02-Matplotlib_Numpy

Unknown Author

March 10, 2014

Part I

Matplotlib - 2D plotting in Python

This notebook contains extensive material from J.R. Johansson's IPython notebook lectures available at the references below:

J.R. Johansson (robert@riken.jp) http://dml.riken.jp/~rob/

The latest version of this IPython notebook lecture is available at `http://github.com/jrjohansson/scientific-python-lectures`.

The other notebooks in this lecture series are indexed at `http://jrjohansson.github.com`.

0.1 Introduction

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library include:

- Easy to get started
- Support for $\LaTeX$ formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

One of the of the key features of matplotlib that I would like to emphasize, and that I think makes matplotlib highly suitable for generating figures for scientific publications is that all aspects of the figure can be controlled *programmatically*. This is important for reproducibility and convenient when one needs to regenerate the figure with updated data or change its appearance.

More information at the Matplotlib web page: http://matplotlib.org/To get started using Matplotlib in a Python program, either include the symbols from the `pylab` module (the easy way):

```
In [2]: from pylab import *
```

or import the `matplotlib.pyplot` module under the name `plt` (the tidy way). In this case we will also find it useful to import the array module `numpy`.

In [3]:
```
import matplotlib.pyplot as plt
import numpy as np
```

0.2 Matplotlib's MATLAB-like interface

The easiest way to get started with plotting using matplotlib is often to use the MATLAB-like API provided by matplotlib.

It is designed to be compatible with MATLAB's plotting functions, so it is easy to get started with if you are familiar with MATLAB.

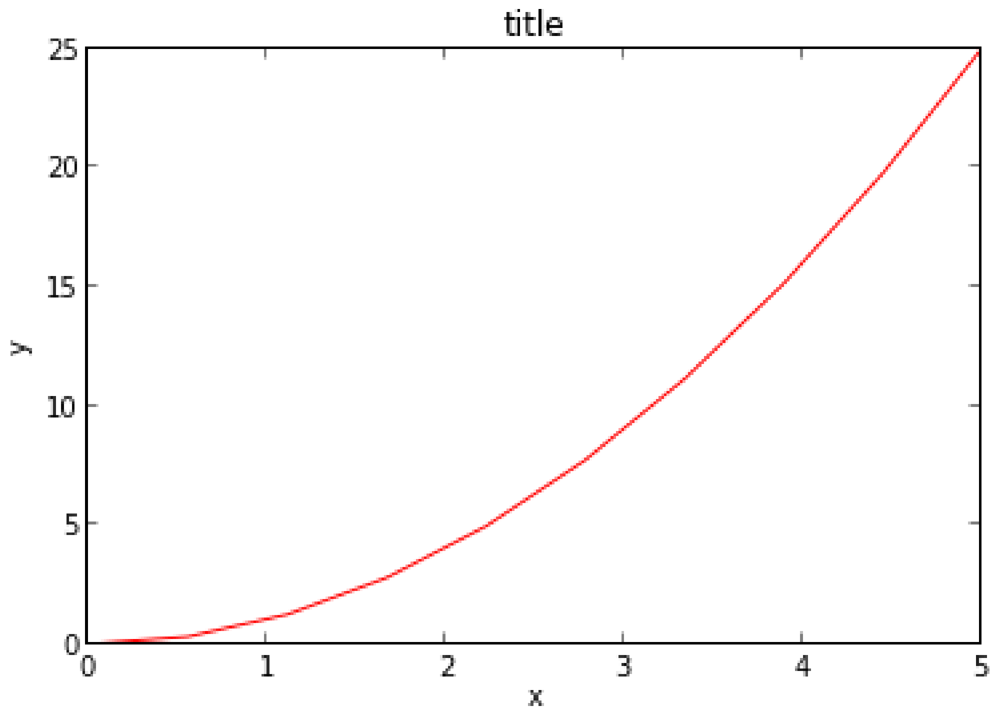
To use this API from matplotlib, we need to include the symbols in the `pylab` module:

In [4]:
```
from pylab import *
```

A simple figure with MATLAB-like plotting API:

In [5]:
```
x = linspace(0, 5, 10)
y = x ** 2
```

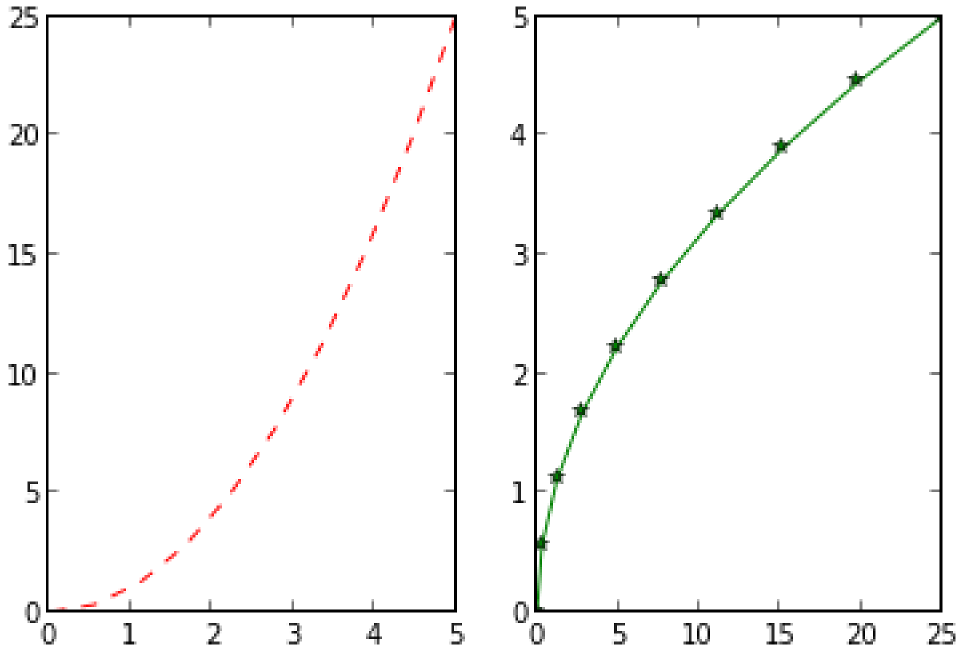
In [6]:
```
figure()
plot(x, y, 'r')
xlabel('x')
ylabel('y')
title('title')
show()
```



Most of the plotting related functions in MATLAB are covered by the `pylab` module. For example, subplot and color/symbol selection:

In [7]:
```
subplot(1,2,1)
plot(x, y, 'r--')
subplot(1,2,2)
plot(y, x, 'g*-')
```

Out [7]:
```
[<matplotlib.lines.Line2D at 0x413f410>]
```



The good thing about the pylab MATLAB-style API is that it is easy to get started with if you are familiar with MATLAB, and it has a minumum of coding overhead for simple plots.

However, I'd encourrage not using the MATLAB compatible API for anything but the simplest figures.

Instead, I recommend learning and using matplotlib's object-oriented plotting API. It is remarkably powerful. For advanced figures with subplots, insets and other components it is very nice to work with.

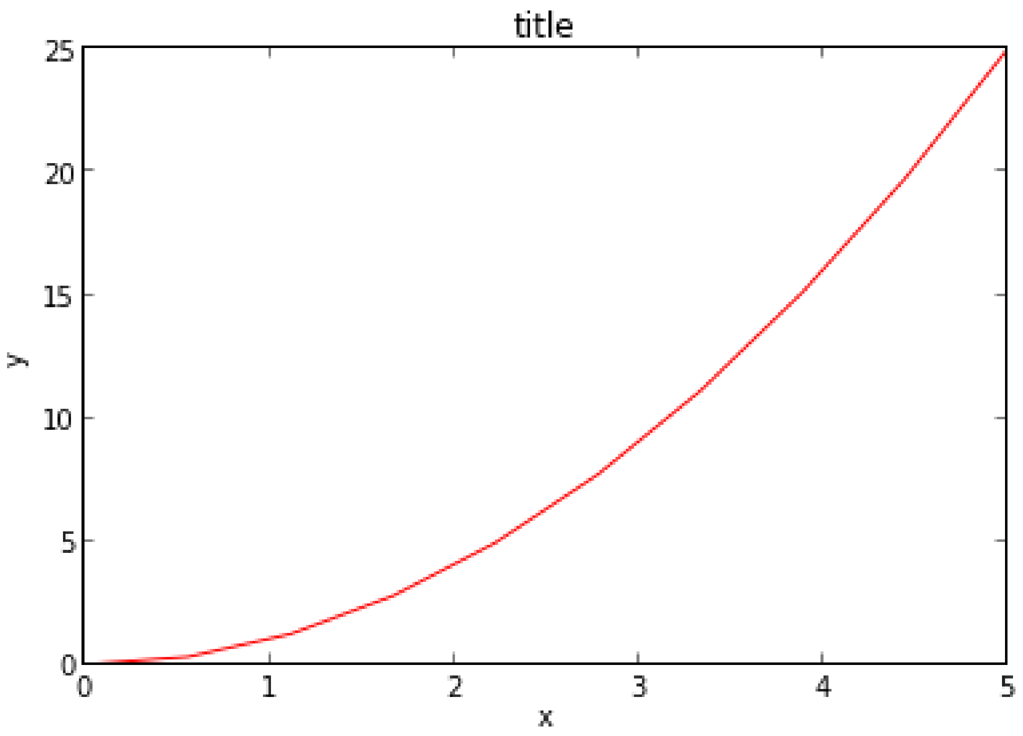
0.3 Exercise 1 : Starting MatPlotlib

0.4 The Matplotlib object-orientated interface

The main idea with object-oriented programming is to have objects that one can apply functions and actions on, and no object or program states should be global (such as the MATLAB-like API). The real advantage of this approach becomes apparent when more than one figure is created, or when a figure contains more than one subplot.

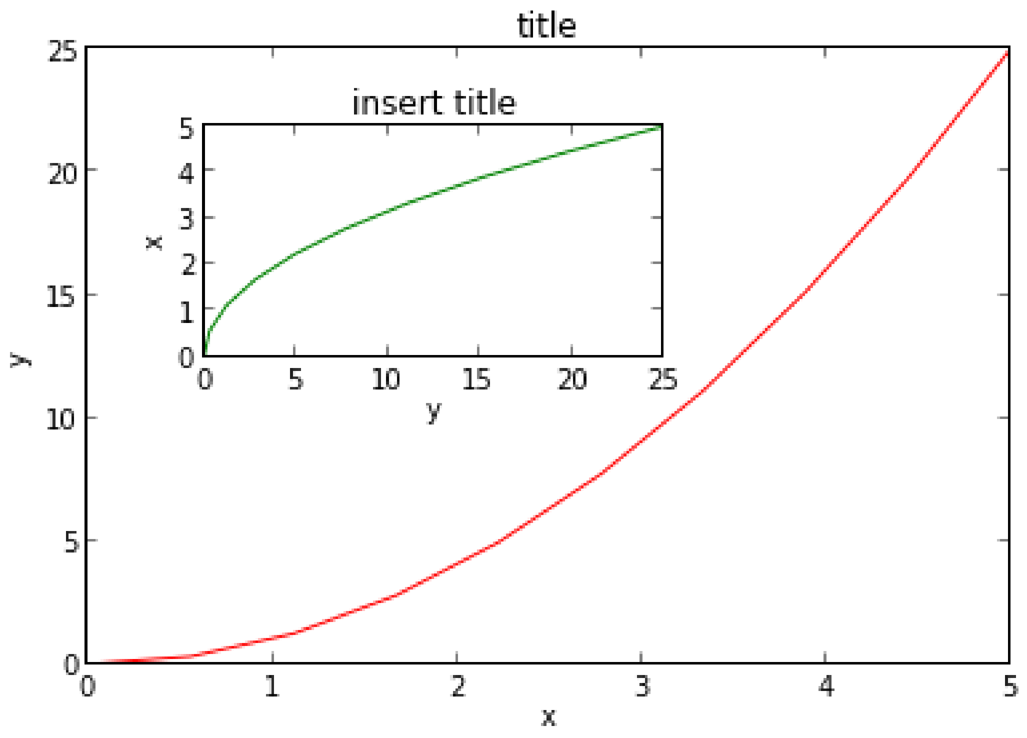
To use the object-oriented API we start out very much like in the previous example, but instead of creating a new global figure instance we store a reference to the newly created figure instance in the `fig` variable, and from it we create a new axis instance `axes` using the `add_axes` method in the `Figure` class instance `fig`:

In [8]:
```
fig = plt.figure()

axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```

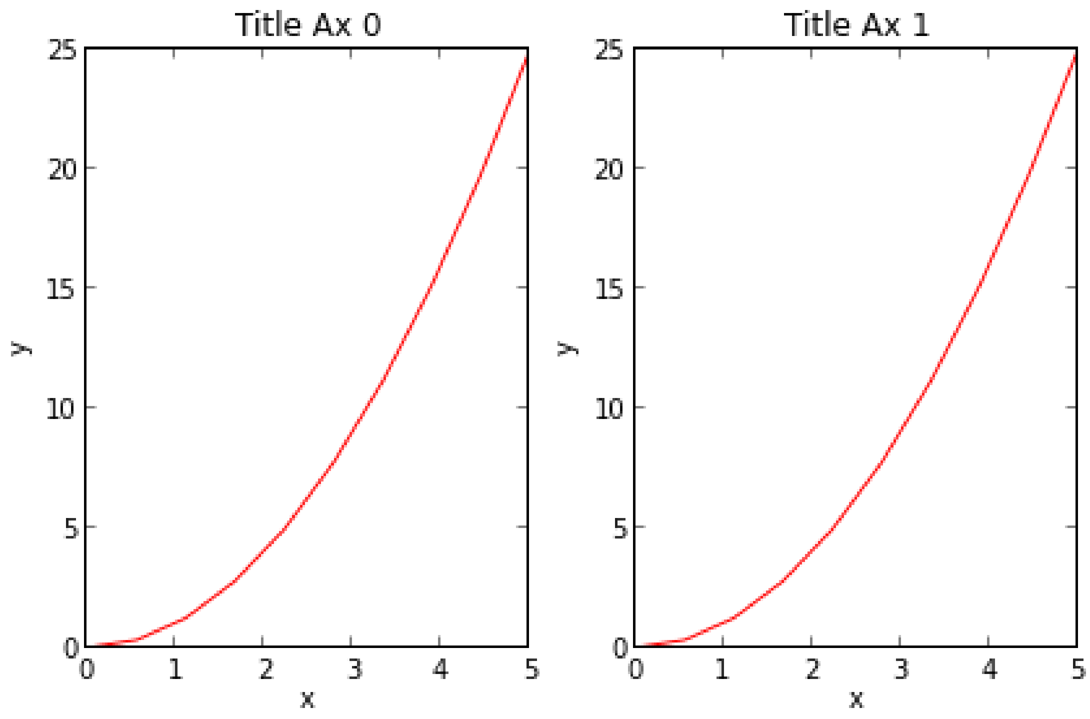


Although a little bit more code is involved, the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

In [9]:
```
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# main figure
axes1.plot(x, y, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('title')

# insert
axes2.plot(y, x, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
axes2.set_title('insert title');
```



```
In [109]: fig, axes = plt.subplots(nrows=1, ncols=2)

          for i, ax in enumerate(axes):
              ax.plot(x, y, 'r')
              ax.set_xlabel('x')
              ax.set_ylabel('y')
              ax.set_title('Title Ax %s' % i)

          # The 'fig.tight_layout' method automatically adjusts the
          # positions of the axes so that there is no overlapping content
          fig.tight_layout()
```



1 Saving figures

To save a figure to a file we can use the `savefig` method in the `Figure` class:

```
In [110]: fig.savefig("filename.png")
```

Here we can also optionally specify the DPI and chose between different output formats. Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, and PDF. For scientific papers, use PDF whenever possible. (LaTeX documents compiled with `pdflatex` can include PDFs using the `includegraphics` command).

```
In [111]: fig.savefig("filename.png", dpi=200)
```

```
In [112]: fig.savefig("filename.svg")
```

1.1 Exercise 2 : Plot layouts and saving

2 2. An introduction to `numpy`

This material is based on a the SciTools Courses notebook available at https://github.com/SciTools/courses

While the Python language is an excellent tool for general-purpose programming, with a highly readable syntax, rich and powerful data types (strings, lists, sets, dictionaries, arbitrary length integers, etc) and a very comprehensive standard library, it was not designed specifically for mathematical and scientific computing. Neither the language nor its standard library have facilities for the efficient representation of multidimensional datasets, tools for linear algebra and general matrix manipulations (essential building blocks of virtually all technical computing), nor any data visualisation facilities.

In particular, Python lists are very flexible containers that can be nested arbitrarily deep and which can hold any Python object in them, but they are poorly suited to represent common mathematical constructs like vectors and matrices. It is for this reason that numpy exists. Typically numpy is imported as np:

```
In [11]: import numpy as np
```

Numpy, at its core, provides a powerful array object. Let's start by exploring how the numpy array differs from Python lists. We start by creating a simple list and an array with the identical contents:

```
In [12]: lst = [10, 20, 30, 40]
         arr = np.array([10, 20, 30, 40])
         print lst
         print arr
```

```
[10, 20, 30, 40]
[10 20 30 40]
```

```
In [14]: print lst[0], arr[0]
```

```
10 10
```

```
In [13]: print lst[2:4], arr[2:4]
```

```
[30, 40] [30 40]
```

Data types

Numpy comes with a most of the common data types (and some uncommon ones too).

The most used (and portable) dtypes are:

- bool
- uint8
- int (machine dependent)
- int8
- int32
- int64
- float (machine dependent)
- float32
- float64

Full details can be found at http://docs.scipy.org/doc/numpy/user/basics.types.html.

```
In [15]: # The information about the type of an array is contained in its dtype at
         arr.dtype
```

```
Out [15]: dtype('int64')
```

Once an array has been created, its dtype is fixed (in this case to an 8 byte/64 bit signed integer) and it can only store elements of the same type. For this example where the dtype is integer, if we try storing a floating point number in the array it will be automatically converted into an integer:

```
In [16]: arr[-1] = 1.234
         arr
```

```
Out [16]: array([10, 20, 30,  1])
```

```
In [17]: arr.shape
```

```
Out [17]: (4,)
```

Creating Arrays

Above we created an array from an existing list; now let us now see other ways in which we can create arrays, which we'll illustrate next. A common need is to have an array initialized with a constant value, and very often this value is 0 or 1 (suitable as starting value for additive and multiplicative loops respectively); `zeros` creates arrays of all zeros, with any desired dtype:

```
In [120]: print '5 zeros:', np.zeros(5, dtype=np.int)
```

```
5 zeros: [0 0 0 0 0]
```

and similarly for `ones`:

```
In [121]: print '5 ones:', np.ones(5, dtype=np.int)
```

```
5 ones: [1 1 1 1 1]
```

```
In [122]: a = np.empty(4, dtype=np.float)
          a.fill(5.5)
          print a

          # Alternatives such as
          print np.ones(4) * 5.5
          print np.zeros(4) + 5.5
```

```
[ 5.5  5.5  5.5  5.5]
[ 5.5  5.5  5.5  5.5]
[ 5.5  5.5  5.5  5.5]
```

Filling arrays with sequences

Numpy also offers the `arange` function, which works like the builtin `range` but returns an array instead of a list:

```
In [123]: np.arange(10, dtype=np.float64)
```

```
Out [123]:array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

```
In [124]: np.arange(5, 7, 0.1)
```

```
Out [124]:array([ 5. ,  5.1,  5.2,  5.3,  5.4,  5.5,  5.6,  5.7,  5.8,  5.9,  6.
,
         6.1,  6.2,  6.3,  6.4,  6.5,  6.6,  6.7,  6.8,  6.9])
```

The `linspace` and `logspace` functions to create linearly and logarithmically-spaced grids respectively, with a fixed number of points and including both ends of the specified interval:

```
In [125]: print "A linear grid between 0 and 1:"
          print np.linspace(0, 1, 5)
```

```
A linear grid between 0 and 1:
[ 0.    0.25  0.5   0.75  1.  ]
```

2.1 Arrays with more than one dimension

Up until now all our examples have used one-dimensional arrays. But Numpy can create arrays of arbitrary dimensions, and all the methods illustrated in the previous section work with more than one dimension. For example, a list of lists can be used to initialize a two dimensional array:

```
In [18]: lst2 = [[1, 2, 3], [4, 5, 6]]
         arr2 = np.array([[1, 2, 3], [4, 5, 6]])
         print arr2
         print arr2.shape
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

With two-dimensional arrays we start seeing the power of numpy: while a nested list can be indexed using repeatedly the `[ ]` operator, multidimensional arrays support a much more natural indexing syntax with a single `[ ]` and a set of indices separated by commas:

```
In [127]: print lst2[0][1]
          print arr2[0, 1]
```

```
2
2
```

The array creation functions listed previously can be used with more than one dimension, for example:

```
In [128]: np.zeros((2, 3))
```

```
Out [128]:array([[ 0.,  0.,  0.],
                 [ 0.,  0.,  0.]])
```

Slices

With multidimensional arrays, you can also use slices, and you can mix and match slices and single indices in the different dimensions:

```
In [129]: arr = np.arange(8).reshape(2, 4)
          print arr

          print 'Second element from dimension 0, last 2 elements from dimension one
          print arr[1, 2:]

          print 'All elements bar the last from dimension 0, third element from dime
          print arr[:-1, 2]

          print 'Second element from dimension 0 (maintaining its dimension), all el
          print arr[1:2, :]
```

```
[[0 1 2 3]
 [4 5 6 7]]
Second element from dimension 0, last 2 elements from dimension one:
[6 7]
All elements bar the last from dimension 0, third element from
dimension one: [2]
Second element from dimension 0 (maintaining its dimension), all
elements from dimension one: [[4 5 6 7]]
```

If you only provide one index, then the slice will be expanded to ":" for all of the remaining dimensions (aka Ellipsis):

```
In [130]: print 'First row:  ', arr[0], 'is equivalent to', arr[0, :]
          print 'Second row: ', arr[1], 'is equivalent to', arr[1, :]
```

```
First row:   [0 1 2 3] is equivalent to [0 1 2 3]
Second row:  [4 5 6 7] is equivalent to [4 5 6 7]
```

2.2 Generating 2D coordinate arrays

A common task is to generate a pair of arrays which represent the coordinates of our data. When the orthogonal 1d coordinate arrays already exist, numpy's meshgrid function is very useful:

```
In [131]: x_g = np.linspace(0, 9, 3)
          y_g = np.linspace(-8, 4, 3)
          x2d, y2d = np.meshgrid(x_g, y_g)
          print x2d
          print y2d
```

```
[[ 0.   4.5  9. ]
 [ 0.   4.5  9. ]
 [ 0.   4.5  9. ]]
[[-8. -8. -8.]
 [-2. -2. -2.]
 [ 4.  4.  4.]]
```

2.3 Operating with arrays

Arrays support all regular arithmetic operators, and the numpy library also contains a complete collection of basic mathematical functions that operate on arrays. It is important to remember that in general, all operations with arrays are applied *element-wise*, i.e., are applied to all the elements of the array at the same time. Consider for example:

```
In [19]: arr1 = np.arange(4)
         arr2 = np.arange(10, 14)
         print arr1, '+', arr2, '=', arr1 + arr2
```

```
[0 1 2 3] + [10 11 12 13] = [10 12 14 16]
```

Importantly, even the multiplication operator is by default applied element-wise, it is *not* the matrix multiplication from linear algebra:

```
In [20]: print arr1, '*', arr2, '=', arr1 * arr2
```

```
[0 1 2 3] * [10 11 12 13] = [ 0 11 24 39]
```

We may also multiply an array by a scalar:

```
In [21]: 1.5 * arr1
```

```
Out [21]: array([ 0. ,  1.5,  3. ,  4.5])
```

This is an example of **broadcasting**.

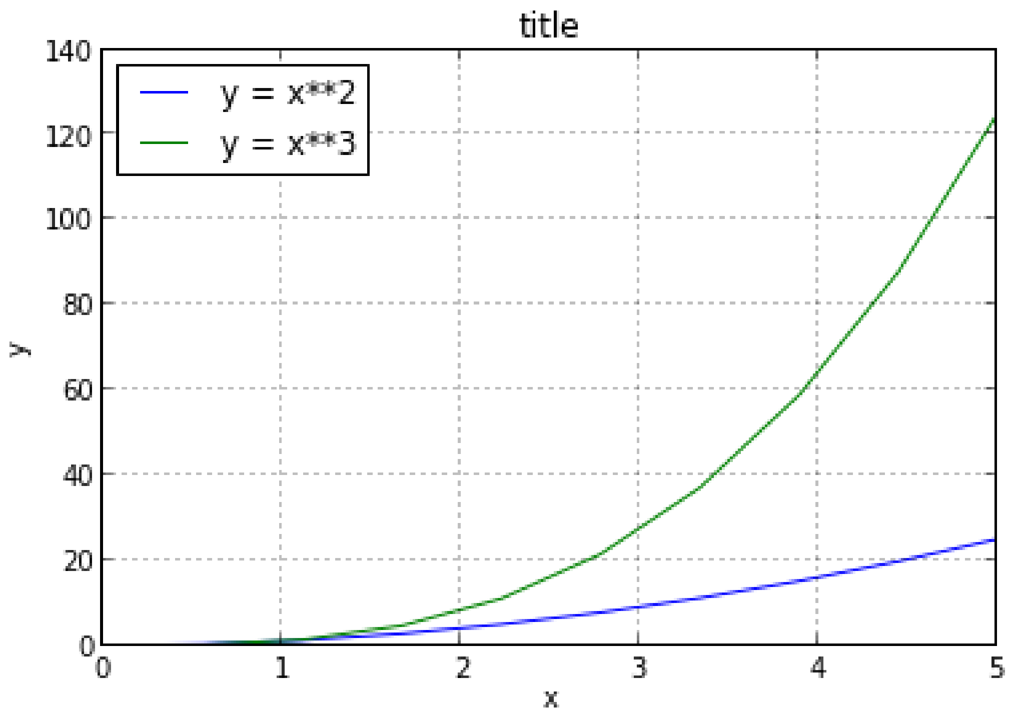
2.4 Exercise 3 : Numpy arrays

3 3. Next steps with Matplotlib

4 Legends, labels and titles

```
In [135]: fig, ax = plt.subplots()

          ax.plot(x, x**2, label="y = x**2")
          ax.plot(x, x**3, label="y = x**3")
          ax.set_xlabel('x')
          ax.set_ylabel('y')
          ax.set_title('title')
          ax.grid(True)
          ax.legend(loc=2); # upper left corner
```

```
Out [135]:<matplotlib.legend.Legend at 0x2a8ec10>
```



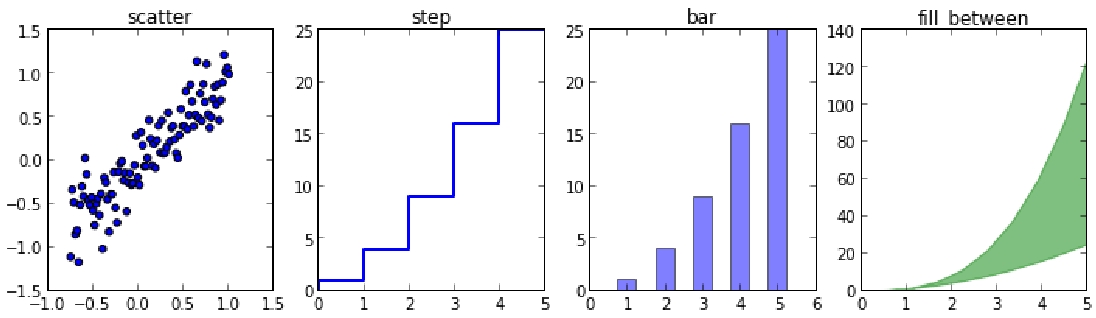
5 Other 2D plot styles

In addition to the regular `plot` method, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of available plot types: http://matplotlib.org/gallery.html. Some of the more useful ones are show below:

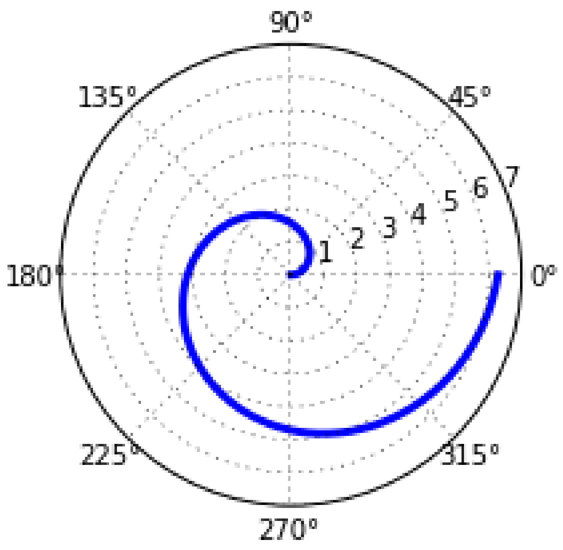
In [23]:
```
n = array([0,1,2,3,4,5])
xx = np.linspace(-0.75, 1., 100)
```

In [24]:
```
fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(xx, xx + 0.25*randn(len(xx)))
axes[0].set_title("scatter")

axes[1].step(n, n**2, lw=2)
axes[1].set_title("step")

axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
axes[2].set_title("bar")

axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
axes[3].set_title("fill_between")
```

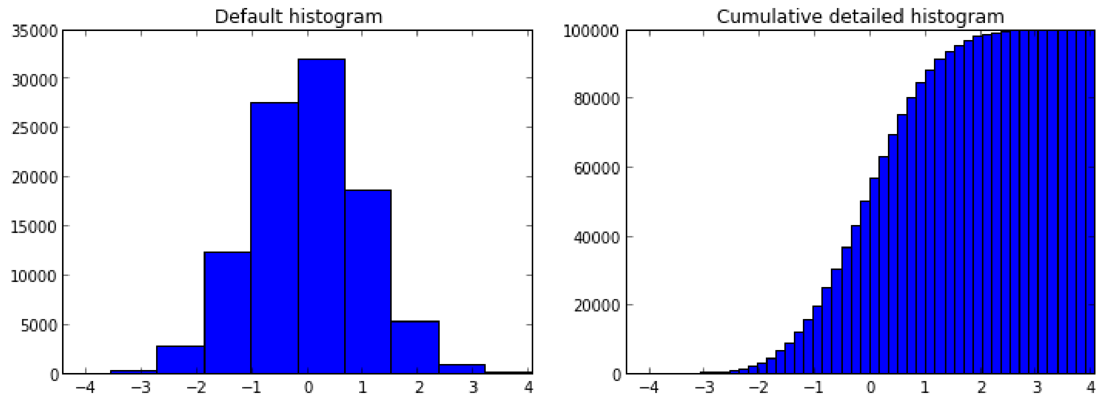
```
Out [24]: <matplotlib.text.Text at 0x4bf8690>
```



```
In [139]: # polar plot using add_axes and polar projection
          fig = plt.figure()
          ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
          t = linspace(0, 2 * pi, 100)
          ax.plot(t, t, color='blue', lw=3);
```



```
In [140]: # A histogram
          n = np.random.randn(100000)
          fig, axes = plt.subplots(1, 2, figsize=(12,4))

          axes[0].hist(n)
          axes[0].set_title("Default histogram")
          axes[0].set_xlim((min(n), max(n)))

          axes[1].hist(n, cumulative=True, bins=50)
          axes[1].set_title("Cumulative detailed histogram")
          axes[1].set_xlim((min(n), max(n)));
```



6 Colormap and contour figures

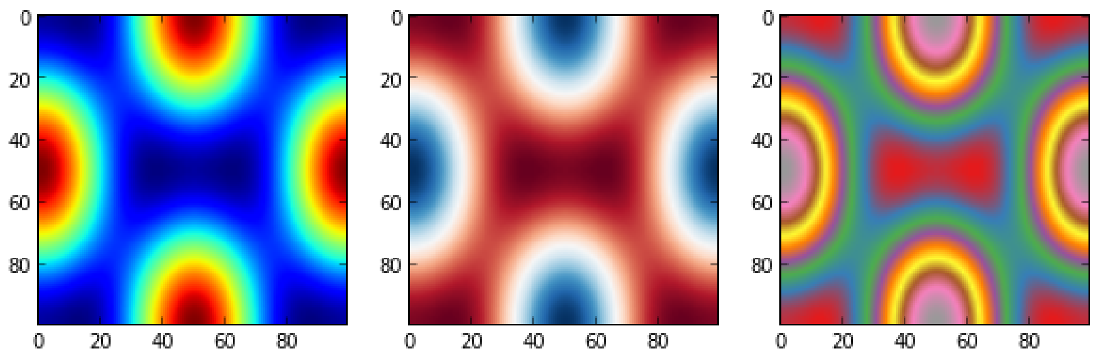
Colormaps and contour figures are useful for plotting functions of two variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps. It is relatively straightforward to define custom colormaps. For a list of pre-defined colormaps, see: http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

In [141]:
```
alpha = 0.7
phi_ext = 2 * pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * cos(phi_p)*cos(phi_m) - alpha * cos(phi_ext -
```

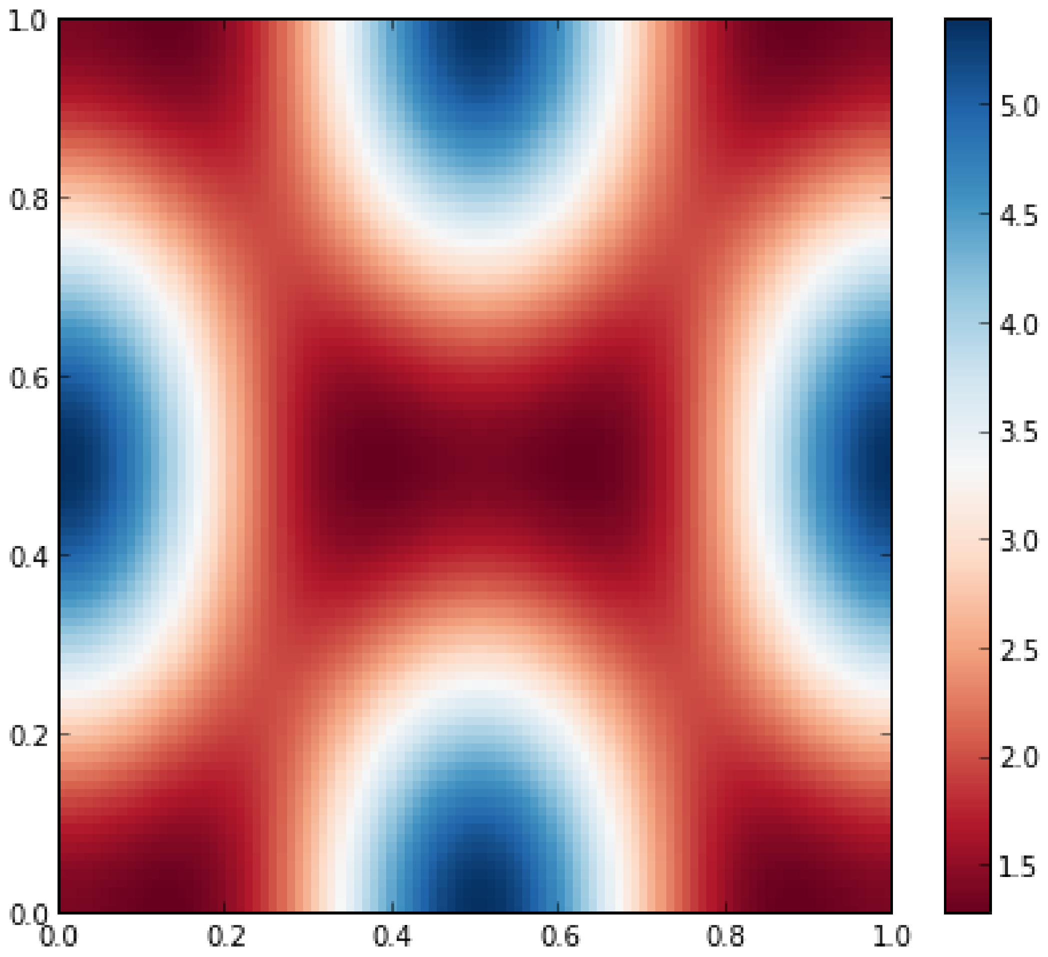
In [142]:
```
phi_m = linspace(0, 2*pi, 100)
phi_p = linspace(0, 2*pi, 100)
X,Y = meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

In [143]:
```
fig, axes = plt.subplots(1, 3, figsize=(10, 4))

axes[0].imshow(Z)
axes[1].imshow(Z, cmap=cm.RdBu)
axes[2].imshow(Z, cmap=cm.Set1)
```

Out [143]:
```
<matplotlib.image.AxesImage at 0x6553490>
```

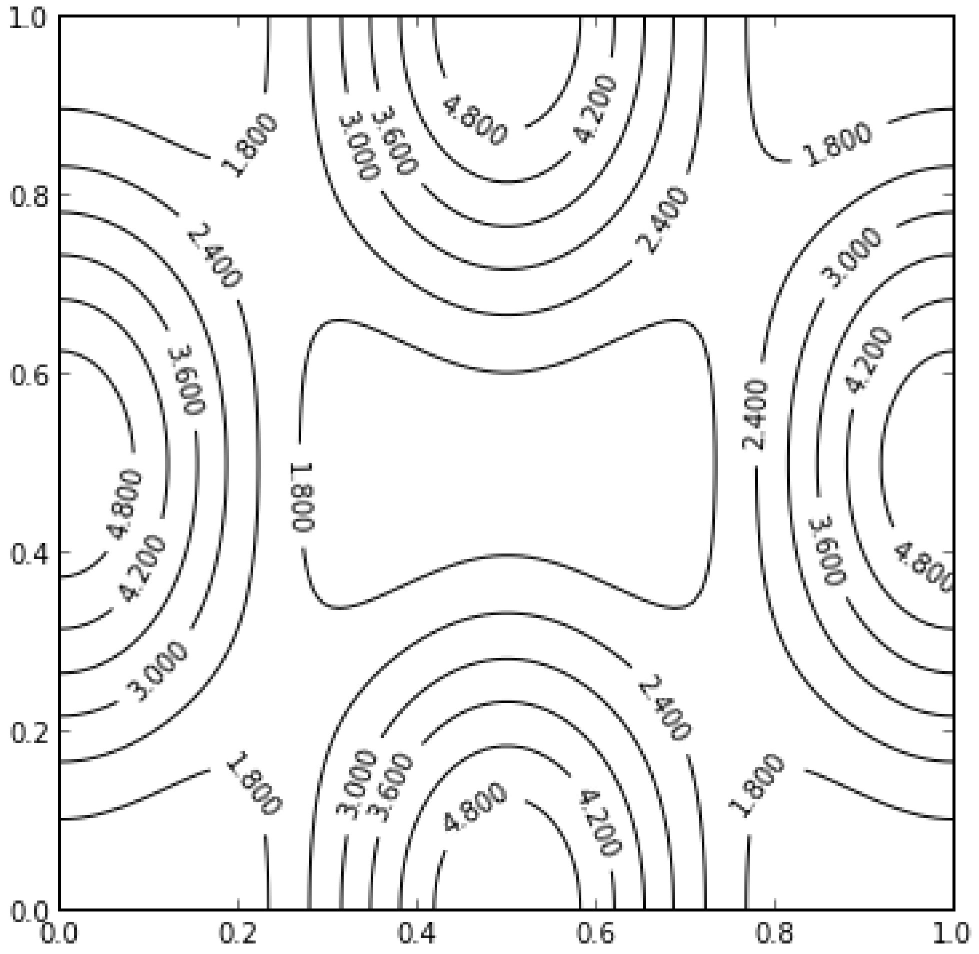


In [144]:
```
fig, ax = plt.subplots(figsize=(7,6))

p = ax.pcolor(X/(2*pi), Y/(2*pi), Z, cmap=cm.RdBu, vmin=abs(Z).min(), vmax
cb = fig.colorbar(p, ax=ax)
```

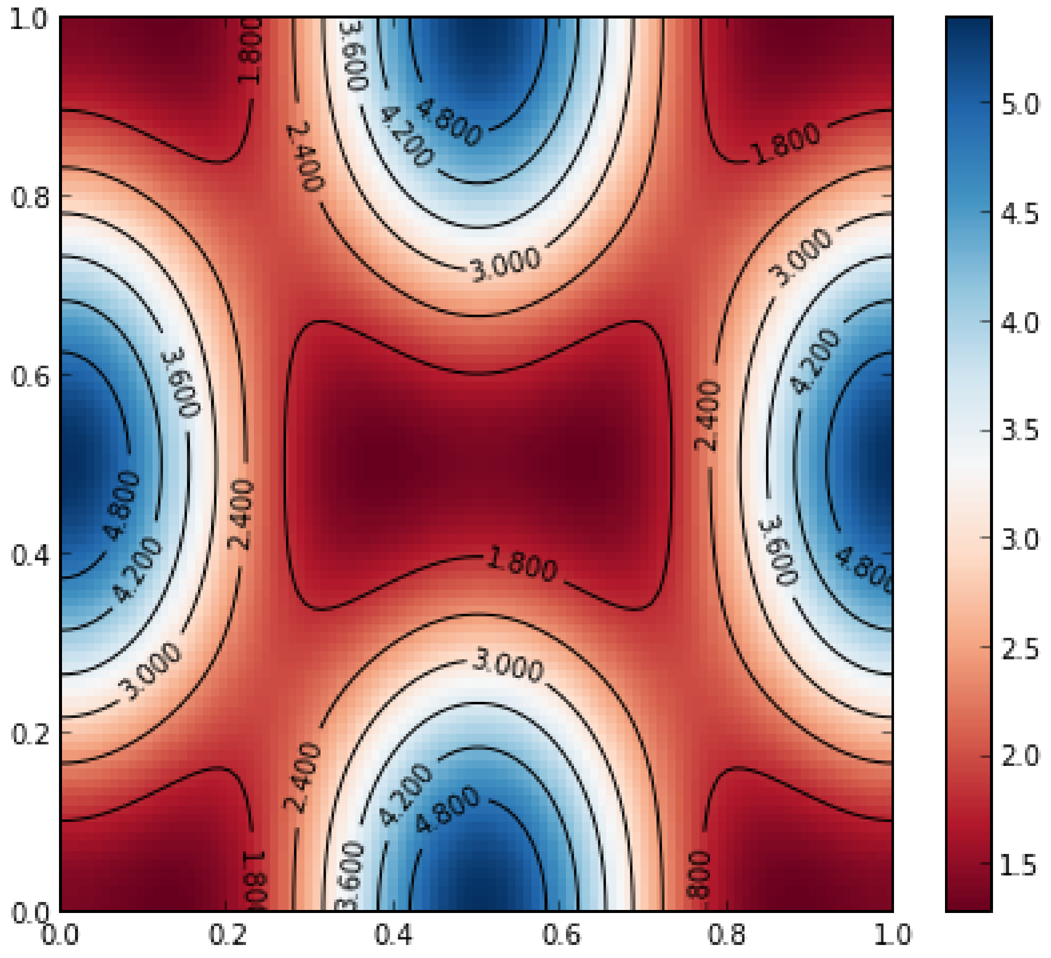


6.1 contour

```
In [145]: fig, ax = plt.subplots(figsize=(6,6))

          p1 = ax.contour(X/(2*pi), Y/(2*pi), Z, colors='k', vmin=abs(Z).min(), vmax
          ax.clabel(p1)

Out [145]:<a list of 25 text.Text objects>
```



```
In [146]: fig, ax = plt.subplots(figsize=(7,6))

          p = ax.pcolor(X/(2*pi), Y/(2*pi), Z, cmap=cm.RdBu, vmin=abs(Z).min(), vmax
          p1 = ax.contour(X/(2*pi), Y/(2*pi), Z, colors='k', vmin=abs(Z).min(), vmax
          ax.clabel(p1)
          cb = fig.colorbar(p, ax=ax)
```



6.2 Further reading

- http://www.matplotlib.org - The project web page for matplotlib.
- https://github.com/matplotlib/matplotlib - The source code for matplotlib.
- http://matplotlib.org/gallery.html - A large gallery showcaseing various types of plots matplotlib can create. Highly recommended!
- http://www.loria.fr/~rougier/teaching/matplotlib - A good matplotlib tutorial.
- http://scipy-lectures.github.io/matplotlib/matplotlib.html - Another good matplotlib reference.