01-Matplotlib

Stephen Pascoe

March 16, 2014

1 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`.

1.1 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 [12]:
```
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 [13]:
```
import matplotlib.pyplot as plt
import numpy as np
```

1.2 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.

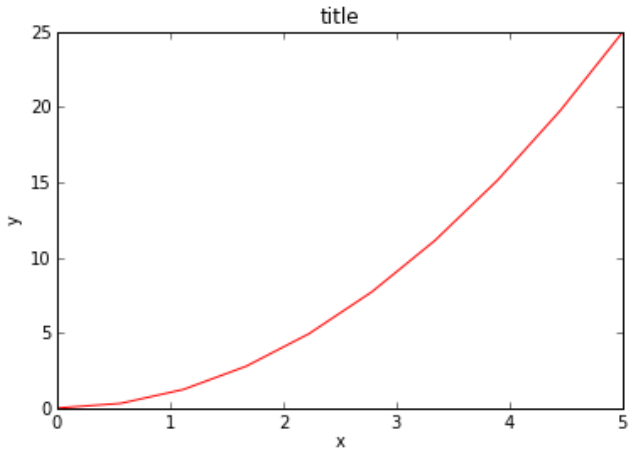
To use this API from matplotlib, we need to include the symbols in the `pylab` module:

In [14]:
```
from pylab import *
```

A simple figure with MATLAB-like plotting API:

In [15]:
```
x = linspace(0, 5, 10)
y = x ** 2
```

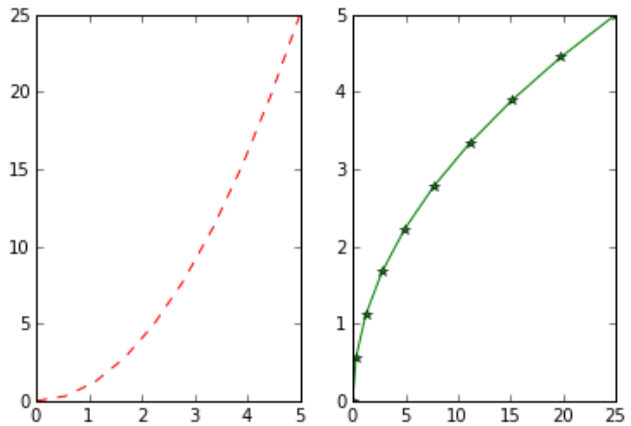
In [16]:
```
figure()
plot(x, y, 'r')
xlabel('x')
ylabel('y')
title('title')
show()
```



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

In [17]:
```
subplot(1,2,1)
plot(x, y, 'r--')
subplot(1,2,2)
plot(y, x, 'g*-')
```

Out [17]:
```
[<matplotlib.lines.Line2D at 0x10338f190>]
```



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 encourage 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.

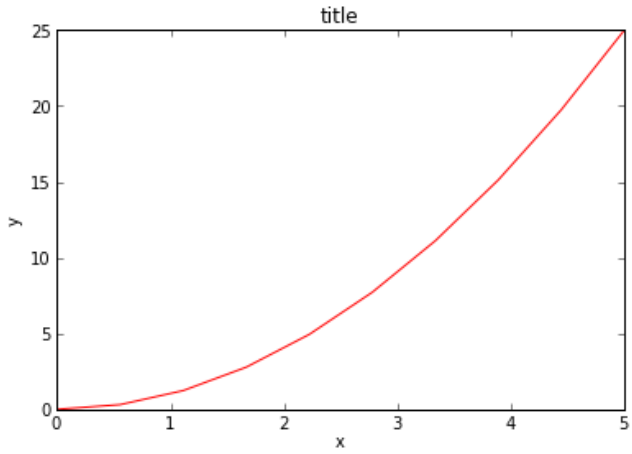
Exercise 1.1 : Starting MatPlotlib

1.3 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`:

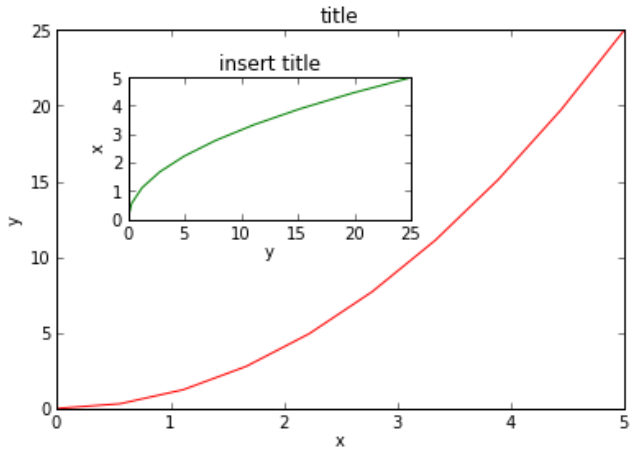
In [18]:
```
fig = plt.figure()

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');
```



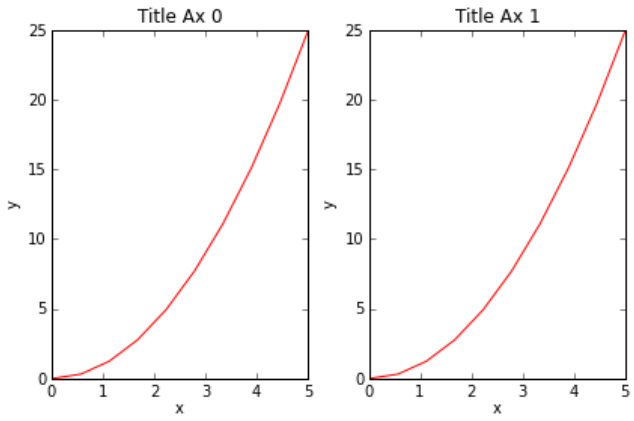
Although a little bit more code is involved, the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

In [19]:
```
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 [20]:
```
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()
```



2 Saving figures

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

In [21]:
```
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).

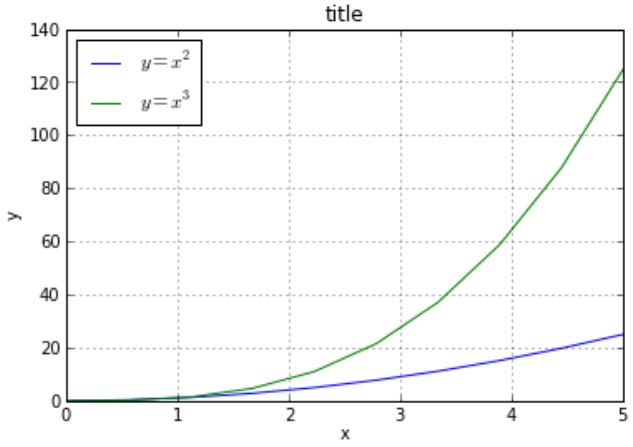
In [22]:
```
fig.savefig("filename.png", dpi=200)
```

In [23]:
```
fig.savefig("filename.svg")
```

3 Legends, labels and titles

In [24]:
```
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 [24]: <matplotlib.legend.Legend at 0x1031d35d0>



Exercise 1.2 : Plot layouts and saving

4 Other 2D plot styles

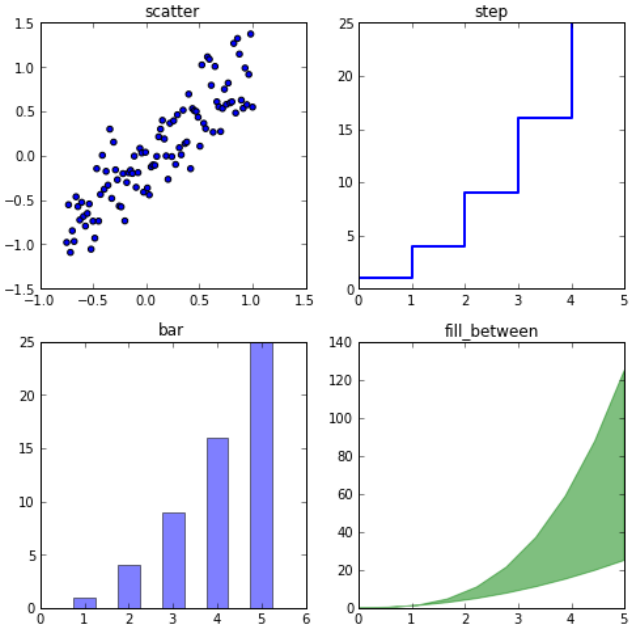
In addition to the regular `plot` method, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of available plot types: http://matplotlib.org/gallery.html. Some of the more useful ones are show below:

In [25]:
```
n = array([0,1,2,3,4,5])
xx = np.linspace(-0.75, 1., 100)
```

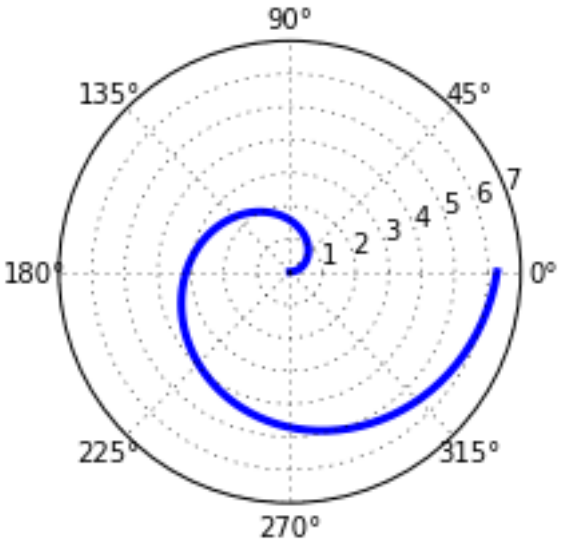
In [26]:
```
fig, axes = plt.subplots(2, 2, figsize=(8, 8))

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

axes[0,1].step(n, n**2, lw=2)
axes[0,1].set_title("step")

axes[1,0].bar(n, n**2, align="center", width=0.5, alpha=0.5)
axes[1,0].set_title("bar")
```

```
axes[1,1].fill_between(x, x**2, x**3, color="green", alpha=0.5);
axes[1,1].set_title("fill_between")
```

Out [26]: <matplotlib.text.Text at 0x103ad8650>

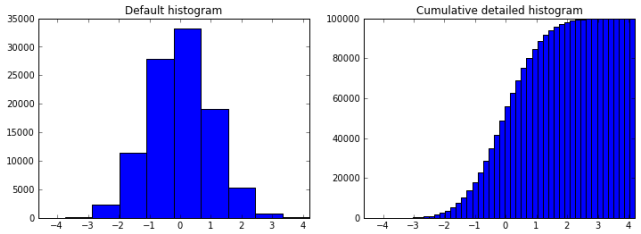


In [27]:
```
# polar plot using add_axes and polar projection
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
t = linspace(0, 2 * pi, 100)
ax.plot(t, t, color='blue', lw=3);
```



In [28]:
```
# 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)));
```



5 Plotting 2D arrays

Matplotlib provides several methods of plotting functions of 2 variables as a 2D field using colour or contour lines to represent the function value. We will look at 3 of these methods.

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 [29]:
```
# Import library of Colour maps
from matplotlib import cm
```

In [30]:
```
x = linspace(0, 2*pi, 50)
y = linspace(0, 2*pi, 50)
X,Y = meshgrid(x, y)
Z = np.cos(X)*np.cos(Y)
```

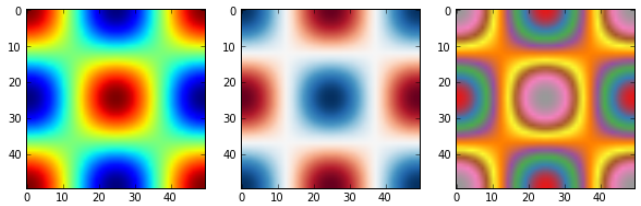
Imshow

`Axes.imshow()` interprets a 2D array as an image by mapping function values to colours within a colour map. This method is the easiest way of visualising a 2D array provided each point represents the same sized region in cartesian space.

By default `imshow()` uses the array-index values as x and y co-ordinates.

In [31]:
```
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)
```

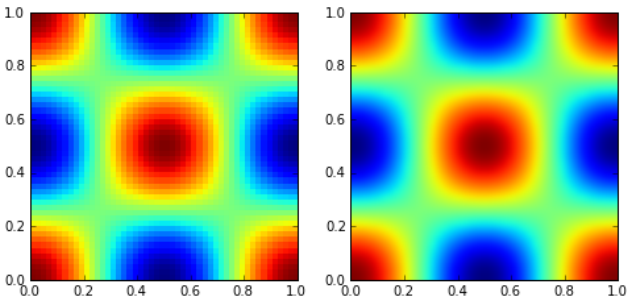
Out [31]:
```
<matplotlib.image.AxesImage at 0x103f1afd0>
```



Specify an extent of the image enables simple mapping of the array to a region in cartesian coordinates. Also note you can select different interpolation methods to smooth pixel boundaries.

In [32]:
```
fix, axes = plt.subplots(1, 2, figsize=(8, 6))
extent = np.array([x[0], x[-1], y[0], y[-1]])

axes[0].imshow(Z, extent=extent/(2*pi), interpolation='nearest')
axes[1].imshow(Z, extent=extent/(2*pi), interpolation='bicubic')
```

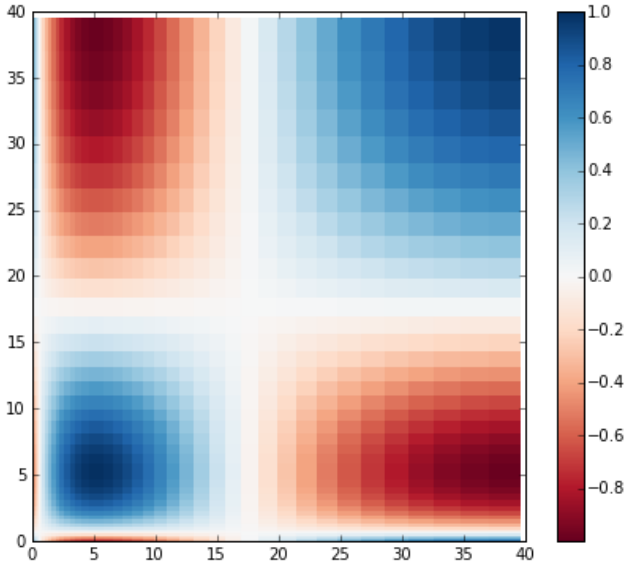
Out [32]: <matplotlib.image.AxesImage at 0x103f03b50>



pcolor

If your grid is not equally spaced in x and y then you cannot use `imshow()`. Instead you can use `Axes.pcolor()` for any rectilinear grids (where all x grid lines are perpendicular y grid lines). Here we stretch the x and y coordinates and use `pcolor()` to plot rectangular grid boxes.

In [39]:
```
X2, Y2 = np.meshgrid(x**3, y**3)

fig, ax = plt.subplots(figsize=(7,6))

p = ax.pcolor(X2/(2*pi), Y2/(2*pi), Z, cmap=cm.RdBu)
cb = fig.colorbar(p, ax=ax)
```



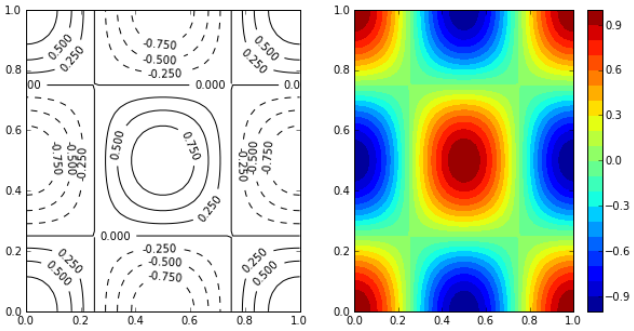
Note: The method `Axes.pcolormesh()` is even more flexible, allowing rotated and curvilinear grids

contour and contourf

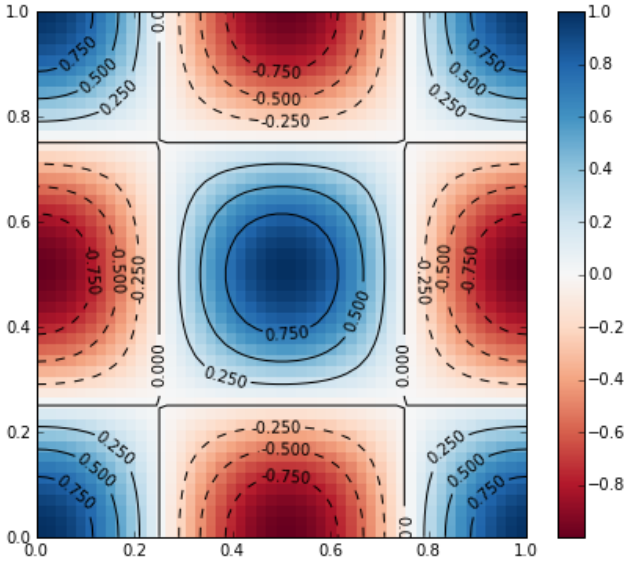
Matplotlib's `Axes.contour()` and `Axes.contourf()` methods create contour and filled contour plots respectively.

In [34]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,5))

p1 = axes[0].contour(X/(2*pi), Y/(2*pi), Z, colors='k')
axes[0].clabel(p1)

p2 = axes[1].contourf(X/(2*pi), Y/(2*pi), Z, 20)
fig.colorbar(p2, ax=axes[1])
```

Out [34]:
```
<matplotlib.colorbar.Colorbar instance at 0x104110e18>
```



In [35]:
```
fig, ax = plt.subplots(figsize=(7,6))

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



Exercise 1.3 : Plotting 2D data

5.1 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.