 9])
```

Strides:

```
>>> x[2:8:3] # Stride every third element
array([99, 5])
```

Index arrays:

```
>>> x[[4,2,4,1]]
array([4, 99, 4, 1])
```

Using results of logical indexing

```
>>> x > 5
array([0,0,1,0,0,0,1,1,1,1], type=Bool)
>>> x[x>5]
array([99, 6, 7, 8, 9])
```

5.5. Indexing multidimensional arrays. Before describing this in detail it is very important to note an item regarding multidimensional indexing that will certainly cause you grief until you become accustomed to it: ARRAY INDICES USE THE OPPOSITE CONVENTION AS FORTRAN REGARDING ORDER OF INDICES FOR MULTIDIMENSIONAL ARRAYS.

```
>>> im = arange(24)
>>> im.shape = 4,6
>>> im
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

To emphasize the point made in the previous paragraph, the index that represents the most rapidly varying dimension in memory is the 2nd index, not the first.

Partial indexing:

```
>>> im[1]
array([6, 7, 8, 9, 10, 11])
```

If only some of the indices for a multidimensional array are specified, then the result is an array with the shape of the "leftover" dimensions, in this case, 1-dimensional. The 2nd row is selected, and since there is no index for the column, the whole row is selected.

All of the indexing tools available for 1-D arrays apply to n-dimensional arrays as well (though combining index arrays with slices is not currently permitted). To understand all the indexing options in their full detail, read sections 4.6, 4.7 and 6 of the numpy manual.

5.6. Compatibility of dimensions. In operations involving combining (e.g., adding) arrays or assigning them there are rules regarding the compatibility of the dimensions involved. For example the following is permitted:

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

since a single value is considered "broadcastable" over a 5 element array. But this is not permitted:

```
>>> x[:5] = array([0,1,2,3])
```

since a 4 element array does not match a 5 element array.

The following explanation can probably be skipped by most on the first reading; it is only important to know that rules for combining arrays of different shapes are quite general. It is hard to precisely specify the rules without getting a bit confusing, but it doesn't take long to get a good intuitive feeling for what is and isn't permitted. Here's an attempt anyway: The shapes of the two involved arrays when aligned on their trailing part must be equal in value or one must have the value one for that dimension. The following pairs of shapes are compatible:

```
(5,4):(4,) -> (5,4)
(5,1):(4,) -> (5,4)
(15,3,5):(15,1,5) -> (15,3,5)
(15,3,5):(3,5) -> (15,3,5)
(15,1,5):(3,1) -> (15,3,5)
```

so that one can add arrays of these shapes or assign one to the other (in which case the one being assigned must be the smaller shape of the two). For the dimensions that have a 1 value that are matched against a larger number, the values in that dimension are simply repeated. For dimensions that are missing, the sub-array is simply repeated for those. The following shapes are not compatible:

```
(3,4):(4,3)
(1,3):(4,)
```

Examples:

```
>>> x = zeros((5,4))
>>> x[:,:] = [2,3,2,3]
>>> x
array([[2, 3, 2, 3],
       [2, 3, 2, 3],
       [2, 3, 2, 3],
       [2, 3, 2, 3],
       [2, 3, 2, 3]])
>>> a = arange(3)
>>> b = a[:] # different array, same data (huh?)
>>> b.shape = (3,1)
>>> b
array([[0],
       [1],
       [2]])
>>> a*b # outer product
```

```
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

5.7. ufuncs. A ufunc (short for Universal Function) applies the same operation or function to all the elements of an array independently. When two arrays are added together, the `add` ufunc is used to perform the array addition. There are ufuncs for all the common operations and mathematical functions. More specialized ufuncs can be obtained from add-on libraries. All the operators have corresponding ufuncs that can be used by name (e.g., `add` for +). These are all listed in table below. Ufuncs also have a few very handy methods for binary operators and functions whose use are demonstrated here.

```
>>> x = arange(9)
>>> x.shape = (3,3)
>>> x
array([0, 1, 2],
      [3, 4, 5],
      [6, 7, 8]])
>>> add.reduce(x) # sums along the first index
array([9, 12, 15])
>>> add.reduce(x, axis=1) # sums along the 2nd index
array([3, 12, 21])
>>> add.accumulate(x) # cumulative sum along the first index
array([[0,  1,  2],
       [3,  5,  7],
       [9, 12, 15]])
>>> multiply.outer(arange(3),arange(3))
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

Standard Ufuncs (with corresponding symbolic operators, when they exist, shown in parentheses)

add (+)	log	greater (>)
subtract (-)	log10	greater_equal (>=)
multiply (*)	cos	less (<)
divide (/)	arcos	less_equal (<=)
remainder (%)	sin	logical_and
absolute, abs	arcsin	logical_or
floor	tan	logical_xor
ceil	arctan	bitwise_and (&)
fmod	cosh	bitwise_or (|)
conjugate	sinh	bitwise_xor (^)
minimum	tanh	bitwise_not (~)
maximum	sqrt	rshift (>>)
power (**)	equal (==)	lshift (<<)
exp	not_equal (!=)	

Note that there are no corresponding Python operators for `logical_and` *and* `logical_or`*. The Python* `and` *and* `or` *operators are NOT equivalent to these respective ufuncs!*

5.8. Array functions. There are many array utility functions. The following lists the more useful ones with a one line description. See the numpy manual for details on how they are used. Arguments shown with argument=value indicate what the default value is if called without a value for that argument.

all(*a*):: are all elements of array nonzero
allclose(*a1, a2, rtol=1.e-5, atol=1.e-8*):: true if all elements within specified amount (between two arrays)
alltrue(*a, axis=0*):: are all elements nonzero along specified axis true.

any(*a*):: are any elements of an array nonzero
argmax(*a, axis=-1*), argmin(*a,axis=-1*):: return array with min/max locations for selected axis
argsort(*a, axis=-1*):: returns indices of results of sort on an array

choose(*selector, population, clipmode=CLIP*):: fills specified array by selecting corresponding values from a set of arrays using integer selection array (population is a tuple of arrays; see tutorial 2)
clip(*a, amin, amax*):: clip values of array *a* at values *amin*, *amax*
dot(*a1, a2*):: dot product of arrays *a1* & *a2*
compress(*condition, a ,axis=0*):: selects elements from array *a* based on boolean arraycondition
concatenate(*arrays, axis=0*):: concatenate arrays contained in sequence of arrays arrays
cumproduct(*a, axis=0*):: net cumulative product along specified axis
cumsum(*a, axis=0*):: accumulate array along specified axis
diagonal(*a, offset=0, axis1=0, axis2=1*):: returns diagonal of 2-d matrix with optional offsets.

fromfile(*file, type, shape=None*):: Use binary data in file to form new array of specified type.
fromstring(*datastring, type, shape=None*):: Use binary data in *datastring* to form new array of specified shape and type
identity(*n, type=None*):: returns identity matrix of size nxn.
indices(*shape, type=None*):: generate array with values corresponding to position of selected index of the array
innerproduct(*a1, a2*):: guess
matrixmultiply(*a1, a2*):: guess
outerproduct(*a1, a2*):: guess
product(*a, axis=0*):: net product of elements along specified axis
ravel(*a*):: creates a 1-d version of an array
repeat(*a, repeats, axis=0*):: generates new array with repeated copies of input array *a*
resize(*a, shape*):: replicate or truncate array to new shape
searchsorted(*bin, a*):: return indices of mapping values of an array *a* into a monotonic array *bin*
sometrue(*a, axis=0*):: are any elements along specified axis true
sort(*a, axis=-1*):: sort array elements along selected axis
sum(*a, axis=0*):: sum array along specified axis
swapaxes(*a, axis1, axis2*):: switch indices for axis of array (doesn't actually move data, just maps indices differently)
trace(*a, offset=0, axis1=0, axis2=1*):: compute trace of matrix *a* with optional offset.
transpose(*a, axes=None*):: transpose indices of array (doesn't actually move data, just maps indices differently)

where(*a*):: find "true" locations in array *a*

5.9. Array methods. Arrays have several methods. They are used as methods are with any object. For example (using the array from the previous example):

```
>>> # sum all array elements
>>> x.sum() # the L indicates a Python Long integer
36L
```

The following lists all the array methods that exist for an array object `a` (a number are equivalent to array functions; these have no summary description shown):

***a*.argmax**(*axis=-1*):
***a*.argmin**(*axis=-1*):
***a*.argsort**(*axis=-1*):
***a*.astype**(*type*):: copy array to specified numeric type
***a*.byteswap**():: perform byteswap on data in place
***a*.byteswapped**():: return byteswapped copy of array
***a*.conjugate**():: complex conjugate
***a*.copy**():: produce copied version of array (instead of view)
***a*.diagonal**():
***a*.info**():: print info about array
***a*.isaligned**():: are data elements guaranteed aligned with memory?
***a*.isbyteswapped**():: are data elements in native processor order?
***a*.iscontiguous**():: are data elements contiguous in memory?
***a*.is_c_array**():: are data elements aligned, not byteswapped, and contiguous?
***a*.is_fortran_contiguous**():: are indicies defined to follow Fortran conventions?
***a*.is_f_array**():: are indices defined to follow Fortran conventions and data are aligned and not byteswapped
***a*.itemsize**():: size of data element in bytes
***a*.max**(type=None):: maximum value in array
***a*.min**():: minimum value in array
***a*.nelements**():: total number of elements in array

a.new():: returns new array of same type and size (data uninitialized)
a.repeat(a,repeats,axis=0)::
a.resize(shape)::
a.size():: same as nelements
a.dtype():: returns type of array
a.tofile(*file*):: write binary data to file
a.tolist():: convert data to Python list format
a.tostring():: copy binary data to Python string
a.transpose(*axes=-1*):: transpose array
a.stddev():: standard deviation
a.sum():: sum of all elements
a.swapaxes(*axis1, axis2*):
a.togglebyteorder():: change byteorder flag without changing actual data byteorder
a.trace():
a.view():: returns new array object using view of same data

5.10. Array attributes:

a.shape:: returns shape of array
a.flat:: returns view of array treating it as 1-dimensional. Doesn't work if array is not contiguous
a.real:: return real component of array (exists for all types)
a.imag, a.imaginary:: return imaginary component (exists only for complex types)

6. Exercises

EXERCISE 7. Load the binary image shown in Figure2. What is the mean pixel value, what are the standard deviation of pixel values? Sum over the rows and make a bar plot for the summated intensity across rows. Do the same for columns. Make a histogram of all the data in the image. (Hint – see `nx.mlab.mean`, `nx.mlab.std`, `pylab.bar` and `pylab.hist`)

EXAMPLE 7.1. this is another test

this is a test

CHAPTER 2

A whirlwind tour of python and the standard library

This is a quick-and-dirty introduction to the python language for the impatient scientist. There are many top notch, comprehensive introductions and tutorials for python. For absolute beginners, there is the *Python Beginner's Guide*.[1] The official *Python Tutorial* can be read online[2] or downloaded[3] in a variety of formats. There are over 100 python tutorials collected online.[4]

There are also many excellent books. Targetting newbies is Mark Pilgrim's *Dive into Python* which in available in print and for free online[5], though for absolute newbies even this may be too hard [30]. For experienced programmers, David Beasley's *Python Essential Reference* is an excellent introduction to python, but is a bit dated since it only covers python2.1 [8]. Likwise Alex Martelli's *Python in a Nutshell* is highly regarded and a bit more current – a 2nd edition is in the works[23]. And *The Python Cookbook* is an extremely useful collection of python idioms, tips and tricks [24].

But the typical scientist I encounter wants to solve a specific problem, eg, to make a certain kind of graph, to numerically integrate an equation, or to fit some data to a parametric model, and doesn't have the time or interest to read several books or tutorials to get what they want. This guide is for them: a short overview of the language to help them get to what they want as quickly as possible. We get to advanced material pretty quickly, so it may be touch sledding if you are a python newbie. Take in what you can, and if you start getting dizzy, skip ahead to the next section; you can always come back to absorb more detail later, after you get your real work done.

1. Hello Python

Python is a dynamically typed, object oriented, interpreted language. Interpreted means that your program interacts with the python interpreter, similar to Matlab, Perl, Tcl and Java, and unlike FORTRAN, C, or C++ which are compiled. So let's fire up the python interpreter and get started. I'm not going to cover installing python – it's standard on most linux boxes and for windows there is a friendly GUI installer. To run the python interpreter, on windows, you can click `Start->All Programs->Python 2.4->Python (command line)` or better yet, install `ipython`, a python shell on steroids, and use that. On linux / unix systems, you just need to type `python` or `ipython` at the command line. The `>>>` is the default python shell prompt, so don't type it in the examples below

```
>>> print 'hello world'
hello world
```

As this example shows, *hello world* in python is pretty easy – one common phrase you hear in the python community is that "it fits your brain". – the basic idea is that coding in python feels natural. Compare python's version with *hello world* in C++

```
// C++
#include <iostream>
int main ()
{
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

[1] http://www.python.org/moin/BeginnersGuide
[2] http://docs.python.org/tut/tut.html
[3] http://docs.python.org/download.html
[4] http://www.awaretek.com/tutorials.html
[5] http://diveintopython.org/toc/index.html

2. Python is a calculator

Aside from my daughter's solar powered cash-register calculator, Python is the only calculator I use. From the python shell, you can type arbitrary arithmetic expressions.

```
>>> 2+2
4
>>> 2**10
1024
>>> 10/5
2
>>> 2+(24.3 + .9)/.24
107.0
>>> 2/3
0
```

The last line is a standard newbie gotcha – if both the left and right operands are integers, python returns an integer. To do floating point division, make sure at least one of the numbers is a float

```
>>> 2.0/3
0.66666666666666663
```

The distinction between integer and floating point division is a common source of frustration among newbies and is slated for destruction in the mythical Python 3000.[6] Since default integer division will be removed in the future, you can invoke the time machine with the `from __future__` directives; these directives allow python programmers today to use features that will become standard in future releases but are not included by default because they would break existing code. From future directives should be among the first lines you type in your python code if you are going to use them, otherwise they may not work. The future division operator will assume floating point division by default,[7]and provides another operator // to do classic integer division.

```
>>> from __future__ import division
>>> 2/3
0.66666666666666663
>>> 2//3
0
```

python has four basic numeric types: int, long, float and complex, but unlike C++, BASIC, FORTRAN or Java, you don't have to declare these types. python can infer them

```
>>> type(1)
<type 'int'>
>>> type(1.0)
<type 'float'>
>>> type(2**200)
<type 'long'>
```

$2^{200}$is a huge number!

```
>>> 2**200
1606938044258990275541962092341162602522202993782792835301376L
```

but python will blithely compute it and much larger numbers for you as long as you have CPU and memory to handle them. The integer type, if it overflows, will automatically convert to a python `long` (as indicated by the appended `L` in the output above) and has no built-in upper bound on size, unlike C/C++ longs.

Python has built in support for complex numbers. Eg, we can verify $i^2 = -1$

```
>>> x = complex(0,1)
>>> x*x
(-1+0j)
```

To access the real and imaginary parts of a complex number, use the `real` and `imag` attributes

[6]Python 3000 is a future python release that will clean up several things that Guido considers to be warts.

[7]You may have noticed that 2/3 was represented as 0.66666666666666663 and not 0.66666666666666666 as might be expected. This is because computers are binary calculators, and there is no exact binary representation of 2/3, just as there is no exact binary representation of 0.1

```
>>> 0.1
0.10000000000000001
```

Some languages try and hide this from you, but python is explicit.

```
>>> x.real
0.0
>>> x.imag
1.0
```

If you come from other languages like Matlab, the above may be new to you. In matlab, you might do something like this (>> is the standard matlab shell prompt)

```
>> x = 0+j
x =
   0.0000 + 1.0000i
>> real(x)
ans =
     0
>> imag(x)
ans =
     1
```

That is, in Matlab, you use a *function* to access the real and imaginary parts of the data, but in python these are attributes of the complex object itself. This is a core feature of python and other object oriented languages: an object carries its data and methods around with it. One might say: "a complex number knows it's real and imaginary parts" or "a complex number knows how to take its conjugate", you don't need external functions for these operations

```
>>> x.conjugate
<built-in method conjugate of complex object at 0xb6a62368>
>>> x.conjugate()
-1j
```

On the first line, I just followed along from the example above with `real` and `imag` and typed `x.conjugate` and python printed the representation `<built-in method conjugate of complex object at 0xb6a62368>`. This means that `conjugate` is a *method*, a.k.a a function, and in python we need to use parentheses to call a function. If the method has arguments, like the `x` in `sin(x)`, you place them inside the parentheses, and if it has no arguments, like `conjugate`, you simply provide the open and closing parentheses. `real`, `imag` and `conjugate` are attributes of the complex object, and `conjugate` is a *callable* attribute, known as a *method*.

OK, now you are an object oriented programmer. There are several key ideas in object oriented programming, and this is one of them: an object carries around with it data (simple attributes) and methods (callable attributes) that provide additional information about the object and perform services. It's one stop shopping – no need to go to external functions and libraries to deal with it – the object knows how to deal with itself.

3. Accessing the standard library

Arithmetic is fine, but before long you may find yourself tiring of it and wanting to compute logarithms and exponents, sines and cosines

```
>>> log(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'log' is not defined
```

These functions are not built into python, but don't despair, they are built into the python standard library. To access a function from the standard library, or an external library for that matter, you must import it.

```
>>> import math
>>> math.log(10)
2.3025850929940459
>>> math.sin(math.pi)
1.2246063538223773e-16
```

Note that the default `log` function is a base 2 logarithm (use `math.log10` for base 10 logs) and that floating point math is inherently imprecise, since analytically$\sin(\pi) = 0$.

It's kind of a pain to keep typing `math.log` and `math.sin` and `math.pi`, and python is accomodating. There are additional forms of `import` that will let you save more or less typing depending on your desires

```
# Appreviate the module name: m is an alias
>>> import math as m
>>> m.cos(2*m.pi)
1.0
# Import just the names you need
>>> from math import exp, log
>>> log(exp(1))
1.0
# Import everything - use with caution!
>>> from math import *
>>> sin(2*pi*10)
-2.4492127076447545e-15
```

To help you learn more about what you can find in the math library, python has nice introspection capabilities – introspection is a way of asking an object about itself. For example, to find out what is available in the math library, we can get a directory of everything available with the `dir` command[8]

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh
```

This gives us just a listing of the names that are in the math module – they are fairly self descriptive, but if you want more, you can call `help` on any of these functions for more information

```
>>> help(math.sin)
Help on built-in function sin:
sin(...)
sin(x)
Return the sine of x (measured in radians).
```

and for the whole math library

```
>>> help(math)
Help on module math:

NAME
    math

FILE
    /usr/local/lib/python2.3/lib-dynload/math.so

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.
```

And much more which is snipped. Likewise, we can get information on the complex object in the same way

```
>>> x = complex(0,1)
>>> dir(x)
['__abs__', '__add__', '__class__', '__coerce__', '__delattr__', '__div__', '__divmod__',
```

[8] In addition to the introdpection and help provided in the python interpreter, the official documentation of the python standard library is very good and up-to-date http://docs.python.org/lib/lib.html .

Notice that called `dir` or `help` on the `math` *module*, the `math.sin` *function*, and the `complex` *number* `x`. That's because modules, functions and numbers are all *objects*, and we use the same object introspection and help capabilites on them. We can find out what type of object they are by calling `type` on them, which is another function in python's introspection arsenal

```
>>> type(math)
<type 'module'>
>>> type(math.sin)
<type 'builtin_function_or_method'>
>>> type(x)
<type 'complex'>
```

Now, you may be wondering: what were all those god-awful looking double underscore methods, like `__abs__` and `__mul__` in the `dir` listing of the complex object above? These are methods that define what it means to be a numeric type in python, and the complex object implements these methods so that complex numbers act like the way should, eg `__mul__` implements the rules of complex multiplication. The nice thing about this is that python specifies an application programming interface (API) that is the definition of what it means to be a number in python. And this means you can define your own numeric types, as long as you implement the required special double underscore methods for your custom type. double underscore methods are very important in python; although the typical newbie never sees them or thinks about them, they are there under the hood providing all the python magic, and more importantly, showing the way to let you make magic.

4. Strings

We've encountered a number of types of objects above: int, float, long, complex, method/function and module. We'll continue our tour with an introduction to strings, which are critical components of almost every program. You can create strings in a number of different ways, with single quotes, double quotes, or triple quotes – this diversity of methods makes it easy if you need to embed string characters in the string itself

```
# single, double and triple quoted strings
>>> s = 'Hi Mom!'
>>> s = "Hi Mom!"
>>> s = """Porky said, "That's all folks!" """
```

You can add strings together to concatenate them

```
# concatenating strings
>>> first = 'John'
>>> last = 'Hunter'
>>> first+last
'JohnHunter'
```

or call string methods to process them: upcase them or downcase them, or replace one character with another

```
# string methods
>>> last.lower()
'hunter'
>>> last.upper()
'HUNTER'
>>> last.replace('h', 'p')
'Hunter'
>>> last.replace('H', 'P')
'Punter'
```

Note that in all of these examples, the string `last` is unchanged. All of these methods operate on the string and return a new string, leaving the original unchanged. In fact, python strings cannot be changed by any python code at all: they are *immutable* (unchangeable). The concept of mutable and immutable objects in python is an important one, and it will come up again, because only immutable objects can be used as keys in python dictionaries and elements of python sets.

You can access individual characters, or slices of the string (substrings), using indexing. A string in sequence of characters, and strings implement the sequence protocol in python – we'll see more examples of python sequences later – and all sequences have the same syntax for accessing their elements. Python

uses 0 based indexing which means the first element is at index 0; you can use negative indices to access the last elements in the sequence

```
# string indexing
>>> last = 'Hunter'
>>> last[0]
'H'
>>> last[1]
'u'
>>> last[-1]
'r'
```

To access substrings, or generically in terms of the sequence protocol, slices, you use a colon to indicate a range

```
# string slicing
>>> last[0:2]
'Hu'
>>> last[2:4]
'nt'
```

As this example shows, python uses "one-past-the-end" indexing when defining a range; eg, in the range `indmin:indmax`, the element of `imax` is not included. You can use negative indices when slicing too; eg, to get everything before the last character

```
>>> last[0:-1]
'Hunte'
```

You can also leave out either the min or max indicator; if they are left out, 0 is assumed to be the `indmin` and one past the end of the sequence is assumed to be `indmax`

```
>>> last[:3]
'Hun'
>>> last[3:]
'ter'
```

There is a third number that can be placed in a slice, a step, with syntax indmin:indmax:step; eg, a step of 2 will skip every second letter

```
>>> last[1:6:2]
'utr'
```

Although this may be more that you want to know about slicing strings, the time spent here is worthwhile. As mentioned above, all python sequences obey these rules. In addition to strings, lists and tuples, which are built-in python sequence data types and are discussed in the next section, the numeric arrays widely used in scientific computing also implement the sequence protocol, and thus have the same slicing rules.

EXERCISE 5. What would you expect last[:] to return?

One thing that comes up all the time is the need to create strings out of other strings and numbers, eg to create filenames from a combination of a base directory, some base filename, and some numbers. Scientists like to create lots of data files like and then write code to loop over these files and analyze them. We're going to show how to do that, starting with the newbie way and progressively building up to the way of python zen master. All of the methods below *work*, but the zen master way will more efficient, more scalable (eg to larger numbers of files) and cross-platform.[9] Here's the newbie way: we also introduce the for-loop here in the spirit of diving into python – note that python uses whitespace indentation to delimit the for-loop code block

```
# The newbie way
for i in (1,2,3,4):
    fname = 'data/myexp0' + str(i) + '.dat'
    print fname
```

Now as promised, this will print out the 4 file names above, but it has three flaws: it doesn't scale to 10 or more files, it is inefficient, and it is not cross platform. It doesn't scale because it hard-codes the '0' after `myexp`, it is inefficient because to add several strings requires the creation of temporary strings, and it is not cross-platform because it hard-codes the directory separator '/'.

[9] "But it works" is a common defense of bad code; my rejoinder to this is "A computer scientist is someone who fixes things that aren't broken".

```
# On the path to elightenment
for i in (1,2,3,4):
    fname = 'data/myexp%02d.dat'%i
    print fname
```

This example uses string interpolation, the funny % thing. If you are familiar with C programming, this will be no surprise to you (on linux/unix systems do `man sprintf` at the unix shell). The percent character is a string formatting character: `%02d` means to take an integer (the `d` part) and print it with two digits, padding zero on the left (the `%02` part). There is more to be said about string interpolation, but let's finish the job at hand. This example is better than the newbie way because is scales up to files numbered 0-99, and it is more efficient because it avoids the creation of temporary strings. For the platform independent part, we go to the python standard library `os.path`, which provides a host of functions for platform-independent manipulations of filenames, extensions and paths. Here we use `os.path.join` to combine the directory with the filename in a platform independent way. On windows, it will use the windows path separator '\' and on unix it will use '/'.

```
# the zen master approach
import os
for i in (1,2,3,4):
    fname = os.path.join('data', 'myexp%02d.dat'%i)
    print fname
```

EXERCISE 6. Suppose you have data files named like

```
data/2005/exp0100.dat
data/2005/exp0101.dat
data/2005/exp0102.dat
...
data/2005/exp1000.dat
```

Write the python code that iterates over these files, constructing the filenames as strings in using `os.path.join` to construct the paths in a platform-independent way. *Hint*: read the help for `os.path.join`!

OK, I promised to torture you a bit more with string interpolation – don't worry, I remembered. The ability to properly format your data when printing it is crucial in scientific endeavors: how many signficant digits do you want, do you want to use integer, floating point representation or exponential notation? These three choices are provided with `%d`, `%f` and `%e`, with lots of variations on the theme to indicate precision and more

```
>>> 'warm for %d minutes at %1.1f C' % (30, 37.5)
'warm for 30 minutes at 37.5 C'
>>> 'The mass of the sun is %1.4e kg'% (1.98892*10**30)
'The mass of the sun is 1.9889e+30 kg'
```

There are two string methods, `split` and `join`, that arise frequenctly in numerical processing, specifically in the context of processing data files that have comma, tab, or space separated numbers in them. `split` takes a single string, and splits it on the indicated character to a sequence of strings. This is useful to take a single line of space or comma separated values and split them into individual numbers

```
# s is a single string and we split it into a list of strings
# for further processing
>>> s = '1.0 2.0 3.0 4.0 5.0'
>>> s.split(' ')
['1.0', '2.0', '3.0', '4.0', '5.0']
```

The return value, with square brackets, indicates that python has returned a list of strings. These individual strings need further processing to convert them into actual floats, but that is the first step. The conversion to floats will be discussed in the next session, when we learn about list comprehensions. The converse method is join, which is often used to create string output to an ASCII file from a list of numbers. In this case you want to join a list of numbers into a single line for printing to a file. The example below will be clearer after the next section, in which lists are discussed

```
# vals is a list of floats and we convert it to a single
# space separated string
>>> vals = [1.0, 2.0, 3.0, 4.0, 5.0]
>>> ' '.join([str(val) for val in vals])
'1.0 2.0 3.0 4.0 5.0'
```

There are two new things in the example above. One, we called the join method directly on a string itself, and not on a variable name. Eg, in the previous examples, we always used the name of the object when accessing attributes, eg `x.real` or `s.upper()`. In this example, we call the `join` method on the string which is a single space. The second new feature is that we use a list comprehension `[str(val) for val in vals]` as the argument to `join`. `join` requires a sequence of strings, and the list comprehension converts a list of floats to a strings. This can be confusing at first, so don't dispair if it is. But it is worth bringing up early because list comprehensions are a very useful feature of python. To help elucidate, compare `vals`, which is a list of floats, with the conversion of `vals` to a list of strings using list comprehensions in the next line

```
# converting a list of floats to a list of strings
>>> vals
[1.0, 2.0, 3.0, 4.0, 5.0]
>>> [str(val) for val in vals]
['1.0', '2.0', '3.0', '4.0', '5.0']
```

7. The basic python data structures

Strings, covered in the last section, are sequences of characters. python has two additional built-in sequence types which can hold arbitrary elements: tuples and lists. tuples are created using parentheses, and lists are created using square brackets

```
# a tuple and a list of elements of the same type
# (homogeneous)
>>> t = (1,2,3,4)  # tuple
>>> l = [1,2,3,4]  # list
```

Both tuples and lists can also be used to hold elements of different types

```
# a tuple and list of int, string, float
>>> t = (1,'john', 3.0)
>>> l = [1,'john', 3.0]
```

Tuples and lists have the same indexing and slicing rules as each other, and as string discussed above, because both implement the python sequence protocol, with the only difference being that tuple slices return tuples (indicated by the parentheses below) and list slices return lists (indicated by the square brackets)

```
# indexing and slicing tuples and lists
>>> t[0]
1
>>> l[0]
1
>>> t[:-1]
(1, 'john')
>>> l[:-1]
[1, 'john']
```

So why the difference between tuples and lists? A number of explanations have been offered on the mailing lists, but the only one that makes a difference to me is that tuples are immutable, like strings, and hence can be used as keys to python dictionaries and included as elements of sets, and lists are mutable, and cannot. So a tuple, once created, can never be changed, but a list can. For example, if we try to reassign the first element of the tuple above, we get an error

```
>>> t[0] = 'why not?'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

But the same operation is perfectly accetable for lists

```
>>> l[0] = 'why not?'
>>> l
['why not?', 'john', 3.0]
```

lists also have a lot of methods, tuples have none, save the special double underscore methods that are required for python objects and sequences

```
# tuples contain only "hidden" double underscore methods
>>> dir(t)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__ge__', '_
# but lists contain other methods, eg append, extend and
# reverse
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '_
```

Many of these list methods change, or mutate, the list, eg append adds an element to the list: `extend` extends the list with a sequence of elements, `sort` sorts the list in place, `reverse` reverses it in place, `pop` takes an element off the list and returns it.

We've seen a couple of examples of creating a list above – let's look at some more using list methods

```
>>> x = []                      # create the empty list
>>> x.append(1)                 # add the integer one to it
>>> x.extend(['hi', 'mom'])     # append two strings to it
>>> x
[1, 'hi', 'mom']
>>> x.reverse()                 # reverse the list, in place
>>> x
['mom', 'hi', 1]
>>> len(x)
3
```

We mentioned list comprehensions in the last section when discussing string methods. List comprehensions are a way of creating a list using a for loop in a single line of python. Let's create a list of the perfect cubes from 1 to 10, first with a for loop and then with a list comprehension. The list comprehension code will not only be shorter and more elegant, it can be much faster (the dots are the indentation block indicator from the python shell and should not be typed)

```
# a list of perfect cubes using a for-loop
>>> cubes = []
>>> for i in range(1,10):
...     cubes.append(i**3)
...
>>> cubes
[1, 8, 27, 64, 125, 216, 343, 512, 729]
# functionally equivalent code using list comprehensions
>>> cubes = [i**3 for i in range(1,10)]
>>> cubes
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```

The list comprehension code is faster because it all happens at the C level. In the simple for-loop version, the python expression which appends the cube of `i` has to be evaluated by the python interpreter for each element of the loop. In the list comprehension example, the single line is parsed once and executed at the C level. The difference in speed can be considerable, and the list comprehension example is shorter and more elegant to boot.

The remaining essential built-in data strucuture in python is the dictionary, which is an associative array that maps arbitrary immutable objects to arbitrary objects. int, long, float, string and tuple are all immutable and can be used as keys; to a dictionary list and dict are mutable and cannot. A dictionary takes one kind of object as the key, and this key points to another object which is the value. In a contrived but easy to comprehent examples, one might map names to ages

```
>>> ages = {}             # create an empty dict
>>> ages['john'] = 36
>>> ages['fernando'] = 33
>>> ages                  # view the whole dict
{'john': 36, 'fernando': 33}
>>> ages['john']
36
>>> ages['john'] = 37     # reassign john's age
>>> ages['john']
37
```

Dictionary lookup is very fast; Tim Peter's once joked that any python program which uses a dictionary is automatically 10 times faster than any C program, which is of course false, but makes two worthy points in jest: dictionary lookup is fast, and dictionaries can be used for important optimizations, eg, creating a cache of frequently used values. As a simple eaxample, suppose you needed to compute the product of two numbers between 1 and 100 in an inner loop – you could use a dictionary to cache the cube of all odd of numbers < 100; if you were inteterested in all numbers, you might simply use a list to store the cached cubes – I am cacheing only the odd numbers to show you how a dictionary can be used to represent a sparse data structure

```
>>> cubes = dict([ ( i, i**3 ) for i in range(1,100,2)])
>>> cubes[5]
125
```

The last example is syntactically a bit challenging, but bears careful study. We are initializing a dictionary with a list comprehension. The list comprehension is made up of length 2 tuples `( i, i**3 )`. When a dictionary is initialized with a sequence of length 2 tuples, it assumes the first element of the tuple `i` is the *key* and the second element i**3is the *value*. Thus we have a lookup table from odd integers to to cube. Creating dictionaries from list comprehensions as in this example is something that hard-core python programmers do almost every day, and you should too.

EXERCISE 8. Create a lookup table of the product of all pairs of numbers less than 100. The key will be a tuple of the two numbers `(i,j)` and the value will be the product. Hint: you can loop over multiple ranges in a list comprehension, eg `[ something for i in range(Ni) for j in range(Nj)]`

9. The Zen of Python

EXERCISE 10. `>>> import this`

11. Functions and classes

You can define functions just about anywhere in python code. The typical function definition takes zero or more arguments, zero or more keyword arguments, and is followed by a documentation string and the function definition, optionally returing a value. Here is a function to compute the hypoteneuse of a right triange

```
def hypot(base, height):
   'compute the hypoteneuse of a right triangle'
   import math
   return math.sqrt(base**2 + height**2)
```

As in the case of the for-loop, leading white space is significant and is used to delimt the start and end of the function. In the example below, x = 1 is not in the function, because it is not indented

```
def growone(l):
   'append 1 to a list l'
   l.append(1)
x = 1
```

Note that this function does not return anything, because the append method modifies the list that was passed in. You should be careful when designing functions that have side effects such as modifying the structures that are passed in; they should be named and documented in such a way that these side effects are clear.

Python is pretty flexible with functions: you can define functions within function definitions (just be mindful of your indentation), you can attach attributes to functions (like other objects), you can pass functions as arguments to other functions. A function keyword argument defines a default value for a function that can be overridden. Below is an example which provides a normalize keyword argument. The default argument is `normalize=None`; the value None is a standard python idiom which usually means either do the default thing or do nothing. If `normalize` is not `None`, we assume it is a function that can be called to normalize our data

```
def psd(x, normalize=None):
    'compute the power spectral density of x'
    if normalize is not None: x = normalize(x)
    # compute the power spectra of x and return it
```

This function could be called with or without a `normalize` keyword argument, since if the argument is not passed, the default of `None` is used and no normalization is done.

```
# no normalize argument; do the default thing
>>> psd(x)
# define a custom normalize function unitstd as pass it
# to psd
>>> def unitstd(x): return x/std(x)
>>> psd(x, normalize=unitstd)
```

In Section2 we noticed that complex objects have the real and imag data attributes, and the conjugate method. An object is an instance of a class that defines it, and in python you can easily define your own classes. In that section, we emphasized that one of the important features of a classes/objects is that they carry around their data and methods in a single bundle. Let's look at the mechnics of defining classes, and creating instances (a.k.a. objects) of these classes. Classes have a special double underscore method `__init__` that is used as the function to initialize the class. For this example, we'll continue with the normalize theme above, but in this case the normalization requires some data parameters. This example arises when you want to normalize an image which may range over 0-255 (8 bit image) or from 0-65535 (16 bit image) to the 0-1 interval. For 16 bit images, you would normally divide everything by 65525, but you might want to configure this to a smaller number if your data doesn't use the whole intensity range to enhance contrast. For simplicitly, let's suppose our normalize class is only interested in the pixel maximum, and will divide all the data by that value.

```
from __future__ import division  # make sure we do float division
class Normalize:
    """
    A class to normalize data by dividing it by a maximum value
    """
    def __init__(self, maxval):
        'maxval will be mapped to 1'
        self.maxval = maxval
    def __call__(self, data):
        'do the normalization'
        # in real life you would also want to clip all values of
        # data>maxval so that the returned value will be in the unit
        # interval
        return data/self.maxval
```

The triple quoted string following the definition of class Normalize is the class documentation stringd, and it will bre shown to the user when they do `help(Normalize)`. A commonly used convention is to name classes with *UpperCase*, but this is not required. self is a special variable that a class can use to refer to its own data and methods, and must be the first argument to all the class methods. The `__init__` method stores the normalization value maxval as a class attribute in `self.maxval`, and this value can later be reused by other class methods (as it is in `__call__`) and it can be altered by the user of the class, as will illustrate below. The `__call__` method is another piece of python double underscore magic, it allows class instances to be used as *functions*, eg you can call them just like you can call any function. OK, now let's see how you could use this.

The first line use used to create an *instance* of the *class* `Normalize`, and the special method `__init__` is implicitly called. The second line implicitly calls the special `__call__`method

```
>>> norm = Normalize(65356) # good for 16 bit images
>>> norm(255)               # call this function
0.0039017075708427688
# We can reset the maxval attribute, and the call method
# is automagically updated
>>> norm.maxval = 255       # reset the maxval
>>> norm(255)               # and call it again
1.0
# We can pass the norm instance to the psd function we defined above, which
# is expecting a function
>>> pdf(X, normalize=norm)
```

Exercise 12. Pretend that `complex` were not built-in to the python core, and write your own complex class `MyComplex`. Provide `real` and `imag` attributes and the `conjugate` method. Define `__abs__`, `__mul__` and `__add__` to implement the absolute value of complex numbers, multiplication

of complex numbers and addition of complex numbers. See the API definition of the python number protocol; although this is written for C programmers, it contains information about the required function call signatures for each of the double underscore methods that define the number protocol in python; where they use `o1` on that page, you would use `self` in python, and where they use `o2` you might use `other` in python.[10] To get you started, I'll show you what the `__add__` method should look like

```
# An example double underscore method required in your MyComplex
# implementation
def __add__(self, other):
    'add self to other and return a new MyComplex instance'
    r = self.real + other.real
    i = self.imag + other.imag
    return MyComplex(r,i)
# When you are finished, test your implementation with
>>> x = MyComplex(2,3)
>>> y = MyComplex(0,1)
>>> x.real
2.0
>>> y.imag
1.0
>>> x.conjugate()
(2-3j)
>>> x+y
(2+4j)
>>> x*y
(-3+2j)
>>> abs(x*y)
3.6055512754639891
```

13. Files and file like objects

Working with files is one of the most common and important things we do in scientific computing because that is usually where the data lives. In Section4, we went through the mechanics of automatically building file names like

```
data/myexp01.dat
data/myexp02.dat
data/myexp03.dat
data/myexp04.dat
```

but we didn't actually do anything with these files. Here we'll show how to read in the data and do something with it. Python makes working with files easy and dare I say fun. The test data set lives in `data/family.csv` and is a standard comma separated value file that contains information about my family: first name, last name, age, height in cm, weight in kg and birthdate. We'll open this file and parse it – note that python has a standard module for parsing CSV files that is much more sophisticated than what I am doing here. Nevertheless, it serves as an easy to understand example that is close enough to real life that it is worth doing. Here is what the data file looks like

```
First,Last,Age,Weight,Height,Birthday
John,Hunter,36,175,180,1968-03-05
Miriam,Sierig,33,135,177,1971-05-04
Rahel,Hunter,7,55,134,1998-02-25
Ava,Hunter,3,45,121,2001-04-26
Clara,Hunter,0,15,55,2004-10-02
```

Here is the code to parse that file

```
# open the file for reading
fh = file('../data/family.csv', 'r')
# slurp the header, splitting on the comma
headers = fh.readline().split(',')
```

[10] http://www.python.org/doc/current/api/number.html

```
# now loop over the remaining lines in the file and parse them
for line in fh:
    # remove any leading or trailing white space
    line = line.strip()
    # split the line on the comma into separate variables
    first, last, age, weight, height, dob = line.split(',')
    # convert some of these strings to floats
    age, weight, height = [float(val) for val in (age, weight, height)]
    print first, last, age, weight, height, dob
```

This example illustrates several interesting things. The syntax for opening a file is `file(filename, mode)` and the `mode` is a string like `'r'` or `'w'` that determines whether you are opening in read or write mode. You can also read and write binary files with `'rb'` and `'wb'`. There are more options and you should do `help(file)` to learn about them. We then use the file `readline` method to read in the first line of the file. This returns a string (the line of text) and we call the string method `split(',')` to split that string wherever it sees a comma, and this returns a list of strings which are the headers

```
>>> headers
['First', 'Last', 'Age', 'Weight', 'Height', 'Birthday\n']
```

The new line character `'\n'` at the end of `'Birthday\n'` indicates we forgot to strip the string of whitespace. To fix that, we should have done

```
>>> headers = fh.readline().strip().split(',')
>>> headers
['First', 'Last', 'Age', 'Weight', 'Height', 'Birthday']
```

Notice how this works like a pipeline: `fh.readline` returns a line of text as a string; we call the string method `strip` which returns a string with all white space (spaces, tabs, newlines) removed from the left and right; we then call the `split` method on this stripped string to split it into a list of strings.

Next we start to loop over the file – this is a nice feature of python file handles, you can iterate over them as a sequence. We've learned our lesson about trailing newlines, so we first strip the line with `line = line.strip()`. The rest is string processing, splitting the line on a comma as we did for the headers, and converting the strings to numbers where approriate by calling `float(val)` for each of `age`, `weight` and `height`. Notice how we use list comprehensions and tuple unpacking – the age, weight, `height = [float(val) for val in (age, weight, height)]` line, to convert several values at once.

Now that we have all this data, how mught we store it. We could store it in a `results` list

```
results = []
for line in fh:
    # process the line as above to get the variables
    results.append( (first, last, age, weight, height, dob) )
# and later when we want to analyze the data
for first, last, age, weight, height, dob in results:
    # do something with the data
```

Exercise 14. `zip` magic. Python has a nice funcion `zip` that lets you do very useful things with lists of tuples. `results` above is a list of tuples – each tuple is the `first`, `last`, `age`, `weight`, `height`, `dob` for a family member. What happens if you do

```
>>> first, last, age, weight, height, dob = zip(*results)
```

What is `age` now?

Exercise 15. Write a class `Person` and store the attributes `first`, `last`, `age`, `weight`, `height`, `dob` in that class. Add a class instance to the results list, eg

```
results.append(Person(first, last, age, weight, height, dob))
```

Python also has a special syntax for printing to an open writable file object

```
# open the file for writing
outfile = file('mydata.data', 'w')
for x,y,z in myresults:
    print >> outfile, '%1.3f %1.3f %1.3f'%(x,y,z)
```

Another really nice thing about file objects is that other classes can implement the file protcol and allow you to use them as if they were files. For example, the StringIO module in the standard library allows you

to read and write to strings as if they were files. The urllib.urlopen function allows you to open a remove web page as a file object. Try this

```
# loop over the lines in google's html
from urllib import urlopen
for line in urlopen('http://www.google.com').readlines():
    print line,
```

CHAPTER 3

A tour of IPython

One of Python's most useful features is its interactive interpreter. This system allows very fast testing of ideas without the overhead of creating test files as is typical in most programming languages. In scientific computing, one of the reasons behind the popularity of systems like Matlab ™, IDL ™or Mathematica ™, is precisely their interactive nature. Scientific computing is an inherently exploratory problem domain, where one is rarely faced with writing a program against a set of well-defined explicit constraints. Being able to load data, process it with different algorithms or test parameters, visualize it, save results, and do all of this in a fluid and efficient way, can make a big productivity difference in day to day scientific work. Even for the development of large codes, a good interactive interpreter can be a major asset, though this is a less commonly held view; later in this document we will discuss this aspect of the problem.

However, the interpreter supplied with the standard Python distribution is somewhat limited for extended interactive use. The IPython project [29] was born out of a desire to have a better Python interactive environment, which could combine the advantages of the Python language with some of the best ideas found in systems like IDL or Mathematica, along with many more enhancements. IPython is a free software project (released under the BSD license) which tries to:

(1) Provide an interactive shell superior to Python's default. IPython has many features for object introspection, system shell access, and its own special command system for adding functionality when working interactively. It tries to be a very efficient environment both for Python code development and for exploration of problems using Python objects (in situations like data analysis).
(2) Serve as an embeddable, ready to use interpreter for your own programs. IPython can be started with a single call from inside another program, providing access to the current namespace. This can be very useful both for debugging purposes and for situations where a blend of batch-processing and interactive exploration are needed.
(3) Offer a flexible framework which can be used as the base environment for other systems with Python as the underlying language. Specifically scientific environments like Mathematica, IDL and Matlab inspired its design, but similar ideas can be useful in many fields.

This document is not meant to replace the comprehensive IPython manual, which ships with the IPython distribution and is also available online at http://ipython.scipy.org/doc/manual. Instead, we will present here some relevant parts of it for everyday use, and refer readers to the full manual for in-depth details.

Additionally, this article by Jeremy Jones provides an introductory tutorial about IPython: http://www.onlamp.com/pub/a/python/2005/01/27/ipython.html.

1. Main IPython features

This section summarizes the most important user-visible features of IPython, which are not a part of the default Python shell or other interactive Python systems. While you can use IPython as a straight replacement for the normal Python shell, a quick read of these will allow you to take advantage of many enhancements which can be very useful in everyday work.

A bird's eye view of IPython's feature set:

- Dynamic object introspection. You can access docstrings, function definition prototypes, source code, source files and other details of any object accessible to the interpreter with a single keystroke ('?'). Adding a second ? produces more details when possible.
- Completion in the local namespace, via the TAB key. This works for keywords, methods, variables and files in the current directory. TAB-completion, especially for attributes, is a convenient way to explore the structure of any object you're dealing with. Simply type object_name.<TAB> and a list of the object's attributes will be printed.

- Numbered input/output prompts with command history (persistent across sessions and tied to each profile), full searching in this history and caching of all input and output.
- User-extensible 'magic' commands. A set of commands prefixed with `%` is available for controlling IPython itself and provides directory control, namespace information and many aliases to common system shell commands.
- Alias facility for defining your own system aliases.
- Complete system shell access. Lines starting with ! are passed directly to the system shell, and using !! captures shell output into python variables for further use.
- The ability to expand python variables when calling the system shell. In a shell command, any python variable prefixed with `$` is expanded. A double `$$` allows passing a literal `$` to the shell (for access to shell and environment variables like `$PATH`).
- Filesystem navigation, via a magic `%cd` command, along with a persistent bookmark system (using `%bookmark`) for fast access to frequently visited directories.
- A macro system for quickly re-executing multiple lines of previous input with a single name, implemented via the `%macro` magic command.
- Session logging and restoring via the `%logstart`, `%logon/off` and `%logstate` magics. You can then later use these log files as code in your programs.
- Verbose and colored exception traceback printouts. Easier to parse visually, and in verbose mode they produce a lot of useful debugging information.
- Auto-parentheses: callable objects can be executed without parentheses: `'sin 3'` is automatically converted to `'sin(3)'`.
- Auto-quoting: using `','` as the first character forces auto-quoting of the rest of the line: `',my_function a b'` becomes automatically `'my_function("a","b")'`.
- Flexible configuration system. It uses a configuration file which allows permanent setting of all command-line options, module loading, code and file execution. The system allows recursive file inclusion, so you can have a base file with defaults and layers which load other customizations for particular projects.
- Embeddable. You can call IPython as a python shell inside your own python programs. This can be used both for debugging code or for providing interactive abilities to your programs with knowledge about the local namespaces (very useful in debugging and data analysis situations).
- Easy debugger access. You can set IPython to call up the Python debugger (pdb) every time there is an uncaught exception. This drops you inside the code which triggered the exception with all the data live and it is possible to navigate the stack to rapidly isolate the source of a bug. The `%run` magic command –with the `-d` option– can run any script under `pdb`'s control, automatically setting initial breakpoints for you.
- Profiler support. You can run single statements (similar to `profile.run()`) or complete programs under the profiler's control. While this is possible with the standard `profile` module, IPython wraps this functionality with magic commands (see `'%prun'` and `'%run -p'`) convenient for rapid interactive work.

2. Effective interactive work

IPython has been designed to try to make interactive work as fluid and efficient as possible. All of its features try to maximize the output-per-keystroke, so that as you work at an interactive console, minimal typing produces results. It makes extensive use of the readline library, has its own control system (magics), caches previous inputs and outputs, has a macro system, etc. Becoming familiar with these features, while not necessary for basic use, will make long-term use of the system much more pleasant and productive.

2.1. Magic functions. The default Python interactive shell only allows valid Python code to be typed at its input prompt. While this appears like a reasonable approach in principle, in practical use it turns out to be rather limiting. A good interactive environment should allow you to control the environment itself, in hopefully the most typing-efficient way.

Verbosity in code is a good thing, since code is a long-lived entity, and deciphering three-letter acronyms for variable names, 6 months after a program was written, is typically an exercise in frustration. However at an interactive prompt, where every keystroke counts and things are not meant to be permanent, compact and efficient control of your environment is an important feature. The default Python shell does not offer this, and the Python language's verbosity, which is an asset for the long-term readability of code, becomes a bit of a liability in this context.

For this reason, IPython offers a system of 'magic' commands, which serve to control IPython itself and perform a number of common tasks. Users of IDL will be familiar with the 'dot' commands, like

.stop, which perform similar functions in that system. In IPython, the magic system covers much more functionality and is fully user-extensible. This allows users to add all the control they may desire to their everyday working environment.

The magics system is patterned after the time-honored Unix shells, with whitespace separating arguments, no parentheses required, and dashes for specifying options to commands. Many builtin magics also are named like the Unix commands they mimic, so that an IPython environment can be used 'out of the box' by any Unix user with ease.

IPython will treat any line whose first character is a % as a special call to a magic function. For example: typing '%cd mydir' (without the quotes) changes you working directory to 'mydir', if it exists. For any magic function, typing its name followed by ? will show you the magic's information and docstring, just like for other regular Python objects. Simply typing magic at the prompt will print an overview of the system, and a list of all the existing magics with their docstrings.

If you have 'automagic' enabled, you don't need to type in the % explicitly. Automagic is enabled by default, and you can configure this in your ipythonrc file, via the command line option -automagic or even toggle it at runtime with the %automagic function. IPython will scan its internal list of magic functions and call one if it exists. With automagic on you can then just type 'cd mydir' to go to directory 'mydir'. The automagic system has the lowest possible precedence in name searches, so defining an identifier with the same name as an existing magic function will shadow it for automagic use. You can still access the shadowed magic function by explicitly using the % character at the beginning of the line.

An example (with automagic on) should clarify all this:

```
In [1]: cd ipython # %cd is called by automagic
/home/fperez/ipython
In [2]: cd = 1 # now cd is just a variable
In [3]: cd .. # and doesn't work as a function anymore
------------------------------------------------------------
    File "<console>", line 1
      cd ..
         ^
SyntaxError: invalid syntax
In [4]: %cd .. # but %cd always works
/home/fperez
In [5]: del cd # if you remove the cd variable
In [6]: cd ipython # automagic can work again
/home/fperez/ipython
```

2.2. Object exploration. Python is a language with exceptional introspection capabilities. This means that, within the language itself, it is possible to extract a remarkable amount of information about all objects currently in memory. However the default Python shell exposes very little of this power in an easy to use manner; IPython provides a lot of functionality to remedy this.

The bulk of IPython's introspection system is accessible via only two keys: the question mark ? and the <TAB> key. Under the hood, these two keys control a fairly complex set of libraries which ultimately rely on the readline and inspect modules from the Python standard library. But for regular use, you should never need to remember anything beyond these two. As an example, consider defining a variable named mylist, which starts as an empty list:

```
In [1]: mylist=[]
```

now you can find out some things about it by using the question mark:

```
In [2]: mylist?
Type:           list
Base Class:     <type 'list'>
String Form:    []
Namespace:      Interactive
Length:         0
Docstring:
    list() -> new list
    list(sequence) -> new list initialized from sequence's items
```

next, by adding a period (the standard Python attribute separator) and hitting TAB, IPython will show you all the attributes which this object has:

```
In [3]: mylist.<The TAB key was pressed here>
```

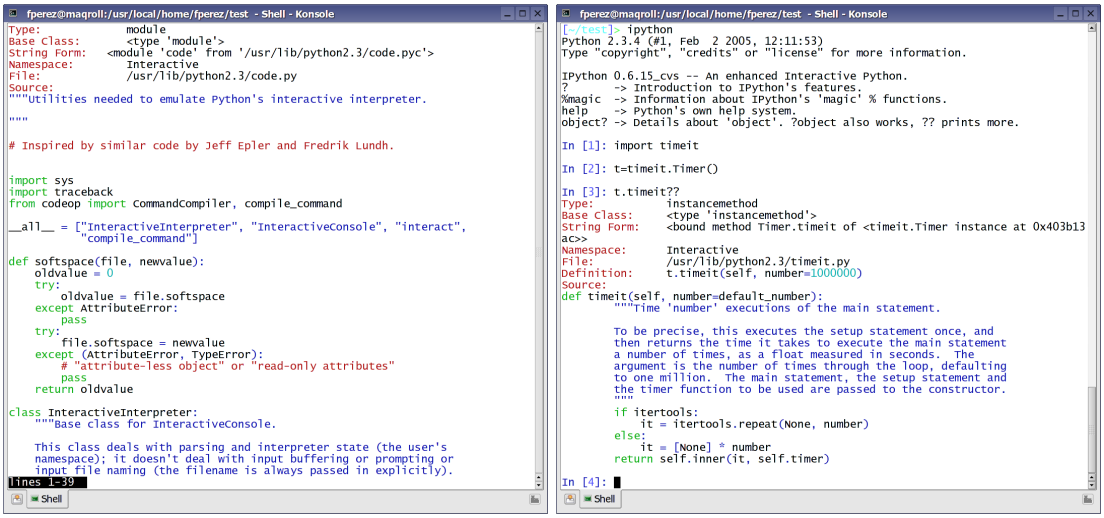

FIGURE 1. IPython can show syntax-highlighted source code for objects whose source is available.

```
mylist.append   mylist.extend   mylist.insert   mylist.remove mylist.sort
mylist.count    mylist.index    mylist.pop      mylist.reverse
```

you can then request further details about any of them:

```
In [3]: mylist.append?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in method append of list object at 0x403b2b6c>
Namespace:      Interactive
Docstring:
    L.append(object) -- append object to end
```

The `?` system can be doubled. The first screenshot in Fig. 1 was generated by typing at the IPython prompt:

```
In [1]: import code
In [2]: code??
```

Using `??` shows the syntax-highlighted source for the `code` module from the Python standard library. This is an excellent way to explore modules or objects which you are not familiar with. As long as Python's `inspect` system is capable of finding the source code for an object, IPython will show it to you, with nice syntax highlights.

This can be done for entire modules, as in the prvious example, for individual functions, or even methods of object instances. The second screenshot in the same figure shows source for the `timeit` method of a `timeit.Timer` object.

The magic commands `%pdoc`, `%pdef`, `%psource` and `%pfile` will respectively print the docstring, function definition line, full source code and the complete file for any object (when they can be found).

2.3. Input and Ouptut cached prompts. In IPython, all output results are automatically stored in a global dictionary named `Out` and variables named `_1`, `_2`, etc. alias them. For example, the result of input line 4 is available either as `Out[4]` or as `_4`. Additionally, three variables named `_`, `__` and `___` are always kept updated with the for the last three results. This allows you to recall any previous result and further use it for new calculations. For example:

```
In [1]: 2+4
Out[1]: 6
In [2]: _+9
Out[2]: 15
In [3]: _+__
Out[3]: 21
In [4]: print _1
```

```
6
In [5]: print Out[1]
6
In [6]: _2**3
Out[6]: 3375
```

You can put a ';' at the end of a line to supress the printing of output. This is useful when doing calculations which generate long output you are not interested in seeing. The _* variables and the Out[] list do get updated with the contents of the output, even if it is not printed. You can thus still access the generated results this way for further processing.

A similar system exists for caching input. All input is stored in a global list called In , so you can re-execute lines 22 through 28 plus line 34 by typing 'exec In[22:29]+In[34]' (using Python slicing notation).

At any time, your input history remains available. The %hist command can show you all previous input, without line numbers if desired (option -n) so you can directly copy and paste code either back in IPython or in a text editor. You can also save all your history by turning on logging via %logstart; these logs can later be either reloaded as IPython sessions or used as code for your programs.

If you need to execute the same set of lines often, you can assign them to a macro with the %macro magic function. Macros are simply short names for groups of input lines, which can be re-executed by only typing that name. Typing macro? at the prompt will show you the function's full documentation. For example, if your history contains:

```
44: x=1
45: y=3
46: z=x+y
47: print x
48: a=5
49: print 'x',x,'y',y
```

You can create a macro with lines 44 through 47 (included) and line 49 called my_macro with:

```
In [51]: %macro my_macro 44:48 49
```

Now, simply typing my_macro will re-execute all this code in one pass. The number range follows standard Python list slicing notation, where n:m means the numbers $(n, n+1, \ldots, m-1)$.

You should note that macros execute in the current context, so if any variable changes, the macro will pick up the new value every time it is executed:

```
In [1]: x=1
In [2]: y=x*5
In [3]: z=x+3
In [4]: print 'y is:',y,'and z is:',z
y is: 5 and z is: 4
# make a macro with lines 2,3,4 (note Python list slice syntax):
In [5]: macro yz 2:5
Macro `yz` created. To execute, type its name (without quotes).
Macro contents:
y=x*5
z=x+3
print 'y is:',y,'and z is:',z
# now, run the macro directly:
In [6]: yz
Out[6]: Executing Macro...
y is: 5 and z is: 4
# we change the value of x
In [7]: x=9
# and now if we rerun the macro, we get the new values:
In [8]: yz
Out[8]: Executing Macro...
y is: 45 and z is: 12
```

2.4. Running code. The %run magic command allows you to run any python script and load all of its data directly into the interactive namespace. %run is a sophisticated wrapper around the Python execfile() builtin function; since the file is re-read from disk each time, changes you make to it are

reflected immediately (in contrast to the behavior of `import`). I rarely use `import` for code I am testing, relying on `%run` instead.

By default,

```
%run myfile arg1 arg2 ...
```

executes `myfile` in a namespace initially consisting only of `__name__=='__main__'` and `sys.argv` being filled with arg1, arg2, etc. This means that using `%run` is functionally very simlar to executing a script at the system command line, but you get all the functionality of IPython (better tracebacks, debugger and profiler access, etc.). The `-n` option prevents `__name__` from being set equal to `'__main__'`, in case you want to test the part of a script which only runs when `imported`.

Additionally, the fact that IPython then updates your interactive namespace with the variables defined in the script is very useful, because you can run your code to do a lot of processing, and then continue using and exploring interactively the objects created by the program.

For example, if the file `ip_simple.py` contains:

```
import sys
print 'sys.argv is:',sys.argv
print '__name__ is:',__name__
x = 1
```

you can run it in IPython as follows:

```
# First, let's check that x is undefined
In [1]: x
---------------------------------------------------------------------------
exceptions.NameError                                 Traceback (most recent call last)
/usr/local/home/fperez/teach/course/problems/<console>
NameError: name 'x' is not defined
# Now we run the script (the .py extension is optional):
In [2]: run ip_simple
sys.argv is: ['ip_simple.py']
__name__ is: __main__
# If we print x, now it has the value from the script
In [3]: x
Out[3]: 1
# Again, but now running with some arguments:
In [4]: run ip_simple -x arg1 "hello world"
sys.argv is: ['ip_simple.py', '-x', 'arg1', 'hello world']
__name__ is: __main__
```

With the `-i` option, the namespace where your script runs is actually your interactive one. This can be used for two sligthly different purposes. The simpler case, is just to quickly type up a set of commands in an editor which you want to execute on your current environment (although the `%edit` command can also be used for this). Consider running the file `ip_simple2.py`:

```
"""This simple file prints a variable which is NOT defined here.
It should be run via IPython's %run with the -i option."""
print 'x is:',x
```

in IPython:

```
# A regular %run will produce an error:
In [1]: run ip_simple2
---------------------------------------------------------------
exceptions.NameError              Traceback (most recent call last)
/usr/local/home/fperez/teach/course/problems/ip_simple2.py
      2
      3 It should be run via IPython's %run with the -i option."""
      4
----> 5 print 'x is:',x
      6
NameError: name 'x' is not defined
WARNING: Failure executing file: <ip_simple2.py>
x is:
```

```
# However, if you do have a variable x defined:
In [2]: x='hello'
# you can use the -i option and the code will see x:
In [3]: run -i ip_simple2
x is: hello
```

A different use of `%run -i`, is to repeatedly run scripts which may have a potentially expensive initialization phase. If this initialization does not need to be repeated on each run (for example, you are debugging some other submodule and can reuse the same expensive object several times), you can avoid it by protecting the expensive object with a `try/except` block. This simple script illustrates the technique:

```
"""Example script with an expensive initialization.

Meant to be used via ipython's %run -i, though it can run standalone."""

# Imagine that bigobject is actually something whose creation is an expensive
# process, though here we are just going to make it a list of numbers for
# demonstration's sake.  The trick is to trap a test for the existence of this
# name in a try/except block.  If the object exists, we don't recreate it, if
# it doesn't exist yet (such as the first time the code is run in any given
# session), we make it.

try:
    bigobject
    print "We found bigobject! No need to initialize it."
except NameError:
    print "bigobject not found, performing expensive initialization..."
    bigobject = range(1000)

# And now you can move on with working on bigobject:
total = sum(bigobject)
print 'total is:',total
```

In IPython, here is how you can use it:

```
# The first time it runs, it will have to initialize
In [1]: run -i ip_expensive_init.py
bigobject not found, performing expensive initialization...
total is: 499500
# but successive runs don't require initialization
In [2]: run -i ip_expensive_init.py
We found bigobject! No need to initialize it.
total is: 499500
# you can still run without -i, to achieve a full reload
# if you need it for any reason
In [3]: run ip_expensive_init.py
bigobject not found, performing expensive initialization...
total is: 499500
```

In the third run, by not using `-i`, your script runs in an empty namespace and this forces a full initialization (the `NameError` exception is triggered).

`%run` also has special flags for timing the execution of your scripts (`-t`) and for executing them under the control of either Python's `pdb` debugger (`-d`) or profiler (`-p`). You can get all of its docstring with the usual `run?` mechanism.

Thanks to all of its various control options, `%run` can be used as the main tool for efficient interactive development of code which you write in your editor of choice. My personal operation mode, which has served me well for several years of scientific work in Python, is to have a good editor (XEmacs in my case) open with all my Python code, and IPython open in a terminal where I run, debug, explore, plot, etc.

3. Access to the underlying Operating System

3.1. Basic usage. IPython allows you to always access the underlying OS very easily. Any lines starting with `!` are passed directly to the system shell:

```
In [6]: !ls ip*.py
ip_expensive_init.py  ip_simple2.py  ip_simple.py
```

and using `!!` captures shell output into python variables for further use:

```
In [7]: !!ls ip*.py
Out[7]: ['ip_expensive_init.py', 'ip_simple2.py', 'ip_simple.py']
```

There is a difference between the two cases: in the first, the `ls` command simply prints its results to the terminal as text, but no value is returned. In the second, IPython actually captures the output of the command, splits it as a list (one line per entry), and returns its value. This allows you to then operate on the results with Python routines.

Additionally, IPython plays a few interesting syntactic tricks for your convenience. Whenever you make a system call, IPython will expand any call of the type `$var` into the actual value of the python variable `var`, so that you can call shell commands on Python values. Continuing the session above, and remembering that _ holds the previously returned value, we can call the '`wc -l`' Unix command (which does a line count on a file) on the files we just obtained:

```
In [8]: for f in _:
   ...:      if 'simple' in f:
   ...:          !wc -l $f
   ...:
3 ip_simple2.py
4 ip_simple.py
```

While this is completely unorthodox (actually, invalid) Python, it is the kind of functionality which can make for extremely efficient uses when working at an interactive command line. Obviously all of this can be done (and it *is* done that way by IPython internally) with regular Python code, but that approach requires a fair amount more typing, the use of `%`-based string interpolation, and making system calls via the `os.system()` function.

If you actually need to pass a `$` character to a shell command, you simply use `$$` in the IPython command line:

```
In [11]: !echo $$SHELL
/bin/tcsh
```

If you want to capture the output of a system command directly to a named Python variable, you can use the `%sc` magic function:

```
# by default, %sc captures to a plain string:
In [16]: %sc astr=ls ip*.py
In [17]: astr
Out[17]: 'ip_expensive_init.py\nip_simple2.py\nip_simple.py'
# but with the -l option, it splits to a list (like !! does)
In [18]: %sc -l alist=ls ip*.py
In [19]: alist
Out[19]: ['ip_expensive_init.py', 'ip_simple2.py', 'ip_simple.py']
```

3.2. System aliases. In IPython, you can also define your own system aliases. Even though IPython gives you access to your system shell via the `!` prefix, it is convenient to have aliases to the system commands you use most often. This allows you to work seamlessly from inside IPython with the same commands you are used to in your system shell:

'`%alias alias_name cmd`' defines '`alias_name`' as an alias for '`cmd`'

Then, typing '`alias_name params`' will execute the system command '`cmd params`' (from your underlying operating system). Aliases have lower precedence than magic functions and Python normal variables, so if '`foo`' is both a Python variable and an alias, the alias can not be executed until '`del foo`' removes the Python variable. If you need to access an alias directly, you can use the builtin function `ipalias` as `ipalias('foo')`.

You can use the `%l` specifier in an alias definition to represent the whole line when the alias is called. For example:

```
In [2]: alias all echo "Input in brackets: <%l>"
In [3]: all hello world
```

```
Input in brackets: <hello world>
```

You can also define aliases with positional parameters using `%s` specifiers (one per parameter):

```
In [1]: alias parts echo first %s second %s
In [2]: %parts A B
first A second B
In [3]: %parts A
Incorrect number of arguments: 2 expected.
parts is an alias to: 'echo first %s second %s'
```

Aliases expand Python variables just like system calls using `!` or `!!` do: all expressions prefixed with '$' get expanded. For details of the semantic rules, see PEP-215: http://www.python.org/peps/pep-0215.html. This is the library used by IPython for variable expansion.

Simply typing `alias` will print a list of the current aliases, and `unalias` can be used to remove an alias. For further details, use `alias?`.

3.3. Directory management. IPython comes with some pre-defined aliases and a complete system for changing directories, both via a stack (see `%pushd`, `%popd` and `%ds`) and via direct `%cd`. The latter keeps a history of visited directories and allows you to go to any previously visited one. You can see this history with the `%dhist` magic:

```
In [1]: cd ~/code/python
/home/fperez/code/python
In [2]: cd ~/teach/
/home/fperez/teach
In [3]: cd ~/research
/home/fperez/research
In [4]: dhist
Directory history (kept in _dh)
0: /home/fperez/teach/course/examples
1: /home/fperez/code/python
2: /home/fperez/teach
3: /home/fperez/research
In [5]: cd -1
/home/fperez/code/python
```

The `%bookmark` magic allows you to create named bookmarks in your filesystem, which `cd` can be directed to go to (with the `-b` flag), and to which it will try to default automatically if no such named directory exists. The system is very easy to use and quite natural in practice:

```
In [8]: bookmark course
In [9]: cd
/home/fperez
In [10]: ls course
ls: course: No such file or directory
In [11]: cd course
(bookmark:course) -> /home/fperez/teach/course
/home/fperez/teach/course
```

3.4. IPython as a system shell. While IPython is *not* a system shell, it ships with a special profile called `pysh`, which you can activate at the command line as `'ipython -p pysh'`. This modifies IPython's behavior and adds some additional facilities and a prompt customized for filesystem navigation.

Note that this does *not* make IPython a full-fledged system shell. In particular, it has no job control, so if you type Ctrl-Z (under Unix), you'll suspend pysh itself, not the process you just started.

What the shell profile allows you to do is to use the convenient and powerful syntax of Python to do quick scripting at the command line. Below we describe some of its features.

3.4.1. *Aliases.* All of your `$PATH` has been loaded as IPython aliases, so you should be able to type any normal system command and have it executed. See `%alias?` and `%unalias?` for details on the alias facilities. See also `%rehash?` and `%rehashx?` for details on the mechanism used to load `$PATH`.

3.4.2. *Special syntax.* Any lines which begin with `'~'`, `'/'` and `'.'` will be executed as shell commands instead of as Python code. The special escapes below are also recognized. `!cmd` is valid in single or multi-line input, all others are only valid in single-line input:

`!cmd`: pass 'cmd' directly to the shell

!!cmd: execute 'cmd' and return output as a list (split on '\n')
$var=cmd: capture output of cmd into var, as a string (shorthand for `%sc var=cmd`)
$$var=cmd: capture output of cmd into var, as a list (split on '\n', shorthand for `%sc -l var=cmd`)

3.4.3. *Useful functions and modules.* The os, sys and shutil modules from the Python standard library are automatically loaded. Some additional functions, useful for shell usage, are listed below. You can request more help about them with '?'.

shell: - execute a command in the underlying system shell
system: - like `shell()`, but return the exit status of the command
sout: - capture the output of a command as a string
lout: - capture the output of a command as a list (split on '\n')
getoutputerror: - capture (output,error) of a shell commandss

`sout/lout` are the functional equivalents of `$/$$`. They are provided to allow you to capture system output in the middle of true python code, function definitions, etc (where `$` and `$$` are invalid)

4. Access to an editor

You can use `%edit` to have almost multiline editing. While IPython doesn't support true multiline editing, this command allows you to call an editor on the spot, and IPython will execute the code you type in there as if it were typed interactively.

`%edit` runs your IPython configured editor. By default this is read from your environment variable `$EDITOR`. If this isn't found, it will default to `vi` under Linux/Unix and to `notepad` under Windows.

You can also set the value of this editor via the command-line option `'-editor'` or in your `ipythonrc` file. This is useful if you wish to use specifically for IPython an editor different from your typical default (and for Windows users who typically don't set environment variables).

This command allows you to conveniently edit multi-line code right in your IPython session.

If called without arguments, `%edit` opens up an empty editor with a temporary file and will execute the contents of this file when you close it (don't forget to save it!).

5. Customizing IPython

5.1. Basics. IPython has a very flexible configuration system. It uses a configuration file which allows permanent setting of all command-line options, module loading, code and file execution. The system allows recursive file inclusion, so you can have a base file with defaults and layers which load other customizations for particular projects.

IPython reads a configuration file which can be specified at the command line (`-rcfile`) or which by default is assumed to be called `ipythonrc`. Such a file is looked for in the current directory where IPython is started and then in your `IPYTHONDIR`, which allows you to have local configuration files for specific projects. The default value for this directory is `$HOME/.ipython` (`_ipython` under Windows). Under Unix operating systems `$HOME` always exists; for Windows, IPython will try to find such an environment variable; if it doesn't exist, it uses `HOMEDRIVE\HOMEPATH` (these are always defined by Windows). This typically gives something like `C:\Documents and Settings\YourUserName`, but your local details may vary. Finally, you can make this directory live anywhere you want by creating an environment variable called `$IPYTHONDIR`.

In this directory you will find all the files that configure IPython's defaults, and you can put there your profiles and extensions. This directory is automatically added by IPython to `sys.path`, so anything you place there can be found by `import` statements.

The syntax of an rcfile is one of key-value pairs separated by whitespace, one per line. Lines beginning with a `#` are ignored as comments, but comments can **not** be put on lines with data (the parser is fairly primitive). You can study the default rcfile created by IPython at startup for customization details, it is extremely commented.

5.2. Profiles. IPython can load any configuration file you want if you give its name at startup with the `-rcfile` flag. However, for convenience it provides a shorthand based on a naming convention for loading such profiles. This system allows you to easily maintain customized versions of IPython for specific purposes.

With the `-profile <name>` flag (you can abbreviate it to `-p`), IPython will assume that your config file is called `ipythonrc-<name>` (it looks in current dir first, then in `IPYTHONDIR`). This is a quick way to keep and load multiple config files for different tasks, especially if you use the include option of config

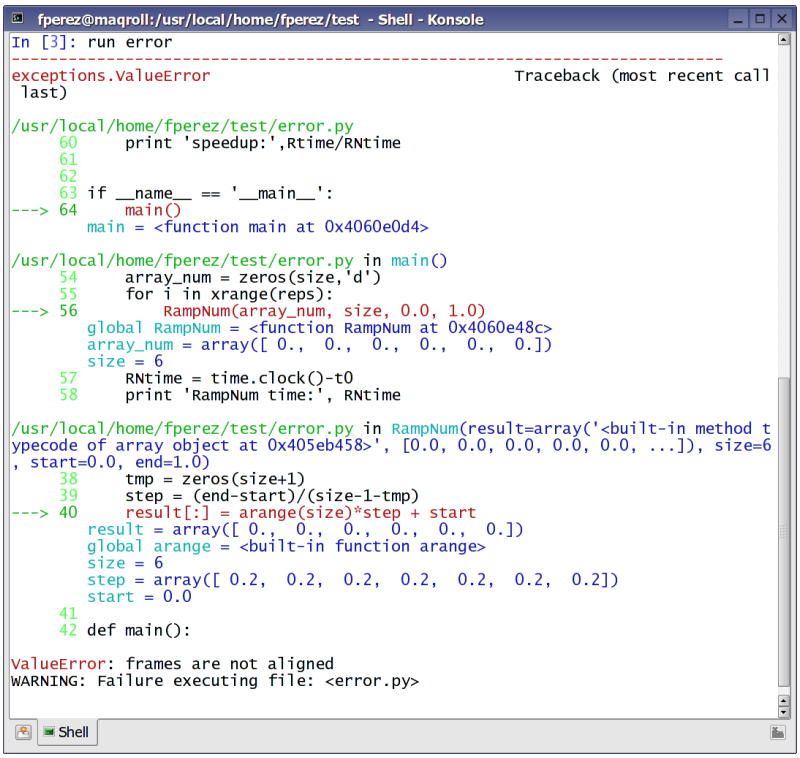

FIGURE 2. IPython can provide extremely detailed tracebacks.

files. You can keep a basic `IPYTHONDIR/ipythonrc` file and then have other profiles which include this one and load extra things for particular tasks. For example:

(1) `$HOME/.ipython/ipythonrc`: load basic things you always want.
(2) `$HOME/.ipython/ipythonrc-math`: load (1) and basic math-related modules.

Since it is possible to create an endless loop by having circular file inclusions, IPython will stop if it reaches 15 recursive inclusions.

6. Debugging and profiling with IPython

The Python standard library includes powerful facilities for debugging and profiling code, but it is common to find even experienced Python programmers who still do not take advantage of them. In part, this is due to the fact that loading and configuring them requires reading an extra documentation section, and keeping a bit of additional information about their use in your head. IPython tries to automate their use to the point where, with a single command, you can use either of these subsystems in a transparent manner. Hopefully they will become part of your daily workflow.

At its most basic, for debugging your programs, you can rely on using `%run` to execute them, see the results, play with all variables loaded into the interactive namespace, etc. A typical working session involves keeping your favorite editor open with the file you are working on, and repeatedly calling `%run` on it as you make changes and save them.

If your program raises an exception, IPython will provide you with a more detailed traceback than the default Python ones. You can even increase the level of detail further by using `%xmode Verbose`, which forces the printing of variable values at all stack frames. This option should be used with care though (and that's why it is not the default), as printing a ten-million-entry array can lock up your computer for a very long time. An example of this kind of very informative traceback is shown in Fig. 2.

6.1. Automatic invocation of pdb on exceptions. IPython, if started with the `-pdb` option (or if the option is set in your rc file) can call the Python `pdb` debugger every time your code triggers an uncaught exception. This feature can also be toggled at any time with the `%pdb` magic command. This can be extremely useful in order to find the origin of subtle bugs, because `pdb` opens up at the point in

your code which triggered the exception, and while your program is at this point 'dead', all the data is still available and you can walk up and down the stack frame and understand the origin of the problem.

Furthermore, you can use these debugging facilities both with the embedded IPython mode and without IPython at all. For an embedded shell (see sec. 7), simply call the constructor with `'-pdb'` in the argument string and automatically `pdb` will be called if an uncaught exception is triggered by your code.

For stand-alone use of the feature in your programs which do not use IPython at all, put the following lines toward the top of your 'main' routine:

```
import sys,IPython.ultraTB
sys.excepthook = IPython.ultraTB.FormattedTB(mode='Verbose',
color_scheme='Linux', call_pdb=1)
```

The `mode` keyword can be either `'Verbose'` or `'Plain'`, giving either very detailed or normal tracebacks respectively. The `color_scheme` keyword can be one of `'NoColor'`, `'Linux'` (default) or `'LightBG'`. These are the same options which can be set in IPython with `-colors` and `-xmode`.

This will give any of your programs detailed, colored tracebacks with automatic invocation of `pdb`.

6.2. Running entire programs via pdb. `pdb`, the Python debugger, is a powerful interactive debugger which allows you to step through code, set breakpoints, watch variables, etc. IPython makes it very easy to start any script under the control of `pdb`, regardless of whether you have wrapped it into a `'main()'` function or not. For this, simply type `'%run -d myscript'` at an IPython prompt. See the `%run` command's documentation (`run?`) for more details, including how to control where `pdb` will stop execution first.

For more information on the use of the `pdb` debugger, read the included `pdb.doc` file (part of the standard Python distribution). On a stock Linux system it is located at `/usr/lib/python2.3/pdb.doc`, but the easiest way to read it is by using the `help()` function of the `pdb` module as follows (in an IPython prompt):

```
In [1]: import pdb
In [2]: pdb.help()
```

This will load the `pdb.doc` document in a file viewer for you automatically.

6.3. Profiling. When dealing with performance issues, the `%run` command with a `-p` option allows you to run complete programs under the control of the Python profiler. The `%prun` command does a similar job for single Python expressions (like function calls, similar to `profile.run()`). While this is possible with the standard `profile` module, IPython wraps this functionality with magic commands convenient for rapid interactive work.

7. Embedding IPython into your programs

A few lines of code are enough to load a complete IPython inside your own programs, giving you the ability to work with your data interactively after automatic processing has been completed.

You can call IPython as a python shell inside your own python programs. This can be used both for debugging code or for providing interactive abilities to your programs with knowledge about the local namespaces (very useful in debugging and data analysis situations).

It is possible to start an IPython instance *inside* your own Python programs. This allows you to evaluate dynamically the state of your code, operate with your variables, analyze them, etc. Note however that any changes you make to values while in the shell do *not* propagate back to the running code, so it is safe to modify your values because you won't break your code in bizarre ways by doing so.

This feature allows you to easily have a fully functional python environment for doing object introspection anywhere in your code with a simple function call. In some cases a simple print statement is enough, but if you need to do more detailed analysis of a code fragment this feature can be very valuable.

It can also be useful in scientific computing situations where it is common to need to do some automatic, computationally intensive part and then stop to look at data, plots, etc[1]. Opening an IPython instance will give you full access to your data and functions, and you can resume program execution once you are done with the interactive part (perhaps to stop again later, as many times as needed).

The following code snippet is the bare minimum you need to include in your Python programs for this to work (detailed examples follow later):

[1]This functionality was inspired by IDL's combination of the `stop` keyword and the `.continue` executive command, which I have found very useful in the past, and by a posting on comp.lang.python by cmkl <cmkleffner@gmx.de> on Dec. 06/01 concerning similar uses of pyrepl.

```
from IPython.Shell import IPShellEmbed
ipshell = IPShellEmbed()
ipshell() # this call anywhere in your program will start IPython
```

You can run embedded instances even in code which is itself being run at the IPython interactive prompt with '`%run <filename>`'. Since it's easy to get lost as to where you are (in your top-level IPython or in your embedded one), it's a good idea in such cases to set the in/out prompts to something different for the embedded instances. The code examples below illustrate this.

You can also have multiple IPython instances in your program and open them separately, for example with different options for data presentation. If you close and open the same instance multiple times, its prompt counters simply continue from each execution to the next.

Please look at the docstrings in the `Shell.py` module for more details on the use of this system.

The following sample file illustrating how to use the embedding functionality is provided in the examples directory as `example-embed.py`. It should be fairly self-explanatory:

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

"""An example of how to embed an IPython shell into a running program.

Please see the documentation in the IPython.Shell module for more details.

The accompanying file example-embed-short.py has quick code fragments for
embedding which you can cut and paste in your code once you understand how
things work.

The code in this file is deliberately extra-verbose, meant for learning."""

# The basics to get you going:

# IPython sets the __IPYTHON__ variable so you can know if you have nested
# copies running.

# Try running this code both at the command line and from inside IPython (with
# %run example-embed.py)
try:
    __IPYTHON__
except NameError:
    nested = 0
    args = ['']
else:
    print "Running nested copies of IPython."
    print "The prompts for the nested copy have been modified"
    nested = 1
    # what the embedded instance will see as sys.argv:
    args = ['-pi1','In <\\#>:','-pi2','   .\\D.:','-po','Out<\\#>:','-nosep']

# First import the embeddable shell class
from IPython.Shell import IPShellEmbed

# Now create an instance of the embeddable shell. The first argument is a
# string with options exactly as you would type them if you were starting
# IPython at the system command line. Any parameters you want to define for
# configuration can thus be specified here.
ipshell = IPShellEmbed(args,
                       banner = 'Dropping into IPython',
                       exit_msg = 'Leaving Interpreter, back to program.')
```

```
# Make a second instance, you can have as many as you want.
if nested:
    args[1] = 'In2<\\#>'
else:
    args = ['-pi1','In2<\\#>:','-pi2','   .\\D.:','-po','Out<\\#>:','-nosep']
ipshell2 = IPShellEmbed(args,banner = 'Second IPython instance.')

print '\nHello. This is printed from the main controller program.\n'

# You can then call ipshell() anywhere you need it (with an optional
# message):
ipshell('***Called from top level. '
        'Hit Ctrl-D to exit interpreter and continue program.')

print '\nBack in caller program, moving along...\n'

#---------------------------------------------------------------------------
# More details:

# IPShellEmbed instances don't print the standard system banner and
# messages. The IPython banner (which actually may contain initialization
# messages) is available as <instance>.IP.BANNER in case you want it.

# IPShellEmbed instances print the following information everytime they
# start:

# - A global startup banner.

# - A call-specific header string, which you can use to indicate where in the
# execution flow the shell is starting.

# They also print an exit message every time they exit.

# Both the startup banner and the exit message default to None, and can be set
# either at the instance constructor or at any other time with the
# set_banner() and set_exit_msg() methods.

# The shell instance can be also put in 'dummy' mode globally or on a per-call
# basis. This gives you fine control for debugging without having to change
# code all over the place.

# The code below illustrates all this.


# This is how the global banner and exit_msg can be reset at any point
ipshell.set_banner('Entering interpreter - New Banner')
ipshell.set_exit_msg('Leaving interpreter - New exit_msg')

def foo(m):
    s = 'spam'
    ipshell('***In foo(). Try @whos, or print s or m:')
    print 'foo says m = ',m
```

```
def bar(n):
    s = 'eggs'
    ipshell('***In bar(). Try @whos, or print s or n:')
    print 'bar says n = ',n

# Some calls to the above functions which will trigger IPython:
print 'Main program calling foo("eggs")\n'
foo('eggs')

# The shell can be put in 'dummy' mode where calls to it silently return. This
# allows you, for example, to globally turn off debugging for a program with a
# single call.
ipshell.set_dummy_mode(1)
print '\nTrying to call IPython which is now "dummy":'
ipshell()
print 'Nothing happened...'
# The global 'dummy' mode can still be overridden for a single call
print '\nOverriding dummy mode manually:'
ipshell(dummy=0)

# Reactivate the IPython shell
ipshell.set_dummy_mode(0)

print 'You can even have multiple embedded instances:'
ipshell2()

print '\nMain program calling bar("spam")\n'
bar('spam')

print 'Main program finished. Bye!'

#********************** End of file <example-embed.py> ***********************
```

Once you understand how the system functions, you can use the following code fragments in your programs which are ready for cut and paste:

```
"""Quick code snippets for embedding IPython into other programs.

See example-embed.py for full details, this file has the bare minimum code for
cut and paste use once you understand how to use the system."""

#-----------------------------------------------------------------------------
# This code loads IPython but modifies a few things if it detects it's running
# embedded in another IPython session (helps avoid confusion)

try:
    __IPYTHON__
except NameError:
    argv = ['']
    banner = exit_msg = ''
else:
    # Command-line options for IPython (a list like sys.argv)
    argv = ['-pi1','In <\\#>:','-pi2','   .\\D.:','-po','Out<\\#>:']
    banner = '*** Nested interpreter ***'
    exit_msg = '*** Back in main IPython ***'
```

```
# First import the embeddable shell class
from IPython.Shell import IPShellEmbed
# Now create the IPython shell instance. Put ipshell() anywhere in your code
# where you want it to open.
ipshell = IPShellEmbed(argv,banner=banner,exit_msg=exit_msg)

#-----------------------------------------------------------------------
# This code will load an embeddable IPython shell always with no changes for
# nested embededings.

from IPython.Shell import IPShellEmbed
ipshell = IPShellEmbed()
# Now ipshell() will open IPython anywhere in the code.

#-----------------------------------------------------------------------
# This code loads an embeddable shell only if NOT running inside
# IPython. Inside IPython, the embeddable shell variable ipshell is just a
# dummy function.

try:
    __IPYTHON__
except NameError:
    from IPython.Shell import IPShellEmbed
    ipshell = IPShellEmbed()
    # Now ipshell() will open IPython anywhere in the code
else:
    # Define a dummy ipshell() so the same code doesn't crash inside an
    # interactive IPython
    def ipshell(): pass

#****************** End of file <example-embed-short.py> ******************
```

8. Integration with Matplotlib

The matplotlib library (http://matplotlib.sourceforge.net) provides high quality 2D plotting for Python. Matplotlib can produce plots on screen using a variety of GUI toolkits, including Tk, GTK and WXPython. It also provides a number of commands useful for scientific computing, all with a syntax compatible with that of the popular Matlab program.

IPython accepts the special option `-pylab`. This configures it to support matplotlib, honoring the settings in the `.matplotlibrc` file. IPython will detect the user's choice of matplotlib GUI backend, and automatically select the proper threading model to prevent blocking. It also sets matplotlib in interactive mode and modifies `%run` slightly, so that any matplotlib-based script can be executed using `%run` and the final `show()` command does not block the interactive shell.

The `-pylab` option must be given first in order for IPython to configure its threading mode. However, you can still issue other options afterwards. This allows you to have a matplotlib-based environment customized with additional modules using the standard IPython profile mechanism: "`ipython -pylab -p myprofile`" will load the profile defined in `ipythonrc-myprofile` after configuring matplotlib.

CHAPTER 4

Introduction to plotting with matplotlib / pylab

1. A bird's eye view

matplotlib is a library for making 2D plots of arrays in python.[1] Although it has its origins in emulating the Matlab graphics commands, it does not require matlab, and has a pure, object oriented API. Although matplotlib is written primarily in python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays. matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn't need to instantiate objects, call methods, set properties, and so on; it should just work.

The matplotlib code is divided into three parts: the *pylab interface* is the set of functions provided by the `pylab` module which allow the user to create plots with code quite similar to matlab figure generating code. The matplotlib frontend or *matplotlib API* is the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots and so on. This is an abstract interface that knowns nothing about output formats. The *backends* are device dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device. Example backends: PS creates postscript hardcopy, SVG creates scalar vector graphics hardcopy, Agg creates PNG output using the high quality antigrain library that ships with matplotlib, GTK embeds matplotlib in a GTK application, GTKAgg uses the antigrain[2] renderer to create a figure and embed it a GTK application, and so on for WX, Tkinter, FLTK, ...

For years, I used to use matlab exclusively for data analysis and visualization. matlab excels at making nice looking plots easy. When I began working with EEG data, I found that I needed to write applications to interact with my data, and developed and EEG analysis application in matlab. As the application grew in complexity, interacting with databases, http servers, manipulating complex data structures, I began to strain against the limitations of matlab as a programming language, and decided to start over in python. python more than makes up for all of matlab's deficiencies as a programming language, but I was having difficulty finding a 2D plotting package – for 3D VTK, which is discussed at length below more than exceeds all of my needs.

When I went searching for a python plotting package, I had several requirements:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc)
- Postscript output for inclusion with LaTeX documents and publication quality printing
- Embeddable in a graphical user interface for application development
- The code should be mostly python so itis easy to understand and extend – users become developers!
- Making plots should be easy – just a few lines of code for simple graphs

Finding no package that suited me just right, I did what any self-respecting python programmer would do: rolled up my sleeves and dived in. Not having any real experience with computer graphics, I decided to emulate matlab's plotting capabilities because that is something matlab does very well. This had the added advantage that many people have a lot of matlab experience, and thus they can quickly get up to steam plotting in python. From a developer's perspective, having a fixed user interface (the pylab interface) has been very useful, because the guts of the code base can be redesigned without affecting user code.

Without further ado, let's create our first figure. This example uses the matplotlib object oriented API. Most users use the pylab interface, which will be discussed next and makes it easier to make plots because a lot of the tedius work of creating and managing figures and figure windows is done for you behind the hood. But since the real core of the library is the object oriented API, I think it is a good place to

[1] This short guide is not meant as a complete guide or tutorial. There is a more comprehensive user's guide and tutorial on the matplotlib web-site at http://matplotlib.sf.net.

[2] http://antigrain.com

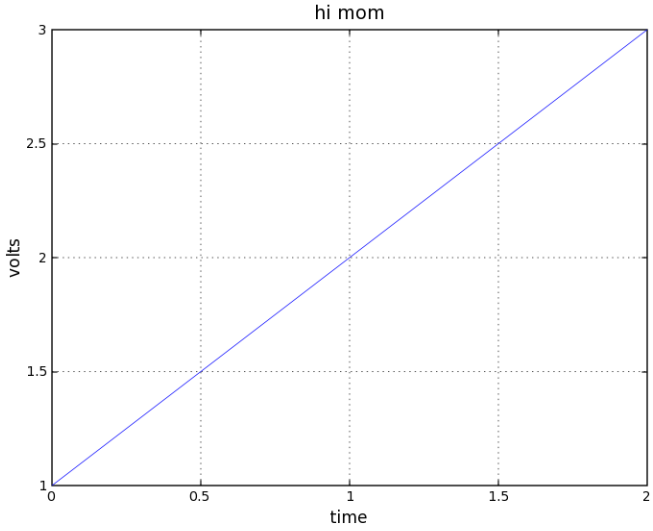

FIGURE 1. A simple plot generated by the antigrain (Agg) backend .

start. If you are developing a graphical user interface or making plots on a web server, you probably want maximal control with no magic going on behind the scenes – this is where the matplotlib API should be used. If you are just trying to make a figure for inclusion in a paper or if your working interactively from the python shell, you'll probably be happy with the pylab interface.

LISTING 4.1. Creating a simple figure with the antigrain backend (generates PNG) using the object oriented matplotlib library

```
"""
A pure object oriented example using the agg backend
"""
# import the matplotlib backend you want to use and the Figure class
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

# the figure is the center of the action, and the canvas is a backend
# dependent container to hold the figure and make backend specific calls
fig = Figure()
canvas = FigureCanvas(fig)

# you can add multiple subplots and axes
ax = fig.add_subplot(111)

# the simplest plot!
ax.plot([1,2,3])

# you can decorate your plot with text and grids
ax.set_title('hi mom')
ax.grid(True)
ax.set_xlabel('time')
ax.set_ylabel('volts')

# and save it to hardcopy
fig.savefig('../fig/mpl_one_two_three.png')
```

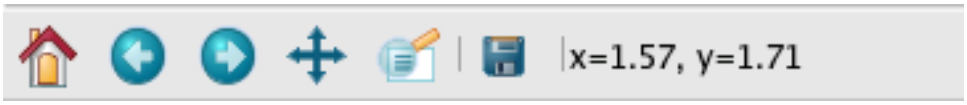

FIGURE 2. The matplotlib toolbar used to navigate around your figure

2. A short pylab tutorial

Here is about the simplest code you can use to create a figure with matplotlib using the pylab interface. In this section, I'm assuming you are using ipython in the pylab mode – see Section8 for details.

```
peds-pc311:~> pylab
Python 2.3.3 (#2, Apr 13 2004, 17:41:29)
Type "copyright", "credits" or "license" for more information.

IPython 0.6.12_cvs -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

  Welcome to pylab, a matplotlib-based Python environment
    help(matplotlib) -> generic matplotlib information
    help(pylab)      -> matlab-compatible commands from matplotlib
    help(plotting)   -> plotting commands

In [1]: plot([1,2,3])
Out[1]: [<matplotlib.lines.Line2D instance at 0xb557a86c>]
```

If your settings are correct, a figure window should popup and you should be able to interact with it. That's a lot less typing than our initial example using the object oriented API in which you had to manually create the Figure, Axes and so on!

Try clicking on the navigation toolbar at the bottom of the figure – the toolbar is shown in Figure2. The first three buttons from left to right in Figure2 are *home*, *back* and *forward*. These byttons are are akin to the web browser buttons. They are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons as described below. This is analogous to trying to click `Back` on your web browser before visiting a new page –nothing happens. The home button always takes you to the first, default view of your data.

The next to button moving right is the pan/zoom button, which looks like a cross with arrows on the end (a *fleur*). The pan/zoom button button has two modes: pan and zoom (no surprise there, right?). Click this toolbar button to activate this mode; you should see "pan/zoom mode" show up in the status bar. Then put your mouse somewhere over an axes. To activate panning: press the left mouse button and hold it, dragging it to a new position. If you press `x` or `y` while panning, the motion will be contrained to the x or y axis, respectively . To activate zooming, press the right mouse button, dragging it to a new position. The x axis will be zoomed in proportionate to the rightward movement and zoomed out proportionate to the leftward movement. Ditto for the yaxis and up/down motions. The point under your mouse when you begin the zoom remains stationary, allowing you to zoom to an arbitrary point in the figure. You can use the modifier keys `x`, `y` or `CONTROL` to constrain the zoom to the x axes, the y axes, or aspect ratio preserve, respectively.

The next button moving right is the *zoom to rectangle button* which has a magnifying glass over a piece of paper. The button is striaghtforward and works in the standard way; when you click it, you should see that it is activated by looking for "Zoom to rect mode" in the status bar, and then you select the rectangular region you want to zoom in on.

The final button is the *save button*, which will save your figure in the current view. All of the *Agg backends know how to save the following image types: PNG, PS, EPS, SVG.

Let's make the same figure we made using the object oriented API above, ie Figure1, but this time using the pylab

LISTING 4.2. Creating a simple figure in pylab

```
from pylab import *
```

```
plot([1,2,3])
title('hi mom')
grid(True)
xlabel('time')
ylabel('volts')
savefig('../fig/mpl_one_two_three.png')
show()
```

As you can see there is basically a direct translation between the OO interface and the pylab interface. When `plot` is called, the pylab interface makes a call to the function `gca()` ("get current axes") to get a reference to the current axes. `gca` in turn, makes a call to `gcf` ("get current figure") to get a reference to the current figure. `gcf`, finding that no figure has been created, creates the default figure using `figure()` and returns it. `gca` will then return the current axes of that figure if it exists, or create the default axes `subplot(111)` if it does not. The last line show is a GUI independent way of actually creating a figure window, and is not required for image backends such as postscript.

Thus a lot happens under the hood when you call plot, but for the most part you don't need to think about it – it just works. The important thing to understand is that the pylab interface has a state, and keeps track of the current figure and axes. All plotting commands target the current axes, and you can manipulate which ones are current

LISTING 4.3. Creating multiple subplots and plotting multiple lines in a single plot command

```
from pylab import *

def f(t):
    s1 = cos(2*pi*t)
    e1 = exp(-t)
    return multiply(s1,e1)

t1 = arange(0.0, 5.0, 0.1)
t2 = arange(0.0, 5.0, 0.02)
t3 = arange(0.0, 2.0, 0.01)

# create and upper subplot and make it current
subplot(211)
l1, l2 = plot(t1, f(t1), 'bo', t2, f(t2), 'k--')
set(l1, markerfacecolor='g')
grid(True)
title('A tale of 2 subplots')
ylabel('Damped oscillation')

# create a lower subplot and make it current
subplot(212)
plot(t3, cos(2*pi*t3), 'r.')
grid(True)
xlabel('time (s)')
ylabel('Undamped')
savefig('../fig/mpl_subplot_demo')
show()
```

In addition to creating multiple subplots, this example contains a couple of new things. In the first plot command, the return value is stored as `l1, l2` and the `set` command is used to change a default line property.

```
l1, l2 = plot(t1, f(t1), 'bo', t2, f(t2), 'k--')
set(l1, markerfacecolor='g')
```

`l1` and `l2` are `matplotlib.lines.Line2D` instances and they are created by the `plot` command and added to the current axes. This is the typical mode of operation of the axes plot commands: they create a bunch of primitive objects (lines, polygons, text, images), add them to the axes, and return them. In

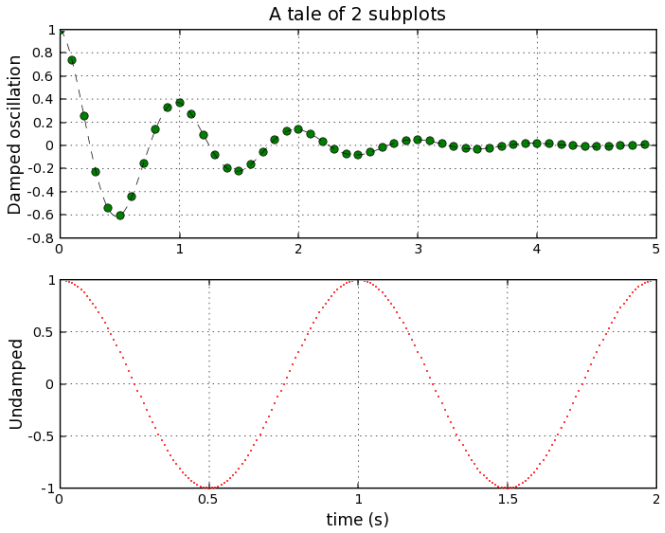

FIGURE 3. It's easy to create multiple axes and subplots.

this example, the line's `markerfacecolor` property is set with the `set` command. In the next section, we'll look into matplotlibs `set` and `get` introspection system and show how to use it to customize your lines, polygons, text instances and images.

3. Set and get introspection

Everything that goes into a matplotlib figure, including the `Figure` itself, are all objects dervied from a single base class `Artist`, and the pylab `set` and `get` commands provide a unified way to configure them. Let's create a simple plot of random circles, and use that to explore how `set` and get work. First the basic plot – we'll store the return value as lines. Note that `plot` always returns a *list* of lines; in the example above there were two lines `l1` and `l2`, and in the example below there is only a single element of the list lines. No matter: `set` and `get` will work on a single instance or a sequence of instances

```
In [2]: x = rand(20); y = rand(20)

In [3]: lines = plot(x,y,'o')

In [4]: type(lines)         # plot always returns a list
Out[4]: <type 'list'>

In [5]: len(lines)          # even if it is length 1
Out[5]: 1
```

The simple figure that was created, a scattering of blue circles at random locations, is shown in Figure4. To see a listing of the properties of the line, and what their current values are, call `get(lines)`

```
In [29]: get(lines)
    alpha = 1.0
    antialiased or aa = True
    clip_on = True
    color or c = blue
    figure = <matplotlib.figure.Figure instance at 0xb40e1cec>
    label =
    linestyle or ls = None
    linewidth or lw = 0.5
    marker = o
```

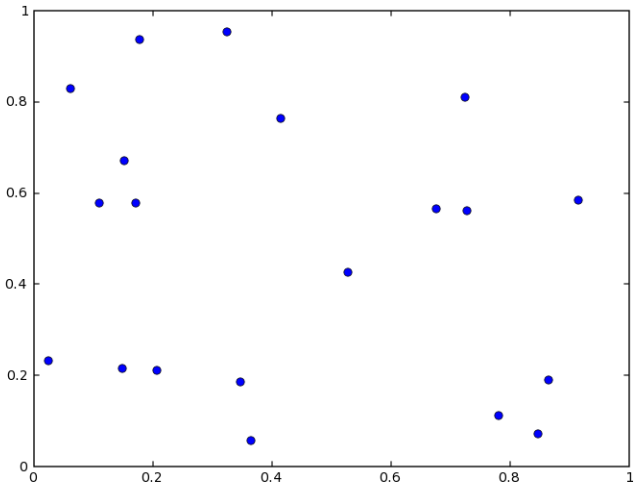

FIGURE 4. The default marker plot, before marker customization

```
    markeredgecolor or mec = black
    markeredgewidth or mew = 0.5
    markerfacecolor or mfc = blue
    markersize or ms = 6.0
    transform = <Affine object at 0x8683c6c>
    visible = True
    xdata = [ 0.16952688  0.59729624  0.16829208  0.51311375  0.7227286
         ...0.45925692]...
    ydata = [ 0.86459035  0.25595992  0.01905832  0.24303582  0.74993261
         ...0.28751132]...
    zorder = 2
```

and to see the same listing of properties with information on legal values you can set them to, call `set(lines)`

```
In [37]: set(lines)
    alpha: float
    antialiased or aa: [True | False]
    clip_box: a matplotlib.transform.Bbox instance
    clip_on: [True | False]
    color or c: any matplotlib color - see help(colors)
    dashes: sequence of on/off ink in points
    data: (array xdata, array ydata)
    data_clipping: [True | False]
    figure: a matplotlib.figure.Figure instance
    label: any string
    linestyle or ls: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]
    linewidth or lw: float value in points
    lod: [True | False]
    marker: [ '+' | ',' | '.' | '1' | '2' | '3' | '4' | '<' | '>' | 'D' | 'H'
         ...| '^' | '_' | 'd' | 'h' | 'o' | 'p' | 's' | 'v' | 'x' | '|' ]
    markeredgecolor or mec: any matplotlib color - see help(colors)
    markeredgewidth or mew: float value in points
    markerfacecolor or mfc: any matplotlib color - see help(colors)
```

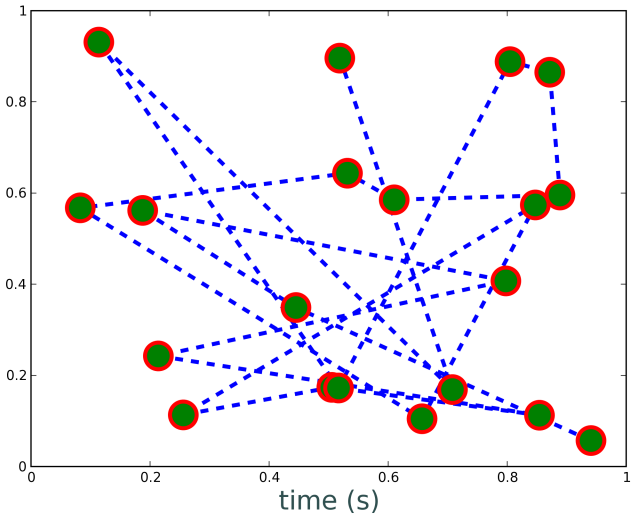

FIGURE 5. The default marker plot, before marker customization

```
markersize or ms: float
transform: a matplotlib.transform transformation instance
visible: [True | False]
xclip: (xmin, xmax)
xdata: array
yclip: (ymin, ymax)
ydata: array
zorder: any number
```

OK, we have a lot of options here. Let's change the marker properties, and add a linesytle

```
In [20]: set(lines, markerfacecolor='green', markeredgecolor='red',
   ....:  markersize=20, markeredgewidth=3,
   ....: linestyle='--', linewidth=3)
```

That's a lot of typing, but to great effect! The same data set now has quite a different appearance, which is shown in Figure5. Note in the long listing output of the set(lines) command above the markerfacecolor settable property is listed as

```
markerfacecolor or mfc: any matplotlib color - see help(colors)
```

The `markerfacecolor` has an alias `mfc` to save typing, and common colornames have abbreviations too, so the `set` command above could just as well be written

```
In [20]: set(lines, mfc='g', mec='r', ms=20, mew=3, ls='--', lw=3)
```

Another nice thing about matplotlib properties is that you can pass them in as keyword arguments to `plot` and they will have the same effect, eg, you can create the identical plot with

```
In [6]: plot(x, y, 'o', mfc='g', mec='r', ms=20, mew=3, ls='--', lw=3)
Out[6]: [<matplotlib.lines.Line2D instance at 0xb40db42c>]
```

As noted above, `set` and `get` work on any `Artist`, so you can configure your axes or text instances this way. Eg, `xlabel` returns a `matplotlib.text.Text` instance

```
In [8]: t = xlabel('time (s)')

In [9]: set(t)
    alpha: float
```

```
backgroundcolor: any matplotlib color - see help(colors)
bbox: rectangle prop dict plus key 'pad' which is a pad in points
clip_box: a matplotlib.transform.Bbox instance
clip_on: [True | False]
color: any matplotlib color - see help(colors)
family: [ 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace' ]
figure: a matplotlib.figure.Figure instance
fontproperties: a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha: [ 'center' | 'right' | 'left' ]
label: any string
lod: [True | False]
multialignment: ['left' | 'right' | 'center' ]
name or fontname: string eg, ['Sans' | 'Courier' | 'Helvetica' ...]
position: (x,y)
rotation: [ angle in degrees 'vertical' | 'horizontal'
size or fontsize: [ size in points | relative size eg 'smaller', 'x-large'
    ... ]    style or fontstyle: [ 'normal' | 'italic' | 'oblique']
text: string
transform: a matplotlib.transform transformation instance
variant: [ 'normal' | 'small-caps' ]
verticalalignment or va: [ 'center' | 'top' | 'bottom' ]
visible: [True | False]
weight or fontweight: [ 'normal' | 'bold' | 'heavy' | 'light' | 'ultrabold
    ...' | 'ultralight']
x: float
y: float
zorder: any number
```

So you have a lot of possibilities to customize your text! The most common things people what to do are change the font size and color; the results of this command on the xlabel are shown in Figure5.

```
In [25]: set(t, fontsize=20, color='darkslategray')
```

4. Customizing the default behavior with the rc file

matplotlib is designed to work in a variety of settings: some people use it in "batch mode" on a web server to create images they never look at. Others use graphical user interfaces (GUIs) to interact with their plots. Thus you must customize matplotlib to work like you want it to with the customization file `.matplotlibrc`, in which you can set whether you want to just create images or use a GUI (the backend setting), and whether you want to work interactively from the shell (the interactive setting). Almost all of the matplotlib settings and figure properties can be customized with this file, which is installed with the rest of the matplotlib data (fonts, icons, etc) into a directory determined by distutils. Before compiling matplotlib, it resides in the same dir as `setup.py` and will be copied into your install path. Typical locations for this file are

```
C:\Python23\share\matplotlib\.matplotlibrc # windows /usr/share/matplotlib/.matplotlibrc
```

By default, the installer will overwrite the existing file in the install path, so if you want to preserve yours, please move it to your `HOME` dir and set the environment variable if necessary. In the rc file, you can set your backend, whether you'll be working interactively and default values for most of the figure properties.

In the RC file, blank lines, or lines starting with a comment symbol, are ignored, as are trailing comments. Other lines must have the format

```
key : val # optional comment
```

where *key* is some property like `backend`, `lines.linewidth`, or `figure.figsize` and *val* is the value of that property. Example entries for these properties are

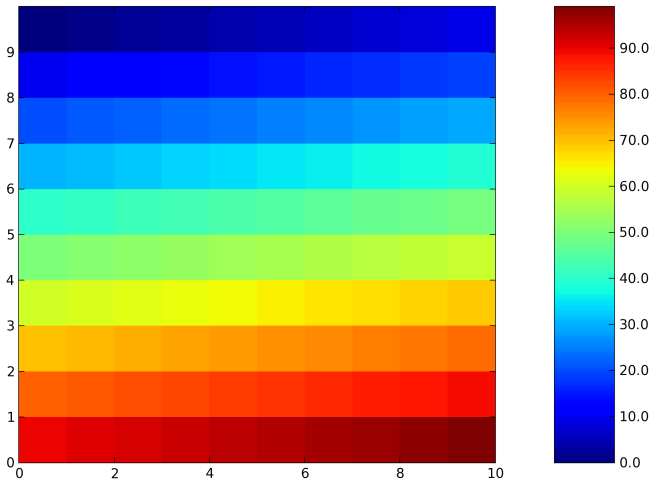

FIGURE 6. A simple image plot of a 2D matrix, using nearest neighbor interpolation and the `jet` colormap.

```
# this is a comment and is ignored
backend         : GTKAgg    # the default backend
lines.linewidth : 0.5       # line width in points
figure.figsize  : 8, 6      # figure size in inches
```

A complete sample rc file is included with the matplotlib distribution and available online.[3]

5. A quick tour of plot types

6. Images

Matplotlib has support for plotting images with imshow and figimage. In imshow, the image data is scaled to fit into the current axes, and many different interpolation schemes are supported to do the resampling, and in figimage, the image data are transferred as a raw pixel dump to the figure canvas without resampling. You can add colorbars, set the default colormaps, and change the interpolation schemes quite easily.

```
In [15]: x = arange(100.0); x.shape = 10,10

In [16]: im = imshow(x, interpolation='nearest')

In [17]: colorbar()
Out[17]: <matplotlib.axes.Axes instance at 0xb455496c>
```

which creates the image shown in Figure 6. You can interactively update the default colormap and change the interpolation scheme, which creates the image show in Figure 7.

```
In [18]: im.set_interpolation('bilinear')

In [19]: hot()
```

There is a lot more you can do with images: you can set the data extent so that you can overlay contours or other plots, you can plot multiple images to the same axes with different colors and transparency values, you can load images with PIL or `imread` and plot them in matplotlib, you can create montages of with `figimage` placed around the figure window at different offsets, you can plot grayscale, rgb or rgba data, and so on. Consult the *Matplotlib User's Guide* and the `examples` subdirectory in the matplotlib source distribution for more information. We'll clost off with a simple example of reading in a PNG and displaying it

[3] http://matplotlib.sourceforge.net/.matplotlibrc

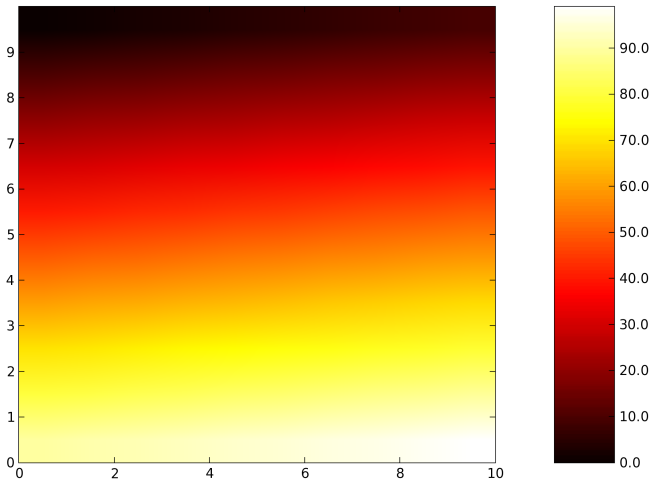

FIGURE 7. The same image data, rendered with the hot colormap and bilinear interpolation. matplotlib has 14 colormaps built-in, and you can define your own with relative ease, and there are 16 interpolation methods.

```
In [35]: im = imread('../data/ratner.png')

In [36]: imshow(im)
Out[36]: <matplotlib.image.AxesImage instance at 0xb3ffba2c>

In [37]: axis('off')
```

7. Customizing text and mathematical expressions

8. Event handling: Tracking the mouse and keyboard

FIGURE 8. Displaying image data from your camera in matplotlib

CHAPTER 5

Interfacing with external libraries

1. weave

Below is a listing of examples of weave use. This needs a lot of cleaning, as some of this code is very old and doesn't actually run with current weave.

```
#!/usr/bin/env python
"""Simple examples of weave use.

Code meant to be used for learning/testing, not production.

Fernando Perez <fperez@colorado.edu>
March 2002, updated 2003."""

from weave import inline,converters
from Numeric import *

#----------------------------------------------------------------------------
def simple_print(input):
    """Simple print test.

    Since there's a hard-coded printf %i in here, it will only work for
         ...numerical
    inputs (ints).  """

    # note in the printf that newlines must be passed as \\n:
    code = '''
    std::cout << "Printing from C++ (using std::cout) : "<<input<<std::endl;
    printf("And using C syntax (printf)     : %i\\n",input);
    '''
    inline(code,['input'],
           verbose=2)  # see inline docstring for details

def py_print(input):
    "Trivial printer, for timing."
    print "Input:",input

def c_print(input):
    "Trivial printer, for timing."
    code = """printf("Input: %i \\n",input);"""
    inline(code,['input'])

def cpp_print(input):
    "Trivial printer, for timing."
    code = """std::cout << "Input: " << input << std::endl;"""
    inline(code,['input'])
```

```
#--------------------------------------------------------------------------
# Returning a scalar quantity computed from a Numeric array.
def trace(mat):
    """Return the trace of a matrix.
    """
    nrow,ncol = mat.shape
    code = \
"""
double tr=0.0;

for(int i=0;i<nrow;++i)
    tr += mat(i,i);
return_val = tr;
"""
    return inline(code,['mat','nrow','ncol'],
                  type_converters = converters.blitz)

#--------------------------------------------------------------------------
# WRONG CODE: trace() version which modifies in-place a python scalar
# variable.  Note that this doesn't work, similarly to how in-place changes in
# python only work for mutable objects. Below is an example that does work.
def trace2(mat):
    """Return the trace of a matrix. WRONG CODE.
    """
    nrow,ncol = mat.shape
    tr = 0.0
    code = \
"""
for(int i=0;i<nrow;++i)
    tr += mat(i,i);
"""
    inline(code,['mat','nrow','ncol','tr'],
           type_converters = converters.blitz)
    return tr

#--------------------------------------------------------------------------
# Operating in-place in an existing Numeric array. Contrary to trying to
     ...modify
# in-place a scalar, this works correctly.
def in_place_mult(num,mat):
    """In-place multiplication of a matrix by a scalar.
    """
    nrow,ncol = mat.shape
    code = \
"""
for(int i=0;i<nrow;++i)
    for(int j=0;j<ncol;++j)
        mat(i,j) *= num;

"""
    inline(code,['num','mat','nrow','ncol'],
           type_converters = converters.blitz)

#--------------------------------------------------------------------------
```

```
# Pure Python version for checking.
def cross_product(a,b):
    """Cross product of two 3-d vectors.
    """
    cross = [0]*3
    cross[0] = a[1]*b[2]-a[2]*b[1]
    cross[1] = a[2]*b[0]-a[0]*b[2]
    cross[2] = a[0]*b[1]-a[1]*b[0]
    return array(cross)

#----------------------------------------------------------------------------
# Here we return a list from the C code. This is probably *much* slower than
# the python version, it's meant as an illustration and not as production
# code.
def cross_productC(a,b):
    """Cross product of two 3-d vectors.
    """
    # py::tuple or py::list both work equally well in this case.
    code = \
"""
py::tuple cross(3);

cross[0] = a(1)*b(2)-a(2)*b(1);
cross[1] = a(2)*b(0)-a(0)*b(2);
cross[2] = a(0)*b(1)-a(1)*b(0);
return_val = cross;
"""
    return array(inline(code,['a','b'],
                        type_converters = converters.blitz))

#----------------------------------------------------------------------------
# C version which accesses a pre-allocated NumPy vector.  Note: when using
# blitz, index access is done with (,,), not [][][]. In fact, [] indexing
# fails silently. See this and the next version for a comparison.
def cross_productC2(a,b):
    """Cross product of two 3-d vectors.
    """

    cross = zeros(3,a.typecode())
    code = \
"""
cross(0) = a(1)*b(2)-a(2)*b(1);
cross(1) = a(2)*b(0)-a(0)*b(2);
cross(2) = a(0)*b(1)-a(1)*b(0);
"""
    inline(code,['a','b','cross'],
           type_converters = converters.blitz)
    return cross

#----------------------------------------------------------------------------
# Just like the previous case, but now we don't use the blitz converters.
# Weave automagically does the type conversions for us.
def cross_productC3(a,b):
    """Cross product of two 3-d vectors.
```

```
    """

    cross = zeros(3,a.typecode())
    code = \
"""
cross[0] = a[1]*b[2]-a[2]*b[1];
cross[1] = a[2]*b[0]-a[0]*b[2];
cross[2] = a[0]*b[1]-a[1]*b[0];
"""
    inline(code,['a','b','cross'])

    return cross

#----------------------------------------------------------------------------
def dot_product(a,b):
    """Dot product of two vectors.

    Implemented in a funny (ridiculous) way to use support_code.

    I want to see if we can call another function from inside our own
    code. This would give us a crude way to implement better modularity by
    having global constants which include the raw code for whatever C
    functions we need to call in various places. These can then be included
    via support_code.

    The overhead is that the support code gets compiled in *every* dynamically
    generated module, but I'm not sure that's a big deal since the big
    compilation overhead seems to come from all the fancy C++ templating and
    whatnot.

    Later: ask Eric if there's a cleaner way to do this."""

    N = len(a)
    support = \
"""
double mult(double x,double y) {
    return x*y;
}
"""
    code = \
"""
double sum = 0.0;
for (int i=0;i<N;++i) {
    sum += mult(a(i),b(i));
}
return_val = sum;
"""
    return inline(code,['a','b','N'],
                  type_converters = converters.blitz,
                  support_code = support,
                  libraries = ['m'],
                  )

#----------------------------------------------------------------------------
```

```
def sumC(x):
    """Return the sum of the elements of a 1-d array.

    An example of how weave accesses a Numeric array without blitz.  """

    num_types = {Float:'double',
                 Float32:'float'}
    x_type = num_types[x.typecode()]

    code = """
           double result=0.0;
           double element;

           for (int i = 0; i < Nx[0]; i++){

               // Note the type of the pointer below is computed in python
               //element = *(%s *)(x->data+i*x->strides[0]);

               // Weave's magic does the above for us:
               element = x[i];

               result += element;
               std::cout << "Element " << i << " = " << element << "\\n";
           }
           std::cout << "size x " << Nx[0] << "\\n";

           return_val = result;
           """ % x_type;

    return inline(code,['x'],verbose=0)

#----------------------------------------------------------------------------
def Cglobals(arr):
    """How to pass data from function to function via globals.

    This allows the kind of 'over the head' parameter passing via globals
    which is ugly but necessary for using things like generic integrators in
    Numerical Recipes with aditional parameters.  """

    support = \
"""
// Declare globals here

/* These blitz guys must be accessed via pointers to avoid a costly copy.
Note that now the type is hardwired in. All python polymorphism is gone.  I
should look into whether this can be fixed by properly using blitz templating.
*/
blitz::Array<int, 1> *G_arr_pt;

// The global M will be visible in the "code" segment
int M = 99;

void aprint(int N) {
  std::cout << "In aprint()\\n";
```

```
  for (int i=0;i<N;++i)
    std::cout << "arr[" << i << "]=" << (*G_arr_pt)(i) << " ";
  std::cout << std::endl;
}

"""
    code = \
"""
// Get the passed array reference so the data becomes global
G_arr_pt = &arr;

std::cout << "global M=" << M << std::endl;
std::cout << "local N=" << N << std::endl;


std::cout << "First, print using the blitz internal printer:\\n";
std::cout << "all arr\\n";
std::cout << arr << std::endl;

std::cout << "all G_arr\\n";
std::cout << *G_arr_pt << std::endl;

std::cout << "now by loop\\n";

for (int i=0;i<N;++i)
  std::cout << "arr[" << i << "]=" << arr(i) << " ";
std::cout << std::endl;


std::cout << "Now calling aprint\\n";

aprint(N);
"""
    N = len(arr)
    return inline(code,['arr','N'],
                  type_converters = converters.blitz,
                  support_code = support,
                  libraries = ['m'],
                  verbose = 0,
                  )


#----------------------------------------------------------------------------
# Two trivial examples using the C math library follow.
def powC(x,n):
    """powC(x,n) -> x**n. Implemented using the C pow() function.
    """
    support = \
"""
#include <math.h>
"""
    code = \
"""
return_val = pow(x,n);
```

```
"""
    return inline(code,['x','n'],
                  type_converters = converters.blitz,
                  support_code = support,
                  libraries = ['m'],
                  )

# Some callback examples
def foo(x,y):
     print "In Python's foo:"
     print 'x',x
     print 'y',y
     return x

def cfoo(x,y):
      code = """
      printf("Attemtping to call back foo() from C...\\n");
      py::tuple foo_args(2);
      py::object z;  // This will hold the return value of foo()
      foo_args[0] = x;
      foo_args[1] = y;
      z = foo.call(foo_args);
      printf("Exiting C code.\\n");
      return_val = z;
      """
      return inline(code,"foo x y".split() )

x=99
y="Hello"

print "Pure python..."
z=foo(x,y)
print "foo returned:",z
print "\nVia weave..."
z=cfoo(x,y)
print "cfoo returned:",z

# Complex numbers
def complex_test():
    a = zeros((4,4),Complex)
    a[0,0] = 1+2j
    a[1,1] = 2+3.5j
    print 'Before\n',a
    code = \
"""
std::complex<double> i(0, 1);
std::cout << a(1,1) << std::endl;
a(2,2) = 3.0+4.5*i;
//a(2,2).imag = 4.5;
"""
    inline(code,['a'],type_converters = converters.blitz)
    print 'After\n',a

complex_test()
```

```
#----------------------------------------------------------------------------
def sinC(x):
    """sinC(x) -> sin(x). Implemented using the C sin() function.
    """
    support = \
"""
#include <math.h>
"""
    code = \
"""
return_val = sin(x);
"""
    return inline(code,['x'],
                  type_converters = converters.blitz,
                  support_code = support,
                  libraries = ['m'],
                  )

def in_place_multNum(num,mat):
    mat *= num


from weave import inline
class bunch: pass

def oaccess():
    x=bunch()

    x.a = 1

    code = """ // BROKEN!
    // Try to emulate Python's: print 'x.a',x.a
    std::cout << "x.a " << x.a << std::endl;
    """
    inline(code,['x'])

main2 = oaccess


def ttest():
    nrun = 10
    size = 6000
    mat = ones((size,size),'d')
    num = 5.6
    tNum = time_test(nrun,in_place_multNum,*(num,mat))
    print 'time Num',tNum
    tC = time_test(nrun,in_place_mult,*(num,mat))
    print 'time C',tC

def main():
    print 'Printing comparisons:'
    print '\nPassing an int - what the C was coded for:'
    simple_print(42)
```

```
print '\nNow passing a float. C++ is fine (cout<< takes care of things)
     ...but C fails:'
simple_print(42.1)
print '\nAnd a string. Again, C++ is ok and C fails:'
simple_print('Hello World!')

A = zeros((3,3),'d')

A[0,0],A[1,1],A[2,2] = 1,2.5,3.3

print '\nMatrix A:\n',A
print 'Trace by two methods. Second fails, see code for details.'
print '\ntr(A)=',trace(A)
print '\ntr(A)=',trace2(A)

a = 5.6
print '\nMultiplying A in place by %s:' % a
in_place_mult(a,A)
print A

# now some simple operations with 3-vectors.
a = array([4.3,1.5,5.6])
b = array([0.8,2.9,3.8])

print '\nPython and C versions follow. Results should be identical:'
print 'a =',a
print 'b =',b

print '\nsum(a_i) =',sum(a)
print 'sum(a_i) =',sumC(a)

print '\na.b =',dot(a,b)
print 'a.b =',dot_product(a,b)

print '\na x b =',cross_product(a,b)
print 'a x b =',cross_productC(a,b)

print '\nIn-place versions.'
print 'a x b =',cross_productC2(a,b)
print 'a x b =',cross_productC3(a,b)

print '\nSimple functions using the C math library:'
import math
x = 3.5
n = 4
theta = math.pi/4.
print '\nx**'+str(n)+'=',x**n
print 'x**'+str(n)+'=',powC(x,n)
print '\nsin('+str(theta)+')=',math.sin(theta)
print 'sin('+str(theta)+')=',sinC(theta)

print '\nGlobal variables and explicitly typed blitz arrays.'
x = array([4,5,6])
print 'x is a Numeric array:\nx=',x
```

```
    print 'Now using weave:'
    Cglobals (x)

if __name__ == '__main__':
    main()
```

2. ctypes

Some quick notes about ctypes, to be finished later.

- **Classes**: _as_parameter_ attribute, one of: [int, str, unicode]. A property can be used to provide custom access. This allows any class to customize how it isseen if one of its instances is used as a parameter in a ctypes call.
- **Functions**: when the underlying ctypes-exposed functions is seen on the Python side, set its `.argtypes` attribute.

3. swig

4. f2py

This is a rough set of notes on how to use f2py. It does NOT substitute the official manual, but is rather meant to be used alongside with it.

For any non-trivial poject involving f2py, one should also keep at hand Pierre Schnizer's excellent 'A short introduction to F2PY', available from http://fubphpc.tu-graz.ac.at/~pierre/f2py_tutorial.tar.gz

4.1. Usage for the impatient. Start by building a scratch signature file automatically from your Fortran sources (in this case all, you can choose only those .f files you need):

```
f2py -m MODULENAME -h MODULENAME.pyf *.f
```

This writes the file MODULENAME.pyf, making the best guesses it can from the Fortran sources. It builds an interface for the module to be accessed as 'import adap1d' from python.

You will then edit the .pyf file to fine-tune the python interface exhibited by the resulting extension. This means for example making unnecessary scratch areas or array dimensions hidden, or making certain parameters be optional and take a default value.

Then, write your setup.py file using distutils, and list the .pyf file along with the Fortran sources it is meant to wrap. f2py will build the module for you automatically, respecting all the interface specifications you made in the .pyf file.

This approach is ultimately far easier than trying to get all the declarations (especially dependencies) right through Cf2py directives in the Fortran sources. While that may seem appealing at first, experience seems to show that it's ultimately far more time-consuming and prone to subtle errors. Using this approach, the first f2py pass can do the bulk of the interface writing and only fine-tuning needs to be done manually. I would only recommend embedded Cf2py directives for very simple problems (where it works very well).

The only drawback of this approach is that the interface and the original Fortran source lie in different files, which need to be kept in sync. This increases a bit the chances of forgetting to update the .pyf file if the Fortran interface changes (adding a parameter, for example). However, the benefit of having explicit, clear control over f2py's behavior far outweighs this concern.

4.2. Choosing a default compiler. Set the FC_VENDOR environment variable. This will then prevent f2py from testing all the compilers it knows about.

4.3. Using Cf2py directives. For simpler cases you may choose to go the route of Cf2py directives. Below are some tips and examples for this approach.

Here's the signature of a simple Fortran routine:

```
      subroutine phipol(j,mm,nodes,wei,nn,x,phi,wrk)

      implicit real *8 (a-h, o-z)
      real *8 nodes(*),wei(*),x(*),wrk(*),phi(*)
      real *8 sum, one, two, half
```

The above is correctly handled by f2py, but it can't know what is meant to be input/output and what the relations between the various variables are (such as integers which are array dimensions). If we add the following f2py directives, the generated python interface is a lot nicer:

```
      subroutine phipol(j,mm,nodes,wei,nn,x,phi,wrk)
c
c     Lines with Cf2py in them are directives for f2py to generate a better
c     python interface.  These must come _before_ the Fortran variable
c     declarations so we can control the dimension of the arrays in Python.
c
c     Inputs:
Cf2py integer check(0<=j && j<mm),depend(mm) :: j
Cf2py real *8 dimension(mm),intent(in) :: nodes
Cf2py real *8 dimension(mm),intent(in) :: wei
Cf2py real *8 dimension(nn),intent(in) :: x
c
c     Outputs:
Cf2py real *8 dimension(nn),intent(out),depend(nn) :: phi
c
c     Hidden args:
c     - scratch areas can be auto-generated by python
Cf2py real *8 dimension(2*mm+2),intent(hide,cache),depend(mm) :: wrk
c     - array sizes can be auto-determined
Cf2py integer intent(hide),depend(x):: nn=len(x)
Cf2py integer intent(hide),depend(nodes) :: mm = len(nodes)
c
      implicit real *8 (a-h, o-z)
      real *8 nodes(*),wei(*),x(*),wrk(*),phi(*)
      real *8 sum, one, two, half
```

Some comments on the above:

- The f2py directives should come immediately after the 'subroutine' line and before the Fortran variable lines. This allows the f2py dimension directives to override the Fortran var(*) directives.
- If the Fortran code uses var(N) instead of var(*), the f2py directives can be placed after the Fortran declarations. This mode is preferred, as there is less redundancy overall. The result is much simpler:

```
      subroutine phipol(j,mm,nodes,wei,nn,x,phi,wrk)
c
c     Lines with Cf2py in them are directives for f2py to generate a better
c     python interface.  These must come _before_ the Fortran variable
c     declarations so we can control the dimension of the arrays in Python.
c
c     Inputs:
Cf2py integer check(0<=j && j<mm),depend(mm) :: j
Cf2py real *8 dimension(mm),intent(in) :: nodes
Cf2py real *8 dimension(mm),intent(in) :: wei
Cf2py real *8 dimension(nn),intent(in) :: x
c
c     Outputs:
Cf2py real *8 dimension(nn),intent(out),depend(nn) :: phi
c
c     Hidden args:
c     - scratch areas can be auto-generated by python
Cf2py real *8 dimension(2*mm+2),intent(hide,cache),depend(mm) :: wrk
c     - array sizes can be auto-determined
Cf2py integer intent(hide),depend(x):: nn=len(x)
Cf2py integer intent(hide),depend(nodes) :: mm = len(nodes)
c
```

```
implicit real *8 (a-h, o-z)
real *8 nodes(*),wei(*),x(*),wrk(*),phi(*)
real *8 sum, one, two, half
```

Since python can automatically manage memory, it is possible to hide the need for manually passed 'work' areas. The C/python wrapper to the underlying fortran routine will allocate the memory for the needed work areas on the fly. This is done by specifying intent(hide,cache). 'hide' tells f2py to remove the variable from the argument list and 'cache' tells it to auto-generate it.

In cases where the allocation cost becomes a performance problem, one can remove the 'hide' part and make it an optional argument. In this case it will only be generated if not given. For this, the line above should be changed to:

```
Cf2py   real *8 dimension(2*mm+2), intent(cache), optional, depend(mm) :: wrk
```

Note that this should only be done after proving that the scratch areas are causing a performance problem. The `cache` directive causes f2py to keep cached copies of the scratch areas, so no unnecessary mallocs should be triggered.

Since f2py relies on NumPy arrays, all dimensions can be determined from the arrays themselves and it is not necessary to pass them explicitly.

With all this, the resulting f2py-generated docstring becomes:

```
phipol - Function signature:
  phi = phipol(j,nodes,wei,x)
Required arguments:
  j : input int
  nodes : input rank-1 array('d') with bounds (mm)
  wei : input rank-1 array('d') with bounds (mm)
  x : input rank-1 array('d') with bounds (nn)
Return objects:
  phi : rank-1 array('d') with bounds (nn)
```

4.4. Debugging. For debugging, use the –debug-capi option to f2py. This causes the extension modules to print detailed information while in operation. In distutils, this must be passed as an option in the f2py_options to the Extension constructor.

4.5. Wrapping C codes with f2py. Below is Pearu Peterson's (the f2py author) response to a question about using f2py to wrap existing C codes. While SWIG provides similar functionality and weave is perfect for inlining C, f2py seems to be an incredibly simple and convenient tool for wrapping C libraries.

Pearu's response follows:

For example, consider the following C file:

```
/* foo.c */
double foo(double *x, int n) {
  int i;
  double r = 0;
  for (i=0;i<n;++i)
    r += x[i];
  return r;
}
/* EOF foo.c */
```

To wrap the C function foo() with f2py, create the following signature file bar.pyf:

```
! -*- F90 -*-
python module bar
  interface
    real*8 function foo(x,n)
      intent(c) foo
      real*8 dimension(n),intent(in) :: x
      integer intent(c,hide),depend(x) :: n = len(x)
    end function foo
  end interface
end python module bar
! EOF bar.pyf
```

(see usersguide for more info about intent(c)) and run

```
f2py -c bar.pyf foo.c
```

Finally, in Python:

```
>>> import bar
>>> bar.foo([1,2,3])
6.0
```

4.6. Passing offset arrays to Fortran routines. It is possible to pass offset arrays (like pointers to the middle of other arrays) by using NumPy's slice notation.

The print_dvec function below simply prints its argument as "print*,'x',x". We show some examples of how it behaves with both 1 and 2-d arrays:

```
In [3]: x
Out[3]: array([ 2.8,  3.4,  4.1])
In [4]: tf.print_dvec(x)
 n 3
 x  2.8  3.4  4.1
In [5]: tf.print_dvec ?
Type:           fortran
String Form:    <fortran object at 0x8306fe8>
Namespace:      Currently not defined in user session.
Docstring:
    print_dvec - Function signature:
      print_dvec(x,[n])
    Required arguments:
      x : input rank-1 array('d') with bounds (n)
    Optional arguments:
      n := len(x) input int
In [6]: tf.print_dvec (x[1])
 n 1
 x  3.4
In [7]: tf.print_dvec (x[1:])
 n 2
 x  3.4  4.1
In [8]: A
Out[8]:
array([[ 3.5,  5.6,  8.2],
       [ 2.1,  4.5,  1.2],
       [ 6.3,  3.4,  3.1]])
In [9]: tf.print_dvec(A)
 n 9
 x  3.5  5.6  8.2  2.1  4.5  1.2  6.3  3.4  3.1
In [10]: A
Out[10]:
array([[ 3.5,  5.6,  8.2],
       [ 2.1,  4.5,  1.2],
       [ 6.3,  3.4,  3.1]])
In [11]: tf.print_dvec(A[1:])
 n 6
 x  2.1  4.5  1.2  6.3  3.4  3.1
In [12]: A[1:]
Out[12]:
array([[ 2.1,  4.5,  1.2],
       [ 6.3,  3.4,  3.1]])
In [13]: A[1:,1:]
Out[13]:
array([[ 4.5,  1.2],
       [ 3.4,  3.1]])
In [14]: tf.print_dvec(A[1:,1:])
```

```
n 4
x  4.5  1.2  3.4  3.1
```

4.7. On matrix ordering and in-memory copies. NumPy (which f2py relies on) is C-based, and therefore its arrays are stored in row-major order. Fortran stores its arrays in column-major order. This means that copying issues must be dealt with. Below we reproduce some comments from Pearu on this topic given in the f2py mailing list in June/2002:

> To avoid copying, you should create array that has internally Fortran data ordering. This is achived, for example, by reading/creating your data in Fortran ordering to NumPy array and then doing numpy.transpose on that. Every f2py generated extension module provides also function
>
> has_column_major_storage
>
> to check if an array is Fortran contiguous or not. If has_column_major_storage(arr) returns true then there will be no copying for the array arr if passed to f2py generated functions (assuming that the types are proper, of cource).
>
> Also note that copying done by f2py generated interface is carried out in C on the raw data and therefore it is extremely fast compared to if you would make a copy in Python, even when using NumPy. Tests with say 1000x1000 matrices show that there is no noticable performance hit when copying is carried out, in fact, sometimes making a copy may speed up things a bit – I was quite surprised about that myself.
>
> So, I think, you should worry about copying only if the sizes of matrices are really large, say, larger than 5000x5000 and efficient memory usage is relevant. The time spent for copying is negligible even for large arrays provided that your computer has plenty of memory (>=256MB).

4.8. Distutils. Below is an example setup.py file which generates a Python extension module from Fortran90 sources and a .pyf interface file generated by f2py and later fine tuned.

```
#!/usr/bin/env python
"""Setup script for F2PY-processed, Fortran based extension modules.

A typical call is:

% ./setup.py install --home=~/usr

This will build and install the generated modules in ~/usr/lib/python.

If called with no args, the script defaults to the above call form (it
automatically adds the 'install --home=~/usr' options)."""

# Global variables for this extension:
name         = "mwadap_tools"  # name of the generated python extension (.so)
description  = "F2PY-wrapped MultiWavelet Tree Toolbox"
author       = "Fast Algorithms Group - CU Boulder"
author_email = "fperez@colorado.edu"

# Necessary sources, _including_ the .pyf interface file
sources = """
binary_decomp.f90 binexpandx.f90 bitsequence.f90 constructwv.f90
display_matrix.f90 findkeypos.f90 findlevel.f90 findnodx.f90 gauleg.f90
gauleg2.f90 gauleg3.f90 ihpsort.f90 invert_f2cmatrix.f90 keysequence2d.f90
level_of_nsi.f90 matmult.f90 plegnv.f90 plegvec.f90 r2norm.f90 xykeys.f90

mwadap_tools.pyf""".split()

# Additional libraries required by our extension module (these will be linked
# in with -l):
```

```
libraries = ['m']

# Set to true (1) to turn on Fortran/C API debugging (very verbose)
debug_capi = 0

#********************************************************************
# Do not modify the code below unless you know what you are doing.

# Required modules
import sys,os
from os.path import expanduser,expandvars
from scipy_distutils.core import setup,Extension

expand_sh = lambda path: expanduser(expandvars(path))

# Additional directories for libraries (besides the compiler's defaults)
fc_vendor = os.environ.get('FC_VENDOR','Gnu').lower()
library_dirs = ["~/usr/lib/"+fc_vendor]

# Modify default arguments (if none are supplied) to install in ~/usr
if len(sys.argv)==1:
    default_args = 'install --home=~/usr'
    print '*** Adding default arguments to setup:',default_args
    sys.argv += default_args.split()  # it must be a list

# Additional options specific to f2py:
f2py_options = []
if debug_capi:
    f2py_options.append('--debug-capi')

# Define the extension module(s)
extension = Extension(name = name,
                      sources = sources,
                      libraries = libraries,
                      library_dirs = map(expand_sh,library_dirs),
                      f2py_options = f2py_options,
                      )

# Call the actual building/installation routine, in usual distutils form.
setup(name = name,
      description  = description,
      author       = author,
      author_email = author_email,
      ext_modules  = [extension],
      )
```

5. Others

boost, pyrex, cxx

6. Distributing standalone applications

py2exe, mcmillan installer

Part 2

Workbook
A Problem Collection

CHAPTER 6

Introduction to the workbook

This document contains a set of small problems, drawn from many different fields, meant to illustrate commonly useful techniques for using Python in scientific computing.

All problems are presented in a similar fashion: the task is explained including any necessary mathematical background and a 'code skeleton' is provided that is meant to serve as a starting point for the solution of the exercise. In some cases, some example output of the expected solution, figures or additional hints may be provided as well.

The accompanying source download for this workbook contains the complete solutions, which are not part of this document for the sake of brevity.

For several examples, the provided skeleton contains pre-written tests which validate the correctness of the expected answers. When you have completed the exercise successfully, you should be able to run it from within IPython and see something like this (illustrated using a trapezoidal rule problem, whose solution is in the file `trapezoid.py`):

```
In [7]: run trapezoid.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.003s

OK
```

This message tells you that 4 automatic tests were successfully executed. The idea of including automatic tests in your code is a common one in modern software development, and Python includes in its standard library two modules for automatic testing, with slightly different functionality: `unittest` and `doctest`. These tests were written using the `unittest` system, whose complete documentation can be found here: http://docs.python.org/lib/module-unittest.html.

Other exercises will illustrate the use of the `doctest` system, since it provides complementary functionality.

CHAPTER 7

Simple non-numerical Problems

1. Sorting quickly with QuickSort

Illustrates: lists, recursion.

Quicksort is one of the best known, and probably the simplest, fast algorithm for sorting n items. It is fast in the sense that it requires on average $\mathcal{O}(n \log n)$ comparisons instead of $\mathcal{O}(n^2)$, although a naive implementation does have quadratic worst-case behavior.

The algorithm uses a simple divide and conquer strategy, and its implementation is naturally recursive. Its basic steps are:

(1) Pick an element from the list, called the pivot p (any choice works).
(2) Select from the rest of the list those elements smaller and those greater than the pivot, and store them in separate lists S and G.
(3) Recursively apply the algorithm to S and G. The final result can be written as $\sigma(S) + [p] + \sigma(G)$, where σ represents the sorting operation, $+$ indicates list concatenation and $[p]$ is the list containing the pivot as its single element.

The listing 7.1 contains a skeleton with no implementation but with tests already written (in the form of *unit tests*, as described in the introduction).

LISTING 7.1. IGNORED

```
"""Simple quicksort implementation.

From http://en.wikipedia.org/wiki/Quicksort we have this pseudocode (see also
the C implementation for comparison).

Note: what follows is NOT python code, it's meant to only illustrate the
algorithm for you.  Below you'll need to actually implement it in real Python.
You may be surprised at how close a working Python implementation can be to
this pseudocode.


function quicksort(array)
     var list less, greater
     if length(array) <= 1
         return array
     select and remove a pivot value pivot from array
     for each x in array
         if x <= pivot then append x to less
         else append x to greater
     return concatenate(quicksort(less), pivot, quicksort(greater))
"""

def qsort(lst):
    """Return a sorted copy of the input list.

    Input:

      lst : a list of elements which can be compared.
```

```
    Examples:

    >>> qsort([])
    []

    >>> qsort([3,2,5])
    [2, 3, 5]
    """

    # Hint: remember that all recursive functions need an exit condition
    raise NotImplementedError('Original solution has 2 lines')

    # Select pivot and apply recursively
    raise NotImplementedError('Original solution has 3 lines')

    # Upon return, make sure to properly concatenate the output lists
    raise NotImplementedError('Original solution has 1 line')


#-----------------------------------------------------------------------------
# Tests
#-----------------------------------------------------------------------------
import random

import nose
import nose, nose.tools as nt

def test_sorted():
    seq = range(10)
    sseq = qsort(seq)
    nt.assert_equal(seq,sseq)

def test_random():
    tseq = range(10)
    rseq = range(10)
    random.shuffle(rseq)
    sseq = qsort(rseq)
    nt.assert_equal(tseq,sseq)

# If called from the command line, run all the tests
if __name__ == '__main__':
    # This call form is ipython-friendly
    nose.runmodule(argv=['-s','--with-doctest'],
                   exit=False)
```

Hints.

- Python has no particular syntactic requirements for implementing recursion, but it does have a maximum recursion depth. This value can be queried via the function `sys.getrecursionlimit()`, and it can be changed with `sys.setrecursionlimit(new_value)`.
- Like in all recursive problems, don't forget to implement an exit condition!
- If `L` is a list, the call `len(L)` provides its length.

2. Dictionaries for counting words

A common task in text processing is to produce a count of word frequencies. While NumPy has a builtin histogram function for doing numerical histograms, it won't work out of the box for couting discrete items, since it is a binning histogram for a range of real values.

But the Python language provides very powerful string manipulation capabilities, as well as a very flexible and efficiently implemented builtin data type, the *dictionary*, that makes this task a very simple one.

In this problem, you will need to count the frequencies of all the words contained in a compressed text file supplied as input.

The listing 7.2 contains a skeleton for this problem, with XXX marking various places that are incomplete.

LISTING 7.2. IGNORED

```
#!/usr/bin/env python
"""Word frequencies - count word frequencies in a string."""

def word_freq(text):
    """Return a dictionary of word frequencies for the given text."""

    freqs = {}
    for word in text.split():
        freqs[word] = freqs.get(word, 0) + 1
    return freqs

def print_vk(lst):
    """Print a list of value/key pairs nicely formatted in key/value order."""

    # Find the longest key: remember, the list has value/key paris, so the key
    # is element [1], not [0]
    #longest_key = max(map(lambda x: len(x[1]),lst))
    longest_key = max([len(word) for count, word in lst])
    # Make a format string out of it
    fmt = '%'+str(longest_key)+'s -> %s'
    # Do actual printing
    for v,k in lst:
        print fmt % (k,v)

def freq_summ(freqs,n=10):
    """Print a simple summary of a word frequencies dictionary.

    Inputs:
      - freqs: a dictionary of word frequencies.

    Optional inputs:
      - n: the number of """

    words,counts = freqs.keys(),freqs.values()
    # Sort by count
    items = zip(counts,words)
    items.sort()

    print 'Number of words:',len(freqs)
    print
    print '%d least frequent words:' % n
    print_vk(items[:n])
```

```
    print
    print '%d most frequent words:' % n
    print_vk(items[-n:])

if __name__ == '__main__':
    import gzip
    text = gzip.open('data/HISTORY.gz').read()
    freqs = word_freq(text)
    freq_summ(freqs,20)
```

Hints.

- The `print_vk` function is already provided for you as a simple way to summarize your results.
- You will need to read the compressed file `HISTORY.gz`. Python has facilities to do this without having to manually uncompress it.
- Consider 'words' simply the result of splitting the input text into a list, using any form of whitespace as a separator. This is obviously a very naïve definition of 'word', but it shall suffice for the purposes of this exercise.
- Python strings have a `.split()` method that allows for very flexible splitting. You can easily get more details on it in IPython:

```
In [2]: a = 'somestring'

In [3]: a.split?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
Namespace:      Interactive
Docstring:
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator.
```

The complete set of methods of Python strings can be viewed by hitting the TAB key in IPython after typing 'a.', and each of them can be similarly queried with the '?' operator as above. For more details on Python strings and their companion sequence types, see http://docs.python.org/library/stdtypes.html#sequence-types-str-unicode-list-tuple-buffer-xrange.

CHAPTER 8

Working with files, the internet, and numpy arrays

This section is a general overview to show how easy it is to load and manipulate data on the file system and over the web using python's built in data structures and numpy arrays. The goal is to exercise basic programming skills like building filename or web addresses to automate certain tasks like loading a series of data files or downloading a bunch of related files off the web, as well as to illustrate basic numpy and pylab skills.

1. Loading and saving ASCII data

The simplest file format is a plain text ASCII file of numbers. Although there are many better formats out there for saveing and loading data, this format is extremely common because it has the advantages of being human readable, and thus will survive the test of time as the *en vogue* programming languages, analysis applications and data formats come and go, it is easy to parse, and it is supported by almost all languages and applications.

In this exercise we will create a data set of two arrays, the first one regularly sampled time t from 0..2 seconds with 20 ms time step , and the second one an array `v` of sinusoidal voltages corrupted by some noise. Let's assume the sine wave has amplitude 2 V, frequency 10 Hz, and zero mean Gaussian distrubuted white noise with standard deviation 0.5 V. Your task is to write two scripts.

The first script should create the vectors `t` and `v`, plot the time series of `t` versus `v`, save them in a two dimensional numpy array `X`, and then dump the array `X` to a plain text ASCII file called `'noisy_sine.dat'`. The file will look like (not identical because of the noise)

```
0.00000000000000000e+00 1.55094782693481602e-02
2.00000000000000042e-02 2.49394458705700472e+00
4.00000000000000083e-02 9.49769407455173797e-01
5.99999999999999778e-02 -9.18577928752441375e-01
8.00000000000000167e-02 -2.81112759068906470e+00
... and so on
```

Here is the exercise skeleton of the script to create and plot the data file

LISTING 8.1. IGNORED

```
from scipy import arange, sin, pi, randn, zeros
import pylab as p

a = 2       # 2 volt amplitude
f = 10      # 10 Hz frequency
sigma = 0.5 # 0.5 volt standard deviation noise

# create the t and v and store them a 2D array X
t = arange(0.0, 2.0, 0.02)                # an evenly sampled time array
v = a*sin(2*f*pi*t) + sigma*randn(len(t)) # a noisy sine wave
X = zeros((len(t),2))                     # an empty output array
X[:,0] = t                                # add t to the first column
X[:,1] = v                                # add s to the 2nd column
p.save('data/noisy_sine.dat', X)            # save the output file as ASCII

# plot the arrays t vs v and label the x-axis, y-axis and title
# save the output figure as noisy_sine.png
p.plot(t, v, 'b-')
```

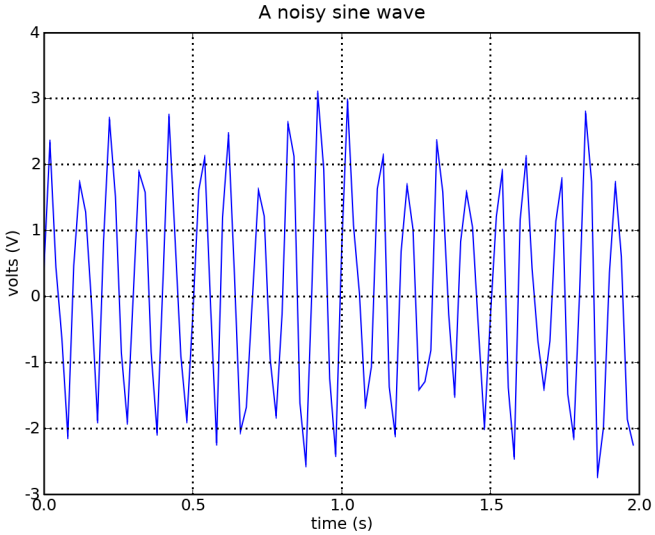

FIGURE 1. A 10 Hz sine wave corrupted by noise

```
p.xlabel('time (s)')
p.ylabel('volts (V)')
p.title('A noisy sine wave')
p.grid()
p.savefig('noisy_sine.png', dpi=150)
p.savefig('noisy_sine.eps')
p.show()
```

and the graph will look something like Figure 1

The second part of this exercise is to write a script which loads data from the data file into an array `X`, extracts the columns into arrays `t` and `v`, and computes the RMS (root-mean-square) intensity of the signal using the `load` command.

2. Working with CSV files

The CSV (Comma Separated Value) file specification is also an ASCII based, human readable format, but it is more powerful than simple flat ASCII files including headers, escape sequences and arbitrary delimiters like TAB, SPACE or COMMA. It is a widely used interchange format for sharing data between operating systems and programs like Excel, Matlab and statistical analysis packages.

A typical CSV file will be a mix of different data types: integers, floating point numbers, dates and strings. Of course, all of these are strings in the file, since all text files are made up of strings, but the data is typically representing some other numeric or date type. Python has very good support for handling different data types, so you don't need to try to force your data to look like a multi dimensional array of floating point numbers if this is not the natural way to describe your data. numpy provides a generalization of the array data structure we used above called record arrays, which allow to store data in a conceptual model similar to a database or spreadsheet: several named fields (eg 'date', 'weight', 'height', 'age') with different types (eg `datetime.date`, `float`, `float`, `int`).

In the example below, we will download some CSV files from Yahoo Financial web pages and load them into numpy record arrays for analysis and visualization. Go to `http://finance.yahoo.com` and enter a stock symbol in the entry boc labeled "Get Quotes". I will use `'SPY'` which is an index fund that tracks the S&P 500. In the left menu bar, there is an entry called "Historical Prices" which will take you to a page where you can download the price history of your stock. Near the bottom of this page you should see a "Download To Spreadsheet" link – instead of clicking on it, right click it and choose "Copy Link Location" and paste this into a python script or ipython session as a string named `url`. Eg, for SPY page better

```
url = 'http://ichart.finance.yahoo.com/table.csv?' +\
   's=SPY&d=9&e=20&f=2007&g=d&a=0&b=29&c=1993&ignore=.csv'
```

I've broken the url into two strings so they will fit on the page. If you spend a little time looking at this pattern, you can probably figure out what is going on. The URL is encoding the information about the stock, the variable `s` for the stock ticker, `d` for the latest month, `e` for the latest day, `f` for the latest year, `c` for the start year, and so on (similarly `a`, `b`, and `c` for the start month, day and year). This is handy to know, because below we will write some code to automate some downloads for a stock universe.

One of the great things about python is it's "batteries included" standard library, which includes support for dates, csv files and internet downloads. The example interactive session below shows how in just a few lines of code using python's `urllib` for retrieving information from the internet, and matplotlib's `csv2rec` function for loading numpy record arrays, we are ready to get to work analyzing some web based data. Comments have been added to a copy-and-paste from the interactive session

```
# import a couple of libraries we'll be needing
In [23]: import urllib
In [24]: import matplotlib.mlab as mlab

# this is the CSV file we'll be downloading
In [25]: url = 'http://ichart.finance.yahoo.com/table.csv?' +\
   's=SPY&d=9&e=20&f=2007&g=d&a=0&b=29&c=1993&ignore=.csv'

# this will grab that web file and save it as 'SPY.csv' on our local
# filesystem
In [27]: urllib.urlretrieve(url, 'SPY.csv')
Out[27]: ('SPY.csv', <httplib.HTTPMessage instance at 0x2118210>)

# here we use the UNIX command head to peak into the file, which is
# a comma separated and contains various types, dates, ints, floats
In [28]: !head SPY.csv
Date,Open,High,Low,Close,Volume,Adj Close
2007-10-19,153.09,156.48,149.66,149.67,295362200,149.67
2007-10-18,153.45,154.19,153.08,153.69,148367500,153.69
2007-10-17,154.98,155.09,152.47,154.25,216687300,154.25
2007-10-16,154.41,154.52,153.47,153.78,166525700,153.78
2007-10-15,156.27,156.36,153.94,155.01,161151900,155.01
2007-10-12,155.46,156.35,155.27,156.33,124546700,156.33
2007-10-11,156.93,157.52,154.54,155.47,233529100,155.47
2007-10-10,156.04,156.44,155.41,156.22,101711100,156.22
2007-10-09,155.60,156.50,155.03,156.48,94054300,156.48

# csv2rec will import the file into a numpy record array, inspecting
# the columns to determine the correct data type
In [29]: r = mlab.csv2rec('SPY.csv')

# the dtype attribute shows you the field names and data types.
# O4 is a 4 byte python object (datetime.date), f8 is an 8 byte
# float, i4 is a 4 byte integer and so on.  The > and < symbols
# indicate the byte order of multi-byte data types, eg big endian or
# little endian, which is important for cross platform binary data
# storage
In [30]: r.dtype
Out[30]: dtype([('date', '|O4'), ('open', '>f8'), ('high', '>f8'),
('low', '>f8'), ('close', '>f8'), ('volume', '>i4'), ('adj_close',
'>f8')])
```

```
# Each of the columns is stored as a numpy array, but the types are
# preserved.  Eg, the adjusted closing price column adj_close is a
# floating  point type, and the date column is a python datetime.date
In [31]: print r.adj_close
[ 149.67  153.69  154.25 ...,   34.68   34.61   34.36]
In [32]: print r.date
[2007-10-19 00:00:00 2007-10-18 00:00:00 2007-10-17 00:00:00 ...,
 1993-02-02 00:00:00 1993-02-01 00:00:00 1993-01-29 00:00:00]
```

For your exercise, you'll elaborate on the code here to do a batch download of a number of stock tickers in a defined stock universe. Define a function `fetch_stock(ticker)` which takes a stock ticker symbol as an argument and returns a numpy record array. Select the rows of the record array where the date is greater than 2003-01-01 and plot the returns $(p - p_0)/p_0$ where p are the prices and p_0 is the initial price. by date for each stock on the same plot. Create a legend for the plot using the matplotlib `legend` command, and print out a sorted list of final returns (eg assuming you bought in 2003 and held to the present) for each stock. Here is the exercise skeleton.:

LISTING 8.2. IGNORED

```
"""
Download historical pricing record arrays for a universe of stocks
from Yahoo Finance using urllib.  Load them into numpy record arrays
using matplotlib.mlab.csv2rec, and do some batch processing -- make
date vs price charts for each one, and compute the return since 2003
for each stock.  Sort the returns and print out the tickers of the 4
biggest winners
"""
import os, datetime, urllib
import matplotlib.mlab as mlab  # contains csv2rec
import numpy as npy
import pylab as p

def fetch_stock(ticker):
    """
    download the CSV file for stock with ticker and return a numpy
    record array.  Save the CSV file as TICKER.csv where TICKER is the
    stock's ticker symbol.

    Extra credit for supporting a start date and end date, and
    checking to see if the file already exists on the local file
    system before re-downloading it
    """
    fname = '%s.csv'%ticker
    url = 'http://ichart.finance.yahoo.com/table.csv?' +\
   's=%s&d=9&e=20&f=2007&g=d&a=0&b=29&c=1993&ignore=.csv'%ticker

    # the os.path module contains function for checking whether a file
    # exists
    if not os.path.exists(fname):
        urllib.urlretrieve(url, fname)
    r = mlab.csv2rec(fname)

    # note that the CSV file is sorted most recent date first, so you
    # will probably want to sort the record array so most recent date
    # is last
    r.sort()
    return r
```

```
tickers = 'SPY', 'QQQQ', 'INTC', 'MSFT', 'YHOO', 'GOOG', 'GE', 'WMT', 'AAPL'

# we want to compute returns since 2003, so define the start date
startdate = datetime.date(2003,1,1)

# we'll store a list of each return and ticker for analysis later
data = []   # a list of (return, ticker) for each stock
fig = p.figure()
for ticker in tickers:
    print 'fetching', ticker
    r = fetch_stock(ticker)

    # select the numpy records where r.date>=startdatre

    r = r[r.date>=startdate]
    price = r.adj_close                # set price equal to the adjusted
        ...close
    returns = (price-price[0])/price[0] # return is the (price-p0)/p0
    data.append((returns[-1], ticker))  # store the data

    # plot the returns by date for each stock
    p.plot(r.date, returns, label=ticker)

p.legend(loc='upper left')

# now sort the data by returns and print the results for each stock
data.sort()
for g, ticker in data:
    print '%s: %1.1f%%'%(ticker, 100*g)


p.savefig('stock_records.png', dpi=100)
p.savefig('stock_records.eps')
p.show()
```

The graph will look something like Figure 2.

3. Loading and saving binary data

ASCII is bloated and slow for working with large arrays, and so binary data should be used if performance is a consideration. To save an array `X` in binary form, you can use the numpy `tostring` method

```
In [16]: import numpy

# create some random numbers
In [17]: x = numpy.random.rand(5,2)

In [19]: print x
[[ 0.56331918  0.519582  ]
 [ 0.22685429  0.18371135]
 [ 0.19384767  0.27367054]
 [ 0.35935445  0.95795884]
 [ 0.37646642  0.14431089]]
```

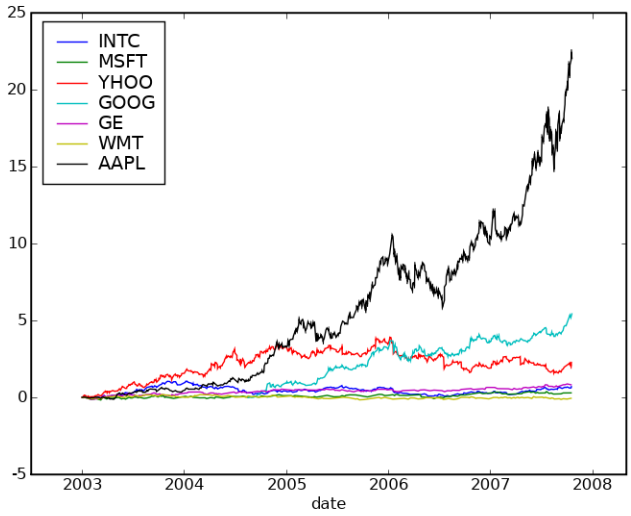

FIGURE 2. Returns for a universe of stocks since 2003

```
# save it to a data file in binary
In [20]: x.tofile(file('myx.dat', 'wb'))

# load it into a new array
In [21]: y = numpy.fromfile(file('myx.dat', 'rb'))

# the shape is not preserved, so we will have to reshape it
In [22]: print y
[ 0.56331918  0.519582    0.22685429  0.18371135  0.19384767
0.27367054
  0.35935445  0.95795884  0.37646642  0.14431089]

In [23]: y.shape
Out[23]: (10,)

# restore the original shape
In [24]: y.shape = 5, 2

In [25]: print y
[[ 0.56331918  0.519582  ]
 [ 0.22685429  0.18371135]
 [ 0.19384767  0.27367054]
 [ 0.35935445  0.95795884]
 [ 0.37646642  0.14431089]]
```

The advantage of numpy `tofile` and `fromfile` over ASCII data is that the data storage is compact and the read and write are very fast. It is a bit of a pain that that meta ata like array datatype and shape are not stored. In this format, just the raw binary numeric data is stored, so you will have to keep track of the data type and shape by other means. This is a good solution if you need to port binary data files between different packages, but if you know you will always be working in python, you can use the python pickle function to preserve all metadata (pickle also works with all standard python data types, but has the disadvantage that other programs and applications cannot easily read it)

```
# create a 6,3 array of random integers
In [36]: x = (256*numpy.random.rand(6,3)).astype(numpy.int)
```

```
In [37]: print x
[[173  38   2]
 [243 207 155]
 [127  62 140]
 [ 46  29  98]
 [  0  46 156]
 [ 20 177  36]]

# use pickle to save the data to a file myint.dat
In [38]: import cPickle

In [39]: cPickle.dump(x, file('myint.dat', 'wb'))

# load the data into a new array
In [40]: y = cPickle.load(file('myint.dat', 'rb'))

# the array type and share are preserved
In [41]: print y
[[173  38   2]
 [243 207 155]
 [127  62 140]
 [ 46  29  98]
 [  0  46 156]
 [ 20 177  36]]
```

CHAPTER 9

Elementary numerics

1. Wallis' slow road to π

Illustrates: arbitrary size integers, simple function definitions.
Wallis' formula is an infinite product that converges (slowly) to π:

$$\pi = \prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}. \tag{1}$$

The listing 9.1 contains a skeleton with no implementation but with some plotting commands already inserted, so that you can visualize the convergence rate of this formula as more terms are kept.

LISTING 9.1. IGNORED

```
#!/usr/bin/env python
"""Simple demonstration of Python's arbitrary-precision integers."""

# We need exact division between integers as the default, without manual
# conversion to float b/c we'll be dividing numbers too big to be represented
# in floating point.
from __future__ import division

def pi(n):
    """Compute pi using n terms of Wallis' product.

    Wallis' formula approximates pi as

    pi(n) = 2 \prod_{i=1}^{n}\frac{4i^2}{4i^2-1}."""

    num = 1
    den = 1
    for i in xrange(1,n+1):
        tmp = 4*i*i
        num *= tmp
        den *= tmp-1
    return 2.0*(num/den)

def part_range(n1,n2,nchunks):
    """Partition a range specification in nchunks"""
    size,rem = divmod(n2-n1,nchunks)
    sizes = [size]*nchunks
    while rem > 0:
        for i in range(nchunks):
            sizes[i] += 1
            rem -= 1
            if rem == 0:
                break

    # The sizes list has the offsets, now we need the actual start,stop pairs
```

```
    ranges = []
    start=n1
    for size in sizes:
        ranges.append((start,start+size))
        start += size
    return ranges

def wpi_nd(range_spec):
    """Compute pi using n terms of Wallis' product.

    Wallis' formula approximates pi as

    pi(n) = 2 \prod_{i=1}^{n}\frac{4i^2}{4i^2-1}."""

    n1,n2 = range_spec

    num = 1
    den = 1
    for i in xrange(n1,n2):
        tmp = 4*i*i
        num *= tmp
        den *= tmp-1

    return num,den

def par_pi(n,num_engines=1):
    """Compute pi using n terms of Wallis' product.

    Wallis' formula approximates pi as

    pi(n) = 2 \prod_{i=1}^{n}\frac{4i^2}{4i^2-1}.

    Parallel version."""

    num,den = reduce(lambda x,y:(x[0]*y[0],x[1]*y[1]),
                     map(wpi_nd,part_range(1,n+1,num_engines)))
    return 2.0*(num/den)


# This part only executes when the code is run as a script, not when it is
# imported as a library
if __name__ == '__main__':
    # Simple convergence demo.

    # A few modules we need
    import pylab as P
    import numpy as N

    # Create a list of points 'nrange' where we'll compute Wallis' formula
    nrange = N.linspace(10,2000,20).astype(int)
    # Make an array of such values
    wpi = N.array(map(pi,nrange))
    # Compute the difference against the value of pi in numpy (standard
    # 16-digit value)
```

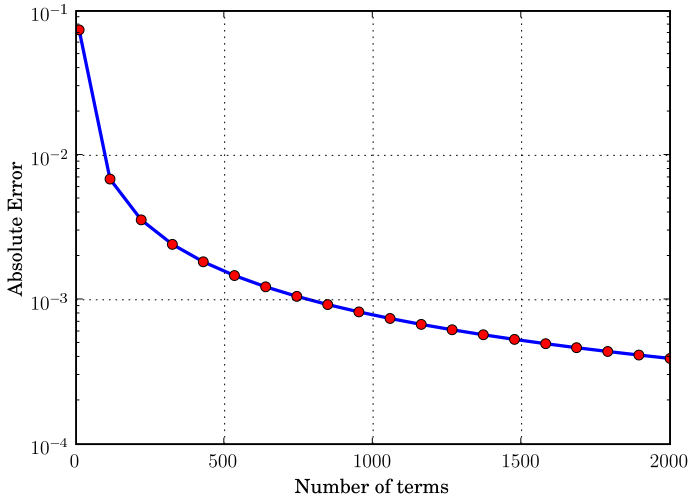

FIGURE 1. Convergence rate for Wallis' infinite product approximation to π.

```
diff = abs(wpi-N.pi)

# Make a new figure and build a semilog plot of the difference so we can
# see the quality of the convergence

P.figure()
# Line plot with red circles at the data points
P.semilogy(nrange,diff,'-o',mfc='red')

# A bit of labeling and a grid
P.title(r"Convergence of Wallis' product formula for $\pi$")
P.xlabel('Number of terms')
P.ylabel(r'Absolute Error')
P.grid()

# Display the actual plot
P.show()
```

After running the script successfully, you should obtain a plot similar to Figure 1.

2. Trapezoidal rule

Illustrates: basic array slicing, functions as first class objects.

In this exercise, you are tasked with implementing the simple trapezoid rule formula for numerical integration. If we want to compute the definite integral

$$\int_a^b f(x)dx \tag{2}$$

we can partition the integration interval $[a, b]$ into smaller subintervals, and approximate the area under the curve for each subinterval by the area of the trapezoid created by linearly interpolating between the two function values at each end of the subinterval. This is graphically illustrated in Figure 2, where the blue line represents the function $f(x)$ and the red line represents the successive linear segments.

The area under $f(x)$ (the value of the definite integral) can thus be approximated as the sum of the areas of all these trapezoids. If we denote by x_i ($i = 0, \ldots, n$, with $x_0 = a$ and $x_n = b$) the abscissas where

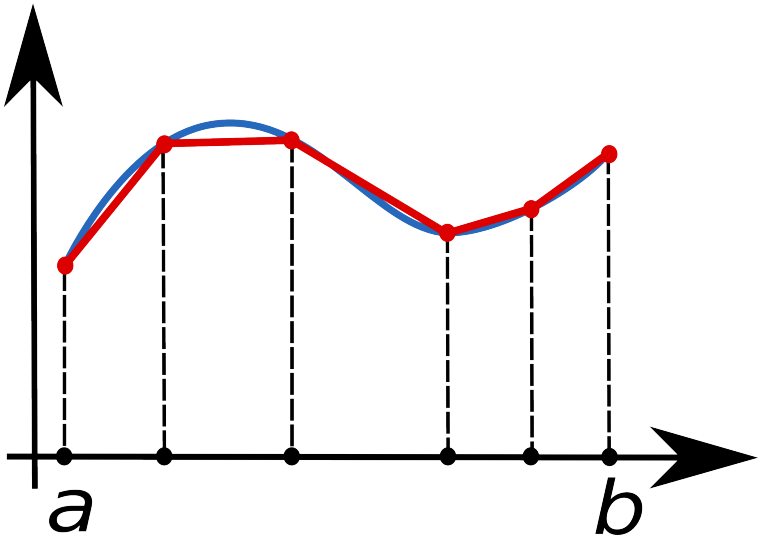

FIGURE 2. Illustration of the composite trapezoidal rule with a non-uniform grid (Image credit: Wikipedia).

the function is sampled, then

$$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{i=1}^{n}(x_i - x_{i-1})\,(f(x_i) + f(x_{i+1}))\,. \tag{3}$$

The common case of using equally spaced abscissas with spacing $h = (b-a)/n$ reads simply

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=1}^{n}(f(x_i) + f(x_{i+1}))\,. \tag{4}$$

One frequently receives the function values already precomputed, $y_i = f(x_i)$, so equation (3) becomes

$$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{i=1}^{n}(x_i - x_{i-1})\,(y_i + y_{i-1})\,. \tag{5}$$

Listing 9.2 contains a skeleton for this problem, written in the form of two incomplete functions and a set of automatic tests (in the form of *unit tests*, as described in the introduction).

LISTING 9.2. IGNORED

```
#!/usr/bin/env python
"""Simple trapezoid-rule integrator."""

import numpy as np

def trapz(x, y):
    """Simple trapezoid integrator for sequence-based innput.

    Inputs:
      - x,y: arrays of the same length (and more than one element).  If the
          ...two
    inputs have different lengths, a ValueError exception is raised.

    Output:
      - The result of applying the trapezoid rule to the input, assuming that
      y[i] = f(x[i]) for some function f to be integrated.

    Minimally modified from matplotlib.mlab."""
```

```
    # Sanity checks.
    #
    # Hint: if the two inputs have mismatched lengths or less than 2
    # elements, we raise ValueError with an explanatory message.
    raise NotImplementedError('Original solution has 4 lines')

    # Efficient application of trapezoid rule via numpy
    #
    # Hint: think of using numpy slicing to compute the moving difference in
    # the basic trapezoid formula.
    raise NotImplementedError('Original solution has 1 line')

def trapzf(f,a,b,npts=100):
    """Simple trapezoid-based integrator.

    Inputs:
      - f: function to be integrated.

      - a,b: limits of integration.

    Optional inputs:
      - npts(100): the number of equally spaced points to sample f at, between
      a and b.

    Output:
      - The value of the trapezoid-rule approximation to the integral."""

    # Hint: you will need to apply the function f to easch element of the
    # vector x.  What are several ways to do this?  Can you profile them to
        ...see
    # what differences in timings result for long vectors x?

    # Generate an equally spaced grid to sample the function.
    raise NotImplementedError('Original solution has 1 line')

    # For an equispaced grid, the x spacing can just be read off from the
        ...first
    # two points and factored out of the summation.
    raise NotImplementedError('Original solution has 1 line')

    # Sample the input function at all values of x
    #
    # Hint: you need to make an array out of the evaluations, and the python
    # builtin 'map' function can come in handy.
    raise NotImplementedError('Original solution has 1 line')

    # Compute the trapezoid rule sum for the final result
    raise NotImplementedError('Original solution has 1 line')


#-----------------------------------------------------------------------------
# Tests
#-----------------------------------------------------------------------------
```

```
import nose, nose.tools as nt
import numpy.testing as nptest

# A simple function for testing
def square(x): return x**2

def test_err():
    """Test that mismatched inputs raise a ValueError exception."""
    nt.assert_raises(ValueError,trapz,range(2),range(3))

def test_call():
    "Test a direct call with equally spaced samples. "
    x = np.linspace(0,1,100)
    y = np.array(map(square,x))
    nptest.assert_almost_equal(trapz(x,y),1./3,4)

def test_square():
    "Test integrating the square() function."
    nptest.assert_almost_equal(trapzf(square,0,1),1./3,4)

def test_square2():
    "Another test integrating the square() function."
    nptest.assert_almost_equal(trapzf(square,0,3,350),9.0,4)


# If called from the command line, run all the tests
if __name__ == '__main__':
    # This call form is ipython-friendly
    nose.runmodule(argv=['-s','--with-doctest'],
                   exit=False)
```

In this exercise, you'll need to write two functions, `trapz` and `trapzf`. `trapz` applies the trapezoid formula to pre-computed values, implementing equation (5), while `trapzf` takes a function f as input, as well as the total number of samples to evaluate, and computes eq. (4).

3. Newton's method

Illustrates: functions as first class objects, use of the scipy libraries.

Consider the problem of solving for t in

$$\int_o^t f(s)ds = u \tag{6}$$

where $f(s)$ is a monotonically increasing function of s and $u > 0$.

This problem can be simply solved if seen as a root finding question. Let

$$g(t) = \int_o^t f(s)ds - u, \tag{7}$$

then we just need to find the root for $g(t)$, which is guaranteed to be unique given the conditions above.

The SciPy library includes an optimization package that contains a Newton-Raphson solver called `scipy.optimize.newton`. This solver can optionally take a known derivative for the function whose roots are being sought, and in this case the derivative can be trivially computed in exact form.

For this exercise, implement the solution for the test function

$$f(t) = t\sin^2(t),$$

using

$$u = \frac{1}{4}.$$

The listing 9.3 contains a skeleton that includes for comparison the correct numerical value.

LISTING 9.3. IGNORED

```
#!/usr/bin/env python
"""Root finding using SciPy's Newton's method routines.
"""

from math import sin

from scipy.integrate import quad
from scipy.optimize import newton

# test input function
def f(t):
    # f(t): t * sin^2(t)
    raise NotImplementedError('Original solution has 1 line')

def g(t):
    "Exact form for g by integrating f(t)"
    u = 0.25
    return .25*(t**2-t*sin(2*t)+(sin(t))**2)-u

def gn(t):
    "g(t) obtained by numerical integration"
    u = 0.25
    # Hint: use quad, see its return value carefully.
    raise NotImplementedError('Original solution has 1 line')

# main
tguess = 10.0

print '"Exact" solution (knowing the analytical form of the integral)'
raise NotImplementedError('Original solution has 1 line')
print "t0, g(t0) =",t0,g(t0)

print
print "Solution using the numerical integration technique"
raise NotImplementedError('Original solution has 1 line')
print "t1, g(t1) =",t1,g(t1)

print
print "To six digits, the answer in this case is t==1.06601."
```

4. Bessel functions

Illustrates: Special functions library, array manipulations to check recursion relation.

In this exercise, you will verify a few simple relations involving the Bessel functions of the first kind. The important relations to keep in mind are the asymptotic form of $J_n(x)$ for $x >> n$:

$$J_n(x) \approx \sqrt{(\frac{2}{\pi x})}\cos(x - (n\frac{\pi}{2} + \frac{\pi}{4})) \tag{8}$$

and the recursion relation

$$J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x) \tag{9}$$

Once you get the code to run, you should see two figures like Figure 3 and Figure 4.

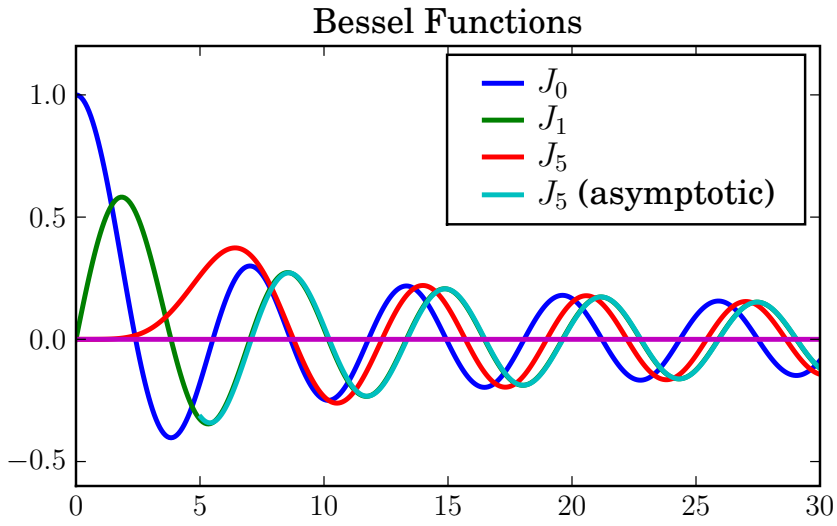

FIGURE 3. A few Bessel functions.

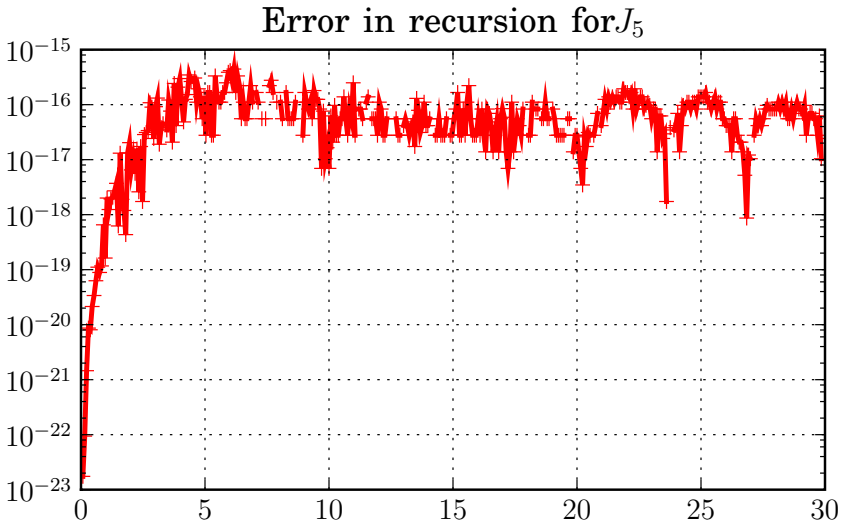

FIGURE 4. Numerical error for J_5.

LISTING 9.4. IGNORED

```
#!/usr/bin/env python
"""Plot some Bessel functions of integer order, using Scipy and pylab"""

from scipy import special

import numpy as np
import matplotlib.pyplot as plt

def jn_asym(n,x):
    """Asymptotic form of jn(x) for x>>n"""
```

```
    # The asymptotic formula is:
    # j_n(x) ~ sqrt(2.0/pi/x)*cos(x-(n*pi/2.0+pi/4.0))

    raise NotImplementedError('Original solution has 1 line')

# build a range of values to plot in
x = np.linspace(0,30,400)

# Start by plotting the well-known j0 and j1
plt.figure()
raise NotImplementedError('Original solution has 2 lines')

# Show a higher-order Bessel function
n = 5
plt.plot(x,special.jn(n,x),label='$J_%s$' % n)

# and compute its asymptotic form (valid for x>>n, where n is the order).  We
# must first find the valid range of x where at least x>n.
# Find where x>n and evaluate the asymptotic relation only there
raise NotImplementedError('Original solution has 2 lines')

# Finish off the plot
plt.legend()
plt.title('Bessel Functions')
# horizontal line at 0 to show x-axis, but after the legend
plt.axhline(0)

# Extra credit: redo the above, for the asymptotic range 0<x<<n.  The
# asymptotic form in this regime is:
#
# J(n,x) = (1/gamma(n+1))(x/2)^n

# Now, let's verify numerically the recursion relation
# J(n+1,x) = (2n/x)J(n,x)-J(n-1,x)
jn = special.jn  # just a shorthand

# Be careful to check only for x!=0, to avoid divisions by zero
# xp contains the positive values of x
raise NotImplementedError('Original solution has 1 line')

# construct both sides of the recursion relation, these should be equal
# Define j_np1  to hold j_(n+1) evaluated at the points xp
j_np1 = jn(n+1,xp)
# Define j_np1_rec to express j_(n+1) via a recursion relation, at points xp
j_np1_rec = (2.0*n/xp)*jn(n,xp)-jn(n-1,xp)

# Now make a nice error plot of the difference, in a new figure
plt.figure()

# We now plot the difference between the two formulas above.  Note that to
# properly display the errors, we want to use a logarithmic y scale.  Search
# the matplotlib docs for the proper calls.
raise NotImplementedError('Original solution has 1 line')
```

```
plt.title('Error in recursion for $J_%s$' % n)
plt.grid()

# Don't forget a show() call at the end of the script
plt.show()
```

CHAPTER 10

Linear algebra

Like matlab, numpy and scipy have support for fast linear algebra built upon the highly optimized LAPACK, BLAS and ATLAS fortran linear algebra libraries. Unlike Matlab, in which everything is a matrix or vector, and the '*' operator always means matrix multiple, the default object in numpy is an `array`, and the '*' operator on arrays means element-wise multiplication.

Instead, numpy provides a `matrix` class if you want to do standard matrix-matrix multiplication with the '*' operator, or the `dot` function if you want to do matrix multiplies with plain arrays. The basic linear algebra functionality is found in `numpy.linalg`

```
In [1]: import numpy as npy
In [2]: import numpy.linalg as linalg

# X and Y are arrays
In [3]: X = npy.random.rand(3,3)
In [4]: Y = npy.random.rand(3,3)

# * operator is element wise multiplication, not matrix matrix
In [5]: print X*Y
[[ 0.00973215  0.18086148  0.05539387]
 [ 0.00817516  0.63354021  0.2017993 ]
 [ 0.34287698  0.25788149  0.15508982]]

# the dot function will use optimized LAPACK to do matrix-matix
# multiply
In [6]: print npy.dot(X, Y)
[[ 0.10670678  0.68340331  0.39236388]
 [ 0.27840642  1.14561885  0.62192324]
 [ 0.48192134  1.32314856  0.51188578]]

# the matrix class will create matrix objects that support matrix
# multiplication with *
In [7]: Xm = npy.matrix(X)
In [8]: Ym = npy.matrix(Y)
In [9]: print Xm*Ym
[[ 0.10670678  0.68340331  0.39236388]
 [ 0.27840642  1.14561885  0.62192324]
 [ 0.48192134  1.32314856  0.51188578]]

# the linalg module provides functions to compute eigenvalues,
# determinants, etc.  See help(linalg) for more info
In [10]: print linalg.eigvals(X)
[ 1.46131600+0.j          0.46329211+0.16501143j  0.46329211-0.16501143j]
```

1. Glass Moiré Patterns

When a random dot pattern is scaled, rotated, and superimposed over the original dots, interesting visual patterns known as Glass Patterns emerge[1] In this exercise, we generate random dot fields using numpy's uniform distribution function, and apply transformations to the random dot field using a scale $\mathbf{S}$ and rotation $\mathbf{R}$ matrix $\mathbf{X_2} = \mathbf{SRX_1}$.

If the scale and rotation factors are small, the transformation is analogous to a single step in the numerical solution of a 2D ODE, and the plot of both $\mathbf{X_1}$ and $\mathbf{X_2}$ will reveal the structure of the vecotr field flow around the fixed point (the invariant under the transformation); see for example the *stable focus*, aka *spiral*, in Figure 1.

The eigenvalues of the tranformation matrix $\mathbf{M} = \mathbf{SR}$ determine the type of fix point: *center*, *stable focus*, *saddle node*, etc.... For example, if the two eigenvalues are real but differing in signs, the fixed point is a *saddle node*. If the real parts of both eigenvalues are negative and the eigenvalues are complex, the fixed point is a *stable focus*. The complex part of the eigenvalue determines whether there is any rotation in the matrix transformation, so another way to look at this is to break out the scaling and rotation components of the transformation $\mathbf{M}$. If there is a rotation component, then the fixed point will be a *center* or a *focus*. If the scaling components are both one, the rotation will be a *center*, if they are both less than one (contraction), it will be a *stable focus*. Likewise, if there is no rotation component, the fixed point will be a *node*, and the scaling components will determine the type of node. If both are less than one, we have a *stable node*, if one is greater than one and the other less than one, we have a *saddle node*.

LISTING 10.1. IGNORED

```
"""
Moire patterns from random dot fields

http://en.wikipedia.org/wiki/Moir%C3%A9_pattern

See L. Glass. 'Moire effect from random dots' Nature 223, 578580 (1969).
"""
import cmath
import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt


def myeig(M):
    """
    compute eigen values and eigenvectors analytically

    Solve quadratic:

      lamba^2 - tau*lambda +/- Delta = 0

    where tau = trace(M) and Delta = Determinant(M)

    if M = | a b |
           | c d |

    the trace is a+d and the determinant is a*d-b*c

    Return value is lambda1, lambda2
    """

    a,b = M[0,0], M[0,1]
    c,d = M[1,0], M[1,1]
```

[1] L. Glass. 'Moiré effect from random dots' Nature 223, 578580 (1969).

```
    tau = a+d       # the trace
    delta = a*d-b*c # the determinant

    lambda1 = (tau + cmath.sqrt(tau**2 - 4*delta))/2.
    lambda2 = (tau - cmath.sqrt(tau**2 - 4*delta))/2.
    return lambda1, lambda2

# 2000 random x,y points in the interval[-0.5 ... 0.5]
X1 = np.random.rand(2,2000)-0.5

#name =  'saddle'
#sx, sy, angle = 1.05, 0.95, 0.

name = 'center'
sx, sy, angle = 1., 1., 2.5

#name= 'stable focus'  # spiral
#sx, sy, angle = 0.95, 0.95, 2.5

theta = angle * cmath.pi/180.

S = np.array([[sx, 0],
              [0, sy]])

R = np.array([[np.cos(theta),  -np.sin(theta)],
              [np.sin(theta), np.cos(theta)],])

M = np.dot(S, R)  # rotate then stretch

# compute the eigenvalues using numpy linear algebra
vals, vecs = linalg.eig(M)
print 'numpy eigenvalues', vals

# compare with the analytic values from myeig
avals = myeig(M)
print 'analytic eigenvalues', avals

# transform X1 by the matrix
X2 = np.dot(M, X1)

# plot the original x,y as green dots and the transformed x, y as red
# dots
fig = plt.figure()
ax = fig.add_subplot(111)

x1 = X1[0]
y1 = X1[1]
x2 = X2[0]
y2 = X2[1]

ax = fig.add_subplot(111)
line1, line2 = ax.plot(x1, y1, 'go', x2, y2, 'ro', markersize=2)
ax.set_title(name)
```

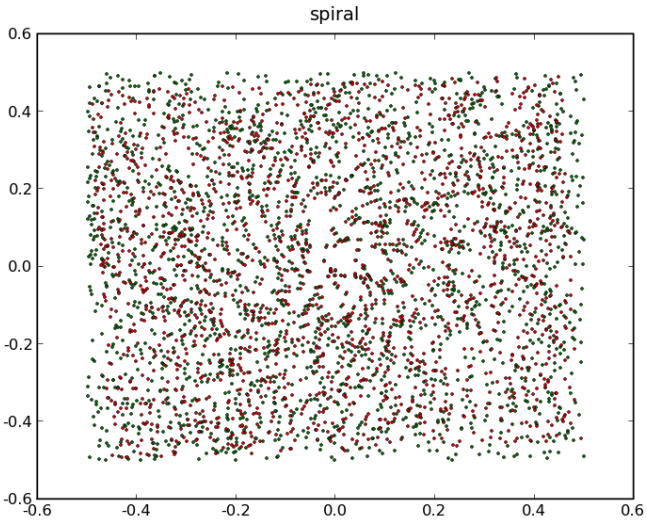

FIGURE 1. Glass pattern showing a stable focus

```
fig.savefig('glass_dots1.png', dpi=100)
fig.savefig('glass_dots1.eps', dpi=100)
plt.show()
```

CHAPTER 11

Signal processing

`numpy` and `scipy` provide many of the essential tools for digital signal processing. `scipy.signal` provides basic tools for digital filter design and filtering (eg Butterworth filters), a linear systems toolkit, standard waveforms such as square waves, and saw tooth functions, and some basic wavelet functionality. `scipy.fftpack` provides a suite of tools for Fourier domain analysis, including 1D, 2D, and ND discrete fourier transform and inverse functions, in addition to other tools such as analytic signal representations via the Hilbert trasformation (`numpy.fft` also provides basic FFT functions). `pylab` provides Matlab compatible functions for computing and plotting standard time series analyses, such as historgrams (`hist`), auto and cross correlations (`acorr` and `xcorr`), power spectra and coherence spectra (`psd`, `csd`, `cohere` and `specgram`).

1. Convolution

The output of a linear system is given by the convolution of its impulse response function with the input. Mathematically

$$y(t) = \int_0^t x(\tau) r(t-\tau) d\tau \tag{10}$$

This fundamental relationship lies at the heart of linear systems analysis. It is used to model the dynamics of calcium buffers in neuronal synapses, where incoming action potentials are represented as Dirac δ-functions and the calcium stores are represented with a response function with multiple exponential time constants. It is used in microscopy, in which the image distortions introduced by the lenses are *deconvolved* out using a measured point spread function to provide a better picture of the true image input. It is essential in structural engineering to determine how materials respond to shocks.

The impulse response function r is the system response to a pulsatile input. For example, in Figure 1 below, the response function is the sum of two exponentials with different time constants and signs. This is a typical function used to model synaptic current following a neuronal action potential. The figure shows three δ inputs at different times and with different amplitudes. The corresponsing impulse response for each input is shown following it, and is color coded with the impulse input color. If the system response is linear, by definition, the response to a sum of inputs is the sum of the responses to the individual inputs, and the lower panel shows the sum of the responses, or equivalently, the convolution of the impulse response function with the input function.

In Figure 1, the summing of the impulse response function over the three inputs is conceptually and visually easy to understand. Some find the concept of a convolution of an impulse response function with a continuos time function, such as a sinusoid or a noise process, conceptually more difficult. It shouldn't be. By the *sampling theorem*, we can represent any finite bandwidth continuous time signal as the sum of Dirac-δ functions where the height of the δ function at each time point is simply the amplitude of the signal at that time point. The only requirement is that the sampling frequency be at least as high as the Nyquist frequency, defined as the highest spectral frequency in the signal divided by 2. See Figure 2 for a representation of a delta function sampling of a damped, oscillatory, exponential function.

In the exercise below, we will convolve a sample from the normal distribution (white noise) with a double exponential impulse response function. Such a function acts as a low pass filter, so the resultant output will look considerably smoother than the input. You can use `numpy.convolve` to perform the convolution numerically.

We also explore the important relationship that a convolution in the tempoeral (or spatial) domain becomes a multiplication in the spectral domain, which is mathematically much easier to work with.

$$Y = R * X$$

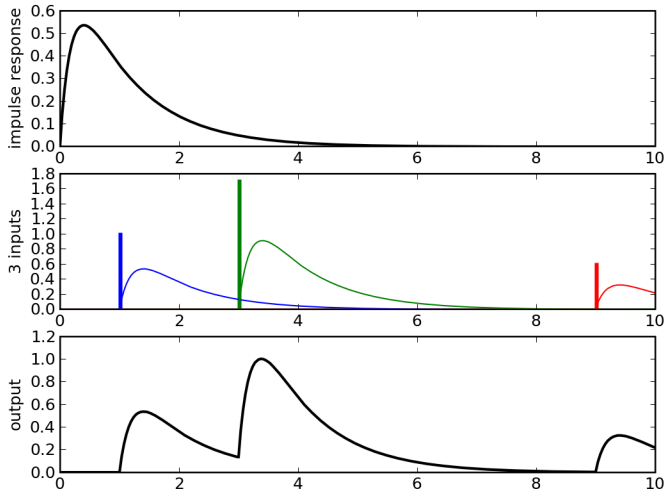

FIGURE 1. The output of a linear system to a series of impulse inputs is equal to the sum of the scaled and time shifted impulse response functions.

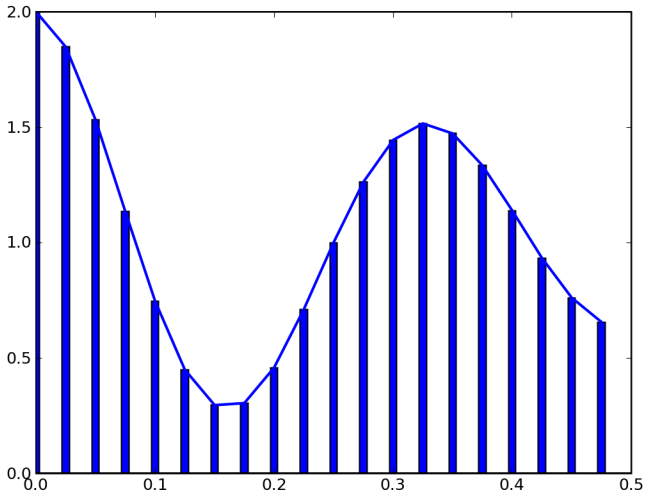

FIGURE 2. Representing a continuous time signal sampled as a sum of delta functions.

where Y, X, and R are the Fourier transforms of the respective variable in the temporal convolution equation above. The Fourier transform of the impulse response function serves as an amplitude weighting and phase shifting operator for each frequency component. Thus, we can get deeper insight into the effects of impulse response function r by studying the amplitude and phase spectrum of its transform R. In the example below, however, we simply use the multiplication property to perform the same convolution in Fourier space to confirm the numerical result from `numpy.convolve`.

LISTING 11.1. IGNORED

```
"""
In signal processing, the output of a linear system to an arbitrary
input is given by the convolution of the impule response function (the
system response to a Dirac-delta impulse) and the input signal.
```

```
Mathematically:

  y(t) = \int_0^\t x(\tau)r(t-\tau)d\tau


where x(t) is the input signal at time t, y(t) is the output, and r(t)
is the impulse response function.

In this exercise, we will compute investigate the convolution of a
white noise process with a double exponential impulse response
function, and compute the results

  * using numpy.convolve

  * in Fourier space using the property that a convolution in the
    temporal domain is a multiplication in the fourier domain
"""

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

# build the time, input, output and response arrays
dt = 0.01
t = np.arange(0.0, 20.0, dt)        # the time vector from 0..20
Nt = len(t)

def impulse_response(t):
    'double exponential response function'
    return (np.exp(-t) - np.exp(-5*t))*dt


x = np.random.randn(Nt)   # gaussian white noise

# evaluate the impulse response function, and numerically convolve it
# with the input x
r = impulse_response(t)               # evaluate the impulse function
y = np.convolve(x, r, mode='full')   # convultion of x with r
y = y[:Nt]

# compute y by applying F^-1[F(x) * F(r)].  The fft assumes the signal
# is periodic, so to avoid edge artificats, pad the fft with zeros up
# to the length of r + x do avoid circular convolution artifacts
R = np.fft.fft(r, len(r)+len(x)-1)
X = np.fft.fft(x, len(r)+len(x)-1)
Y = R*X

# now inverse fft and extract just the part up to len(x)
yi = np.fft.ifft(Y)[:len(x)].real

# plot t vs x, t vs y and yi, and t vs r in three subplots
fig = plt.figure()
ax1 = fig.add_subplot(311)
```

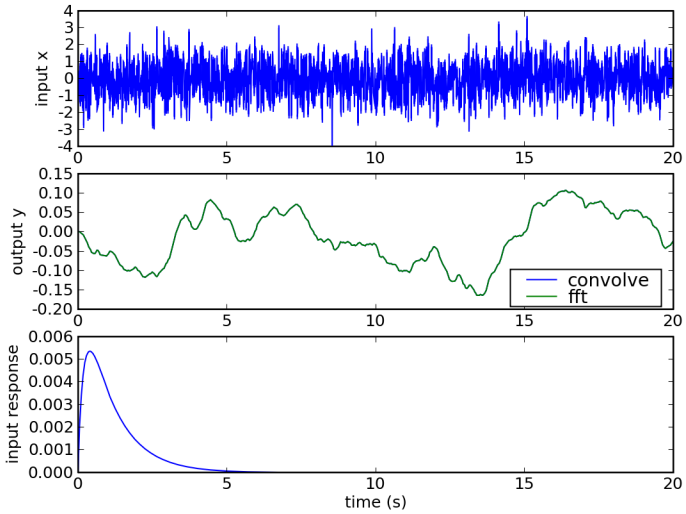

FIGURE 3. Convolution of a white noise process with a double exponential function computed with `numpy.fft` and `numpy.convolve`

```
ax1.plot(t, x)
ax1.set_ylabel('input x')

ax2 = fig.add_subplot(312)
ax2.plot(t, y, label='convolve')
ax2.set_ylabel('output y')

ax3 = fig.add_subplot(313)
ax3.plot(t, r)
ax3.set_ylabel('input response')
ax3.set_xlabel('time (s)')

ax2.plot(t, yi, label='fft')
ax2.legend(loc='best')

fig.savefig('convolution_demo.png', dpi=150)
fig.savefig('convolution_demo.eps')
plt.show()
```

2. FFT Image Denoising

Illustrates: 2-d image denoising, use of the scipy FFT library, array manipulations, image plotting.

Convolution of an input with with a linear filter in the termporal or spatial domain is equivalent to multiplication by the fourier transforms of the input and the filter in the spectral domain. This provides a conceptually simple way to think about filtering: transform your signal into the frequency domain, dampen the frequencies you are not interested in by multiplying the frequency spectrum by the desired weights, and then apply the inverse transform to the modified spectrum, back into the original domain. In the example below, we will simply set the weights of the frequencies we are uninterested in (the high frequency noise) to zero rather than dampening them with a smoothly varying function. Although this is not usually the best thing to do, since sharp edges in one domain usually introduce artifacts in another (eg high frequency "ringing"), it is easy to do and sometimes provides satisfactory results.

The image in the upper left panel of Figure 4 is a grayscale photo of the moon landing. There is a banded pattern of high frequency noise polluting the image. In the upper right panel we see the 2D spatial frequency spectrum. The FFT output in `scipy` is packed with the lower freqeuencies starting in the upper left, and proceeding to higher frequencies as one moves to the center of the spectrum (this is the most efficient way numerically to fill the output of the FFT algorithm). Because the input signal is real, the output spectrum is complex and symmetrical: the transformation values beyond the midpoint of the frequency spectrum (the Nyquist frequency) correspond to the values for negative frequencies and are simply the mirror image of the positive frequencies below the Nyquist (this is true for the 1D, 2D and ND FFTs in `numpy`).

In this exercise we will compute the 2D spatial frequency spectra of the luminance image, zero out the high frequency components, and inverse transform back into the spatial domain. We can plot the input and output images with the `pylab.imshow` function, but the images must first be scaled to be within the 0..1 luminance range. For best results, it helps to *amplify* the image by some scale factor, and then *clip* it to set all values greater than one to one. This serves to enhance contrast among the darker elements of the image, so it is not completely dominated by the brighter segments

LISTING 11.2. IGNORED

```
#!/usr/bin/env python
"""Simple image denoising example using 2-dimensional FFT."""

import sys

import numpy as np
from matplotlib import pyplot as plt

def plot_spectrum(F, amplify=1000):
    """Normalise, amplify and plot an amplitude spectrum."""

    # Note: the problem here is that we have a spectrum whose histogram is
    # *very* sharply peaked at small values.  To get a meaningful display, a
    # simple strategy to improve the display quality consists of simply
    # amplifying the values in the array and then clipping.

    # Compute the magnitude of the input F (call it mag).  Then, rescale mag
        ...by
    # amplify/maximum_of_mag.  Numpy arrays can be scaled in-place with ARR *=
    # number.  For the max of an array, look for its max method.
    raise NotImplementedError('Original solution has 2 lines')

    # Next, clip all values larger than one to one.  You can set all elements
    # of an array which satisfy a given condition with array indexing syntax:
    # ARR[ARR<VALUE] = NEWVALUE, for example.
    raise NotImplementedError('Original solution has 1 line')

    # Display: this one already works, if you did everything right with mag
    plt.imshow(mag, plt.cm.Blues)

if __name__ == '__main__':

    try:
        # Read in original image, convert to floating point for further
        # manipulation; imread returns a MxNx4 RGBA image.  Since the image is
        # grayscale, just extract the 1st channel
        # Hints:
        # - use plt.imread() to load the file
        # - convert to a float array with the .astype() method
```

```
        # - extract all rows, all columns, 0-th plane to get the first
        # channel
        # - the resulting array should have 2 dimensions only
        raise NotImplementedError('Original solution has 1 line')
        print "Image shape:",im.shape
    except:
        print "Could not open image."
        sys.exit(-1)

    # Compute the 2d FFT of the input image
    # Hint: Look for a 2-d FFT in np.fft.
    # Note: call this variable 'F', which is the name we'll be using below.
    raise NotImplementedError('Original solution has 1 line')

    # In the lines following, we'll make a copy of the original spectrum and
    # truncate coefficients.  NO immediate code is to be written right here.

    # Define the fraction of coefficients (in each direction) we keep
    keep_fraction = 0.1

    # Call ff a copy of the original transform.  Numpy arrays have a copy
    # method for this purpose.
    raise NotImplementedError('Original solution has 1 line')

    # Set r and c to be the number of rows and columns of the array.
    # Hint: use the array's shape attribute.
    raise NotImplementedError('Original solution has 1 line')

    # Set to zero all rows with indices between r*keep_fraction and
    # r*(1-keep_fraction):
    raise NotImplementedError('Original solution has 1 line')

    # Similarly with the columns:
    raise NotImplementedError('Original solution has 1 line')

    # Reconstruct the denoised image from the filtered spectrum, keep only the
    # real part for display.
    # Hint: There's an inverse 2d fft in the np.fft module as well (don't
    # forget that you only want the real part).
    # Call the result im_new,
    raise NotImplementedError('Original solution has 1 line')

    # Show the results
    # The code below already works, if you did everything above right.
    plt.figure()

    plt.subplot(221)
    plt.title('Original image')
    plt.imshow(im, plt.cm.gray)

    plt.subplot(222)
    plt.title('Fourier transform')
    plot_spectrum(F)
```

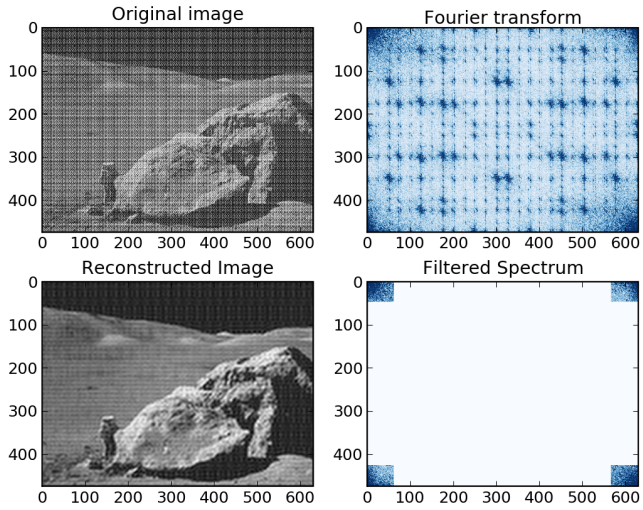

FIGURE 4. High freqeuency noise filtering of a 2D image in the Fourier domain. The upper panels show the original image (left) and spectral power (right) and the lower panels show the same data with the high frequency power set to zero. Although the input and output images are grayscale, you can provide colormaps to `pylab.imshow` to plot them in psudo-color

```
plt.subplot(224)
plt.title('Filtered Spectrum')
plot_spectrum(ff)

plt.subplot(223)
plt.title('Reconstructed Image')
plt.imshow(im_new, plt.cm.gray)

# Adjust the spacing between subplots for readability
plt.subplots_adjust(hspace=0.32)
plt.show()
```

CHAPTER 12

Dynamical systems

TODO

1. Lotka-Volterra Equations

LISTING 12.1. IGNORED

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

def dr(r, f):
    """
    return the derivative of *r* (the rabbit population) evaulated as a
    function of *r* and *f*.  The function should work whether *r* and *f*
    are scalars, 1D arrays or 2D arrays.  The return value should have
    the same dimensionality (shape) as the inputs *r* and *f*.
    """
    raise NotImplementedError('Original solution has 1 line')

def df(r, f):
    """
    return the derivative of *f* (the fox population) evaulated as a
    function of *r* and *f*.  The function should work whether *r* and *f*
    are scalars, 1D arrays or 2D arrays.  The return value should have
    the same dimensionality (shape) as the inputs *r* and *f*.
    """
    raise NotImplementedError('Original solution has 1 line')

def derivs(state, t):
    """
    Return the derivatives of R and F, stored in the *state* vector::

       state = [R, F]

    The return data should be [dR, dF] which are the derivatives of R
    and F at position state and time *t*
    """
    raise NotImplementedError('Original solution has 4 lines')

# the parameters for rabbit and fox growth and interactions
alpha, delta = 1, .25
beta, gamma = .2, .05

# the initial population of rabbits and foxes
r0 = 20
f0 = 10
```

```
# create a time array from 0..100 sampled at 0.1 second steps
raise NotImplementedError('Original solution has 1 line')


y0 = [r0, f0]  # the initial [rabbits, foxes] state vector

# integrate your ODE using scipy.integrate.  Read the help to see what
# is available
# HINT: see scipy.integrate.odeint
raise NotImplementedError('Original solution has 1 line')

# the return value from the integration is a Nx2 array.  Extract it
# into two 1D arrays caled r and f using numpy slice indexing
raise NotImplementedError('Original solution has 2 lines')

# time series plot: plot the population of rabbits and foxes as a
# funciton of time
plt.figure()
plt.plot(t, r, label='rabbits')
plt.plot(t, f, label='foxes')
plt.xlabel('time (years)')
plt.ylabel('population')
plt.title('population trajectories')
plt.grid()
plt.legend()
plt.savefig('lotka_volterra.png', dpi=150)
plt.savefig('lotka_volterra.eps')

# phase-plane plot: plot the population of foxes versus rabbits
# make sure you include and xlabel, ylabel and title
raise NotImplementedError('Original solution has 5 lines')


# Create 2D arrays for R and F to represent the entire phase plane --
# the point (R[i,j], F[i,j]) is a single (rabbit, fox) combinations.
# pass these arrays to the functions dr and df above to get 2D arrays
# of dR and dF evaluated at every point in the phase plance.
raise NotImplementedError('Original solution has 6 lines')


# Now find the nul-clines, for dR and dF respectively.  These are the
# points where dR=0 and dF=0 in the (R, F) phase plane.  You can use
# matplotlib's countour routine to find the zero level.  See the
# levels keyword to contour.  You will need a fine mesh of R and F,
# reevaluate dr and df on the finer grid, and use contour to find the
# level curves
raise NotImplementedError('Original solution has 7 lines')

plt.savefig('lotka_volterra_pplane.png', dpi=150)
plt.savefig('lotka_volterra_pplane.eps')
plt.show()
```

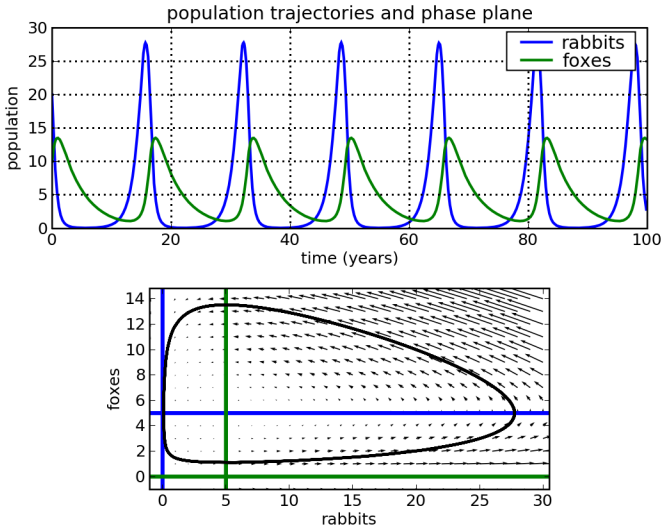

FIGURE 1. Upper panel shows population trajectories for rabbits (blue) and foxes (green) simulated using the Lotka-Volterra population dynamics equations. Lower panel shows the phase-plane trajectories, direction field and nullclines.

CHAPTER 13

Statistics

R, a statistical package based on S, is viewd by some as the best statistical software on the planet, and in the open source world it is the clear choice for sophisticated statistical analysis. Like python, R is an interpreted language written in C with an interactive shell. Unlike python, which is a general purpose programming language, R is a specialized statistical language. Since python is a excellent glue language, with facilities for providing a transparent interface to FORTRAN, C, C++ and other languages, it should come as no surprise that you can harness R's immense statistical power from python, through the rpy third part extension library.

However, R is not without its warts. As a language, it lacks python's elegance and advanced programming constructs and idioms. It is also GPL, which means you cannot distribute code based upon it unhindered: the code you distribute must be GPL as well (python, and the core scientific extension libraries, carry a more permissive license which support distribution in closed source, proprietary application).

Fortunately, the core tools scientific libraries for python (primarily numpy and scipy.stats) provide a wide array of statistical tools, from basic descriptive statistics (mean, variance, skew, kurtosis, correlation, ...) to hypothesis testing (t-tests, χ-Square, analysis of variance, general linear models, ...) to analytical and numerical tools for working with almost every discrete and continuous statistical distribution you can think of (normal, gamma, poisson, weibull, lognormal, levy stable, ...).

1. Descriptive statistics

The first step in any statistical analysis should be to describe, charaterize and importantly, visualize your data. The normal distribution (aka Gaussian or bell curve) lies at the heart of much of formal statistical analysis, and normal distributions have the tidy property that they are completely characterized by their mean and variance. As you may have observed in your interactions with family and friends, most of the world is not normal, and many statistical analyses are flawed by summarizing data with just the mean and standard deviation (square root of variance) and associated signficance tests (eg the T-Test) as if it were normally distributed data.

In the exercise below, we write a class to provide descriptive statistics of a data set passed into the constructor, with class methods to pretty print the results and to create a battery of standard plots which may show structure missing in a casual analysis. Many new programmers, or even experienced programmers used to a proceedural environment, are uncomfortable with the idea of classes, having hear their geekier programmer friends talk about them but not really sure what to do with them. There are many interesting things one can do with classes (aka object oriented programming) but at their hear they are a way of bundling data with methods that operate on that data. The self variable is special in python and is how the class refers to its own data and methods. Here is a toy example

```
In [115]: class MyData:
   .....:     def __init__(self, x):
   .....:         self.x = x
   .....:     def sumsquare(self):
   .....:         return (self.x**2).sum()
   .....:
   .....:

In [116]: nse = npy.random.rand(100)

In [117]: mydata.sumsquare()
Out[117]: 29.6851135284
```

LISTING 13.1. IGNORED

```
import scipy.stats as stats
from matplotlib.mlab import detrend_linear, load

import numpy
import pylab


class Descriptives:
    """
    a helper class for basic descriptive statistics and time series plots
    """
    def __init__(self, samples):
        self.samples = numpy.asarray(samples)
        self.N = len(samples)
        self.median = stats.median(samples)
        self.min = numpy.amin(samples)
        self.max = numpy.amax(samples)
        self.mean = stats.mean(samples)
        self.std = stats.std(samples)
        self.var = self.std**2.
        self.skew = stats.skew(samples)
        self.kurtosis = stats.kurtosis(samples)
        self.range = self.max - self.min

    def __repr__(self):
        """
        Create a string representation of self; pretty print all the
        attributes:

         N, median, min, max, mean, std, var, skew, kurtosis, range,
        """

        descriptives = (
            'N        = %d'        % self.N,
            'Mean     = %1.4f' % self.mean,
            'Median   = %1.4f'   % self.median,
            'Min      = %1.4f'  % self.min,
            'Max      = %1.4f'  % self.max,
            'Range    = %1.4f'  % self.range,
            'Std      = %1.4f' % self.std,
            'Skew     = %1.4f'     % self.skew,
            'Kurtosis = %1.4f' % self.kurtosis,
        )
        return '\n'.join(descriptives)

    def plots(self, figfunc, maxlags=20, Fs=1, detrend=detrend_linear,
              fmt='bo', bins=100,
              ):
        """
        plots the time series, histogram, autocorrelation and spectrogram

        figfunc is a figure generating function, eg pylab.figure
```

```
return an object which stores plot axes and their return
values from the plots.  Attributes of the return object are
'plot', 'hist', 'acorr', 'psd', 'specgram' and these are the
return values from the corresponding plots.  Additionally, the
axes instances are attached as c.ax1...c.ax5 and the figure is
c.fig

keyword args:

  Fs      : the sampling frequency of the data

  maxlags : max number of lags for the autocorr

  detrend : a function used to detrend the data for the correlation
       ...and spectral functions

  fmt     : the plot format string

  bins : the bins argument to hist
"""
data = self.samples

# Here we use a rather strange idiom: we create an empty do
# nothing class C and simply attach attributes to it for
# return value (which we carefully describe in the docstring).
# The alternative is either to return a tuple a,b,c,d or a
# dictionary {'a':someval, 'b':someotherval} but both of these
# methods have problems.  If you return a tuple, and later
# want to return something new, you have to change all the
# code that calls this function.  Dictionaries work fine, but
# I find the client code harder to use d['a'] vesus d.a.  The
# final alternative, which is most suitable for production
# code, is to define a custom class to store (and pretty
# print) your return object
class C: pass
c = C()
N = 5
fig = c.fig = figfunc()
fig.subplots_adjust(hspace=0.3)
ax = c.ax1 = fig.add_subplot(N,1,1)
c.plot = ax.plot(data, fmt)
ax.set_ylabel('data')

ax = c.ax2 = fig.add_subplot(N,1,2)
c.hist = ax.hist(data, bins)
ax.set_ylabel('hist')

ax = c.ax3 = fig.add_subplot(N,1,3)
c.acorr = ax.acorr(data, detrend=detrend, usevlines=True,
  maxlags=maxlags, normed=True)
ax.set_ylabel('acorr')

ax = c.ax4 = fig.add_subplot(N,1,4)
c.psd = ax.psd(data, Fs=Fs, detrend=detrend)
```

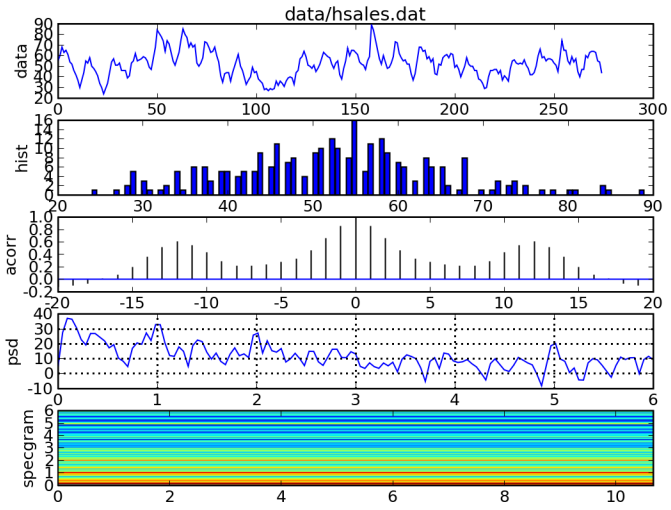

FIGURE 1.

```
        ax.set_ylabel('psd')

        ax = c.ax5 = fig.add_subplot(N,1,5)
        c.specgtram = ax.specgram(data, Fs=Fs, detrend=detrend)
        ax.set_ylabel('specgram')
        return c


if __name__=='__main__':

    # load the data in filename fname into the list data, which is a
    # list of floating point values, one value per line.  Note you
    # will have to do some extra parsing
    data = []
    fname = 'data/nm560.dat'  # tree rings in New Mexico 837-1987
    fname = 'data/hsales.dat'  # home sales
    for line in file(fname):
        line = line.strip()
        if not line: continue
        vals = line.split()
        val = vals[0]
        data.append(float(val))

    desc = Descriptives(data)
    print desc
    c = desc.plots(pylab.figure, Fs=12, fmt='-')
    c.ax1.set_title(fname)

    c.fig.savefig('stats_descriptives.png', dpi=150)
    c.fig.savefig('stats_descriptives.eps')
    pylab.show()
```

2. Statistical distributions

We explore a handful of the statistical distributions in `scipy.stats` module and the connections between them. The organization of the distribution functions in `scipy.stats` is quite elegant, with each distribution providing random variates (`rvs`), analytical moments (mean, variance, skew, kurtosis), analytic density (`pdf`, `cdf`) and survival functions (`sf`, `isf`) (where available) and tools for fitting empirical distributions to the analytic distributions (`fit`).

in the exercise below, we will simulate a radioactive particle emitter, and look at the empirical distribution of waiting times compared with the expected analytical distributions. Our radioative particle emitter has an equal likelihood of emitting a particle in any equal time interval, and emits particles at a rate of 20 Hz. We will discretely sample time at a high frequency, and record a 1 of a particle is emitted and a 0 otherwise, and then look at the distribution of waiting times between emissions. The probability of a particle emission in one of our sample intervals (assumed to be very small compared to the average interval between emissions) is proportional to the rate and the sample interval Δt, ie $p(\Delta t) = \alpha \Delta t$ where α is the emission rate in particles per second.

```
# a uniform distribution [0..1]
In [62]: uniform = scipy.stats.uniform()

# our sample interval in seconds
In [63]: deltat = 0.001

# the emission rate, 20Hz
In [65]: alpha = 20

# 1000 random numbers
In [66]: rvs = uniform.rvs(1000)

# a look at the 1st 20 random variates
In [67]: rvs[:20]
Out[67]:
array([ 0.71167172,  0.01723161,  0.25849255,  0.00599207,  0.58656146,
        0.12765225,  0.17898621,  0.77724693,  0.18042977,  0.91935639,
        0.97659579,  0.59045477,  0.94730366,  0.00764026,  0.12153159,
        0.82286929,  0.18990484,  0.34608396,  0.63931108,  0.57199175])

# we simulate an emission when the random number is less than
# p(Delta t) = alpha * deltat
In [84]: emit = rvs < (alpha * deltat)


# there were 3 emissions in the first 20 observations
In [85]: emit[:20]
Out[85]:
array([False,  True, False,  True, False, False, False, False, False,
       False, False, False, False,  True, False, False, False, False,
       False, False], dtype=bool)
```

The waiting times between the emissions should follow an exponential distribution (see `scipy.stats.expon`) with a mean of $1/\alpha$. In the exercise below, you will generate a long array of emissions, compute the waiting times between emissions, between 2 emissions, and between 10 emissions. These should approach an 1st order gamma (aka exponential) distribution, 2nd order gamma, and 10th order gamma (see `scipy.stats.gamma`). Use the probability density functions for these distributions in `scipy.stats` to compare your simulated distributions and moments with the analytic versions provided by `scipy.stats`. With 10 waiting times, we should be approaching a normal distribution since we are summing 10 waiting times and under the central limit theorem the sum of independent samples from a finite variance process approaches the normal distribution (see `scipy.stats.norm`). In the final part of

the exercise below, you will be asked to approximate the 10th order gamma distribution with a normal distribution. The results should look something like those in Figure 2.

LISTING 13.2. IGNORED

```
"""
Illustrate the connections bettwen the uniform, exponential, gamma and
normal distributions by simulating waiting times from a radioactive
source using the random number generator.  Verify the numerical
results by plotting the analytical density functions from scipy.stats
"""
import numpy
import scipy.stats
from pylab import figure, show, close


# N samples from a uniform distribution on the unit interval.  Create
# a uniform distribution from scipy.stats.uniform and use the "rvs"
# method to generate N uniform random variates
N = 100000
uniform = scipy.stats.uniform()  # the frozen uniform distribution
uninse = uniform.rvs(N)          # the random variates

# in each time interval, the probability of an emission
rate = 20.  # the emission rate in Hz
dx = 0.001  # the sampling interval in seconds
t = numpy.arange(N)*dx  # the time vector

# the probability of an emission is proportionate to the rate and the interval
emit_times = t[uninse < rate*dx]

# the difference in the emission times is the wait time
wait_times = numpy.diff(emit_times)

# plot the distribution of waiting times and the expected exponential
# density function lambda exp( lambda wt) where lambda is the rate
# constant and wt is the wait time; compare the result of the analytic
# function with that provided by scipy.stats.exponential.pdf; note
# that the scipy.stats.expon "scale" parameter is inverse rate
# 1/lambda.  Plot all three on the same graph and make a legend.
# Decorate your graphs with an xlabel, ylabel and title
fig = figure()
ax = fig.add_subplot(311)
p, bins, patches = ax.hist(wait_times, 100, normed=True)
l1, = ax.plot(bins, rate*numpy.exp(-rate * bins), lw=2, color='red')
l2, = ax.plot(bins, scipy.stats.expon.pdf(bins, 0, 1./rate),
        lw=2, ls='--', color='green')

ax.set_ylabel('PDF')
ax.set_title('waiting time densities of a %dHz Poisson emitter'%rate)
ax.text(0.05, 0.9, 'one interval', transform=ax.transAxes)
ax.legend((patches[0], l1, l2), ('simulated', 'analytic', 'scipy.stats.expon')
     ...)


# plot the distribution of waiting times for two events; the
```

```
# distribution of waiting times for N events should equal a N-th order
# gamma distribution (the exponential distribution is a 1st order
# gamma distribution.  Use scipy.stats.gamma to compare the fits.
# Hint: you can stride your emission times array to get every 2nd
# emission
wait_times2 = numpy.diff(emit_times[::2])
ax = fig.add_subplot(312)
p, bins, patches = ax.hist(wait_times2, 100, normed=True)
l1, = ax.plot(bins, scipy.stats.gamma.pdf(bins, 2, 0, 1./rate),
        lw=2, ls='-', color='red')

ax.set_ylabel('PDF')
ax.text(0.05, 0.9, 'two intervals', transform=ax.transAxes)
ax.legend((patches[0], l1), ('simulated', 'scipy.stats.gamma'))

# plot the distribution of waiting times for 10 events; again the
# distribution will be a 10th order gamma distribution so plot that
# along with the empirical density.  The central limit thm says that
# as we add N independent samples from a distribution, the resultant
# distribution should approach the normal distribution.  The mean of
# the normal should be N times the mean of the underlying and the
# variance of the normal should be 10 times the variance of the
# underlying.  HINT: Use scipy.stats.expon.stats to get the mean and
# variance of the underlying distribution.  Use scipy.stats.norm to
# get the normal distribution.  Note that the scale parameter of the
# normal is the standard deviation which is the square root of the
# variance
expon_mean, expon_var = scipy.stats.expon(0, 1./rate).stats()
mu, var = 10*expon_mean, 10*expon_var
sigma = numpy.sqrt(var)
wait_times10 = numpy.diff(emit_times[::10])
ax = fig.add_subplot(313)
p, bins, patches = ax.hist(wait_times10, 100, normed=True)
l1, = ax.plot(bins, scipy.stats.gamma.pdf(bins, 10, 0, 1./rate),
        lw=2, ls='-', color='red')
l2, = ax.plot(bins, scipy.stats.norm.pdf(bins, mu, sigma),
        lw=2, ls='--', color='green')

ax.set_xlabel('waiting times')
ax.set_ylabel('PDF')
ax.text(0.1, 0.9, 'ten intervals', transform=ax.transAxes)
ax.legend((patches[0], l1, l2), ('simulated', 'scipy.stats.gamma', 'normal
    ...approx'))

fig.savefig('stats_distributions.png', dpi=150)
fig.savefig('stats_distributions.eps')


show()
```

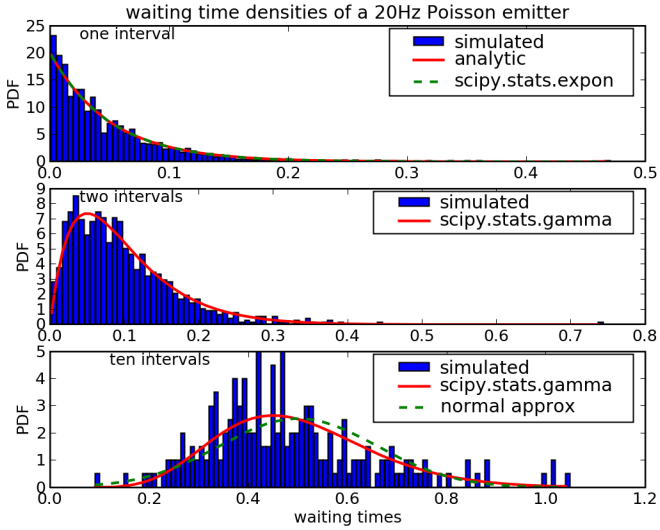
waiting time densities of a 20Hz Poisson emitter
one interval
simulated
analytic
scipy.stats.expon
PDF
two intervals
simulated
scipy.stats.gamma
PDF
ten intervals
simulated
scipy.stats.gamma
normal approx
PDF
waiting times

FIGURE 2.

CHAPTER 14

Plotting on maps

The matplotlib basemap toolkit is an add-on for matplotlib that provides the capability to draw maps of the earth in various map projections, and plot data on those maps. This section shows how to use basemap to create simple maps, draw coastlines and political boundaries, draw lines of constant latitude and longitude, and plot geophysical data on the maps.

1. Setting up the map.

In order to represent the curved surface of the earth in a two-dimensional map, a map projection is needed. Since this cannot be done without distortion, there are many map projections, each with it's own advantages and disadvantages. Basemap provides 19 different map projections. Some are global, some can only represent a portion of the globe. When a Basemap class instance is created, the desired map projection must be specified, along with information about the portion of the earth's surface that the map projection will describe. There are two basic ways of doing this. One is to provide the latitude and longitude values of each of the four corners of the rectangular map projection region. The other is to provide the lat/lon value of the center of the map projection region along with the width and height of the region in map projection coordinates. The first script illustrates how to use both of these methods to create a simple map. It also shows how to draw the continents and political boundaries on the map.

Here is an example script that creates a map by specifying the latitudes and longitudes of the four corners

LISTING 14.1. IGNORED

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
# create map by specifying lat/lon values at corners.
resolution = 'l'; projection = 'lcc'
lat_0 = 60; lon_0 = -50
llcrnrlat, llcrnrlon = 8, -92
urcrnrlat, urcrnrlon  = 39, 63
m = Basemap(lat_0=lat_0,lon_0=lon_0,\
            llcrnrlat=llcrnrlat,llcrnrlon=llcrnrlon,\
            urcrnrlat=urcrnrlat,urcrnrlon=urcrnrlon,\
            resolution=resolution,projection=projection)
# draw coastlines. Make liness a little thinner than default.
m.drawcoastlines(linewidth=0.5)
# background fill color will show ocean areas.
m.drawmapboundary(fill_color='aqua')
# fill continents, lakes within continents.
m.fillcontinents(color='coral',lake_color='aqua')
# draw states and countries.
m.drawcountries()
m.drawstates()
plt.title('map region specified using corner lat/lon values')
plt.show()
```

After running this script, you should see a plot that looks similar to Figure 1.

Here is an example script that creates a map by specifying the center of the map, plus the width and height in meters.

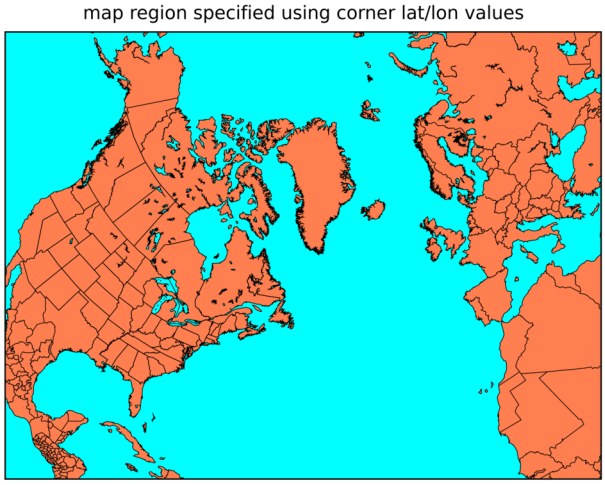

FIGURE 1. A map created by specifying the latitudes and longitudes of the four corners.

LISTING 14.2. IGNORED

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
# create map by specifying width and height in km.
resolution = 'l'; projection = 'lcc'
lon_0 = -50; lat_0 = 60
width = 12000000; height = 0.75*width
m = Basemap(lon_0=lon_0,lat_0=lat_0,\
            width=width,height=height,\
            resolution=resolution,projection=projection)
m.drawcoastlines(linewidth=0.5)
m.drawmapboundary(fill_color='aqua')
m.fillcontinents(color='coral',lake_color='aqua')
m.drawcountries()
m.drawstates()
plt.title('map region specified using width and height')
plt.show()
```

After running this script, you should see a plot that looks nearly identical to Figure 1.

The Basemap class instance can be used to convert latitudes and longitudes to coordinates on the map. To do this, simply call the instance as if it were a function, passing it the longitude and latitudes values to convert. The corresponding x and y values in map projection coordinates will be returned. The following example script shows how to use this to plot the locations of two cities (New York and London). The Basemap method drawgreatcircle is then used to draw the great circle route between these cities on the map.

LISTING 14.3. IGNORED

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
# create map by specifying width and height in km.
resolution = 'l'; projection = 'lcc'
lon_0 = -50; lat_0 = 60
width = 12000000; height = 0.75*width
m = Basemap(lon_0=lon_0,lat_0=lat_0,width=width,height=height,
```

```
            resolution=resolution,projection=projection)
# nylat, nylon are lat/lon of New York
nylat = 40.78
nylon = -73.98
# lonlat, lonlon are lat/lon of London.
lonlat = 51.53
lonlon = 0.08
# convert these points to map projection coordinates
# (using __call__ method of Basemap instance)
ny_x, ny_y = m(nylon, nylat)
lon_x, lon_y = m(lonlon, lonlat)
# plot black dots at the two points.
# make sure dots are drawn on top of other plot elements (zorder=10)
m.scatter([ny_x,lon_x],[ny_y,lon_y],25,color='k',marker='o',zorder=10)
# connect the dots along a great circle.
m.drawgreatcircle(nylon,nylat,lonlon,lonlat,linewidth=2,color='k')
# put the names of the cities to the left of each dot, offset
# by a little. Use a bold font.
plt.text(ny_x-100000,ny_y+100000,'New York',fontsize=12,\
           color='k',horizontalalignment='right',fontweight='bold')
plt.text(lon_x-100000,lon_y+100000,'London',fontsize=12,\
           color='k',horizontalalignment='right',fontweight='bold')
m.drawcoastlines(linewidth=0.5)
m.drawmapboundary(fill_color='aqua')
m.fillcontinents(color='coral',lake_color='aqua')
m.drawcountries()
m.drawstates()
plt.title('NY to London Great Circle')
plt.show()
```

This should produce something similar to Figure 2.

FIGURE 2. Drawing the locations of two cities, and connecting them along a great circle.

Most maps include a graticule grid, a reference network of labelled latitude and longitude lines. Basemap does this with the drawparallels and drawmeridians instance methods. The longitude and latitude

lines can be labelled where the intersect the map projection boundary. Following is an example script that draws a graticule on the map we've been working with.

LISTING 14.4. IGNORED

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np
# create map by specifying width and height in km.
resolution = 'l'
lon_0 = -50
lat_0 = 60
projection = 'lcc'
width = 12000000
height = 0.75*width
m = Basemap(lon_0=lon_0,lat_0=lat_0,width=width,height=height,
            resolution=resolution,projection=projection)
m.drawcoastlines(linewidth=0.5)
m.drawmapboundary(fill_color='aqua')
m.fillcontinents(color='coral',lake_color='aqua')
m.drawcountries()
m.drawstates()
# label meridians where they intersect the left, right and bottom
# of the plot frame.
m.drawmeridians(np.arange(-180,181,20),labels=[1,1,0,1])
# label parallels where they intersect the  left, right and top
# of the plot frame.
m.drawparallels(np.arange(-80,81,20),labels=[1,1,1,0])
plt.title('labelled meridians and parallels',y=1.075)
plt.show()
```

Running this script should produce a plot that looks like Figure 3.

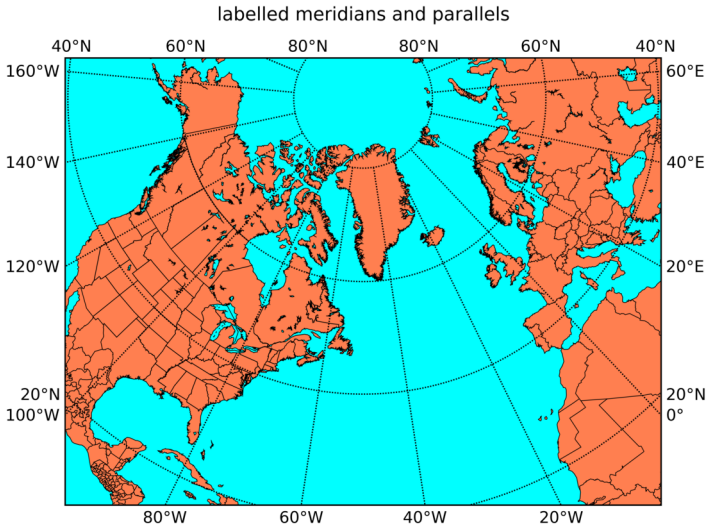

FIGURE 3. Drawing labelled meridians and parallels on the map (a graticule grid).

2. Plotting geophysical data on the map.

One of the most common uses of Basemap is to visualize earth science data, such as output from climate models. These data often come on latitude/longitude grids. One common data format for storing such grids is NetCDF. Basemap includes a NetCDF file reader (written in pure python by Roberto De Almeida). You can also access remote datasets over the web using the OPeNDAP protocol - just give the NetCDFFile function a URL instead of a local file name and Roberto's pydap module (`http://pydap.org`) will be used. The pydap client is included in Basemap. If the PyNIO module (`http://www.pyngl.ucar.edu/Nio.shtml`) is installed, the NetCDFFile function can also be used to open the formats that PyNIO supports, like GRIB and HDF. Following is an example of how to read sea-surface temperature data from a NetCDF file and plot it on a global mollweide projection.

LISTING 14.5. IGNORED

```
from mpl_toolkits.basemap import Basemap, NetCDFFile
import matplotlib.pyplot as plt
import numpy as np
# read in netCDF sea-surface temperature data
# can be a local file, a URL for a remote opendap dataset,
# or (if PyNIO is installed) a GRIB or HDF file.
ncfile = NetCDFFile('data/sst.nc')
sst = ncfile.variables['sst'][:]
lats = ncfile.variables['lat'][:]
lons = ncfile.variables['lon'][:]
# create Basemap instance for mollweide projection.
# coastlines not used, so resolution set to None to skip
# continent processing (this speeds things up a bit)
m = Basemap(projection='moll',lon_0=0,lat_0=0,resolution=None)
# compute map projection coordinates of grid.
x, y = m(*np.meshgrid(lons, lats))
# plot with pcolor
im = m.pcolormesh(x,y,sst,shading='flat',cmap=plt.cm.gist_ncar)
# draw parallels and meridians, but don't bother labelling them.
m.drawparallels(np.arange(-90.,120.,30.))
m.drawmeridians(np.arange(0.,420.,60.))
# draw line around map projection limb.
# color map region background black (missing values will be this color)
m.drawmapboundary(fill_color='k')
# draw horizontal colorbar.
plt.colorbar(orientation='horizontal')
plt.show()
```

The resulting plot should look like Figure 4.

Basemap also is capable of reading ESRI shapefiles, a very common GIS format. The script fillstates.py in the examples directory of the basemap source distribution shows how to read and plot polygons in a shapefile. There are many other useful examples in that directory that illustrate various ways of using basemap.

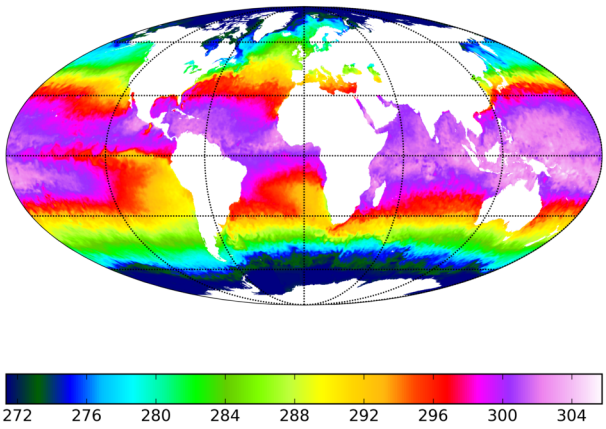

FIGURE 4. Sea surface temperature on a global mollweide projection.

CHAPTER 15

Performance python: interfacing with other languages

pyrex is a pure python packages that utilizes a custom language which is a hybid of C and python to write code that looks like python, but is converted by pyrex into python C extension code. It can be used to write custom C extension modules in a python like module to remove performance bottlenecks in code, as well as to wrap and existing C API with a python binding. pyrex generates C code, so you can use it to automatically generate C extensions that you can ship with your code and users can build your code without pyrex installed.

1. Writing C extensions pyrex

The canonical pyrex example generates a list of N prime numbers, and illustrates the hybrid nature of pyrex syntax

```
# name this file with the pyx extension for pyrex, rather than the py
# extension for python, eg primes.pyx
def primes(int kmax):
    # pyrex uses cdef to declare a c type
    cdef int n, k, i
    cdef int p[1000]

    # you can use normal python too, eg a python list
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] <> 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

To build our python extension, we will use the pyrex.distutils extensions. Here is a typical setup.py

```
from distutils.core import setup

# we use the Pyrex distutils Extension class rather than the standard
# python one
#from distutils.extension import Extension

from Pyrex.Distutils.extension import Extension
from Pyrex.Distutils import build_ext
```

```
setup(
  name = 'Demos',
  ext_modules=[
    Extension("primes",       ["primes.pyx"]),
    ],
  cmdclass = {'build_ext': build_ext}
)
```

and we can build it in place using

```
python setup.py build_ext --inplace
```

This creates a primes.c module which is the generated C code that we can ship with our python code to users who may not have `pyrex` installed, and a primes.so file which is the python shared library extension. We can now fire up ipython, import primes, and call our function with C performance. Here is an example shell session in which we build and test our new extension code

```
# our single pyx file from above
pyrex_demos> ls primes*
primes.pyx

# build the module in place
pyrex_demos> python setup.py build_ext --inplace
running build_ext
pyrexc primes.pyx --> primes.c
building 'primes' extension
creating build
creating build/temp.macosx-10.3-fat-2.5
gcc -arch ppc -arch i386 -isysroot /Developer/SDKs/MacOSX10.4u.sdk -fno-strict
     ...-aliasing -Wno-long-double -no-cpp-precomp -mno-fused-madd -fno-common
     ...-dynamic -DNDEBUG -g -O3 -I/Library/Frameworks/Python.framework/
     ...Versions/2.5/include/python2.5 -c primes.c -o build/temp.macosx-10.3-
     ...fat-2.5/primes.o
gcc -arch i386 -arch ppc -isysroot /Developer/SDKs/MacOSX10.4u.sdk -g -bundle
     ...-undefined dynamic_lookup build/temp.macosx-10.3-fat-2.5/primes.o -o
     ...primes.so

# now we have the original pyx and also the autogenerated C file and
# the extension module
pyrex_demos> ls primes*
primes.cprimes.pyxprimes.so

# let's test drive this in ipython
pyrex_demos> ipython
IPython 0.8.3.svn.r2876 -- An enhanced Interactive Python.

In [1]: import primes

In [2]: dir(primes)
Out[2]: ['__builtins__', '__doc__', '__file__', '__name__', 'primes']

In [3]: print primes.primes(20)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```

2. Working with numpy arrays

numpy arrays are the core of high performance computing in python, and one of the most common data formats for passing large data sets around between pyhton code and other wrappers. There are many things that arrays do very well and are practically as fast as a native C or FORTRAN implementations, eg convolutions and FFTs. But there are somethings that can be painfully slow in python when working with arrays, for example iterative algorithms over an array of values. For these cases, it is nice to be able to quickly generate some python extension code for working with numpy array data.

numpy provides a file which exposes its C API for use in pyrex extension code, you can find it, and another file which numpy uses to expose the requisite bits of the Python C API which it needs, in the numpy source code directory numpy/doc/pyrex. These files are c_numpy.pxd and c_python.pxd. In addition, numpy provides and example file numpyx.pyxthat shows you how to build a pyx extension file for multi-dimensional array sof different data types (eg int, float, python object). Here we will be a little less ambitious for starters, and write a simple toy function that sums a 1D array of floats.

```
# import the numpy c API (you need to have c_python.pxd and
# c_numpy.pxd from the numpy source directory in your build directory
cimport c_numpy

# since this is pyrex, we can import normal python modules too
import numpy

# numpy must be initialized -- don't forget to do this when writing
# numpy extension code.  It's a common gotcha
c_numpy.import_array()

def sum_elements(c_numpy.ndarray arr):
    cdef int i
    cdef double x, val

    x = 0.
    val = 0.
    for i from 0<=i<arr.dimensions[0]:
        val = (<double*>(arr.data + i*arr.strides[0]))[0]
        x = x + val

    return x
```

This exercise introduces pyrex to wrap a C library for trailing statistics.

Computation of trailing windowed statistics is common in many quantitative data driven disciplines, particularly where there is noisy data. Common uses of windowed statistics are the trailing moving average, standard deviation, minumum and maximum. Two common use cases which pose computational challenges for python: real time updating of trailing statistics as live data comes in, and posthoc computation of trailing statistics over a large data array. In the second case, for some statistics we can use convolution and related techniques for efficient computation, eg of the trailing 30 sample average

```
numpy.convolve(x, numpy.ones(30), mode=valid')[:len(x)]
```

but for other statistics like the trailing 30 day maximum at each point, efficient routines like convolution are of no help.

This exercise introduces pyrex to efficiently solve the problem of trailing statistics over arrays as well as for a live, incoming data stream. A pure C library, ringbuf, defines a circular C buffer and attached methods for efficiently computing trailing averages, and pyrex is used to provide a pythonic API on top of this extension code. The rigid segregation between the C library and the python wrappers insures that the C code can be used in other projects, be it a matlab (TM) extension or some other C library. The goal of the exercise is to compute the trailing statistics *mean*, *median*, *stddev*, *min* and *max* using three approaches:

- with brute force using numpy arrays, slices and methods
- with python bindings to the ringbuf code ringbuf.Ringbuf.

- using a `pyrex` extension to the `ringbuf.runstats` code

Bibliography

[1] *Python Success Stories: 8 True Tales of Flexibility, Speed, and Improved Productivity.* O'Reilly Associates, 2002.

[2] *Python Success Stories Volume II: 12 More True Tales.* O'Reilly Associates, 2005.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] K. Arnold, J. G., and D. Holmes. *Java(TM) Programming Language.* Addison-Wesley Professional, 2005.

[5] J. W. Backus et al. Algol 60 — revised report on the algorithmic language. *Communications of the ACM*, 6(1):1–17, January 1963.

[6] C. H. Barker and W. P. Healy. Statistical analysis of oil spill response options: A NOAA-U.S. Navy joint project. In *Proceedings of the International Oil Spill Conference*, pages 883–890, 2001.

[7] P. Barrett, J.D. Hunter, and P. Greenfield. Matplotlib - A portable Python plotting package. In *Astronomical Data Analysis Software & Systems XIV.*, 2004.

[8] D. Beasley. *Python Essential Reference.* New Riders Publishing, 2nd edition, 2001.

[9] David Beazley. SWIG and automated C/C++ scripting extensions. *Dr. Dobb's Journal of Software Tools*, 23(2):30, 32, 34–36, 100, February 1998.

[10] Thomas J. Bergin, Richard G. Gibson, and Richard G. Gibson. *History of Programming Languages.* Addison-Wesley Professional, 1996.

[11] J.B. Buckheit and D.L. Donoho. *Wavelets and Statistics*, chapter WaveLab and Reproducible Research. Springer-Verlag, 1995.

[12] A. Butterfield, V. Vedagiri, E. Lang, C. Lawrence, M. J. Wakefield, A. Isaev, and G. A. Huttley. Pyevolve: A toolkit for statistical modelling of molecular evolution. *BMC Bioinformatics*, 5(1):1, 2004.

[13] P. F. Dubois, K. Hinsen, and J. Hugunin. Numerical Python. *Computers in Physics*, 10(3):262–267, May/June 1996.

[14] Paul F. Dubois and T.-Y. Yang. Scientific programming: Extending Python. *Computers in Physics*, 10(4):359–??, ???? 1996.

[15] Jr. Drake G. Van Rossum, F. L., editor. *An Introduction to Python.* Network Theory Ltd., 2003.

[16] A. Goldberg and D. Robson. *Smalltalk 80 : The Language.* Addison-Wesley Professional, 1989.

[17] Duane C. Hanselman and Bruce L. Littlefield. *Mastering MATLAB 7.* Prentice Hall, 2004.

[18] Jim Hugunin. Python and java - The best of both worlds. In *In Proceedings of the 6th International Python Conference*, pages 11–20, 1997.

[19] JD Hunter, J Reimer, DM Hanan, KE Hecox, and VL Towle. Locating chronically implanted subdural electrodes using 3-D rendering. *Clinical Neurophysiology*, 2005.

[20] Gavin A Huttley. Modeling the impact of DNA methylation on the evolution of BRCA1 in mammals. *Mol Biol Evol*, 21(9):1760–8, 2004.

[21] I. K. Kominis, T. W. Kornack, J.C.. Allred, and M. V. Romalis. A subfemtotesla multichannel atomic magnetometer. *Nature*, 422(6932):596–599, April 2003.

[22] T. W. Kornack and M. V. Romalis. Dynamics of two overlapping spin ensembles interacting by spin exchange. *Phys Rev Lett*, 89(25):253002, December 2002.

[23] A. Martelli. *Python in a Nutshell.* O'Reilly, 1st edition, 2003.

[24] A. Martelli and D. Ascher. *Python Cookbook.* O'Reilly, 1st edition, 2004.

[25] Peter Naur al. Revised report on the algorithmic language ALGOL 60. 6(1):1–17, January 1963.

[26] NOAA. *TAP II 1.2 User Manual.* National Oceanic and Atmospheric Administration, Hazardous Material Response Division, Seattle, WA, 2000.

[27] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 1998.

[28] H. Parker-Hall and C. H. Barker. Do trajectories belong in area plans? a new approach in california using the trajectory analysis planner (TAP II). In *Proceedings of the International Oil Spill Conference*, pages 685–691, 2001.

[29] Fernando Pérez. IPython – an enhanced interactive Python shell, 2001. `http://ipython.scipy.org`.

[30] M. Pilgrim. *Dive into Python.* Apress, 1st edition, 2004.

[31] S. M. Ransom, J. W. T. Hessels, I. H. Stairs, P. C. C. Freire, F. Camilo, V. M. Kaspi, and D. L. Kaplan. Twenty-One Millisecond Pulsars in Terzan 5 Using the Green Bank Telescope. *Science*, 307:892–896, February 2005.

[32] S. M. Ransom, V. M. Kaspi, R. Ramachandran, P. Demorest, D. C. Backer, E. D. Pfahl, F. D. Ghigo, and D. L. Kaplan. Green Bank Telescope Measurement of the Systemic Velocity of the Double Pulsar Binary J0737-3039 and Implications for Its Formation. *Astrophysical Journal*, 609:L71–L74, July 2004.

[33] S. Rosen. *Programming Systems and Languages*, chapter Programming Systems and Languages–A Historical Survey. McGraw-Hill, New York, 1967.

[34] G. Van Rossum. *Python Library Reference*. To Excel Inc, 2001.

[35] M.F. Sanner. A component-based software environment for visualizing large macromolecular assemblies. *Structure*, 13:447–462, 2005.

[36] M.F. Sanner. *Encyclopedia of Genomics, Proteomics and Bioinformatics*, chapter Using the Python Programming Language for Bioinformatics. John Wiley & Sons, Ltd, 2005.

[37] W. Schoeder, K. Martin, and B Lorense. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Kitware, Inc., 3rd edition, 2002.

[38] G. Stein. Python at Google. In *Pycon2005*, 2005.

[39] M Strous. Python - executable pseudocode. *PC Update*, 2001.

[40] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 3rd edition edition, 2000.

[41] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Professional, 1994.