Python 3: Plotting simple graphs

Bruce Beckles `mbb10@cam.ac.uk`

Bob Dowling `rjd4@cam.ac.uk`

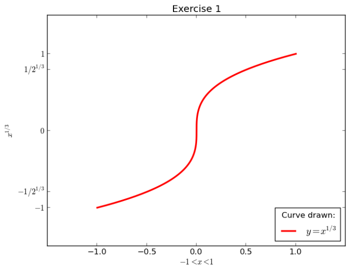
4 February 2013

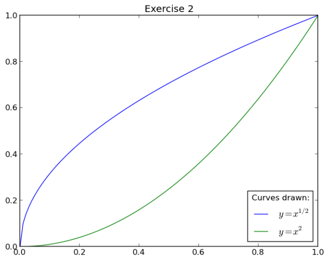
What's in this course

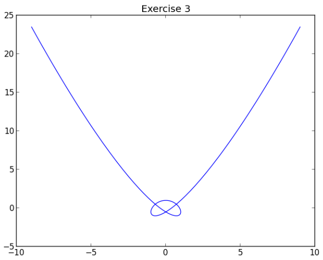
This course runs through a series of types of graph. You will learn most by doing all of them, obviously. However, if you want to fast-forward to just the sort of graph you want to do then you need to do the first three sections and then you can jump to the type of graph you want.

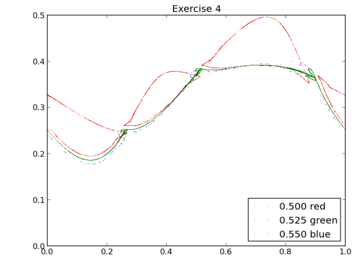
1. **Drawing a basic line graph**
2. **Drawing multiple curves**
3. **Parametric graphs**
4. Scatter graphs
5. Error bar graphs
6. Polar graphs
7. Histograms
8. Bar charts

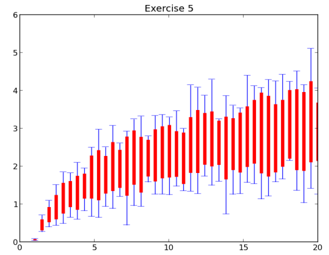
By the end of each section you will be creating graphs like these, the end-of-section exercises:

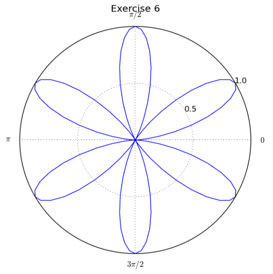


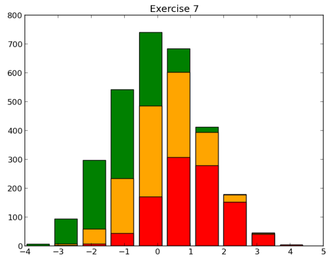


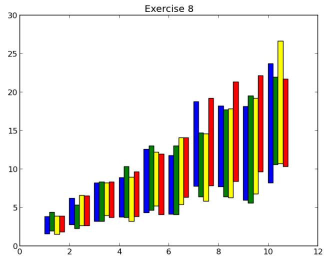












Prerequisites

This self-paced course assumes that you have a knowledge of Python 3 equivalent to having completed one or other of

- Python 3: Introduction for Absolute Beginners, or
- Python 3: Introduction for Those with Programming Experience

Some experience beyond these courses is always useful but no other course is assumed.

The course also assumes that you know how to use a Unix text editor (`gedit`, `emacs`, `vi`, …).

The Python module used in this course is built on top of the numerical python module, `numpy`. You do not need to know this module for most of the course as the module maps Python lists into `numpy`'s one-dimensional arrays automatically. There is *one* place, in the bar charts chapter, where using `numpy` makes life much easier, and we will do it both ways. You can safely skip the `numpy` way if you do not know about it.

Facilities for this session

- The computers in this room have been prepared for these self-paced courses. They are already logged in with course IDs and have home directories specially prepared. Please do not log in under any other ID.
- These courses are held in a room with two demonstrators. If you get stuck or confused, or if you just have a question raised by something you read, please ask!
- The home directories contain a number of subdirectories one for each topic. For this topic please enter directory `graphs`. All work will be completed there:

```
$ cd graphs
$ pwd
/home/x250/graphs
$
```

- At the end of this session the home directories will be cleared. Any files you leave in them will be deleted. Please copy any files you want to keep.
- These handouts and the prepared folders to go with them can be downloaded from `www.ucs.cam.ac.uk/docs/course-notes/unix-courses/pythontopics`
- You are welcome to annotate and keep this handout.
- The formal documentation for the topics covered here can be found online at `matplotlib.org/1.2.0/index.html`

Notation

Warnings

Warnings are marked like this. These sections are used to highlight common mistakes or misconceptions.

Exercises

Exercise 0

Exercises are marked like this.

Input and output

Material appearing in a terminal is presented like this:

```
$ more lorem.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
--More--(44%)
```

The material you type is presented like this: **`ls`**. (Bold face, typewriter font.)

The material the computer responds with is presented like this: “`Lorem ipsum`”. (Typewriter font again but in a normal face.)

Keys on the keyboard

Keys on the keyboard will be shown as the symbol on the keyboard surrounded by square brackets, so the “A key” will be written “[A]”. Note that the return key (pressed at the end of every line of commands) is written “[↲]”, the shift key as “[⇧]”, and the tab key as “[↹]”. Pressing more than one key at the same time (such as pressing the shift key down while pressing the A key) will be written as “[⇧]+[A]”. Note that pressing [A] generates the lower case letter “a”. To get the upper case letter “A” you need to press [⇧]+[A].

Content of files

The content[1] of files (with a comment) will be shown like this:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis nulla pariatur.                                This is a comment about the line.
```

[1] The example text here is the famous “lorem ipsum” dummy text used in the printing and typesetting industry. It dates back to the 1500s. See `http://www.lipsum.com/` for more information.

!

This course uses version 3 of the Python language. Run your scripts as

```
$ python3 example1.py
```

etc.

Drawing a basic line graph

We start by importing the graphics module we are going to use. The module is called "`matplotlib.pyplot`" which is a bit long so we import it under an alias, "`pyplot`":

```
import matplotlib.pyplot as pyplot
```

We also need some data to plot. Rather than spend time preparing data in our scripts the course has a local module which consists of nothing but dummy data for plotting We import this, but remember it's just a local hack:

```
import plotdata

x = plotdata.data01x

y = plotdata.data01y
```

The names "x" and "y" are attached to two lists of floating point values, the x-values and the y-values. (If you know the `numpy` module then these can also be bound to two numpy 1-d arrays of values too.)

The list `plotdata.data01x` is just a list of numbers from -1 to +1 and `plotdata.plot01y` is a list of those numbers cubed.

Now we have to actually plot the data:

```
pyplot.plot(x,y)
```

And that's it for a graph with all the default settings. Now we just need to save the graph to a file or display it on the screen:

```
pyplot.savefig('example01.png')
```

The `pyplot.savefig()` function saves the current graph to a file identified by name. It can take a Python file object, but if you do that remember to open it in binary mode. (So open it with `open('example01.png', 'wb')`.)

!

There is an alternative to `pyplot.savefig()` called `pyplot.show()`. This launches an interactive viewer. There is a problem with using it in an interactive session, though:
`pyplot.show()` resets the state of the graphics subsystem and you have to repeat most of the commands you had previously typed in to get a second graph in the same interactive session. It's often simpler to quite Python 3 and restart it.
This is why this course uses `pyplot.savefig()` and scripts.

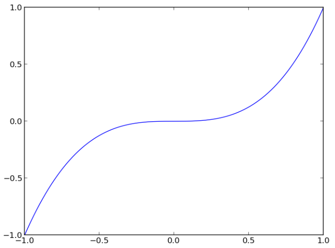
We can now run the script:

```
$ python3 example01.py
```

If you want to view a graphics file's contents, either double-click on it in the graphical file browser or open it from the command line with the "`eog`" program[2]:

[2] "`eog`" stands for **E**ye **o**f **G**nome and is the default graphics viewer for the Gnome desktop environment that we use on the Managed Cluster Service.

```
$ eog example01.png &
[1] 5537
$
```



(This example runs eog in the background. Lose the ampersand at the end to run it in the foreground.)

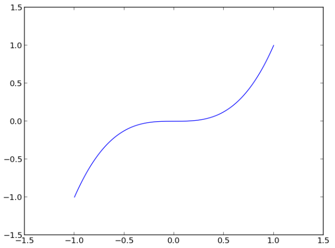
We can observe a few aspects of the default behaviour that we might want to modify:

- the axes exactly match the data ranges
- the tick marks are every 0.5 units
- there is no title
- there are no labels on the axes
- there is no legend
- the line is blue

We will address these points one-by-one. A series of example scripts called “example01a.py”, “example01b.py”, etc. are provided to illustrate each modification in isolation. Each script is accompanied by its associated graph “example01a.png”, “example01b.png”, etc.

Setting the axes limits

(Script: example01a.py, graph: example01a.png)



The axes on the graph are automatically derived from the data ranges in x-values and y-values. This is often what you want and is not a bad default.

There is a function `pyplot.axis()` which explicitly sets the limits of the axes to be drawn. It takes a single argument which is a list of four values: `pyplot.axis([`X_{min}`,`X_{max}`,`Y_{min}`,`Y_{max}`])`.

For example, to set the line floating with 0.5 units of space on either side of our curve we would add the following line *before* the call to `pyplot.plot(x,y)`:

```
pyplot.axis([-1.5, 1.5, -1.5, 1.5])
```

Typically all the settings calls are run before `pyplot.plot()`. The only call that comes afterwards is `pyplot.savefig()` or `pyplot.show()`.

!

There is another function with a very similar name, `pyplot.axes()`, with very similar arguments that does something related but different.

You want `pyplot.axis.()` (n.b.: singular *axis*)

Setting the tick marks

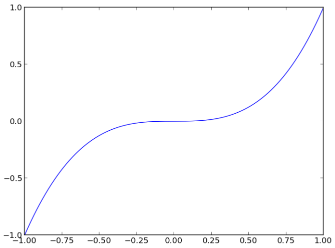
There is a default set of tick marks that are created, based on the axis limits. These are often sufficient but you can change them if you want to.

There is a function `pyplot.xticks()` which sets the ticks along the x-axis and a corresponding function, `pyplot.yticks()`, for the y-axis. These can set the labels on the ticks as well as the position of the ticks.

In its simplest use, `pyplot.xticks()` is passed a list of x-values at which it should draw ticks.

(Script: example01b.py, graph: example01b.png)

For example, to set x-ticks every 0.25 rather than every 0.5 we could say



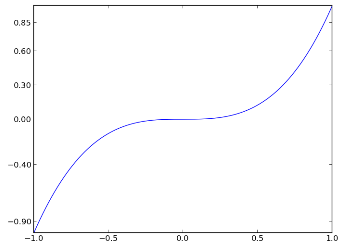
```
pyplot.xticks([0.25*k for k in range(-4,5)])
```

where we use Python’s “list comprehension” syntax:

```
>>> [0.25*k for k in range(-4,5)]
[-1.0, -0.75, -0.5, -0.25, 0.0, 0.25, 0.5, 0.75, 1.0]
```

If we really wanted to we could have arbitrary ticks:

```
pyplot.yticks([-0.9, -0.4, 0.0, 0.3, 0.6, 0.85])
```

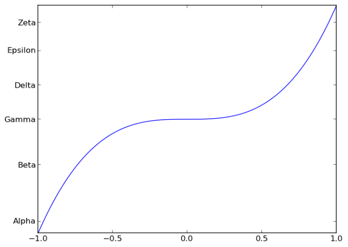


(Script: example01c.py, graph: example01c.png)

This is not completely pointless. The specific values might correspond to critical points in some theory, in which case you might want to label them as such. The pyplot.xticks() and pyplot.yticks() functions take an optional second argument which is a list of labels to attach to the tick marks instead of their numerical values.

We can name the ticks like this:

```
pyplot.yticks([-0.9, -0.4, 0.0, 0.3, 0.6, 0.85],
    ['Alpha','Beta','Gamma','Delta','Epsilon','Zeta'])
```



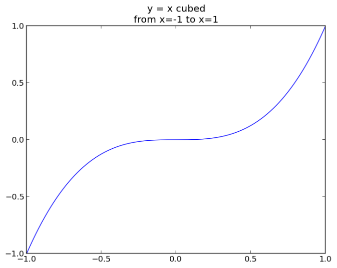
(Script: example01d.py, graph: example01d.png)

Setting the title

By default the graph comes with no title.

There is a function pyplot.title() which sets a title. The title is just a simple text string which can contain multiple lines but does not need to end with a new line character.

```
pyplot.title('y = x cubed\nfrom x=-1 to x=1')
```

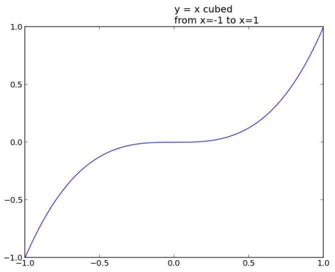


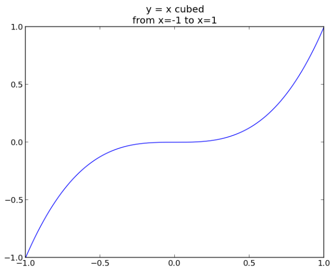
(Script: example01e.py, graph: example01d.png)

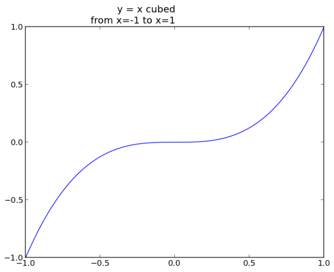
Note that the title goes above the box and is centre-aligned.

There is an optional argument on pyplot.title(), horizontalalignment=… which can takes the string values 'left', 'center', or 'right'. This sets the title text’s alignment relative to the centre of the figure.

(Files: example01f.py, example01g.py, example01h.py for “left”, “center” and “right” respectively.)

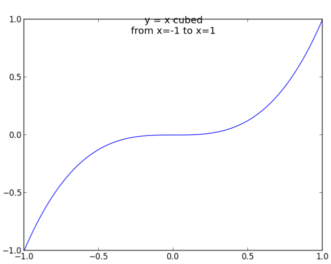


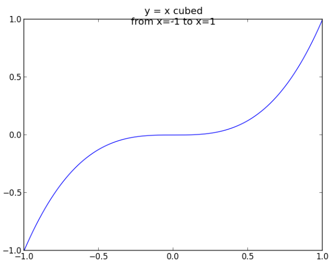


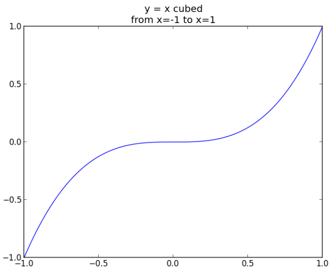


In addition to horizontal alignment we can also control the vertical alignment with the verticalalignment=… parameter. It can take one of four string values: 'top', 'center', 'bottom', and 'baseline' (the default).

(Files: example01i.py, example01j.py, example01k.py for “top”, “center” and “bottom” respectively, with the corresponding graphs in example01i.png, example01j.png, example01k.png.)





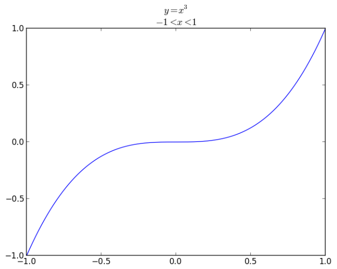


Of course, the title would benefit from proper mathematical notation; “y=x^{3}” would look much better than “y = x cubed”. To support this, `matplotlib` supports a subset of Teχ’s syntax[3] if the text is enclosed in dollar signs (also from Teχ):

```
pyplot.title('$y = x^{3}$\n$-1<x<1$')
```

(Script: `example01l.py`, graph: `example01l.png`)

This mathematical notation can be used in *any* of `pyplot`’s text functions.



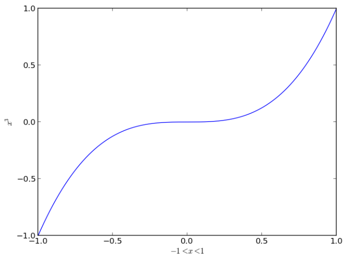
Labelling the axes

By default the axes are unlabelled. (Don’t confuse this with the individual tick marks being unlabelled.)

There are functions, `pyplot.xlabel()` and `pyplot.ylabel()`, which set the labels on the two axes.

```
pyplot.xlabel('$-1<x<1$')
pyplot.ylabel('$y=x^{3}$')
```

(Script: `example01m.py`, graph: `example01m.png`)



Note that the text of the y-axis label is rotated to match the axis. Use

```
pyplot.ylabel('$y=x^{3}$', rotation='horizontal')
```

to write horizontally. The alternative, “`rotation='vertical'`” is the default. (See the script `example01n.py` and its corresponding graph `example01n.png` for an example.)

Adding a legend

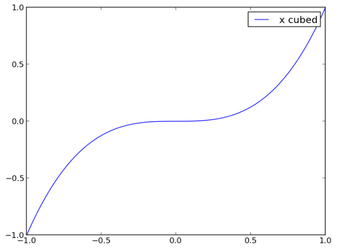
The last piece of labelling that might be required is a legend. This identifies the curve and is typically more use when there are multiple curves on the graph. For completeness, though, we will show it here.

The best approach to creating a legend is to attach text to the lines as they are plotted with `pyplot.plot()` using an optional `label=`… parameter. This does not, in itself, add any text to the plotted graph but rather makes the text available for other routines to use. One of those routines is `pyplot.legend()` which plots a legend box using the label information stored with the lines. This function must be called *after* any lines are plotted.

```
pyplot.line = pyplot.plot(x,y, label='x cubed')
pyplot.legend()
```

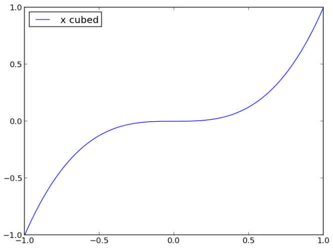
(Script: `example01o.py`, graph: `example01o.png`)

Note that the figure appears in the top right corner by default, obscuring part of our line. We can change its position by adding a named parameter, `loc=`…, to our call to `pyplot.legend()` specifying where the legend should go.



[3] Teχ is a mathematical typesetting system developed by Donald Knuth.

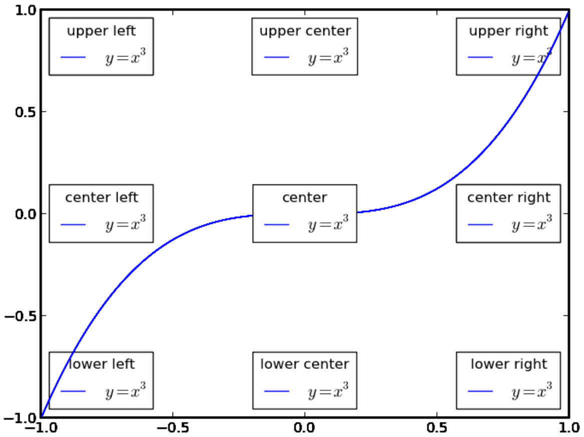
```
pyplot.line = pyplot.plot(x,y, label='x cubed')
pyplot.legend(loc='upper left')
```



(Script: `example01p.py`, graph: `example01p.png`)

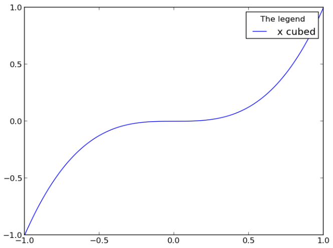
The `loc=`… parameter can takes either a text or an integer value from the following set:

Text	Number	
best	0	
upper right	1	
upper left	2	
lower left	3	
lower right	4	
right	5	
center left	6	= left
center right	7	= right
lower center	8	
upper center	9	
center	10	



We can also attach a title to the legend with the `title=`… parameter in the `pyplot.legend()` function:

```
pyplot.legend(title='The legend')
```



(Script: `example01q.py`, graph: `example01q.png`)

We can use `loc=`… and `title=`… in the same call; they are quite independent.

Setting the line’s properties

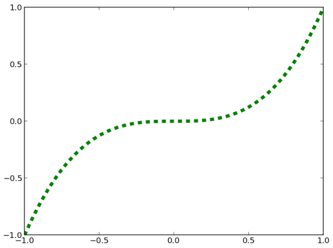
When we start plotting multiple lines we will see lines of various colours. For now, we will change the colour of the single line we are drawing. We also observe that it is a line. We might want to see the specific points, marked with dots or crosses etc.

We set these properties of the line in the same way as we set the text label property of the line: with the `pyplot.plot()` function. This time we see the properties immediately in the line itself, though.

There is an optional formatting parameter which specifies the colour and style of the data plot with a single string. This can get to be a complex little micro-language all of its own. There is an easier, though more long-winded, way to do it using a set of named parameters. These parameters can specify colour, marker style (line, points, crosses, …), and line width.

For example:

```
pyplot.plot(x,y, color='green', linestyle='dashed',
linewidth=5.0)
```



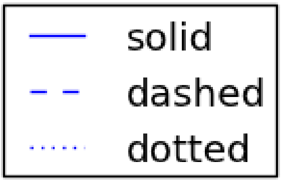
(Script: `example01r.py`, graph: `example01r.png`)

Note the American spelling of “color”. Colour can be specified by name, single character abbreviated name or in hexadecimal RGB format as “`#RRGGBB`”.

The line width is specified in points.

The line style can be identified by a word or a short code. The basic line styles are these:

Name	Code
`solid`	`-`
`dashed`	`--`
`dotted`	`:`
`None`	*(empty string)*



The named colours are these:

Long form	Single letter
`red`	`r`
`green`	`g`
`blue`	`b`
`cyan`	`c`
`magenta`	`m`
`yellow`	`y`
`black`	`k`
`white`	`w`

We don't have to have a line at all. For scatter plots (which we will review below) we want to turn off the line altogether and to mark the (x,y) data points individually.

Setting the graph's file format

Strangely, changing the image's file format is one of the harder things to do. The `matplotlib` system has a number of "back ends" that generate its final output. To output files in a different format we have to select the appropriate back end at the very start of the script (before any graphical work is done) and then use `pyplot.savefig()` to save a file with a corresponding file name suffix ("`.png`", "`.svg`" etc.) at the end.

To pick a back end we have to import the `matplotlib` module itself and use its `matplotlib.use()` function to identify the back end to use. We must do this before importing `matplotlib.pyplot`.

(Script: `example01t.py`):

```
import matplotlib
matplotlib.use('SVG')

import matplotlib.pyplot as pyplot
```

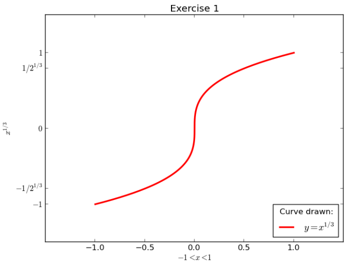
At the top of the script, and

```
pyplot.savefig('example01t.svg')
```

at the bottom.

Exercise 1

Recreate this graph. You may start with the file `exercise01.py` which will take care of including the data for you.
If the graph is printed here in greyscale you can see the target graph in the file `exercise01a.png`. The curve is red and is 2½pt wide.



On the machines used for these courses the three available back ends are

- the default, which saves PNG files,
- an SVG back end, as illustrated above, and
- a PDF back end for generating PDF files, `example1.pdf` etc.

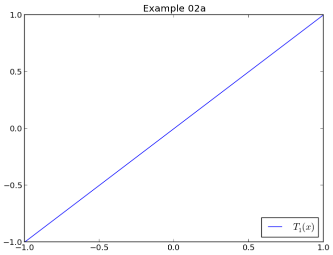
Multiple graphs on a single set of axes

To date we have plotted a single curve in our graphs. Next we are going to plot multiple curves. When we come on to plot different types of graph exactly the same approach will apply to them as to multiple simple curves.

We will start with a single graph. The Python script `example02a.py` plots the first Chebyshev polynomial, which is spectacularly dull. The data for this graph is in `plotdata.data02ya` and we draw the graph with

```
pyplot.plot(plotdata.data02x, plotdata.data02ya,
label='$T_{1}(x)$')
```

(Script: `example02a.py`, graph: `example02a.png`)

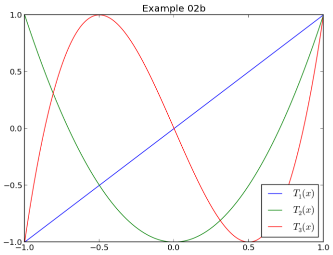


The array `plotdata.data02yb` contains the data for the second Chebyshev polynomial, and `plotdata.data02yc` for the third. The script `example02b.py` simply runs three `pyplot.plot()` functions one after the other:

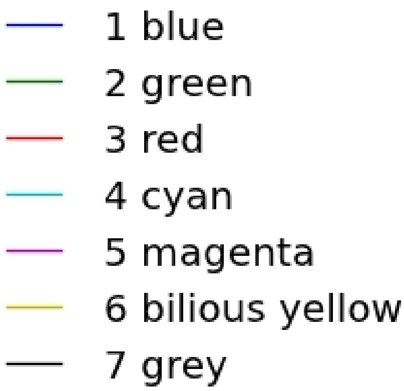
```
pyplot.plot(plotdata.data02x, plotdata.data02ya, label='$T_{1}(x)$')
pyplot.plot(plotdata.data02x, plotdata.data02yb, label='$T_{2}(x)$')
pyplot.plot(plotdata.data02x, plotdata.data02yc, label='$T_{3}(x)$')
```

and gives us the graph we want.

(Script: `example02b.py`, graph: `example02b.png`)

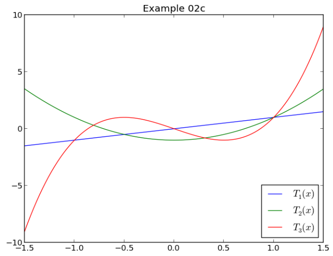


We can adjust the properties of these lines by adding parameters to the `pyplot.plot()` function calls just as we did with individual lines.



The default colour sequence is given here. (Though the lines aren't much use if this handout is printed in in greyscale!) The colours start from blue again if you plot eight or more curves.

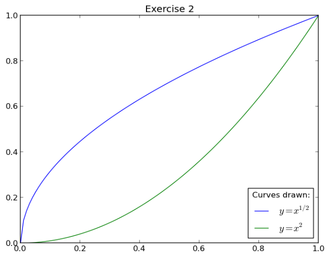
Also note that it doesn't matter if later graphs have different data ranges; the actual calculation of the axes is deferred until the graph is created with `pyplot.savefig()`. The script `example02c.py` extends outside the range *–1<x<1* and the y-axes scale to the largest numbers seen.



(Script: `example02c.py`, graph: `example02c.png`)

Exercise 2

Recreate this graph. You may start with the file `exercise02.py` which will take care of including the data for you.
If the graph is printed here in greyscale you can see the target graph in the file `exercise02a.png`.

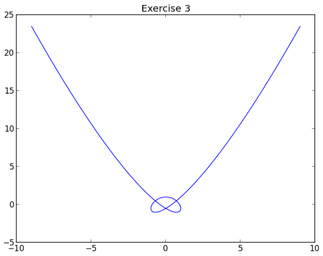


Parametric graphs

To date our x-axes have all been monotonically increasing from the lower bound to the upper bound. There is no obligation for that to be the case, though; the curve simply connects the (x,y) points in order.

Exercise 3

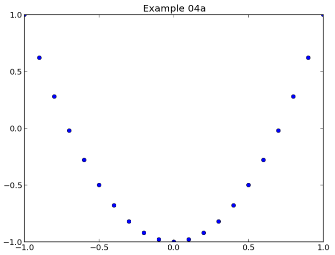
Recreate this graph. Here we plot two of the different “y-value” datasets against each other. You may start with the file `exercise03.py` which will take care of including the data for you. The target graph is in the file `exercise03a.png`.
Please do play with other combinations of Chebyshev polynomials. The data is provided for $T_1(x)$ to $T_7(x)$ in `plotdata.data03ya` to `plotdata.data03yg`.



Scatter graphs

Let’s do away with the curve altogether and plot the pure data points. We will revisit the `pyplot.plot()` function and see some more parameters to turn off the line entirely and specify the point markers. To do this we have to do two things: turn off the line and turn on the markers we will use for the points.

We can do this with the `pyplot.plot()` function:



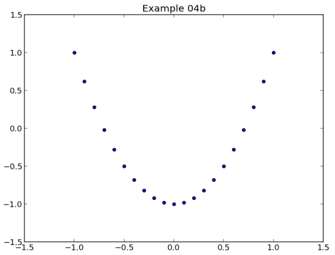
```
x = plotdata.data04x
y = plotdata.data04yb

pyplot.plot(x, y, linestyle='', marker='o')
```

(Script: `example04a.py`, graph: `example04a.png`)

The line is turned off by setting the `linestyle=…` parameter to the empty string.

There is a convenience function, `pyplot.scatter()`, which does this automatically.

```
pyplot.scatter(x, y)
```



(Script: `example04b.py`, graph: `example04b.png`)

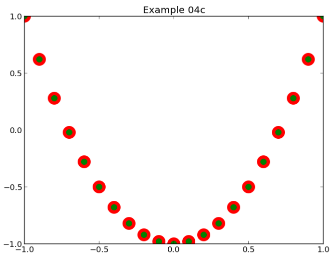
The `pyplot.scatter()` function is not as flexible as `pyplot.plot()`, though, and we will continue to use the more general command.

The most common set of changes wanted with a scatter graph relate to the selection of the marker. The default marker is a filled circle. For any shape of marker we have three or four parameters we might want to change and each of these has a corresponding parameter in the call to `pyplot.plot()`:

- the size of the marker as a whole
  `markersize=…`
- the colour of the area inside the marker (for those shapes that enclose an area)
  `markerfacecolor=…`
- the width of the line that forms the marker’s perimeter
  `markeredgewidth=…`
- the colour of the line that forms the marker’s perimeter
  `markeredgecolor=…`

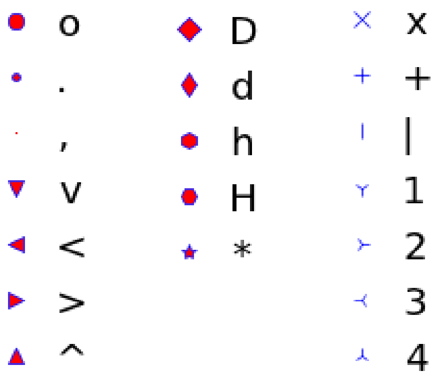
The `markersize=…` and `markeredgewidth=…` parameters take a floating point number which is the dimension in points. The `markerfacecolor=…` and `markeredgecolor=…` parameters take any of the colour names or codes that worked for lines.
The following script uses all four parameters:

```
pyplot.plot(x, y, linestyle='', marker='o',
markersize=15.0, markeredgewidth=5.0,
markerfacecolor='green', markeredgecolor='red')
```



(Script: `example04c.py`, graph: `example04c.png`)

The `matplotlib` system supports a wide variety of markers, including some you can define yourselves. The pre-defined ones are given in the following table:



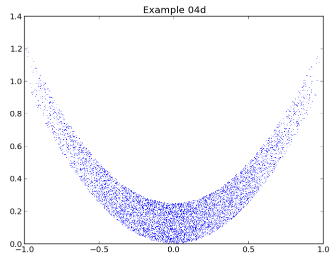
●	o	◆	D	×	x
•	.	♦	d	+	+
·	,	⬢	h	|	|
▼	v	⬣	H	Y	1
◀	<	★	*	≻	2
▶	>			≺	3
▲	^			⅄	4

(File: `markers.png`)

For scatter graphs involving large amounts of data, however, large markers are inappropriate and the graph uses points on the graph to indicate data points. For this the `','` marker — a single pixel — is most appropriate. (Note that this takes the `markerfacecolor` parameter to define its colour.)

The script `example04d.py` draws a scatter graph with 10,000 blue pixels:

```
x = plotdata.data04u
y = plotdata.data04v

pyplot.plot(x, y, linestyle='', marker=',',
markerfacecolor='blue')
```

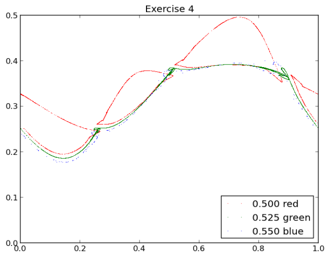


(Script: `example04d.py`, graph: `example04d.png`)

The problem with pixels is that they can be *too* small. Where the transition between `marker=','` and `marker='.'` falls depends on your data.

Exercise 4

The script `exercise04.py` runs some iterations over an initial set of points and then needs to plot the resultant points as a series of scatter graphs on the same axes. The comments in the script explain what you need to write Python for.
You should get a graph like this (`exercise04a.png`)
Feel free to play with other parameter values.



Error bars

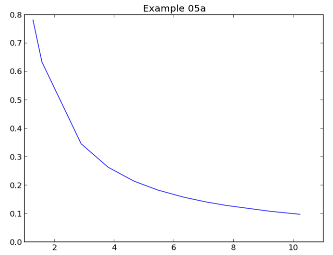
Experimental data points, whether drawn as lines or points typically come with error bars to indicate the uncertainties in the data. These error bars can be symmetric or asymmetric and can be in just the y-coordinate or in both the x- and y-coordinate.

We will start with a graph with asymmetric error bars in the y-coordinate but no error bars for x.

The `pyplot` module contains a function `pyplot.errorbar()` which has the same arguments as `pyplot.plot()` but with some additional ones for plotting the error bars. Because of this it can be used for scatter graphs with error bars by turning off the line (`linestyle=''`). In its simplest form we can use it just as we use `pyplot.plot()`:

```
x = plotdata.data05x
y = plotdata.data05y

pyplot.errorbar(x,y)
```

(Script: `example05a.py`, graph: `example05a.png`)

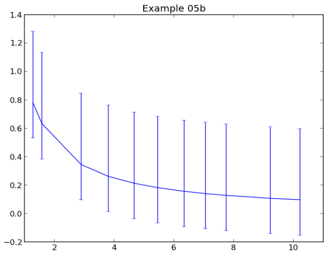


Obviously, we want to add some error bars to this. We can add the error bars with the `yerr=…` parameter. This takes a pair of values, `yerr=(yminus,yplus)`. Each value is a list of errors, as long as the x and y lists. In each case the error value is positive and represents the offset of the error bar from the base curve.

To make it clear, consider the following example where the `yminus` list is `[0.25, 0.25 …, 0.25]` and the `yplus` list is `[0.5, 0.5, …, 0.5]`:

```
half = plotdata.data05half
quarter = plotdata.data05quarter

pyplot.errorbar(x,y, yerr=(quarter,half))
```

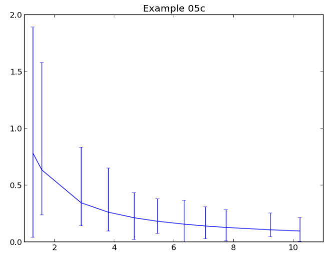
(Script: `example05b.py`, graph: `example05b.png`)



Future examples will use rather more realistic error data:

```
yminus = plotdata.data05ym
yplus  = plotdata.data05yp

pyplot.errorbar(x,y, yerr=(yminus,yplus))
```

(Script: `example05c.py`, graph: `example05c.png`)



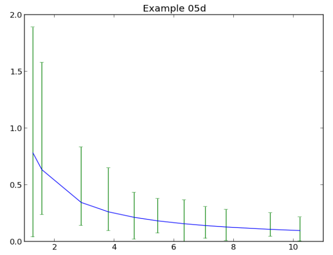
There are a few aspects of these error bars that users typically want to change:

- the colours of the error bars,
- the width of the error bars, and
- the width of the caps on the error bars.

To set the colour of the error bar, `pyplot.errorbar()` supports a parameter `ecolor=…` to override the error bars having the same colour as the main curve:

```
pyplot.errorbar(x,y, yerr=(yminus,yplus),
ecolor='green')
```

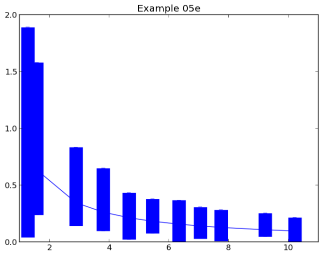
(Script: `example05d.py`, graph: `example05d.png`)



The (blue) data curve is drawn in front of the (green) error bars. There is no parameter to make the upper and lower error bars different colours.

The `elinewidth=…` parameter sets the width of the error bar line and is measured in points and given as a floating point number. If the parameter is not given then the same width is used for the error bar as for the main curve.

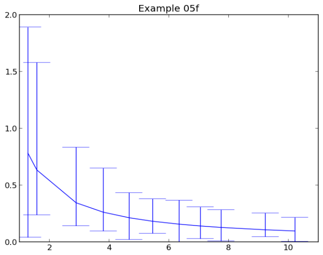
```
pyplot.errorbar(x,y, yerr=(yminus,yplus),
elinewidth=20.0)
```



(Script: `example05e.py`, graph: `example05e.png`)

Note that the caps on the error bars are not widened proportionately to the width of the error bar itself. That width must be separately defined and the `pyplot.errorbar()` function supports the `capsize=…` parameter to do that. This specifies the length of the cap line in points; the width of the cap line cannot be set. Setting `capsize=0` eliminates the cap altogether.

```
pyplot.errorbar(x,y, yerr=(yminus,yplus), capsize=20.0)
```

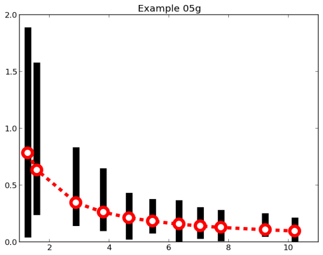


(Script: `example05f.py`, graph: `example05f.png`)

The cap line is always drawn in the same colour as the error bar line itself.

As a (rather gaudy) example of combining the main `pyplot.plot()` parameters with the error bar-specific parameters, consider the following example that draws a 5pt red dashed data line, red and blue data point markers, and 10pt error bars without caps in black:

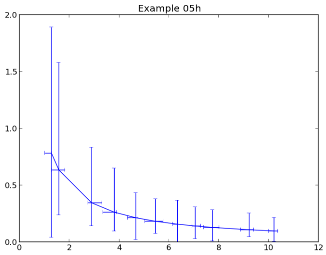
```
pyplot.errorbar(x,y, yerr=(yminus,yplus),
linestyle='dashed', color='red', linewidth=5.0,
marker='o', markeredgecolor='red', markeredgewidth=5,
markerfacecolor='white', markersize=15,
ecolor='black', elinewidth=10.0, capsize=0)
```



(Script: `example05g.py`, graph: `example05g.png`)

We often want error bars in the x-direction as well as the y-direction. The `pyplot.errorbar()` function has an `xerr=…` parameter which behaves in exactly the same way as `yerr=…`:

```
xminus = plotdata.data05xm
xplus  = plotdata.data05xp

pyplot.errorbar(x,y, xerr=(xminus,xplus),
yerr=(yminus,yplus))
```



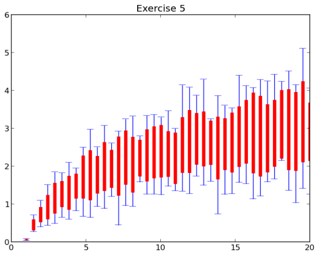
(Script: `example05h.py`, graph: `example05h.png`)

The line can be suppressed with `linestyle=''` as usual.

The settings for the error bar lines (colour, width, cap size, etc.) must be the same in both directions.

Exercise 5

The script `exercise05.py` sets up some data for a graph like this with “inner” and “outer” error ranges (e.g. for quartiles and extrema).
Write the `pyplot.errorbar()` lines to generate the graph shown.



Polar graphs

Not all graphs want orthogonal axes, *x* and *y*. Cyclical data is typically drawn on polar axes, *θ* and *r*. The `pyplot` module provides a function `pyplot.polar(`*θ*`,`*r*`,`…`)` which is takes exactly the same arguments as `pyplot.plot(`*x*`,`*y*`,`…`)` but draws a polar graph. The *θ* list of values must be in radians.

! Some of the polar plotting functions misbehave when the values of *r* are negative. It is better to avoid negative values altogether.

While we can get away without it in certain circumstances, it is a good idea to let `pyplot` know that we are going to be working with polar axes rather than orthogonal ones as early in the process as possible. We do this with a function call

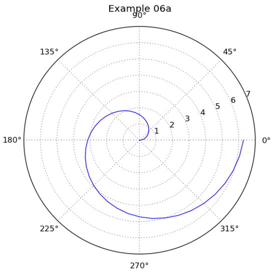
```
pyplot.axes(polar=True)
```

The default is `False` and is what we have been using for our previous graphs.

The Python script `example06a.py` plots a graph of *r=t* for *0<θ<π*. The laying down of the line is done with this `pyplot.polar()` command:

```
t = plotdata.data06t
r = plotdata.data06r

pyplot.polar(t,r)
```

(Script: `example06a.py`, graph: `example06a.png`.)



As with the orthogonal axis graphs, there are a number of properties we might want to change:

- the line parameters (colour, style, points rather than lines),
- the angular position of the radial labels (which defaults to 22½° above the horizontal 0° line),
- the radial labels (1, 2, …, 7),
- the angular labels (which default to every 45° and are labelled in degrees),
- the dotted polar grid.

We will work through these as we did for the orthogonal graphs.

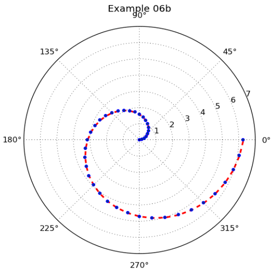
Line parameters

The properties of the data line can be set with exactly the same parameters as are used for `pyplot.plot()`:

`linestyle`	Style of the line. Use `''` for no line.
`linewidth`	Width of the line in points.
`color`	Colour of the line. n.b. USA spelling.
`marker`	Selection of the data point marker.
`markersize`	Size of the data point marker in points.
`markeredgewidth`	Width of the line drawing the marker.
`markeredgecolor`	Colour of the line drawing the marker.
`markerfacecolor`	Colour of the area within the marker if any.

So, if we go overboard for illustrative purposes we could define a plot like this:

```
pyplot.polar(t,r, linestyle='dashed', linewidth=2.5,
color='red', marker='.', markersize=10.0,
markeredgecolor='blue', markeredgewidth=1.5,
markerfacecolor='green')
```



(Script: `example06b.py`, graph: `example06b.png`.)

The angular grid lines and their labels

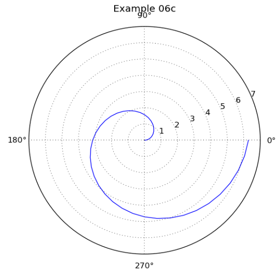
The `pyplot.thetagrids()` function controls the position and labels of the lines running from the centre to the edge of the graph. This allows us to set the necessary properties:

- what angles the lines should be at,
- what their labels should be,
- how far out their labels should be.

We specify the angular position of the radial lines with the function's compulsory argument of a list of values as its argument.

```
pyplot.thetagrids([0.0, 90.0, 180.0, 270.0])
```

(Script: `example06c.py`, graph: `example06c.png`.)



!

The angles are specified in *degrees*.
Yes, this is inconsistent with `pyplot.polar()`!

There are two ways to specify the labels appearing at the ends of the lines. If neither is used then you get the degree values as seen in the graphs above.

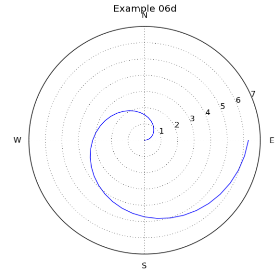
You can specify a `labels=`… parameter just as we did with the tick marks on orthogonal graphs. This takes a list of text objects and must have the same length as the list of angles.

For example "`labels=['E', 'N', 'W', 'S']`".

So we could label the graph in our running example as follows:

```
pyplot.thetagrids([0, 90, 180, 270],
labels=['E', 'N', 'W', 'S'])
```

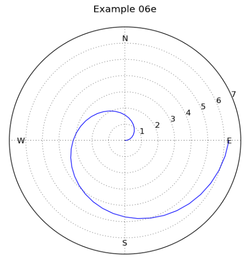
(Script: `example06d.py`, graph: `example06d.png`.)



Finally we can change the position of the labels. We specify their position relative to the position of the outer ring with a parameter `frac=`…. This gives their radial distance as a fraction of the outer ring's radius and defaults to a value of 1.1. A value less than 1.0 puts the labels inside the circle:

```
pyplot.thetagrids([0, 90, 180, 270],
labels=['E', 'N', 'W', 'S'], frac=0.9)
```

(Script: `example06e.py`, graph: `example06e.png`.)



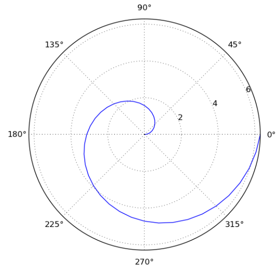
The concentric grid lines and their labels

The `pyplot.rgrids()` function is the equivalent of `pyplot.thetagrids()` for the concentric circles (labelled 1 to 7 in our graphs so far).

In its simplest use it specifies the radii of the circles as a list of values:

```
pyplot.rgrids([2.0, 4.0, 6.0, 8.0])
```

(Script: `example06f.py`, graph: `example06f.png`.)



!

Note that we don't get a ring at *r=8.0*!

The polar graph only extends out as far as the data.

Yes, this is a bug.

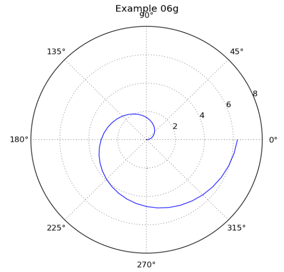
We can explicitly set the outer limit of the rings, but we have to expose the inner workings of `pyplot` to do it. Internally, `pyplot` treats *θ* as an *x*-coordinate and *r* as a *y*-coordinate. We can explicitly set the upper value of *r* with this line:

```
pyplot.ylim(ymax=8.0)
```

So the following lines give us the graph we want:

```
pyplot.ylim(ymax=8.0)

pyplot.rgrids([2.0, 4.0, 6.0, 8.0])
```

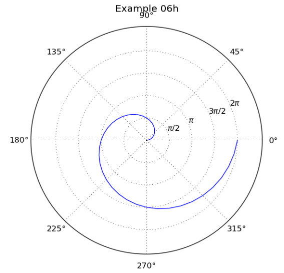
(Script: `example06g.py`, graph: `example06g.png`.)



We can set our own labels for these values in the same way as for the radial grid:

```
pyplot.ylim(ymax=8.0)

pyplot.rgrids([1.57, 3.14, 4.71, 6.28], labels=['π/2',
'π', '3π/2', '2π'])
```

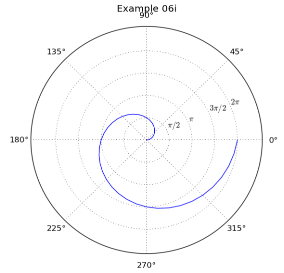
(Script: `example06h.py`, graph: `example06h.png`.)



For those of you not comfortable with direct use of non-keyboard characters you can get the more mathematical version with the $…$ notation:

```
pyplot.ylim(ymax=8.0)

pyplot.rgrids([1.57, 3.14, 4.71, 6.28],
labels=['$\pi/2$', '$\pi$', '$3\pi/2$', '$2\pi$'])
```

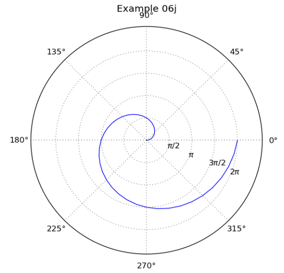
(Script: `example06i.py`, graph: `example06i.png`.)



There is also a control over the angle at which the line of labels is drawn. By default it is 22½° above the *θ=0* line, but this can be adjusted with the `angle=…` parameter:

```
pyplot.ylim(ymax=8.0)

pyplot.rgrids([1.57, 3.14, 4.71, 6.28],
labels=['π/2', 'π', '3π/2', '2π'], angle=-22.5)
```

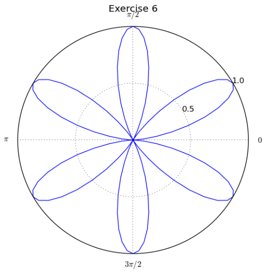
(Script: `example06j.py`, graph: `example06j.png`.)



Angle takes its value in degrees.

Exercise 6

The script `exercise06.py` sets up some data for this graph. Write the `pyplot` lines to generate the graph shown.
(The curve is $r=sin^2(3θ)$.)



Histograms

For the purposes of this tutorial we are going to be pedantic about the *different* definitions of histograms and bar charts.

A **histogram** counts how many data items fit in ranges known as “bins”.

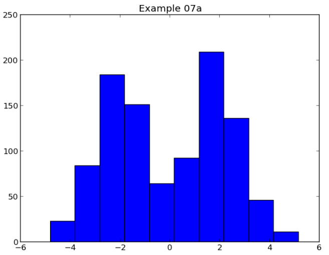
A **bar chart** is simply a set of y-values marked with vertical bars rather than by points or a curve.

The basic `pyplot` function for histograms is `pyplot.hist()` and it works with raw data. You do not give it fifty values (say) and ask for those heights to be plotted as a histogram. Instead you give it a list of values and tell it how many bins you want the histogram to have.

The simplest use of the function is this:

```
data = plotdata.data07

pyplot.hist(data)
```

(Script: `example07a.py`, graph: `example07a.png`.)

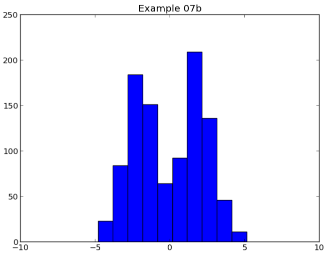


Note that the defaults provide for ten bins, and blue boxes with black borders. We will adjust these shortly.

We can adjust the limits of the axes in the usual way:

```
pyplot.axis([-10.0, 10.0, 0.0, 250])
pyplot.hist(data)
```

(Script: `example07b.py`, graph: `example07b.png`.)



Cosmetic changes

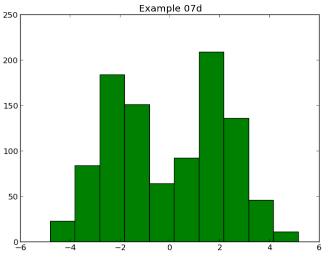
We will start with the cosmetic appearance of the histogram.

We can set the colour of the boxes with the `color=`… parameter:

```
pyplot.hist(data, color='green')
```

(Script: `example07d.py`, graph: `example07d.png`.)

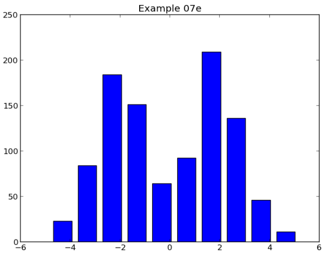
(There is no `example07c.py`; you haven't skipped it by mistake.)



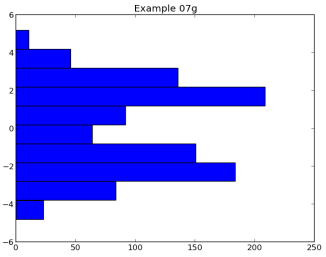
We can set the width of the bars with the `rwidth=`… parameter. This sets the width of the bar relative to the width of the bin. So it takes a value between 0 and 1. A value of 0.75 would give bars that spanned three quarters of the width of a bin:

```
pyplot.hist(data, rwidth=0.75)
```

(Script: `example07e.py`, graph: `example07e.png`.)



We can also have the bars run horizontally.



The `pyplot.hist()` function has an `orientation=`… parameter. There are two accepted vales: `'vertical'` and `'horizontal'`.

```
pyplot.hist(data, orientation='horizontal')
```

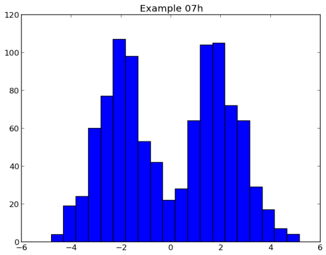
(Script: `example07g.py`, graph: `example07g.png`.)

Setting the bins

The number of bins in the histogram is set with the `bins=`… parameter. In its simplest form it can be an integer, specifying the number of bins to use:

```
pyplot.hist(data, bins=20)
```

(Script: `example07h.py`, graph: `example07h.png`.)

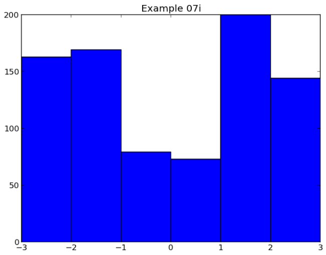


Alternatively we can specify exactly what the bins should be. To do this we specify the boundaries of the bins, so for *N* bins we need to specify *N+1* boundaries:

```
pyplot.hist(data,
    bins=[-3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0])
```

(Script: `example07i.py`, graph: `example07i.png`.)

Note that any data that lies outside the boundaries is ignored.

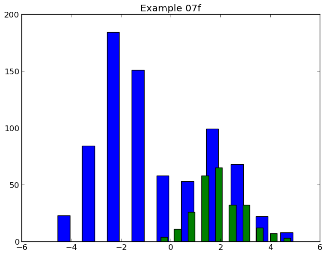


Plotting multiple data sets

We can plot multiple histograms with repeated calls to `pyplot.hist()` but a measure of care is called for. Each call calculates its own bin sizes so if the sizes of the datasets are too different one set of bars can be noticeably wider than the others:

```
pyplot.hist(data[:750], rwidth=0.5)
pyplot.hist(data[750:], rwidth=0.5)
```

(Script: `example07f.py`, graph: `example07f.png`.)



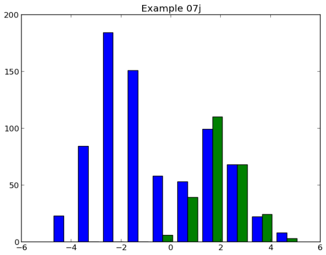
At a minimum you should set the bins explicitly. There is, however, a better way to do it.

Because repeated calls to `pyplot.hist()` can prove so problematic the function can take a list of datasets (not necessarily all of the same size) and plot them together:

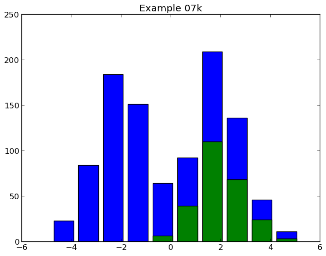
```
pyplot.hist([data[:750], data[750:]])
```

(Script: `example07j.py`, graph: `example07j.png`.)

If it is plotting multiple histograms the bin sizes are based on the union of all the datasets. The widths of the blocks are automatically scaled down to fit the bars alongside each other with a thin gap between them.



The `pyplot.hist()` function provides a way to provide multiple datasets as "stacked" histograms. This is enabled with the `histtype=`… parameter which specifies what type of histogram we want. To date, all our histograms have used the default `histtype='bar'` setting. If we change this to `histtype='barstacked'` we get the second dataset above the first:



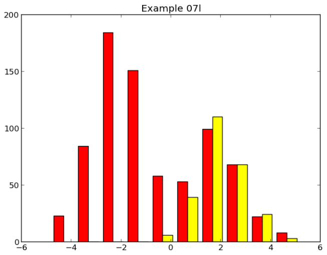
```
pyplot.hist([data[:750], data[750:]], histtype='barstacked')
```

(Script: `example07k.py`, graph: `example07k.png`.)

If you use a list of datasets then you can use a list of colours with `color=…` too:

```
pyplot.hist([data[:750], data[750:]],
color=['red', 'yellow'])
```

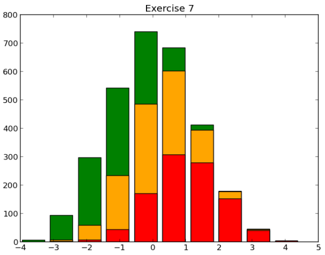
(Script: `example07l.py`, graph: `example07l.png`.)



Exercise 7

The script `exercise07.py` sets up some data for this graph. Write the `pyplot` lines to generate the histogram shown.

(Green is `data1`, yellow is `data2`, and red is `data3`.)



Bar charts

Recall that we are being pedantic about the *different* definitions of histograms and bar charts.

A **histogram** counts how many data items fit in ranges known as “bins”.

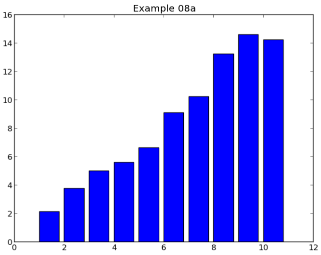
A **bar chart** is simply a set of uniformly spaced y-values marked with vertical bars rather than by points or a curve.

The basic `pyplot` function for drawing bar charts is `pyplot.bar()`.

In its simplest use it takes a series of x- and y-values, just like `pyplot.plot()`:

```
x = plotdata.data08x
y = plotdata.data08y

pyplot.bar(x,y)
```

(Script: `example08a.py`, graph: `example08a.png`.)

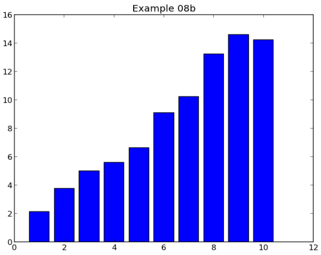


Technically, the x-values are the values at the left edge of the bar. In the example shown the x-values run from 1 to 10 inclusive. The left-most bar *starts* at x=1 and the right-most one *starts* at x=10.

We can change this with the `align=…` parameter. This can take two values: `align='edge'`, which is the default, or `align='center'` which centres the bar on the x-value:

```
pyplot.bar(x,y, align='center')
```

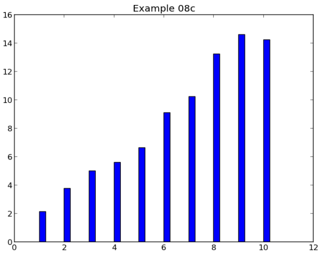
(Script: `example08b.py`, graph: `example08b.png`.)



The bars default to having a width of 0.8. The width can be changed with the parameter `width=…`:

```
pyplot.bar(x,y, width=0.25)
```

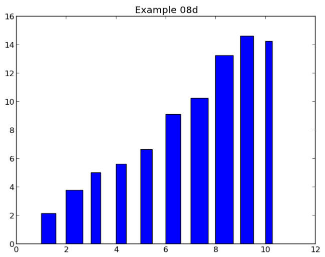
(Script: `example08c.py`, graph: `example08c.png`.)



We can also set the widths of each bar independently by passing a list of values. This list must be the same length as the x- and y-value lists:

```
wlist = plotdata.data08w

pyplot.bar(x,y, width=wlist)
```

(Script: `example08d.py`, graph: `example08d.png`.)



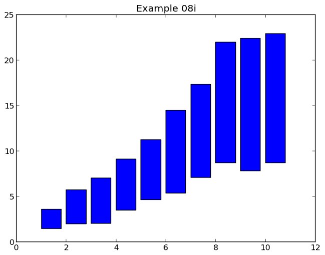
Indicating ranges with bar charts

As well as varying with widths of the bars we can also vary their bases, or “bottoms” as `pyplot` calls them. This can be very useful for showing ranges of values and could be used, for example, to indicate ranges of values either size of a line or scatter graph drawn with `pyplot.plot()`.

For example, if we have arrays of upper and lower values we can do this:

```
pyplot.bar(x,upper, bottom=lower)
```

(Script: `example08i.py`, graph: `example08i.png`.)

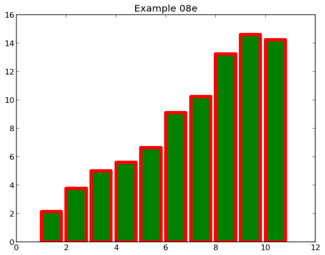


Cosmetic changes to bar charts

We can set the colour of the bars with `color=`…, taking the usual named values. We can set the colour of the lines around the bars and the widths of those lines with `edgecolor=`… and `linewidth=`… respectively:

```
pyplot.bar(x,y, color='green', edgecolor='red',
linewidth=5.0)
```

(Script: `example08e.py`, graph: `example08e.png`.)



Plotting multiple datasets on a single graph requires some finesse. Multiple calls to `pyplot.bar()` simply superimposes the graphs. While this might work for line graphs or scatter graphs it is no use for bar graphs.

Instead, we call `pyplot.bar()` multiple times and use the `width=`… parameter to narrow the bars. Then we explicitly offset the x-value for each dataset so that the bars do not overlap.

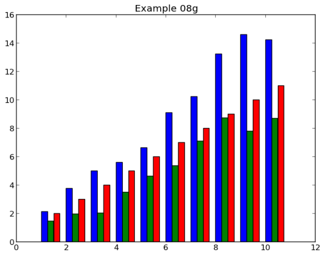
! People using plain Python lists have to generate new lists of offset x-values for this purpose. People using NumPy arrays have a short cut. We will do each in turn. Readers unfamiliar with NumPy arrays can simply skip that bit with no impact on their ability to create bar charts.

We will plot three datasets in parallel, with each bar being ¼ width, leaving a thin gap between sets. Both scripts below produce the same graph:

Multiple bar charts using Python lists

Using Python lists we have to create new x-value lists. The simplest way to do this is with the bit of syntactic sugar called a “list comprehension” that we met earlier when we set the tick marks.

```
x1 = plotdata.data08x
x2 = [ x + 0.25 for x in x1 ]
x3 = [ x + 0.50 for x in x1 ]

y1 = plotdata.data08y1
y2 = plotdata.data08y2
y3 = plotdata.data08y3
```



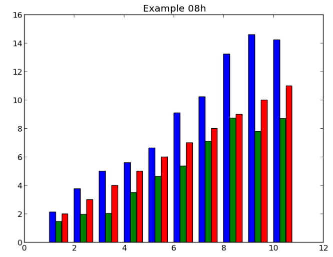
```
pyplot.bar(x1,y1, width=0.25, color='blue')
pyplot.bar(x2,y2, width=0.25, color='green')
pyplot.bar(x3,y3, width=0.25, color='red')
```

(Script: `example08g.py`, graph: `example08g.png`.)

Multiple bar charts using NumPy arrays

If we use NumPy arrays then we can create the offset versions with a simpler expression. Everything else is the same:

```
import numpy

x1 = numpy.array(plotdata.data08x)
x2 = x1 + 0.25
x3 = x3 + 0.50

y1 = plotdata.data08y1
y2 = plotdata.data08y2
y3 = plotdata.data08y3

pyplot.bar(x1,y1, width=0.25, color='blue')
pyplot.bar(x2,y2, width=0.25, color='green')
pyplot.bar(x3,y3, width=0.25, color='red')
```

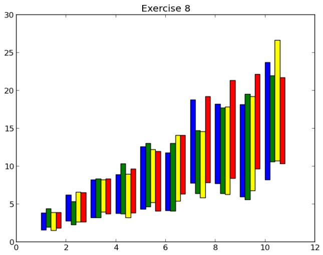


(Script: `example08h.py`, graph: `example08h.png`.)

Of course, if we were using NumPy arrays we would usually have NumPy arrays for the y-values too.

Exercise 8

The script `exercise08.py` sets up some data for this graph. Write the `pyplot` lines to generate the bar chart shown.


Exercise 8