# Scientific Plotting with Matplotlib

## A Tutorial at PyCon US 2012

## March 8, 2012 Santa Clara, CA, USA

**author:** Dr.-Ing. Mike Müller
**email:** mmueller@python-academy.de
**version:** 1.1

# Contents

**1 Introduction** **4**

**2 IPython** **4**

**3 pylab** **4**

**4 Simple Plots** **4**

4.1 Exercises 7

**5 Properties** **7**

5.1 Exercise 9

**6 Text** **10**

6.1 Exercise 11

**7 Ticks** **11**

7.1 Where and What 11

7.2 Tick Locators 11

7.3 Tick Formatters 12

7.4 Exercises 13

**8 Figures, Subplots, and Axes** **13**

8.1 The Hierarchy 13

8.2 Figures 13

8.3 Subplots 14

8.4 Axes 15

8.5 Exercises 15

**9 Other Types of Plots** **15**

9.1 Many More 15

9.2 Bar Charts 15

9.3 Horizontal Bar Charts 16

9.4 Broken Horizontal Bar Charts 16

9.5 Box and Whisker Plots 17

9.6 Contour Plots 17

9.7 Histograms 18

9.8 Loglog Plots 19

9.9 Pie Charts 19

9.10 Polar Plots 20

9.11 Arrow Plots 20

9.12 Scatter Plots 21

9.13 Sparsity Pattern Plots 21

9.14 Stem Plots 22

9.15 Date Plots 22

**10 The Class Library 23**

10.1 The Figure Class 23

10.2 The Classes Axes and Subplot 24

10.3 Other Classes 24

10.4 Example 24

10.5 Exercises 25

**11 Creating New Plot Types 25**

11.1 Exercises 27

**12 Animations 28**

12.1 Exercises 30

## 1 Introduction

`matplotlib` is probably the single most used Python package for 2D-graphics. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats. We are going to explore matplotlib in interactive mode covering most common cases. We also look at the class library which is provided with an object-oriented interface.

## 2 IPython

IPython is an enhanced interactive Python shell that has lots of interesting features including named inputs and outputs, access to shell commands, improved debugging and many more. When we start it with the command line argument `-pylab`, it allows interactive `matplotlib` sessions that has Matlab/Mathematica-like functionality.

## 3 pylab

`pylab` provides a procedural interface to the `matplotlib` object-oriented plotting library. It is modeled closely after Matlab(TM). Therefore, the majority of plotting commands in `pylab` has Matlab(TM) analogs with similar arguments. Important commands are explained with interactive examples.

## 4 Simple Plots

Let's start an interactive session:

```
$python ipython.py -pylab
```

This brings us to the IPython prompt:

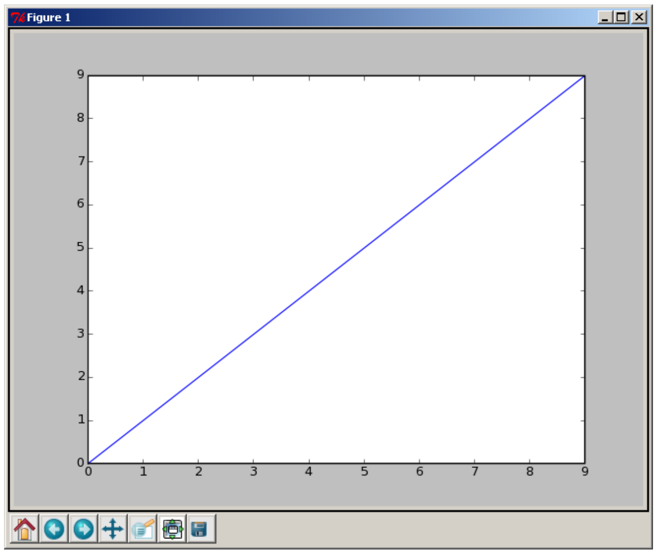
```
IPython 0.8.1 -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.
```

```
In [1]:
```

Now we can make our first, really simple plot:

```
In [1]: plot(range(10))
Out[1]: [<matplotlib.lines.Line2D instance at 0x01AA26E8>]

In [2]:
```

The numbers form 0 through 9 are plotted:



Now we can interactively add features to or plot:

```
In [2]: xlabel('measured')
Out[2]: <matplotlib.text.Text instance at 0x01A9D210>

In [3]: ylabel('calculated')
Out[3]: <matplotlib.text.Text instance at 0x01A9D918>

In [4]: title('Measured vs. calculated')
Out[4]: <matplotlib.text.Text instance at 0x01A9DF80>

In [5]: grid(True)

In [6]:
```

We get a reference to our plot:

```
In [6]: my_plot = gca()
```

and to our line we plotted, which is the first in the plot:

```
In [7]: line = my_plot.lines[0]
```

Now we can set properties using `set_something` methods:

```
In [8]: line.set_marker('o')
```

or the `setp` function:

```
In [9]: setp(line, color='g')
Out[9]: [None]
```

To apply the new properties we need to redraw the screen:

```
In [10]: draw()
```

We can also add several lines to one plot:

```
In [1]: x = arange(100)

In [2]: linear = arange(100)

In [3]: square = [v * v for v in arange(0, 10, 0.1)]

In [4]: lines = plot(x, linear, x, square)
```

Let's add a legend:

```
In [5]: legend(('linear', 'square'))
Out[5]: <matplotlib.legend.Legend instance at 0x01BBC170>
```

This does not look particularly nice. We would rather like to have it at the left. So we clean the old graph:

```
In [6]: clf()
```

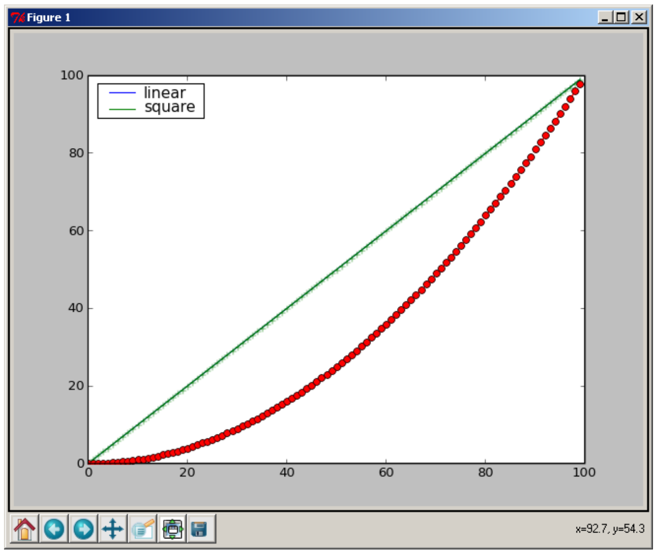
and print it anew providing new line styles (a green dotted line with crosses for the linear and a red dashed line with circles for the square graph):

```
In [7]: lines = plot(x, linear, 'g:+', x, square, 'r--o')
```

Now we add the legend at the upper left corner:

```
In [8]: l = legend(('linear', 'square'), loc='upper left')
```

The result looks like this:



## 4.1 Exercises

1. Plot a simple graph of a sinus function in the range 0 to 3 with a step size of 0.01.
2. Make the line red. Add diamond-shaped markers with size of 5.
3. Add a legend and a grid to the plot.

# 5 Properties

So far we have used properties for the lines. There are three possibilities to set them:

1) as keyword arguments at creation time: `plot(x, linear, 'g:+', x, square, 'r--o')`.

2. with the function `setp`: `setp(line, color='g')`.
3. using the `set_something` methods: `line.set_marker('o')`

Lines have several properties as shown in the following table:

| Property | Value |
|---|---|
| alpha | alpha transparency on 0-1 scale |
| antialiased | True or False - use antialised rendering |
| color | matplotlib color arg |
| data_clipping | whether to use numeric to clip data |
| label | string optionally used for legend |
| linestyle | one of - : -. - |
| linewidth | float, the line width in points |
| marker | one of + , o . s v x > <, etc |
| markeredgewidth | line width around the marker symbol |

| | |
|---|---|
| markeredgecolor | edge color if a marker is used |
| markerfacecolor | face color if a marker is used |
| markersize | size of the marker in points |

There are many line styles that can be specified with symbols:

| Symbol | Description |
|---|---|
| - | solid line |
| -- | dashed line |
| -. | dash-dot line |
| : | dotted line |
| . | points |
| , | pixels |
| o | circle symbols |
| ^ | triangle up symbols |
| v | triangle down symbols |
| < | triangle left symbols |
| > | triangle right symbols |
| s | square symbols |
| + | plus symbols |
| x | cross symbols |
| D | diamond symbols |
| d | thin diamond symbols |
| 1 | tripod down symbols |
| 2 | tripod up symbols |
| 3 | tripod left symbols |
| 4 | tripod right symbols |
| h | hexagon symbols |
| H | rotated hexagon symbols |
| p | pentagon symbols |
| \| | vertical line symbols |
| _ | horizontal line symbols |
| steps | use gnuplot style 'steps' # kwarg only |

Colors can be given in many ways: one-letter abbreviations, gray scale intensity from 0 to 1, RGB in hex and tuple format as well as any legal html color name.

The one-letter abbreviations are very handy for quick work. With following you can get quite a few things done:

| Abbreviation | Color |
|---|---|
| b | blue |
| g | green |
| r | red |
| c | cyan |
| m | magenta |
| y | yellow |
| k | black |
| w | white |

Other objects also have properties. The following table list the text properties:

| Property | Value |
|---|---|
| alpha | alpha transparency on 0-1 scale |
| color | matplotlib color arg |
| family | set the font family, eg 'sans-serif', 'cursive', 'fantasy' |
| fontangle | the font slant, one of 'normal', 'italic', 'oblique' |
| horizontalalignment | 'left', 'right' or 'center' |
| multialignment | 'left', 'right' or 'center' only for multiline strings |
| name | font name, eg, 'Sans', 'Courier', 'Helvetica' |
| position | x,y location |
| variant | font variant, eg 'normal', 'small-caps' |
| rotation | angle in degrees for rotated text |
| size | fontsize in points, eg, 8, 10, 12 |
| style | font style, one of 'normal', 'italic', 'oblique' |
| text | set the text string itself |
| verticalalignment | 'top', 'bottom' or 'center' |
| weight | font weight, e.g. 'normal', 'bold', 'heavy', 'light' |

## 5.1 Exercise

1. Apply different line styles to a plot. Change line color and thickness as well as the size and the kind of the marker. Experiment with different styles.

# 6 Text

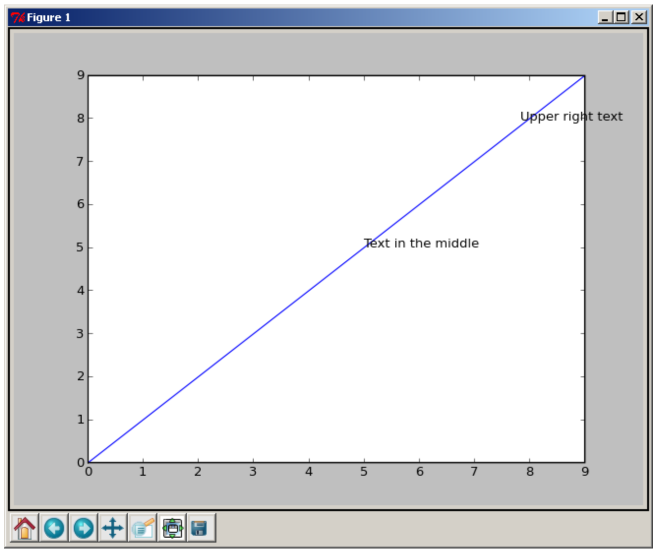
We've already used some commands to add text to our figure: `xlabel` `ylabel`, and `title`.

There are two functions to put text at a defined position. `text` adds the text with data coordinates:

```
In [2]: plot(arange(10))
In [3]: t1 = text(5, 5, 'Text in the middle')
```

`figtext` uses figure coordinates form 0 to 1:

```
In [4]: t2 = figtext(0.8, 0.8, 'Upper right text')
```



`matplotlib` supports TeX mathematical expression. So `r'$\pi$'` will show up as:
\pi

If you want to get more control over where the text goes, you use annotations:

```
In [4]: ax.annotate('Here is something special', xy = (1, 1))
```

We will write the text at the position (1, 1) in terms of data. There are many optional arguments that help to customize the position of the text. The arguments `textcoords` and `xycoords` specifies what `x` and `y` mean:

| argument | coordinate system |
|---|---|
| 'figure points' | points from the lower left corner of the figure |
| 'figure pixels' | pixels from the lower left corner of the figure |
| 'figure fraction' | 0,0 is lower left of figure and 1,1 is upper, right |
| 'axes points' | points from lower left corner of axes |
| 'axes pixels' | pixels from lower left corner of axes |

| | |
|---|---|
| 'axes fraction' | 0,1 is lower left of axes and 1,1 is upper right |
| 'data' | use the axes data coordinate system |

If we do not supply `xycoords`, the text will be written at `xy`.

Furthermore, we can use an arrow whose appearance can also be described in detail:

```
In [14]: plot(arange(10))
Out[14]: [<matplotlib.lines.Line2D instance at 0x01BB15D0>]

In [15]: ax = gca()

In [16]: ax.annotate('Here is something special', xy = (2, 1), xytext=(1,5))
Out[16]: <matplotlib.text.Annotation instance at 0x01BB1648>

In [17]: ax.annotate('Here is something special', xy = (2, 1), xytext=(1,5),
   ....: arrowprops={'facecolor': 'r'})
```

## 6.1 Exercise

1. Annotate a line at two places with text. Use green and red arrows and align it according to `figure points` and `data`.

# 7 Ticks

## 7.1 Where and What

Well formated ticks are an important part of publishing-ready figures. `matplotlib` provides a totally configurable system for ticks. There are tick locators to specify where ticks should appear and tick formatters to make ticks look like the way you want. Major and minor ticks can be located and formated independently from each other. Per default minor ticks are not shown, i.e. there is only an empty list for them because it is as `NullLocator` (see below).

## 7.2 Tick Locators

There are several locators for different kind of requirements:

| Class | Description |
|---|---|
| NullLocator | no ticks |
| IndexLocator | locator for index plots (e.g. where `x = range(len(y))`) |
| LinearLocator | evenly spaced ticks from min to max |
| LogLocator | logarithmically ticks from min to max |
| MultipleLocator | ticks and range are a multiple of base; either integer or float |
| AutoLocator | choose a MultipleLocator and dynamically reassign |

All of these locators derive from the base class `matplotlib.ticker.Locator`. You can make your own locator deriving from it.

Handling dates as ticks can be especially tricky. Therefore, `matplotlib provides special locators in ``matplotlib.dates`:

| Class | Description |
|---|---|
| MinuteLocator | locate minutes |
| HourLocator | locate hours |
| DayLocator | locate specified days of the month |
| WeekdayLocator | locate days of the week, e.g. MO, TU |
| MonthLocator | locate months, e.g. 10 for October |
| YearLocator | locate years that are multiples of base |
| RRuleLocator | locate using a matplotlib.dates.rrule |

## 7.3 Tick Formatters

Similarly to locators, there are formatters:

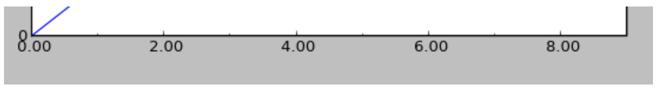
| Class | Description |
|---|---|
| NullFormatter | no labels on the ticks |
| FixedFormatter | set the strings manually for the labels |
| FuncFormatter | user defined function sets the labels |
| FormatStrFormatter | use a sprintf format string |
| IndexFormatter | cycle through fixed strings by tick position |
| ScalarFormatter | default formatter for scalars; autopick the fmt string |
| LogFormatter | formatter for log axes |
| DateFormatter | use an strftime string to format the date |

All of these formatters derive from the base class `matplotlib.ticker.Formatter`. You can make your own formatter deriving from it.

Now we set our major locator to 2 and the minor locator to 1. We also format the numbers as decimals using the `FormatStrFormatter`:

```
In [5]: major_locator = MultipleLocator(2)

In [6]: major_formatter = FormatStrFormatter('%5.2f')

In [7]: minor_locator = MultipleLocator(1)

In [8]: ax.xaxis.set_major_locator(major_locator)

In [9]: ax.xaxis.set_minor_locator(minor_locator)
In [10]: ax.xaxis.set_major_formatter(major_formatter)

In [10]: draw()
```

After we redraw the figure our x axis should look like this:



## 7.4 Exercises

1. Plot a graph with dates for one year with daily values at the x axis using the built-in module `datetime`.
2. Format the dates in such a way that only the first day of the month is shown.
3. Display the dates with and without the year. Show the month as number and as first three letters of the month name.

# 8 Figures, Subplots, and Axes

## 8.1 The Hierarchy

So far we have used implicit figure and axes creation. This is handy for fast plots. We can have more control over the display using `figure`, `subplot`, and `axes` explicitly. A `figure` in `matplotlib` means the whole window in the user interface. Within this `figure` there can be subplots. While `subplot` positions the plots in a regular grid, `axes` allows free placement within the `figure`. Both can be useful depending on your intention. We've already work with figures and subplots without explicitly calling them. When we call `plot` `matplotlib` calls `gca()` to get the current axes and `gca` in turn calls `gcf()` to get the current figure. If there is none it calls `figure()` to make one, strictly speaking, to make a `subplot(111)`. Let's look at the details.

## 8.2 Figures

A `figure` is the windows in the GUI that has "Figure #" as title. Figures are numbered starting from 1 as opposed to the normal Python way starting from 0. This is clearly MATLAB-style. There are several parameters that determine how the figure looks like:

| Argument | Default | Description |
|---|---|---|
| `num` | `1` | number of figure |
| `figsize` | `figure.figsize` | figure size in in inches (width, height) |
| `dpi` | `figure.dpi` | resolution in dots per inch |
| `facecolor` | `figure.facecolor` | color of the drawing background |
| `edgecolor` | `figure.edgecolor` | color of edge around the drawing background |
| `frameon` | `True` | draw figure frame or not |

The defaults can be specified in the resource file and will be used most of the time. Only the number of the figure is frequently changed.

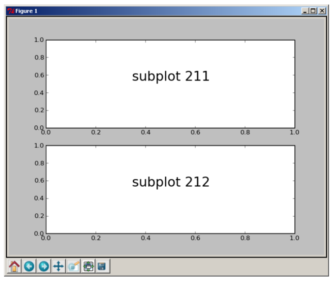
When you work with the GUI you can close a figure by clicking on the x in the upper right corner. But you can close a figure programmatically by calling `close`. Depending on the argument it closes (1) the current figure (no argument), (2) a specific figure (figure number or figure instance as argument), or (3) all figures (`all` as argument).

As with other objects, you can set figure properties also `setp` or with the `set_something` methods.

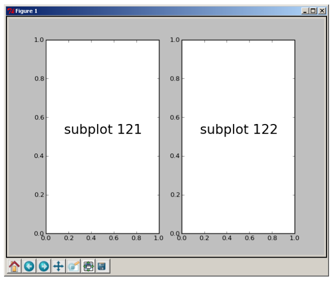
## 8.3 Subplots

With `subplot` you can arrange plots in regular grid. You need to specify the number of rows and columns and the number of the plot.

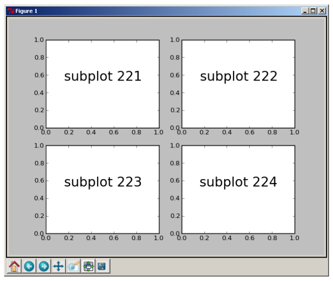
A plot with two rows and one column is created with `subplot(211)` and `subplot(212)`. The result looks like this:



If you want two plots side by side, you create one row and two columns with `subplot(121)` and `subplot(112)`. The result looks like this:



You can arrange as many figures as you want. A two-by-two arrangement can be created with `subplot(221)`, `subplot(222)`, `subplot(223)`, and `subplot(224)`. The result looks like this:



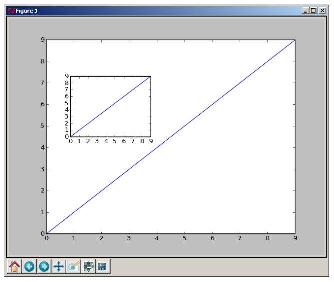
Frequently, you don't want all subplots to have ticks or labels. You can set the `xticklabels` or the `yticklabels` to an empty list (`[]`). Every subplot defines the methods `'is_first_row`, `is_first_col`, `is_last_row`, `is_last_col`. These can help to set ticks and labels only for the outer pots.

## 8.4 Axes

Axes are very similar to subplots but allow placement of plots at any location in the figure. So if we want to put a smaller plot inside a bigger one we do so with `axes`:

```
In [22]: plot(x)
Out[22]: [<matplotlib.lines.Line2D instance at 0x02C9CE90>]

In [23]: a = axes([0.2, 0.5, 0.25, 0.25])

In [24]: plot(x)
```

The result looks like this:



## 8.5 Exercises

1. Draw two figures, one 5 by 5, one 10 by 10 inches.
2. Add four subplots to one figure. Add labels and ticks only to the outermost axes.
3. Place a small plot in one bigger plot.

# 9 Other Types of Plots

## 9.1 Many More

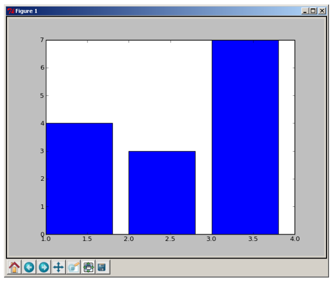
So far we have used only line plots. `matplotlib` offers many more types of plots. We will have a brief look at some of them. All functions have many optional arguments that are not shown here.

## 9.2 Bar Charts

The function `bar` creates a new bar chart:

```
bar([1, 2, 3], [4, 3, 7])
```

Now we have three bars starting at 1, 2, and 3 with height of 4, 3, 7 respectively:



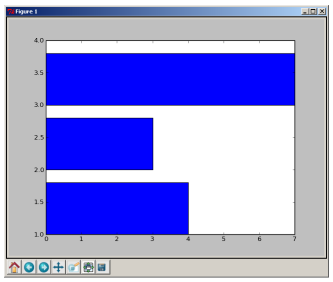
The default column width is 0.8. It can be changed with common methods by setting `width`. As it can be `color` and `bottom`, we can also set an error bar with `yerr` or `xerr`.

## 9.3 Horizontal Bar Charts

The function `barh` creates an vertical bar chart. Using the same data:

```
barh([1, 2, 3], [4, 3, 7])
```

We get:



## 9.4 Broken Horizontal Bar Charts

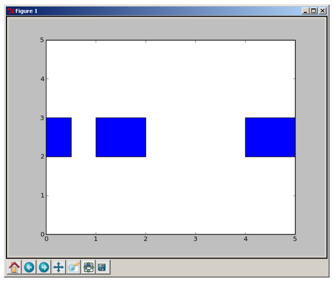
We can also have discontinuous vertical bars with `broken_barh`. We specify start and width of the range in y-direction and all start-width pairs in x-direction:

```
yrange = (2, 1)
xranges = ([0, 0.5], [1, 1], [4, 1])
broken_barh(xranges, yrange)
```

We changes the extension of the y-axis to make plot look nicer:

```
ax = gca()
ax.set_ylim(0, 5)
(0, 5)
draw()
```

and get this:



## 9.5 Box and Whisker Plots

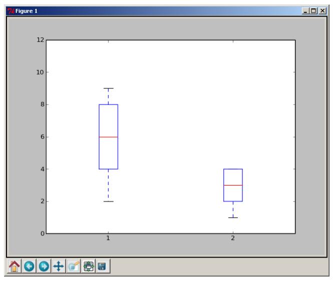
We can draw box and whisker plots:

```
boxplot((arange(2, 10), arange(1, 5)))
```

We want to have the whiskers well within the plot and therefore increase the y axis:

```
ax = gca()
ax.set_ylim(0, 12)
draw()
```

Our plot looks like this:



The range of the whiskers can be determined with the argument `whis`, which defaults to 1.5. The range of the whiskers is between the most extreme data point within `whis*(75%-25%)` of the data.

## 9.6 Contour Plots

We can also do contour plots. We define arrays for the x and y coordinates:

```
x = y = arange(10)
```

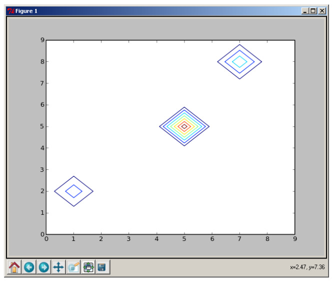
and also a 2D array for z:

```
z = ones((10, 10))
z[5,5] = 7
z[2,1] = 3
z[8,7] = 4
z
array([[ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  3.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  7.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  4.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]])
```

Now we can make a simple contour plot:

```
contour(x, x, z)
```

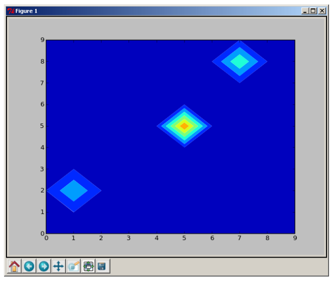
Our plot looks like this:



We can also fill the area. We just use numbers form 0 to 9 for the values v:

```
v = x
contourf(x, x, z, v)
```

Now our plot area is filled:



## 9.7 Histograms

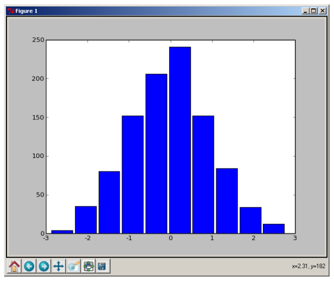
We can make histograms. Let's get some normally distributed random numbers from numpy:

```
import numpy as N
r_numbers = N.random.normal(size= 1000)
```

Now we make a simple histogram:

```
hist(r_numbers)
```

With 100 numbers our figure looks pretty good:



## 9.8 Loglog Plots

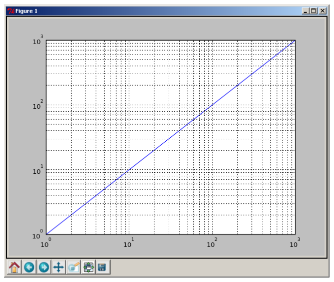
Plots with logarithmic scales are easy:

```
loglog(arange(1000))
```

We set the mayor and minor grid:

```
grid(True)
grid(True, which='minor')
```

Now we have loglog plot:



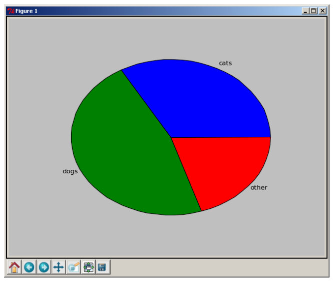
If we want only one axis with a logarithmic scale we can use `semilogx` or `semilogy`.

## 9.9 Pie Charts

Pie charts can also be created with a few lines:

```
data = [500, 700, 300]
labels = ['cats', 'dogs', 'other']
pie(data, labels=labels)
```

The result looks as expected:



## 9.10 Polar Plots

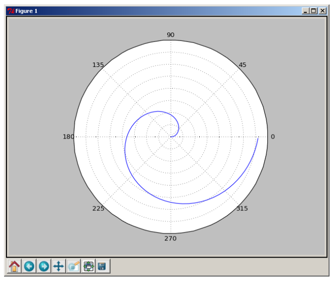
Polar plots are also possible. Let's define our `r` from 0 to 360 and our `theta` from 0 to 360 degrees. We need to convert them to radians:

```
r = arange(360)
theta = r / (180/pi)
```

Now plot in polar coordinates:

```
polar(theta, r)
```

We get a nice spiral:



## 9.11 Arrow Plots

Plotting arrows in 2D plane can be achieved with `quiver`. We define the x and y coordinates of the arrow shafts:

```
x = y = arange(10)
```

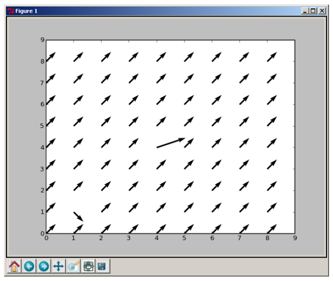
The x and y components of the arrows are specified as 2D arrays:

```
u = ones((10, 10))
v = ones((10, 10))
u[4, 4] = 3
v[1, 1] = -1
```

Now we can plot the arrows:

```
quiver(x, y, u, v)
```

All arrows point to the upper right, except two. The one at the location (4, 4) has 3 units in x-direction and the other at location (1, 1) has -1 unit in y direction:



## 9.12 Scatter Plots

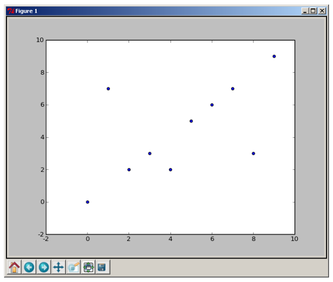
Scatter plots print x vs. y diagrams. We define x and y and make some point in y random:

```
x = arange(10)
y = arange(10)
y[1] = 7
y[4] = 2
y[8] = 3
```

Now make a scatter plot:

```
scatter(x, y)
```

The three different values for y break out of the line:



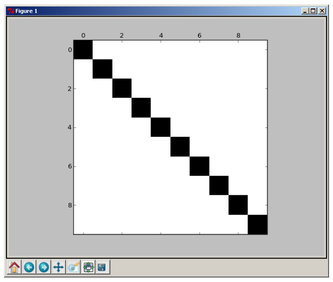
## 9.13 Sparsity Pattern Plots

Working with sparse matrices, it is often of interest as how the matrix looks like in terms of sparsity. We take an identity matrix as an example:

```
i = identity(10)
i
array([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]])
```

Now we look at it more visually:

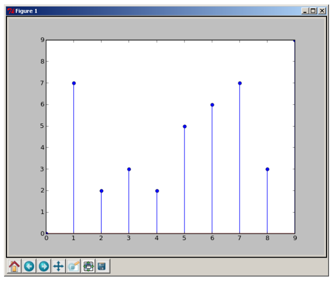
```
spy(i)
```



## 9.14 Stem Plots

Stem plots vertical lines at the given x location up to the specified y location. Let's reuse x and y from our scatter (see above):

```
stem(x, y)
```



## 9.15 Date Plots

There is a function for creating date plots. Let's define 10 dates starting at January 1st 2000 at 15.day intervals:

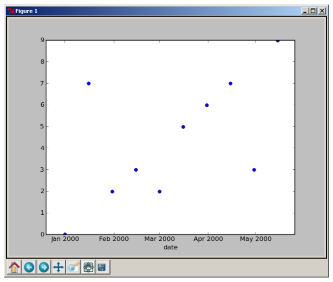
```
import datetime
d1 = datetime.datetime(2000, 1, 1)
delta = datetime.timedelta(15)
dates = [d1 + x * delta for x in range(1
dates
[datetime.datetime(2000, 1, 1, 0, 0),
 datetime.datetime(2000, 1, 16, 0, 0),
 datetime.datetime(2000, 1, 31, 0, 0),
 datetime.datetime(2000, 2, 15, 0, 0),
 datetime.datetime(2000, 3, 1, 0, 0),
 datetime.datetime(2000, 3, 16, 0, 0),
 datetime.datetime(2000, 3, 31, 0, 0),
 datetime.datetime(2000, 4, 15, 0, 0),
 datetime.datetime(2000, 4, 30, 0, 0),
 datetime.datetime(2000, 5, 15, 0, 0)]
```

We reuse our data from the scatter plot (see above):

```
y
array([0, 7, 2, 3, 2, 5, 6, 7, 3, 9])
```

Now we can plot the dates at the x axis:

```
plot_date(dates, y)
```



# 10 The Class Library

So far we have used the `pylab` interface only. As the name suggests it is just wrapper around the class library. All `pylabb` commands can be invoked via the class library using an object-oriented approach.

## 10.1 The Figure Class

The class `Figure` lives in the module `matplotlib.figure`. Its constructor takes these arguments:

```
figsize=None, dpi=None, facecolor=None, edgecolor=None,
linewidth=1.0, frameon=True, subplotpars=None
```

Comparing this with the arguments of `figure` in `pylab` shows significant overlap:

```
num=None, figsize=None, dpi=None, facecolor=None
edgecolor=None, frameon=True
```

Figure provides lots of methods, many of them have equivalents in pylab. The methods add_axes and add_subplot are called if new axes or subplot are created with axes or subplot in pylab. Also the method gca maps directly to pylab as do legend, text and many others.

There are also several set_something method such as set_facecolor or set_edgecolor that will be called through pylab to set properties of the figure. Figure also implements get_something methods such as get_axes or get_facecolor to get properties of the figure.

## 10.2 The Classes Axes and Subplot

In the class Axes we find most of the figure elements such as Axis, Tick, Line2D, or Text. It also sets the coordinate system. The class Subplot inherits from Axes and adds some more functionality to arrange the plots in a grid.

Analogous to Figure, it has methods to get and set properties and methods already encountered as functions in pylab such as annotate. In addition, Axes has methods for all types of plots shown in the previous section.

## 10.3 Other Classes

Other classes such as text, Legend or Ticker are setup very similarly. They can be understood mostly by comparing to the pylab interface.

## 10.4 Example

Let's look at an example for using the object-oriented API:

```python
#file matplotlib/oo.py

import matplotlib.pyplot as plt                 #1

figsize = (8, 5)                                #2
fig = plt.figure(figsize=figsize)               #3
ax = fig.add_subplot(111)                       #4
line = ax.plot(range(10))[0]                    #5
ax.set_title('Plotted with OO interface')       #6
ax.set_xlabel('measured')
ax.set_ylabel('calculated')
ax.grid(True)                                   #7
line.set_marker('o')                            #8

plt.savefig('oo.png', dpi=150)                  #9
plt.show()                                      #10
```

Since we are not in the interactive pylab-mode, we need to import pyplot (#1). We set the size of our figure to be 8 by 5 inches (#2). Now we initialize a new figure (#3) and add a subplot to the figure (#4). The 111 says one plot at position 1, 1 just as in MATLAB. We create a new plot with the numbers from 0 to 9 and at the same time get a reference to our line (#5). We can add several things to our plot. So we set a title and labels for the x and y axis (#6). We also want to see the grid (#7) and would like to have little filled circles as markers (#8).

Finally, we save our figure as a PNG file specifying the desired resolution in dpi (#9) and show our figure on the screen (#10).

## 10.5 Exercises

1. Use the object-oriented API of matplotlib to create a png-file with a plot of two lines, one linear and square with a legend in it.

# 11 Creating New Plot Types

We can come up with our own new plot type. Matplotlib provides all the building blocks to compose new plots. We would like to have a stacked plot, that is lines are drawn on top of each other and the y values represent the difference to the other plot rather than to zero. We implement our own plot type. It assumes all y values are positive and the lowest plot will be filled starting from zero, i.e the x axis.

```
"""A simple stacked plot.
"""

import matplotlib as mpl                                                #1
import matplotlib.pyplot as plt                                         #2
import numpy                                                            #3
```

First we need to import `matplotlib` itself (#1) as well as `pyplot` (#2) and `numpy` (#3).

```
StackPlot(object):                                                #4
    """A stacked plot.

    Data are stacked on top of each other and the space
    between the lines is filled with color.
    """

    def __init__(self, x, y_data, axes=None, colors=None, names=None,
                 loc='best'):                                           #5
        """Show the plot.
        """
        # We've got plenty of default arguments.
        # pylint: disable-msg=R0913
        # Make sure we can work with numpy arrays.
        self.x = numpy.array(x)                                         #6
        self.y_data = numpy.array(y_data)
        self.axes = axes
        if not axes:
            self.axes = plt.gca()                                       #7
        default_colors = ['r', 'b', 'g', 'y', 'm', 'w']                 #8
        self.colors = colors
        self.names = names
        self.loc = loc
        if not self.colors:                                             #9
            # Repeat colors as needed.
            ncolors = len(self.y_data)
            colors = default_colors * (1 + (ncolors // len(default_colors)))
            self.colors = colors[:ncolors]
        self.stacked = None
        self._stack_data()
```

Now we define a new class `StackedPlot` (#4). It takes the x values and a two dimensional array of y values, where each row will become one plot, as well as a bunch of optional arguments that we will meet later (#5). We convert all data into `numpy` arrays (#6). This won't hurt if they are arrays already. If no `axes` was supplied, we just get the current axes (#7). This is consistent with the behavior of the standard plots. We define some default colors that will be used if none are given in the `__init__` (#8). We don't know how many plots might be stacked. Therefore, we just repeat the default colors again and again until all data have a color (#9).

```
def _stack_data(self):                                          #10
        """Accumulate data.
        """
        nlines = self.y_data.shape[0] + 1                           #11
        self.stacked = numpy.zeros((nlines, self.y_data.shape[1]))  #12
        for index in xrange(1, nlines):
            self.stacked[index] = (self.stacked[index - 1] +         #13
                                   self.y_data[index - 1])
```

We stack our data (#10). We need one more entry than y values (#11) and the first one is all zeros (#12). Than we just add the new values on top of the ones below (#13).

```
def draw(self):                                                 #14
        """Draw the plot.
        """
        for data1, data2, color in zip(self.stacked[:-1], self.stacked[1:],
                                       self.colors):                #15
            self.axes.fill_between(self.x, data1, data2, color=color) #16
            self.axes.plot(self.x, data2, 'k', linewidth=0.1)         #17
        if self.names:
            rects = []
            for color in self.colors:                               #18
                rects.append(plt.Rectangle((0, 0), 1, 1, fc=color))
            self.axes.legend(rects, self.names, loc=self.loc,
                        prop=mpl.font_manager.FontProperties(size=10))
```

We draw our new plot (#14). Using `zip`, we go through all stacked data each time with a lower and upper boundary as well as through our colors (#15). We fill the space between both plots (#16) and also plot the line in black (#17). Furthermore, we make a nice legend for all colors (#18).

```
def __getattr__(self, name):                                    #19
        """Delegate not found attributes to axes.

        This works for set_tile, set_xlabel etc.
        """
        try:
            return getattr(self.axes, name)
        except AttributeError:
            raise AttributeError("'StackPlot' object has no attribute '%s'"
                                 % name)
```

We delegate all attributes that are not found (#19) to or axes. This allows to provide all the `set_<property>` methods without actually implementing them.

```
if __name__ == '__main__':

    def test():                                                  #20
        """Check if it works.
        """
        x = range(10)                                            #21
        y_data = numpy.ones((5, 10), dtype=numpy.float)          #22
        y_data[1, 1] = 2
        y_data[2, 1] = 2
        y_data[3, 1] = 2
        y_data[1, 2] = 0.5
        y_data[2, 3] = 0.5
        y_data[3, 4] = 0.5
        fig = fig = plt.figure()
        s_plot = StackPlot(x, y_data,
                           axes=fig.add_subplot(111),            #23
                           names=['a', 'b', 'c', 'd', 'e'])
        s_plot.set_title('My Stacked Plot')                      #24
        s_plot.set_xlabel('x')
        s_plot.set_ylabel('y')
        s_plot.draw()                                            #25
        plt.show()                                               #26

    test()
```

Finally, we test our program (#20). We create data for x (#22) and y (#23). We replace some data in y, that is only ones, with some different numbers. Now we can add our new stacked plot to our axes (#23). This argument is optional. Leaving it out, the class will use the current axes. We set some properties (#24), create the plot (#25) and show our figure (#26).

## 11.1 Exercises

1. Change some y data in the test function and see how this changes the graph. Increase the number of stacked plots to 50 and 200. See what happens.
2. Place four stacked plots in one figure. Change the data slightly for each stacked plot.
3. Move the legend to different place for each of the four plots.

# 12 Animations

Animations can be useful to show how things change over time. There are two basic ways to produce an animation with matplotlib:

1. Saving pixel graphics for each animation frame and stitch them together with tools like imagemagick or ffmpeg.
2. Having an animation directly in the figure.

The first approach is straight forward. The second typically needs some interaction with the GUI library. Let's have a look at an example:

```
"""Animation with matplolib.
"""

import random

import matplotlib                                                       #1
matplotlib.use('TkAgg')                                                 #2
import matplotlib.pyplot as plt
```

After importing `matplotlib` (#1), we need to tell what GUI to use. We use TKinter here (#2). Other supported tool kits are Qt, wxPython and GTK. The solution presented here is Tkinter specific.

```
class Residual(object):                                                 #3
    """Generator for residual values.

    This somehow mimics the residual a solver for
    system of equations would produces.
    """

    def __init__(self, start, limit):                                   #4
        self.value = start
        self.limit = limit
        self.counter = 0
        self.done = False

    def __call__(self):                                                 #5
        if self.done:
            return None                                                 #6
        diff = random.random() * self.value                             #7
        if self.counter == 2:                                           #8
            self.value += diff
            self.counter = 0
        else:                                                           #9
            self.value -= diff
            self.counter += 1
        if self.value <= self.limit:                                    #10
            self.done = True
        return self.value
```

The class `Residual` (#3) will be used as function that keeps its state. It is a placeholder for a numerical program that solves something interactively and is finished if a certain value of the residual is reached. The `__int__` (#4) takes the starting value and the limit to be reached. We implement the special method `__call__` (#5) making an instance of this class behave just like a function but with the additional feature of keeping state information in `self`. If we are done, the limit was reached, we return `None` (#6). The difference leading to a new residual is determined by multiplying the old value with a random number between 0 and 1 (#7). We increment the residual by `diff` once (#8) after decrementing it twice (#9). If the residual is less or equal to the limit, we are done (#10).

```
class ResidualGraph(object):                               #11
    """Semilog plot with matplotlib.

    This plot is updated for every calculation time step.
    Therefore it grows dynamically.
    """

    def __init__(self, start):
        # make a figure
        self.fig = plt.figure()                            #12
        self.axes = self.fig.add_subplot(111)              #13
        self.counter = 0                                   #14
        self.x = [self.counter]                            #15
        self.y = [start]                                   #16
        self.show_initial_graph()                          #17
        self.window = plt.get_current_fig_manager().window #18
```

The `ResidualGraph` (#11) is responsible for displaying the animated figure. After creating a new figure (#12) and adding one subplot (#13), we start with a counter of zero (#14) and use it as the first and so far only value for x (#15). The value makes the first entry in our y values (#16). After initializing the graph, we need to get a hold of the window (#18). This will be needed to start the animation as wee will see later.

```
def show_initial_graph(self):
        """Show a first version of the graph without calculated residuals.
        """
        self.axes.semilogy(self.x, self.y)                 #19
        self.axes.set_title('Solver test program')
        self.axes.set_xlabel('Number of steps')
        self.axes.set_ylabel('Nonlinear residual')
        self.fig.canvas.draw()                             #20
```

Initializing the graph is straight forward. We use a semi-log plot (#19). After adding title and labels, we draw our plot (#20).

```
def update(self, value):                                 #21
        """Redraw the graph with an added value for the residual.
        """
        self.counter += 1                                  #22
        self.x.append(self.counter)                        #23
        self.y.append(value)                               #24
        plt.semilogy(self.x, self.y, color='blue')          #25
        plt.draw()                                          #26
```

The update-method (#21) is important, because it will redraw the graph for each animation step. After incrementing the counter (#22) and adding it to the x values (#23), we append the new residual value to our y values (#24) and make new plot (#25) that we draw (#26).

```
def start_animation(start, limit):                          #27
    """Start the animation.
    """

    def animate():                                          #28
        """Animation function will be called by GUI library (Tkinter).
        """
        residual = get_residual()                           #29
        # show value and update graph
        if residual is not None:                            #30
            graph.window.after(300, animate)                #31
            print residual
            graph.update(residual)                          #32
        else:
            print 'done'                                    #33
    get_residual = Residual(start, limit)                   #34
    graph = ResidualGraph(start)                            #35
    graph.window.after(300, animate)                        #36
    plt.show()                                              #37
```

In start_animation (#27), we define a nested function animate (#28). This function will be called by Tkinter after a specified amount of time. We get the new value for the residual from get_residual(), which is an instance of the class Residual (#34). As long as we have a value for the residual (#30), we call window.after with a delay of 300 milliseconds and the function animate, i.e. the function we just define (#31). The object window is from Tkinter (#18). We also update the graph (#32) which is an instance of ResidualGraph (#35). The first call to window.after (#36) starts the animation. We also want the plot stay alive after the animation is finished and therefore call show (#37).

```
if __name__ == '__main__':

    start_animation(1e4, 1e-7)                              #38
```

Finally, we start everything with some numbers spanning eleven orders of magnitude (#38).

## 12.1 Exercises

1. Alternate the color of the plotted line with each animation step.