Diagra Documentation

February 2010

ReportLab Europe Ltd
Thornton House, Thornton Road
Wimbledon
London SW19 4NG, UK

CONTENTS

1 Overview

1.1 About This Document

1.2 What is Diagra?

1.3 Other Documentation

1.4 Limitations and Bugs

2 Installation

3 Using the Drawing Editor to Create Static Charts

3.1 Introduction to the Drawing Editor

3.2 Creating a New Project

3.3 Adding Widgets

3.4 Setting Widget Attributes

3.5 Saving Your Project

3.6 Exporting to Different Formats

3.7 Opening Existing Files

3.8 Adding Primitives - and another way to add widgets

3.9 Miscellaneous Features

4 Working with Charts

4.1 BongoCorps's Drawing Classes

4.2 Working with Colours

4.3 Working with Fonts

4.4 Working with Collections

4.5 Formats and Formatters: DecimalFormatter, NA_Label and Format Strings

5 Creating Data Aware Charts

5.1 Introduction to DataAwareDrawings: Plotmode, fileNamePattern and outDir

5.2 Setting the Data Source

5.3 Setting the Data Associations

5.4 A practical example: Slidebox.py

5.5 Running the resulting scripts

5.6 Example 2: Dotbox.py

5.7 Example 3: Horizontalbarchart.py

5.8 Example 4: Verticalbarchart.py

5.9 Example 5: Linechart.py

5.10 Example 6: SectorCylinderChart

6 More about Using Axes

6.1 ValueAxis

6.2 BongoYValueAxis

7 More on Data Sources

7.1 Making a Simple Chart Data Aware

7.2 Changing the Data Associations on an Existing Chart

7.3 Data Association Types: scalar, vector, matrix, tmatrix and rowmap

8 New Drawing Editor Features

8.1 Test Mode

8.2 Errors

8.3 Command Line Arguments

8.4 Changing Environments

8.5 The Diagnostics

9 Using 'Knockout'

9.1 'Knockout' - what it is, and why use it

1 Overview

1.1 About this document

This document started out as specialised documentation for a large customer's in-house systems which used Diagra and the Diagra Drawing Editor. It is in the middle of being sanitised and made more generic to become the Diagra documentation. This is a work in progress.

As such, it will be an overview of using the Drawing Editor with some very specific examples. It will cover the Drawing Editor (guiedit) and some other software - how to install it and how to use it to create charts. It is *not* intended as a complete set of documentation for every chart and every method for each chart. See the section on 'Other Documentation' (below) for info on how to get that.

1.2 What is EPSCHARTS?

EPSCHARTS was the original name for what has now evolved into Diagra and the Diagra chart server. You may still see the name in some code and in a few other places in this manual. This is a hangover from previous versions - it is not a separate product.

1.3 What is Diagra?

Diagra is a process for batch-generating charts - originally in Encapsulated Postscript but now in many other formats as well. This has evolved through a number of iterations.

The current iteration is hopefully the right long-term workflow and goes like this:

- The *reportlab/graphics* package contains code to let programmers generate standard chart types in several formats
- ReportLab has also written some special chart types for various customers with extra features
- Programmers define some base "Drawing" classes including these charts.
- Designers use a GUI tool (the Diagra Drawing Editor) to create several variants of these base classes. These specify chart formatting attributes (size, color, label style etc.) and also data-processing attributes (input data file name, output directory, file format (not just EPS). They save these as special python modules.
- When you run one of these modules (e.g. with a double-click), it looks for a data source (CSV file or ODBC database), reads it, and generates a batch of charts. These will look like the prototype, except that the data for each chart is of course different.

1.4 Other Documentation

xxxxxxxx.epscharts.pdf and updatedoc.py

The most up-to-date and complete documentation on the various charts on your system will be the documentation that is autogenerated.

There is a program called updatedoc.py (which should be in your xxxxxxxx/epscharts directory). When you run this, it will run over your epscharts directory and create a document called 'xxxxxxxx.epscharts.pdf' in your documentation directory.

This document contains all the classes (i.e. charts) that are available to you on your machine, along with a brief description of what it does, the attributes you can set for it ('Public Attributes') and an example of what one looks like.

Other documents

Both the core ReportLab toolkit and the ReportLab graphics package come with their own documentation. That for the core toolkit is called 'userguide.pdf', and that for graphics is called 'graphguide.pdf'. Both these files should be in the reportlab\docs directory under your Python directory.

1.5 Limitations and Bugs

At the time of writing we hope we now have an overall process which makes sense. However, each chart has dozens of different attributes which need debugging. We therefore expect a period of rapid change while this stabilizes and before all of the attributes work.

Unfortunately, some of these changes (especially if we completely delete some attribute) will means 'starting again' with existing drawing files.

Occasionally, the drawing editor may get your settings confused (e.g. forcing you to have a full screen for your editor screen). If this is the case, find the file called ***_guiedit.ini*** (which should be in the same folder as the drawing editor program itself - guiedit.py). Delete it. This will be regenerated the next time you start up the Drawing Editor, and should clear your problem.

2 Installation

Please follow http://www.reportlab.com/software/installation/ for the most up-to-date instructions on installing our libraries along with their dependencies.

- **Optional step (for Windows only): Creating a shortcut to the Drawing Editor**

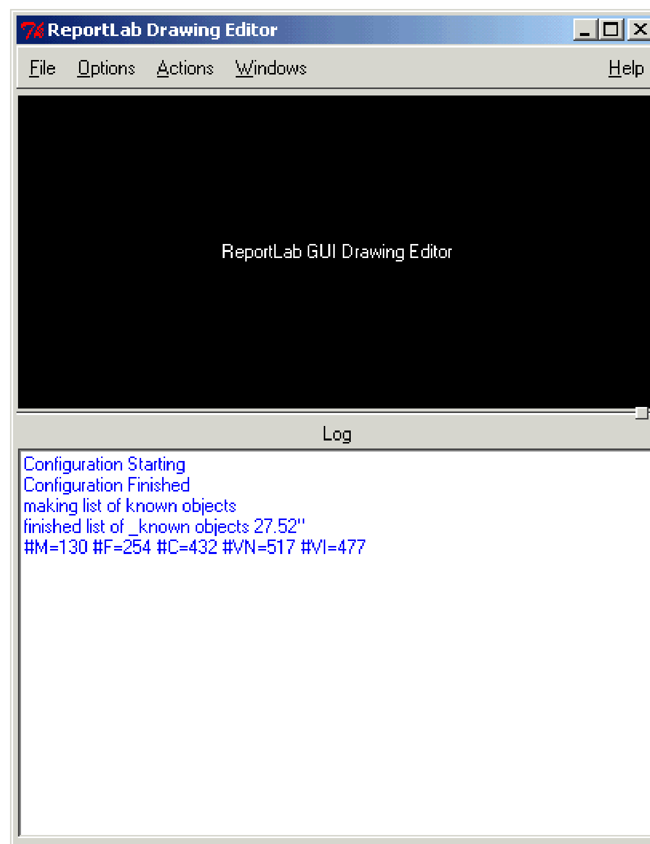
  If you will be using the Drawing Editor on a regular basis, you should put a shortcut to it on your desktop to make life easier. To do this, right click on your desktop and 'New' then 'Shortcut'. Then copy this line c:\Python26\pythonw.exe c:\python26\Lib\site-packages\rlextra\graphics\guiedit\guiedit.py and paste it into the location text box that appears in the dialogue, and save.

  Double clicking on this icon will now have the same effect as double-clicking on the original: it will start up the Drawing Editor, but without having to navigate to it in Windows Explorer. And the change to the target in the Properties box means that it doesn't open an extra shell window when it runs.

3 Using the Drawing Editor to Create Static Charts

3.1 Introduction to the Diagra Drawing Editor

The Diagra Drawing Editor (guiedit.py) is the graphical editor for EPS charts. If you look in your 'rlextra' directory (under Python22), you should find a 'graphics' directory which should in turn have a 'guiedit' subdirectory. Inside that is the GUI Editor program - 'guiedit.py'. Double-clicking on that should give you something like this:



From the top, the elements of this display are:

- *Title bar*
  Just like any other program, this is where the name of the program is displayed, and which contains the Minimize, Maximize and Close buttons.
- *Menu bar*
  This is the area which contains drop-down menus - click on 'File', 'Options', 'Actions', 'Windows' or 'Help' to see the associated menu. Also, notice how if you leave the cursor over one of the menu titles, a floating box appears with some text telling you what it is (you might know these as 'tooltips' if you are a Windows user, or 'balloon help' if you are a Mac user). Most buttons and fields in the Drawing Editor have these. If you find them annoying, you can turn them off by using the 'Tooltips off'/'Tooltips on' item in the Options menu.
- *Main Window*
  The large black area which currently has the words 'ReportLab GUI Drawing Editor' in it is the 'Main Window'. This is the area which will contain the graphical displays of your charts.
- *Log Window*
  The Drawing Editor keeps a log of its actions. These are displayed in their own window (the Log Window). This window can be turned off if you don't find them useful or to make more room on screen. You can do

this by going to the 'Windows' menu and selecting 'src window on', which will then toggle to off (and change colour from green to red).

Some other points to notice before we move on to doing something useful using the Drawing Editor. The frame the contains all these other windows is resizable. If you move the cursor to the left or right hand edges of the Drawing Editor window, it will change into an horizontal arrow with arrowheads at either end. When the cursor looks like this, you can click-and-drag to move the edges out horizontally. Similarly, when you place the cursor over the top or bottom edges of the window, it changes to a vertical arrow with arrowheads at the top and bottom. This means that you can click-and-drag to move the edges vertically. And when you place it over the bottom left hand corner, it changes to a diagonal arrow - you can then move it horizontally and vertically at the same time to resize it in both directions.

There is a thin line over the divider between the main window and the log window (just over where it says 'Log'). If you move the cursor over it, you see it changes to the vertical arrowheads. You can click and drag to shrink or grow this window vertically. If you look at the line, it has a small square box or button almost at the right hand edge. If you notice a box like this on any other divider in the Drawing Editor, then you can click-and-drag it to resize the windows it borders.

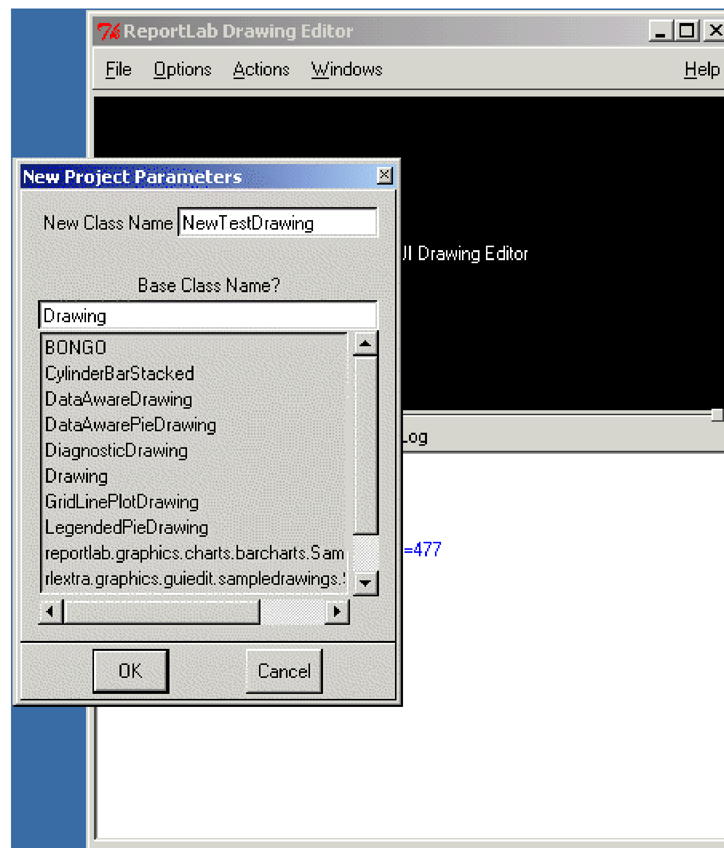
From the Options menu, select the item 'Searched Packages'. You should get a dialogue that looks like this:

This lists the packages that the Drawing Editor searches to find charts. You may have to add a package (i.e. a directory which contains the reusable Python code to create the charts, as well as a file called __init__.py) if you cannot find the charts that you need (but you know they exist on your hard disk). 'Package Exclusions' from the same menu provides you with a similar dialogue which does the same thing for the excluded packages - those that are deliberately ignored when the packages are being imported.

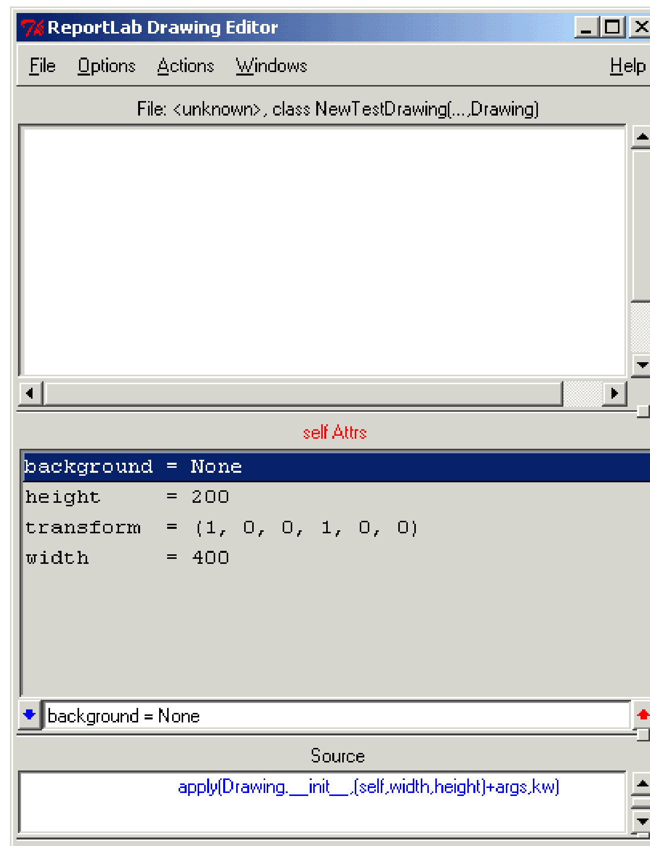
As well as the immediate function of this pop-up, it's worth noting the two small coloured arrows at the left and right. Whenever you see arrows like this in a dialogue in the Drawing Editor, you will know that they have to do with navigation. In this case they point up and down - if you have clicked to highlight a package, these up and down arrows move you up and down the list. As we will see later, they can also be used for navigating the attributes in a list as well as just moving up and down the list itself.

3.2 Creating Static Charts: Creating a New Project

The 'New' item from the File menu allows you to create a new Project. This will pop-up a dialogue which prompts you for the name of the project, and allows you to chose the name of the class from which it will be inheriting.



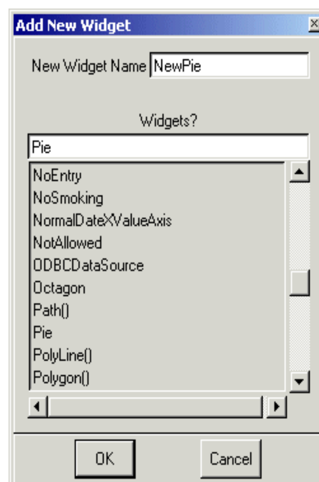
As you can see from this illustration, we'll be using the class called 'Drawing' as the basis for our new static chart. The illustration below shows you what we get when we click on the OK button to open it up. This doesn't look very exciting, since the Drawing class just gives us a blank canvas to work on. All of our static drawings will inherit (directly or indirectly) from this class. You should also notice some more windows appearing - more on those in a few paragraphs.



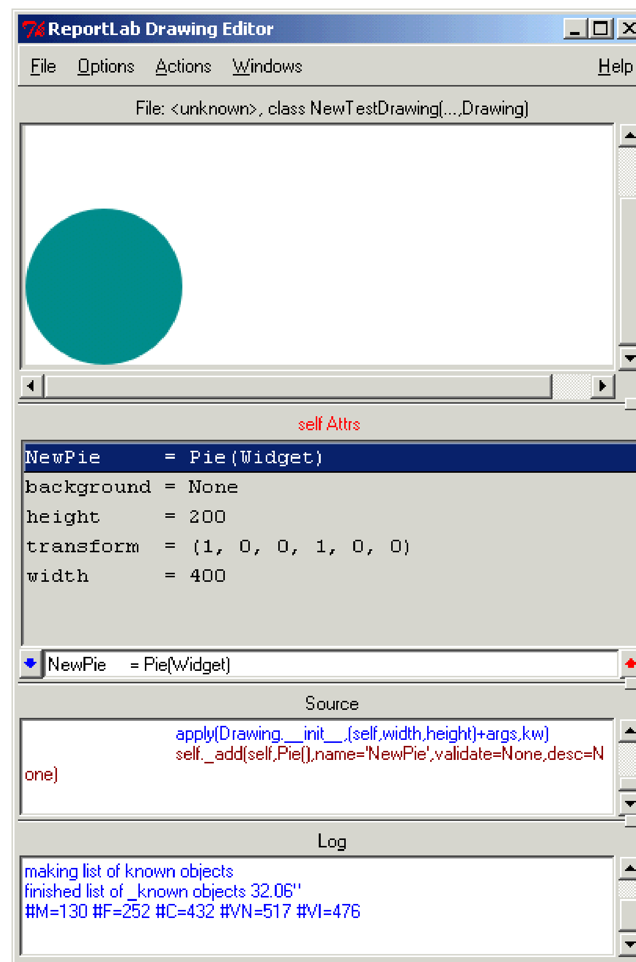
3.3 Adding Widgets

We can add contents to this drawing. In the 'Actions' menu there is a menu item called 'Add Widget'. A widget is a reusable shape that can be 'drawn'. Widgets can be as simple or as complex as you want. You can create your own widgets, but in most cases you will be modifying existing, pre-created widgets.

If you go 'Actions' and select 'Add Widget', you should get a dialog appearing that looks similar to the one below. For the purposes of this example, scroll down it until you see 'Pie' and click on it. You should see 'Pie' appear in the box labelled 'Widgets?'. Then give it a name - enter 'NewPie' in the box labelled 'New Widget Name'.



As you can see (below), as well as the main and log windows we already had when we started the Drawing Editor, we now have three new windows: the 'Attributes' window (labelled 'self Attrs'), the entry field and the 'Source' window. If you can't see any of these windows, click on the dividers to resize things until you do (e.g., you may have to grab and drag the bottom edge to see the Log window).



The *attributes window* shows a list of attributes that this class has. The 'self' in 'self Attrs' refers to the fact that these attributes are for the base class (rather than for one of its attributes). Some of these attributes themselves have attributes that you can change. For example, double click on the line that says 'NewPie = Pie(Widget)'. The 'self Attrs' changes to 'self.pieChart Attrs' to show that you are now changing the attributes for the pie chart rather than for the class as a whole, and the list of attributes displayed changes to those for the pie chart (NewPie).

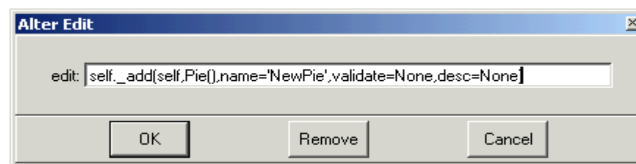
If you have finished with the attributes for NewPie, you can go back to the attributes for the whole chart by clicking on the red up-arrow on the right hand side of the entry field. The blue down-arrow on the left takes you down a level ('drilling down') if that attribute has attributes that you can change. Double-clicking on a class with attributes does the same as the blue arrow, but double clicking on the red text with the class name ('self Attrs') does the same as the red up-arrow (i.e. takes you up a level).

The *entry field* shows the attribute you have currently selected from the Attribute window. You can make changes in this one-line window, and these will be reflected in the Attributes window as soon as you hit enter. These changes also show up in a more graphical way in the main window, so you can see exactly what those changes do to the look of a chart.

The *source window* shows the Python source code. This may be especially useful to you if you are a programmer, but is also very useful if you're not. Looking in the source window, you should see a line that looks like this:

```
self._add(self,Pie(),name='NewPie',validate=None,desc=None)
```

This is the line the actually adds the widget for the Pie chart. If you right click on it, you should see another dialog pop up:



This alter edit dialog is very handy. It allows you to alter the text (to do things such as changing the name of a widget - if you decided to call it PieChart rather than NewPie), and to totally remove an object that you have added by mistake. If you do want to close the source window, go to the Windows menu and select 'Src window on' - this will toggle it to off. You can always toggle it back on later.

3.4 Setting Widget Attributes

By default, the width and height for this pie are 100x100. This is pretty small - but you can change this. If you have clicked on the line that said 'NewPie = Pie(Widget)' in the attributes window, you should see a scrollable list of all the attributes for the pie appear in the Attributes window. Click on the line that says 'height = 100' - this should now appear in the entry field below it. You can click in this and edit it. Change the 100 into 200. Do the same for the width attribute. The chart in the main window should now be bigger. You can also change the x and y attributes - these are the x and y co-ordinates for the bottom left hand corner of the widget. Change the x to 25, and the y to 10 - you'll see why later.

It is still just a circle though - it has no data to display. Most of our charts (except for a couple of extremely simple ones) have an attribute called 'data'. Our NewPie is no exception. The data attribute is a list, and currently it just contains the number 1. You can change this by clicking on the line that says 'data' in the attributes window, and then editing it in the text entry box. The data must be a list (i.e., be contained in square brackets), and it must be separated by commas. It doesn't matter if you use whitespace between the elements or not. So,

```
[23]
[1, 2, 3]
[2,3,4]
```

are all valid lists for the data attribute, but these are not:

```
(1,2,3)
1,2,3
'1,2,3'
```

Most widgets will have attribute checking switched on. This means that they check if what you are entering is a valid data type for that attribute. If you try and enter an invalid list for data - or if you try and enter something that isn't suitable for any other attribute, say a string instead of a number - you will get a 'traceback'. The one for this attribute would look something like this:

```
Traceback (most recent call last):
  File "\rlextra\graphics\guiedit\guiedit.py", line 920, in attrChange
    self.redraw(action='redraw')
  File "\rlextra\graphics\guiedit\guiedit.py", line 718, in redraw
    self.newDrawing()
  File "\rlextra\graphics\guiedit\guiedit.py", line 826, in newDrawing
    self.project.getSample()
  File "\rlextra\graphics\guiedit\guiedit.py", line 392, in getSample
    exec "%s\n_x=%s()\n" % (self.buildText(),self.className) in locals()
  File "<string>", line 14, in ?
  File "<string>", line 13, in __init__
  File "reportlab\graphics\widgetbase.py", line 49, in __setattr__
  File "reportlab\lib\attrmap.py", line 72, in validateSetattr
AttributeError: Illegal assignment of '23' to 'data' in class Pie
```

The last line is the most important - that's the one telling you what has gone wrong. In this case, you'd have entered '23' instead of '[23]'

Width, height, x, y and data are common attributes for most of the widgets you will be working with.

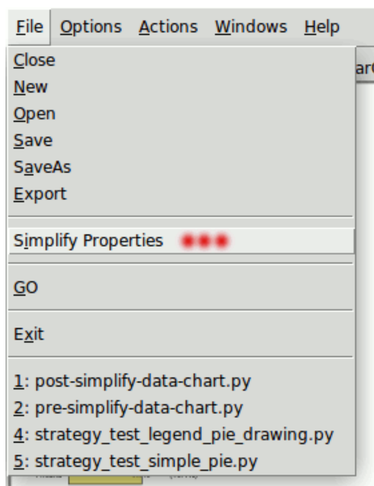
Simplifying Attributes

When you change attribute properties you will see lines being added to the source code in the source window. When you are assigning a new value to an attribute that you have already modified, the new attribute assignment is simply appended to the code, with the last assignment being the one that is effected. An example of this in the screen shot below showing multiple attempts to set the chart.height.

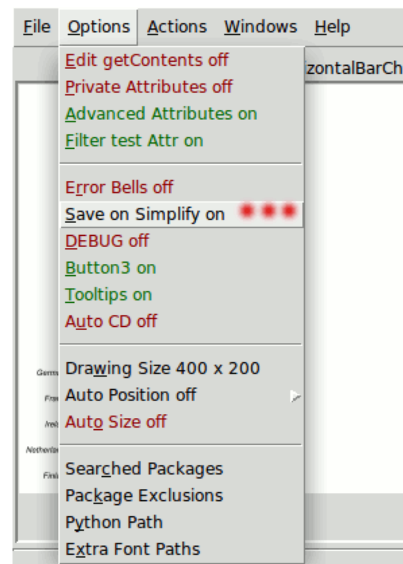
```
#Autogenerated by ReportLab guiedit do not edit
from reportlab.graphics.samples.line_chart import LineChart
from reportlab.graphics.shapes import _DrawingEditorMixin

class LineChart_000(_DrawingEditorMixin,LineChart):
	def __init__(self,width=400,height=200,*args,**kw):
		LineChart.__init__(self,width,height,*args,**kw)
		self.height      = 300
		self.width       = 500
		self.chart.height          = 180
		self.chart.width           = 255
		self.chart.height          = 200
		self.chart.strokeWidth     = 3

if __name__=="__main__": #NORUNTESTS
	LineChart_000().save(formats=['pdf'],outDir='.',fnRoot=None)
```

A list of the attributes, might be handy for viewing a history of what has been changed. However after a period of time seeing particular attributes assigned multiple times might seem confusing. If you want to remove the duplicate settings you have two options, either manually by selecting the 'File' menu then selecting 'Simplify properties' or automatically when you save your chart by selecting 'Save on Simplify' in the 'Options' menu (saving you chart is covered the subsequent section).

Manualy Selecting 'Simplify properties'



Setting 'Save on Simplify'



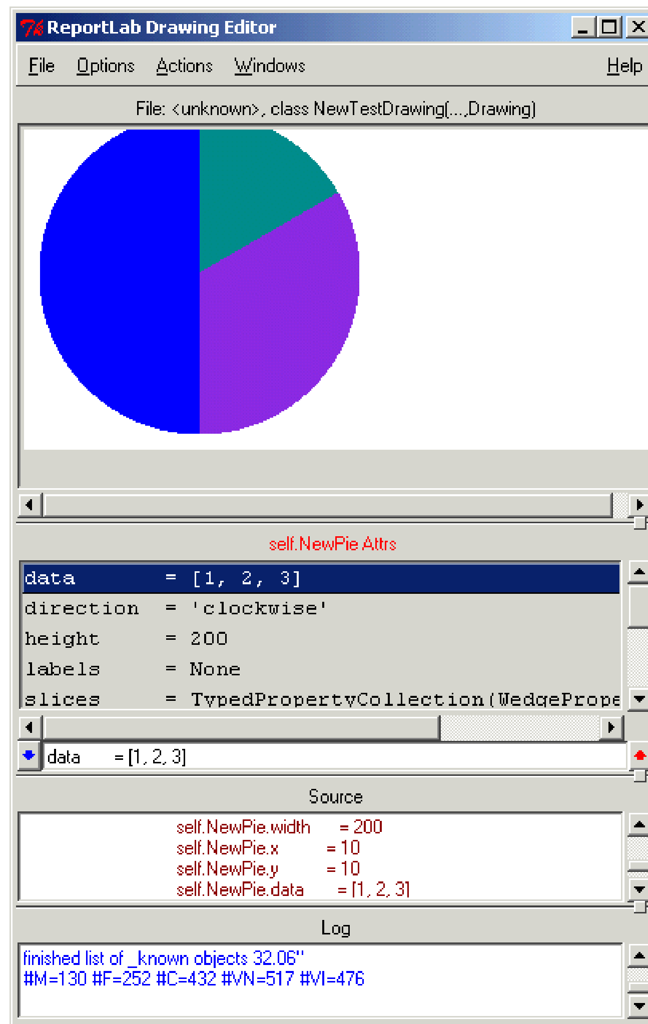
After you have run 'Simplify Properties', or have 'Save on Simplify' set to 'on', if you look at the 'log' window you should see that the duplicates have been removed.

```
#Autogenerated by ReportLab guiedit do not edit
from reportlab.graphics.samples.line_chart import LineChart
from reportlab.graphics.shapes import _DrawingEditorMixin

class LineChart_000(_DrawingEditorMixin,LineChart):
	def __init__(self,width=400,height=200,*args,**kw):
		LineChart.__init__(self,width,height,*args,**kw)
		self.height      = 300
		self.width       = 500
		self.chart.width           = 255
		self.chart.height          = 200
		self.chart.strokeWidth     = 3

if __name__=="__main__": #NORUNTESTS
	LineChart_000().save(formats=['pdf'],outDir='.',fnRoot=None)
```

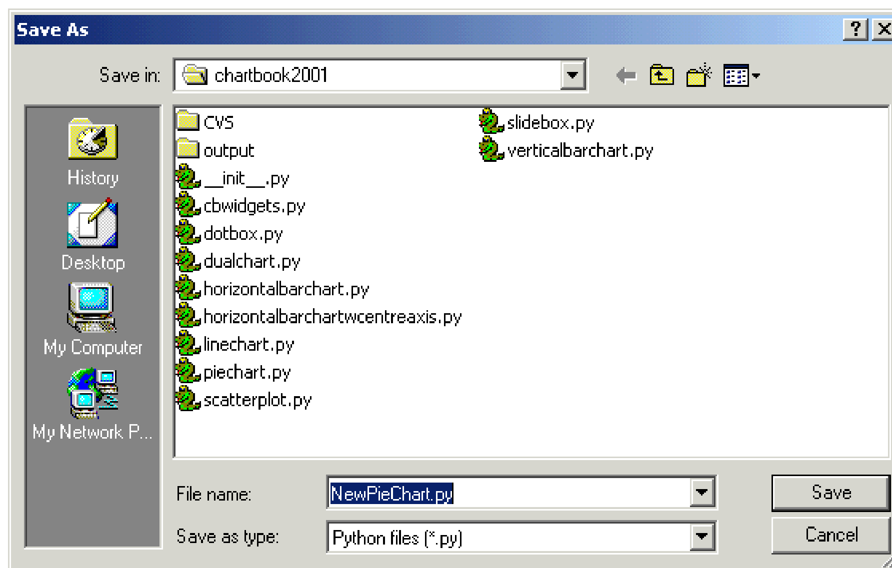
```
Simplifying edits
edit[2]: self.chart.height [0:30] removed
Assignments: 1 removed, 0 not removed 1 lines totally removed
project modified
```

3.5 Saving Your Project

If you have made all the changes described above, you should have something that looks like this:

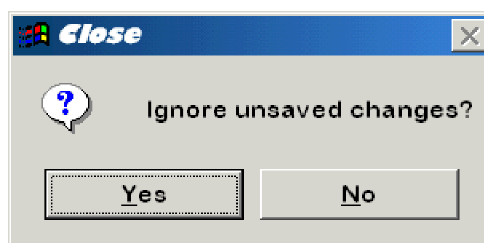


We've now made enough changes to be obvious that the chart has changed. Now we can save it. From the File menu, select the option 'Save'. This gives you a save dialog which allows you to navigate the file system. You don't have to save the file on your local machine - you can save it on any mapped drive or server that is available to you. Enter the filename of 'NewPieChart' and click on 'OK' to save this chart. The Drawing Editor will automatically append a .py suffix onto the filename (to show it is a Python file).



When the main window is visible again, notice how the file line above the main window now contains the file name.

Now we have finished with the immediate changes and saved the chart, we can close it. Select 'Close' from the file menu. The main window goes back to black with the 'ReportLab GUI Drawing Editor' line to show that we have no chart loaded.



If you have forgotten to save the changes you have made to the chart, you will see a dialog box like the one above. Click on 'No' to keep this chart open and go back and save the chart, or click on 'Yes' to close it and lose any changes you have made.

3.6 Exporting to Different Formats

The Drawing Editor also allows you to save your file in a bitmapped graphic format. To do this, go to the 'File' menu and select 'Export'. A 'Save As' dialogue box will appear, but it has the addition of a 'save as type' dropdown box. You can select from the available formats (including Jpeg or GIF suitable for web use, Postscript or Encapsulated Postscript, TIFF and PNG).

It's not advisable to export a file without saving it using the 'save' or 'save as' options. Once a file has been exported, it is impossible to convert it back again. Exporting a file is a strictly one way process.

3.7 Opening Existing Files

Once you have created a chart, you shouldn't have to use the 'New' menu item again. That should be a once-per-project step.

Back in the Drawing Editor, you can select 'Open' from the file menu to open an existing chart. For this example, open the file NewPieChart.py. Let's make another couple of changes:

Double click on the 'NewPie = Pie(Widget)' to take you back into the attributes for the PieChart. Click on the line 'labels = None', which should appear in the entry field ready to be changed. None in this context is a Python object that means 'null', and should be used where you don't want an attribute to be used (rather than trying to use empty strings, empty lists etc which may cause unexpected behaviour). You can change it to something like

'labels = ['a', 'b', 'c']' to make labels appear. The labels attribute accepts a list in the same way data does. (You should also see why we suggested making the x for the pie 25 - if it wasn't this big, you wouldn't have been able to see the label).

If you select 'Save' from the file menu, the chart will be saved to the file that it came from. Since the Drawing Editor already knows the file name, you will not be asked to give one, and no save dialogue will appear. If you want to save this to a different file name, use the 'Save as' item from the file menu instead.

3.8 Adding Primitives - and another way to add widgets

As well as adding widgets from the menu, you can also add them from the text entry box (ie, the "entry box" with the white background). So to add the pie chart, we could have typed

```
add(Pie(), 'NewPie')
```

You may have to highlight and remove what is already there to do this - in this case highlight "background = None", hit delete, and type in the line above.

The 'Pie' is the name of the widget, and the 'NewPie' is the name that we will be referring to it by. Pie must be followed by those brackets. If you know the structure of the Pie widget, you can use those brackets to pass in attributes for it to use, but in most cases it is easier to leave them empty and set them using the Drawing Editor.

The Drawing Editor doesn't just allow you to add widgets in this way. You can also add graphics primitives. If you type

```
add(Rect(0,0,20,20), 'Rectangle')
```

you will see a rectangle appear in the bottom left hand corner. The numbers in the brackets represent the following attributes: x, y, width, height. You can also do this for other graphics primitives (such as Circle and Ellipse). Notice that because these are primitives rather than widgets, you have to give them certain required parameters, rather than just let them use their own defaults and edit them later. For more information about what these are and how to use them, look in the Graphics Guide documentation (or read the code and docstrings in reportlab\graphics\shapes.py).

Some examples:

```
add(Circle(20,20,20), 'MyCircle')
add(Ellipse(20,20,20,20), 'MyEllipse')
add(Line(0,0,20,20), 'MyLine')
```

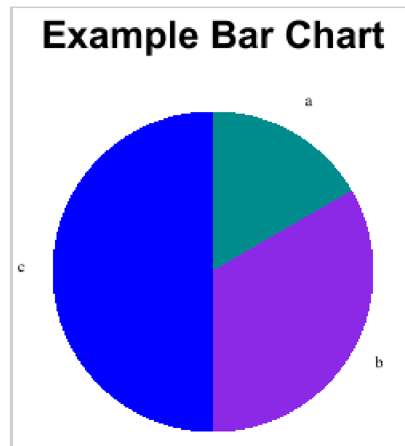
As well as the more obvious graphics primitives, you can also add lines of text to a chart. As an example, type this into the text entry box:

```
add(String(125,250, 'Example Bar Chart'), 'Title')
```

Looking at the chart, nothing seems to have changed. This is because, while the string has been added it's been added in a place off the top of the chart. To make it visible, edit the height and width attributes (of the chart, not the pie). Make them something sensible like a width of 250 and a height of 275 (to allow for the title). The other alternative is of course to move the string to a different place on the chart - but the important thing to notice is that there isn't any sanity checking of where you put it.

When we added the string, we gave it a y attribute of 125, which is in the centre of the chart. To make sure the string actually is centred, you need to edit its attributes. Click on the line in the attributes window which says 'Title = String(Shape)'. Right-click on the textAnchor line, and you can see the options to select from. 'Start' makes the x position the start of the line of text, 'end' makes the end of the line appear at the x position, and 'middle' centres it. You can also change the font to be used for the string, its size and various other attributes from here.

By this point, you should have enough knowledge to put together a static chart at least as complex as the one below:


Example Bar Chart
a
b
c

Miscellaneous Features

A few quick notes about Drawing Editor features that we haven't mentioned yet:

- Zoom set

  This item (from the Windows menu) allows you to zoom in or out of the chart. In other words, it magnifies or decreases the size of the image in the main window.

- Clear log window

  This item (from the Windows menu) removes all the text from the log window. Useful if you have large amounts that you no longer want to scroll through.

- Reload module

  This item (from the Actions menu) make the Drawing Editor reload a module. Useful if you have edited or updated the Python file for one a class or drawing while you have the Drawing Editor still open. Instead of shutting it down and restarting to make sure the Drawing Editor notices the change, you can use 'reload module' and select the module to reload and update.

- Debug

  This item (from the Option menu) turns on some debugging information printouts. These are normally only of use to developers.

- Exit

  This item, in the File menu, quits out of the Drawing Editor program.

4 Working with Charts

This chapter covers some general details of the chart classes and their attributes.

4.1 Working with Collections

A number of attributes for charts are 'Collections'. A collection in this instance is a 'smart collection class' under the hood which allows you to change attributes for just one or for all of the items in a set. These sets can be the slices in a pie chart, or the labels or barLabels in a line chart. They even allow you to change the attributes for an item which doesn't exist yet!

If you have a source distribution, open the file named 'example_collections_1.py' from the 'samples' directory. Click down into the attributes for the pie chart. You should see a line like this:

```
slices              = TypedPropertyCollection(WedgeProperties(PropHolder))
```

This line represents a collection. Double click on the line to show the attributes of the collection:

```
[0].fillColor   = darkcyan
[1].fillColor   = blueviolet
[2].fillColor   = blue
[3].fillColor   = cyan
[8].fillColor   = green
fillColor   = pink
```

These attributes fall into two different categories.

If a line in a collection (such as the 'slices' collection above) starts with a number in square brackets, that line refers only to that numbered element in the collection. So '[2].fillColor = blue' sets the filling only for the third wedge in the chart (since we start counting from 0). Any attribute that appears in a collection can have a numeric index added as a prefix to define that attribute for one element of the collection. In our pie example, you could set *strokeWidth*, *strokeDashArray*, *strokeColor*, *popout*, *labelRadius* or *fillColor* for any individual slice just by adding a prefix like '[0].' to the start of the line.

In contrast, if a line in a collection begins with an ordinary attribute name, then that line sets the default value for all of the analogously-named numbered attributes in the collection. For example, the last line above makes all the wedges in your pie chart pink by default.

You should be careful when mixing the two kinds of statements. One of the second type (with no numeric index) sets the attributes for all the elements in the collection - but only if none of the second type are used. The kind with the numeric index always over-ride the more general kind of statement. These general, un-numbered statements then become a default which is used as a fallback when no specific statement applies to a particular element. (e.g. if you define a default fillColor with one of these un-numbered statements, then define the fillColor for elements [0], [2] and [4], the default fillColor statement would apply to elements [1] and [3]).

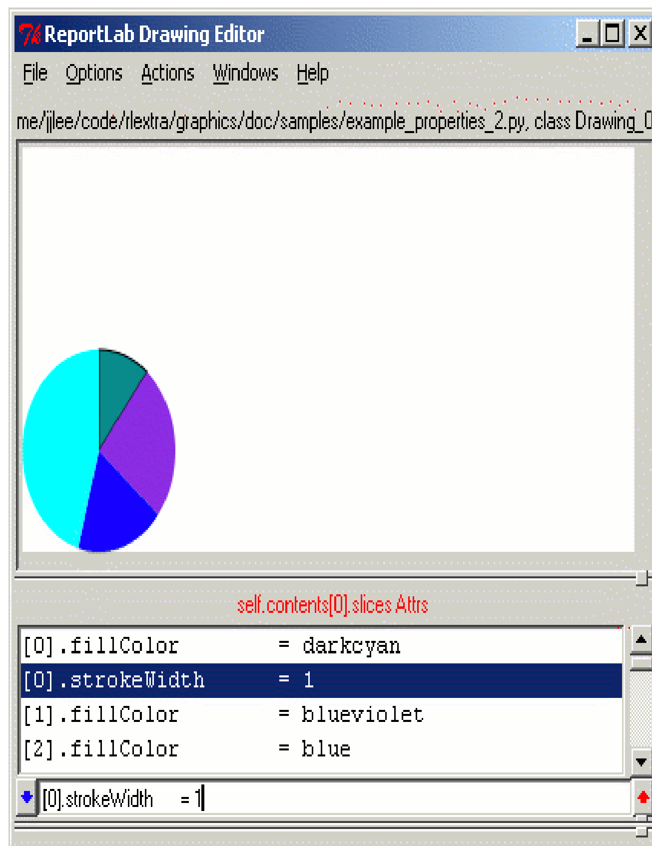
If our chart has fewer than 9 slices, then you might expect the second-to-last line in the pie chart example above to cause a problem. It doesn't. You should think of these statements as prescriptions for what to do to a member of the collection if it exists - if it doesn't this prescription is silently ignored. This also means that when you are using the Drawing Editor to edit a chart, you can edit the attributes for an element which isn't even in the list which appears in the attributes window.

Another thing to be aware of with collections is the way that they 'rollover'. When you set the attributes for numbered members of a collection, if there are more members than you have set the attributes for, then the attributes will rollover. If you have set '[0].fillColor' as red, '[1].fillColor' as green and '[2].fillColor' as blue, then the fillColor for '[3]' would flip back to the start and become red, [4] becomes green, and so on. (Actually, this isn't quite the case for the standard pie chart since it has the first four fillColors set in the base class, but this principle works in most other places).

Adding New Attributes to a Collection

To add a new numbered attribute to a collection, simply enter a new atrribute in the entry field. For example, to add a new attribute specifying the stroke width for the first pie slice:

```
[0].strokeWidth = 1
```



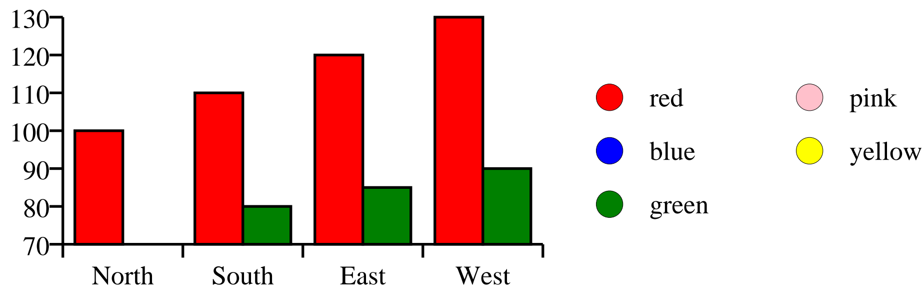
4.2 Working with Legends

This section shows how to construct legends. Legends are a little more involved than most other chart components, in that they commonly need to pull other information from other things on the drawing; their size and shape may depend on the number of data series in a chart, which isn't always known at design time. Legends have just undergone an upgrade in December 2004 to allow them to 'connect to' and configure from the charts they are attached to.

If you have a source distribution, the example charts herein will be located in a subdirectory called 'samples' and are named as 'example_legend*.py'. All of these can be opened in the Drawing Editor, and directly executed to create a PDF output chart.

The drawing below shows us getting started with a simple drawing containing a vertical bar chart, title string, and Legend object. The Legend has all the normal default values EXCEPT THAT we have positioned it by setting x=220, y=20, and boxAnchor="southeast". The latter choice means that the (220,20) reference point is at the bottom left of the legend. When you first add the legend, you might not even see it as y = 0 and the legend 'grows down' from the top of the drawing.

Chart with Legend - example 3



The legend is, of course, the bit on the right with the multiple circles. This does not agree with the chart because, at the moment, it has no knowledge whatsoever of the chart's existence.

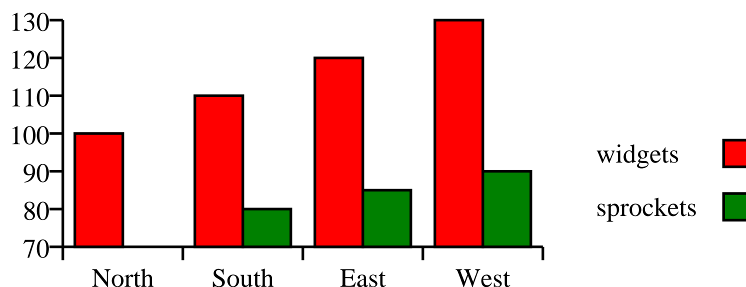
We'll just give a quick guided tour of the main attributes; we advise you to open up this example in the drawing editor or to create one yourself and explore. A legend consists at minimum, of a number of 'swatches' of colour, each with a string attached. The overall position is set with the *x*, *y* and *boxAnchor* attributes. The actual data can be explicitly set through the *colorNamePairs* attribute, if you wish. Let's say we know there will be two series, and want to configure automatically.

There are two main ways to get a legend working correctly. First, you can explicitly configure it. If (as is very often the case), you know exactly how many series there will be, and the labels to go with them, you could set this:

```
colorNamePairs = [(red, 'widgets'), (green, 'sprockets')]
```

And you would then see a legend like this:

Chart with Legend - example 2



The other thing you may want to do is adjust the position of the swatches, especially if there are a lot of them. Pie charts might have a dozen or more. The basic layout is to arrange them in columns, up to a maximum numbe of swatches per column; the legend has an attribute *columnMaximum* with default value 3, meaning that after 3 swatches are created it will start a new column. If you had twelve colorNamePairs entries and left the *columnMaximum* on 3, you would get 4 columns. You can set this to a high number to arrange all your swatches vertically, or to 1 to force them to spread out horizontally. There are also a number of attributes to specify the size of the swatch rectangle itself *(dx, dy)*, the vertical and horizontal separation between swatches (*deltax, deltay*), the space between swatch and text (*dxTextSpace*), and whether text appears to left or right of the swatches (*alignment*).

Until v1.20 (Dec 2004) all swatches were simple rectangles, and a separate LineLegend would draw lines with markers. From Jan 2005 onwards, you are advised to use the plain old Legend class, and if you want a different shape under direct control you can set the *swatchMarker* attribute. For example, if we assign as follows:

```
legend.swatchMarker = makeMarker('Circle')
```

...then we get circle shaped markers, and can then drill down into the marker object to further configure the size, colour, thickness etc. The default kinds of Marker at present are *'Square', 'Diamond', 'Circle', 'Cross', 'Triangle', 'StarSix', 'Pentagon', 'Hexagon', 'Heptagon', 'Octagon', 'StarFive', 'FilledSquare', 'FilledCircle', 'FilledDiamond', 'FilledCross', 'FilledTriangle','FilledStarSix', 'FilledPentagon', 'FilledHexagon', 'FilledHeptagon', 'FilledOctagon', 'FilledStarFive', 'Smiley'*

This kind of direct control is simple and should let a designer position a legend exactly the way they want. While it may seem perverse to set the shapes and colours yourself, it's also probably the simplest way to do things in cases where there are many charts arranged on a drawing.

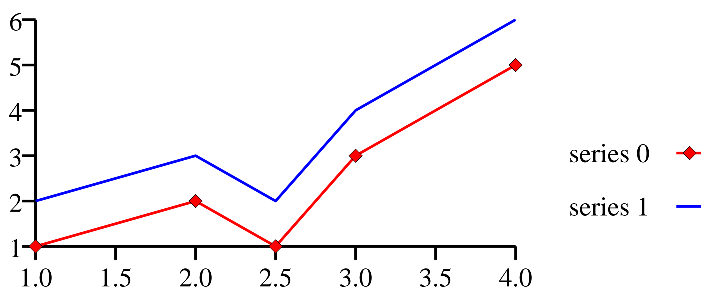
Connecting charts to legends

In other cases, one really wants the legend to intelligently connect to the chart and configure itself, as the chart data might vary dramatically at runtime. Two cases where you might often want this are for Pie charts with many slices, where it's tedious to repeat a long list of colour names and the number is variable; and for line charts with markers, where quite a lot of attribute-setting would be needed to make the legend match the chart. The new *Auto* feature allows this to be done at several levels. *(This has been added in a backward compatible way and is therefore not very elegant - be warned! In Version 2.0 we hope to reimplement some attributes more sensibly)*. To do things automatically, you must set the colorNamePairs object to a new magic 'Auto' value, and tell it how to find the chart to configure itself from. In the example below, we have switched to a LinePlot object called 'plot', and defined markers for the lines. The Drawing object can thus refer to it as 'self.plot', and we pass this to the Auto object as follows:

```
colorNamePairs = Auto(chart=self.plot)
```

That's the main thing we need to do - the chart now picks up the number and colour for each series, and draws the lines. It cannot yet say anything more intelligent than 'series 0' and 'series 1' for the series names. We'll cover this in a moment.

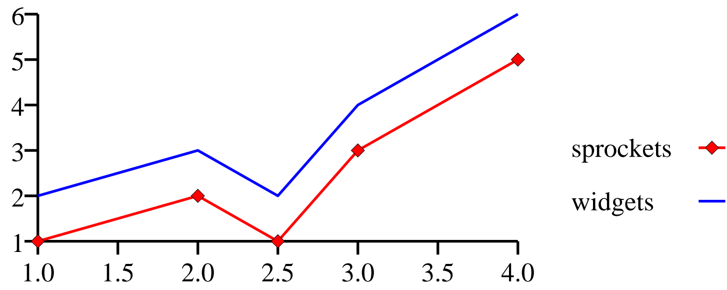
Example 4 - auto legend



The problem is that until now we have had no need for any 'series name' in the chart classes. We've added a hidden attribute to most of the charts so you can set this if you wish, and legends can pull it out. Each chart has an collection attribute, whose name regrettably varies, for saying how each series should appear; for a bar chart this is called 'bars', for a pie it is 'slices' and for a line plot it is 'lines'. If one drills into this in the editor, one can set these on specific series by typing "[n].name='myname'" for series n. Defining two of these will result in the following chart:

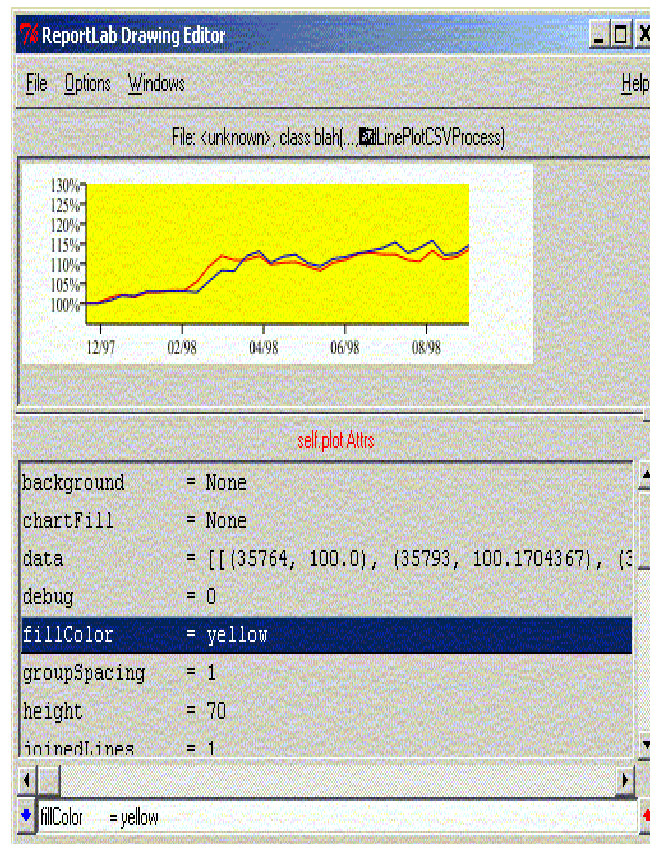
```
self.plot.lines[0].name = 'sprockets'
self.plot.lines[1].name = 'widgets'
```

Example 5 - auto legend with series names



4.3 Working with Colours

Our graphics framework supports the RGB and CMYK color models as well as specific spot colours. The examples so far have all involved attributes like 'red' and 'yellow', which actually refer to RGB colours. Here's what we have done so far when editing a color:



Some users prefer to use CMYK colours while others,for example web designers, may want a specific RGB value. Here are the things you could type into the line currently saying 'yellow':

For the RGB model, with each number from 0 to 255, we use the standard *Color* class from *reportlab.lib.colors*:

```
fillColor = Color(128,0,0)
```

People used to writing HTML documents may find "HTML-style" colors useful:

```
fillColor = toColor('#FF0000')
```

For the CMYK model, with each number ranging from 0 to 1, we use the class *CMYKColor*:

```
fillColor = CMYKColor(1.00,0,0.83,0.47)
```

Many people in the graphics world think in percentages and like to specify a range from 0 to 100. We therefore have a *PCMYKColor* class as well - the 'P' standard for 'Percentage' - which is exactly the same except for the numeric range:

```
fillColor = PCMYKColor(100,0,83,47)
```

CMYKColor and *PCMYKColor* accept the *spotName* and *density* attributes. In our core open source model, they don't do anything and are ignored when we write Postscript, but an additional (commercial) renderer will know what to do with this information. So we can now specify an "Institutional Green" if we wish, or do inks of varying densities:

```
CMYKColor(1.00,0,0.83,0.47,spotName='PANTONE 349 CV',density=1.00)
```

Finally and most usefully, we can just define some constants in an external file. Create a module called, say, "yourcolors.py", and put in on the Drawing Editor's search path somewhere (for example in *rlextra/graphics/guiedit*). In it we can put named constants like this:

```
from reportlab.lib.colors import CMYKColor

#Your Company's Institutional colors
```

```
InstGreen = CMYKColor(1.00,0,0.83,0.47,spotName='PANTONE 349 CV',
                      density=1.00)
InstKhaki = CMYKColor(0.32,0.23,0.51,0.07,  spotName='PANTONE 875 CV',
                      density=1.00)
```

the Drawing Editor will find these colors, and you can use them just like the standard ones. We suggest that where required each department creates a module of their own like this.*Warning: don't try to redefine 'red', and don't define the same name in more than one module; we don't want to promise which definition would be found and used.*

4.4 Working with Fonts

Diagra can use exactly the same fonts in all output formats. It uses Type 1 fonts and needs an AFM and a PFB file for each. We covered their installation earlier. If creating fonts outside of the 'standard 14' (Times/Helvetica/Courier with bold and italic, Symbol and ZapfDingbats), you may need to add a couple of lines when rendering a drawing.

To use a Type 1 font (*aka* a Postscript font), you will need at least two files. So for example, for the font Avenir you could have files including:

```
aeblo___.PFB
aebl____.PFB
aeblo___.AFM
aebl____.AFM
```

AFM file

The AFM file is the 'Adobe font metrics' file. The AFM file contains important info about the characters in the font (the 'glyphs'). This includes information such as their height, width, info on the bounding box, hints and kerning and other things that are collectively known as the 'font metrics'. It is vital to have the AFM file if you want to use any ReportLab software to embed fonts.

- If you don't have the AFM file for an Adobe font, you can get them from the Adobe site. ftp://ftp.adobe.com/pub/adobe/type/win/all/afmfiles/
- To find the Adobe package number, you can check their type catalogue: http://www.adobe.com/type/main.html

PFB file

The PFB file is the 'printer outline file' or 'Printer Font Binary'. The PFB file is used by our software if you need to produce bitmaps (by setting 'formats' to be gif, jpg etc), and to produce the previews in EPS files.

PFM files

You may also see PFM files. The PFM file is the 'metrics file' or 'Printer Font Metrics' file. Essentially, this is a reduced version of the AFM file. Most of the information our software needs is missing from this file.

Once you have these files, they have to be located somewhere where our software can find them.

If you look in the file rl_config.py (located in the Python22/reportlab/ directory), you will find a section that looks something like this:

```
# places to look for T1Font information

T1SearchPath =  ('c:/Program Files/Adobe/Acrobat 5.0/Resource/Font', #Win32, Acrobat 5
                 'c:/Program Files/Adobe/Acrobat 4.0/Resource/Font', #Win32
                 '%(disk)s/Applications/Python %(sys_version)s/reportlab/fonts', #Mac?
                 '/usr/lib/Acrobat5/Resource/Font', #Linux, Acrobat 5?
                 '/usr/lib/Acrobat4/Resource/Font', #Linux
                 '%(REPORTLAB_DIR)s/fonts' #special
                 )
```

If you have extra fonts that you want to use and embed, you have two choices. You can edit the rl_config.py file and add the existing directory where your AFM and PFB files are located to this section. Or, you could add your font files to one of these directories which the Diagra software already know about. A good one is '%(REPORTLAB_DIR)s/fonts' (i.e. C:\Python22\reportlab\fonts on a standard PC installation).

Typically, if you are running in a server environment, you'd put the AFM and PFB files in the fonts directory under the reportlab installation. If you are running in a desktop environment, it might be easier for you to add your font directories to the section in rl_config.py instead.

Once you have done all this, you can then use any of these fonts from the Drawing Editor. Anything which has a 'fontName' attribute can be given the name of one of these fonts. (If in doubt, the name we use is the same as the one which appears in the AFM file on the line which begins 'FontName').

4.5 Formats and Formatters

DecimalFormatter

In any place in a chart where you might require a format, you can use the decimal formatter. The most obvious examples are the *barLabelFormat* and *valueAxis.labelTextFormat* in a bar chart.

What you would do is find the correct line in the window, highlight it so that you can edit it in the Drawing Editor's text entry field, and replace the text after the equals sign with 'DecimalFormatter()'. Don't forget the brackets at the end! You should then see something which resembles one of these lines (though of course the details would change with the location):

```
barLabelFormat   = DecimalFormatter(places=2, decimalSep='.',
                                    thousandSep=None,
                                    prefix=None,
                                    suffix=None)
valueAxis.labelTextFormat    = DecimalFormatter(places=2,
                                                decimalSep='.',
                                                thousandSep=None,
                                                prefix=None,
                                                suffix=None)
```

This then allows you to do things like change the decimal separator from a full stop to a comma (to have numbers appearing in the continental style of '1,00' rather than '1.00'), and add a prefix (for things like '$' and '£') or suffix (for things like '%'). This also allows you to specify how many numbers should appear after the decimal point (or comma) - this is set by the 'places' attribute.

NA_Label and naLabel

In some circumstances, you will have a chart which takes data from a database which contains nulls (i.e. total empty cells which return a 'None' value to Python or a 'Null' value to SQL). By default, these cells are ignored (so, for example, no bar and barLabel appear for them in a bar chart which instead just leaves a gap where the bar should be).

This isn't always what you want. The required behaviour may be to leave out the bar but to insert a label which says 'n/a' (or 'results not in' or some other snippet of text).

This is what the attribute *naLabel* is designed for. A number of the charts have an attribute called *nalabel* . Initially, this starts off as:

```
naLabel          = None
```

You should change this to read

```
naLabel          = NA_Label()
```

The Drawing Editor should then expand it to read something like:

```
naLabel          = NA_Label(BarChartLabel)
```

These na_labels start of as Times-Roman with a fillColor of black and the text being 'n/a'. They do not start with the same formatting as your normal barLabels, and use their own separate formatting, which allows you to set them to a different colour or style if you want them to stand out.

NB: These labels only get used for cells which return Null (or None). '0' is often a perfectly valid piece of data, so if you want your chart to display these na_labels in these situations, you should change the behaviour of your database to returns nulls rather than zeroes in these locations.

Format Strings

You will probably be using format strings in a number of places when you use the Drawing Editor to create or modify charts. As an example, here is a fileNamePattern taken from a DataAwareDrawing. (DataAwareDrawings are designed to pull data from a data source and produce multiple files using this data, and their fileNamePattern allows us to do give them distinct names which share a common root).

```
fileNamePattern = 'drawing%03d'
```

The '%' sign means that we are using a *format string*. Anyone who has used C's sprintf() format or is familiar with string formatting in Python will recognise this. The 'd' in this case stands for 'decimal' (i.e. a decimal integer). Others you may (rarely) want to use include 'f' for a floating point number, 'o' for octal, 'x' or 'X' for hexadecimal, 'i' for integer (identical to 'd'), and 's' for string. The '03' refers to a zero fill - in this case to three digits. If you use 'f', the chartId is converted to a floating point number, and you can use a '.' between two numbers to specify a field width and a precision.

Examples:

The following examples assume that we are using chartIds of 1, 2, 3 and that we are only producing PDF output:

fileNamePattern	result
`drawing%d`	`drawing1.pdf, drawing2.pdf, drawing3.pdf`
`drawing%03d`	`drawing001.pdf, drawing002.pdf, drawing003.pdf`
`drawing%07d`	`drawing0000001.pdf, drawing0000002.pdf, drawing0000003.pdf`
`drawing%f`	`drawing1.000000.pdf, drawing2.000000.pdf, drawing3.000000.pdf`
`drawing%1.2f`	`drawing1.00.pdf, drawing2.00.pdf, drawing3.00.pdf`
`drawing%s`	`drawing1.pdf, drawing2.pdf, drawing3.pdf`
`drawing%5.5f`	`drawing1.00000.pdf, drawing2.00000.pdf, drawing3.00000.pdf`
`drawing3.2%f`	`drawing3.21.000000.pdf, drawing3.22.000000.pdf, drawing3.23.000000.pdf`
`drawing03%d`	`drawing031.pdf, drawing032.pdf, drawing033.pdf`

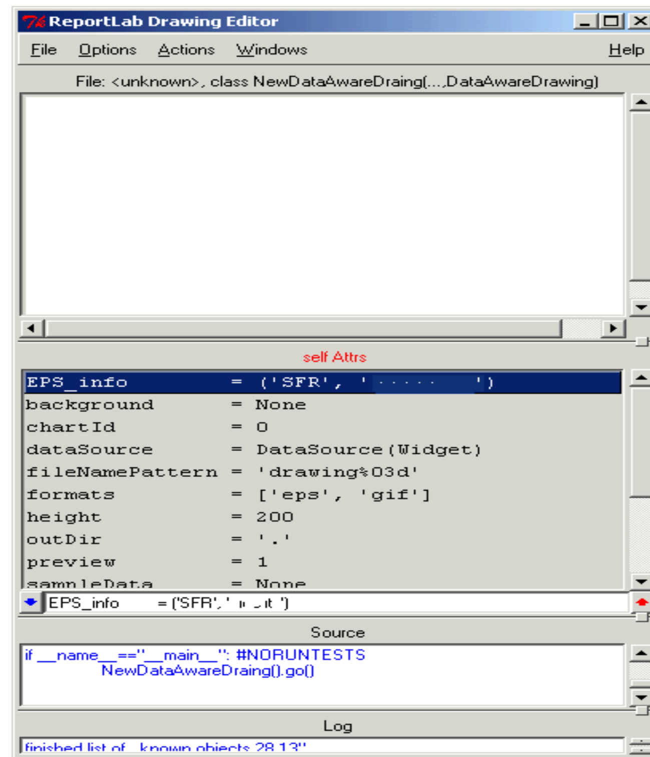
NB Notice how the last two are wrong: they both have the numbers *before* the '%' sign rather than after it, so the numbers are just treated as part of the alphanumeric filename prefix rather than the formatting string. These examples all use 'drawing' as this prefix, but you could of course have used 'Foobar' or anything that you required.

5 Creating Data Aware Charts

5.1 Introduction to DataAwareDrawings: Plotmode, fileNamePattern and outDir

The main difference between a static drawing and a data aware drawing is that a data aware drawing inherits from the 'DataAwareDrawing' class (which in turn inherits from 'Drawing'). In practice, you can do anything with a DataAwareDrawing that you can with a Drawing, but it has a number of additional abilities.

To start a DataAwareDrawing, start up the Drawing Editor, select 'New' from the File menu and choose 'DataAwareDrawing' for the Base Class Name. Give it a name - such as 'NewDataAwareDrawing'. This is also a blank canvas in the same way that Drawing was.



The attribute window should contain something like the following:

```
EPS_info        = ('Dept', 'Your Company')
background      = None
chartId         = 0
dataSource      = DataSource(Widget)
fileNamePattern = 'drawing%03d'
formats         = ['eps', 'gif']
height          = 200
outDir          = '.'
preview         = 1
sampleData      = None
showBorder      = 0
test            = 0
transform       = (1, 0, 0, 1, 0, 0)
verbose         = 0
width           = 400
```

Plotmode

One attribute that we didn't see when starting off with a Drawing was the 'plotmode' (the line that looks like:

```
formats         = ['eps', 'gif']
```

This is the format that the output will appear in. The square brackets around it show it is a list. Each item in it should be surrounded by quotes. We can have multiple output formats (as long as the renderers can handle them).

Valid plotmodes are currently 'pdf', 'gif', 'png', 'tif', 'jpg', and 'eps'.

fileNamePattern

DataAwareDrawings are designed to pull data from a data source and produce multiple files using this data. Therefore we need to have some sort of filename convention to tell them apart. The fileNamePattern allows us to do this. The default of :

```
fileNamePattern = 'drawing%03d'
```

will give each file a name in the form of 'drawing001'(plus a file type suffix), 'drawing002', 'drawing003' and so on. You can edit this to change both the prefix text and the number added to it.

The '%' sign means that we are using a *format string*. See section 4.5 for more information about these.

outDir

As well as changing the name of the files produces, you can change where they are saved. The line

```
outDir          = '.'
```

sets the output directory for these files to be the same as where the script is. You can change this to a subdirectory, use '..' for navigation back up the file system, and either give absolute or relative path names.

EPS_info

This only useful when producing EPS output. It contains the Department and Company info that are inserted into the EPS header. It is ignored if you use any other output format.

background

This allows you to use a background for the whole drawing (not just a chart inside the drawing). You can use any colour that we recognise (look in reportlab\lib\colors.py and any other files where you have defined colours such as perhaps yourcompany\yourcolors.py) for lists of these. The RML2PDF has a good colorized list of these in Appendix A("Colors recognized by RML")). The following is an example of how to do one of these coloured backgrounds - this uses pink, but you can use any colour. Once you have added the background, you should see a new attribute appear in the attribute window which reads 'background = Rect(SolidShape)'. You can now double click on this to change the colour, add a stroke color and stroke width (the stroke in this case being an outline around the Drawing), and various other attributes.

```
background      = Rect(0,0,self.width, self.height, fillColor=pink, strokeColor=None)
```

As well as plain colours, you can also use other widgets. For example, entering the following into the entry box

```
background      = Grid()
```

gives you a grid as the background, whose attributes you can then edit in the usual way (making it horizontal instead of vertical, changing whether it uses coloured stripes or lines and so on).

5.2 Setting the Data Source

A DataAwareDrawing can contain one (and only one) 'DataSource'. Currently, all our data-aware classes use an ODBC database, but you can use a CSV file - and eventually you would be able to use other sources (such as an Excel spreadsheet).

For this example, you'll need to set up your database before you do anything involving data in the Drawing Editor. We provide an example Access database called 'sampledata.mdb'. You should register it with the data source name 'samplechartdata'. On a Windows machine, this involves going to the Control Panel and selecting 'Data Sources (ODBC)' (or a control panel with a similar name, which may or may not be under 'Administrative Settings' depending on which version of Windows you are using).

To make the chart connect to a database, you need to edit the line that reads

```
dataSource      = DataSource(Widget)
```

so that it reads

```
dataSource      = ODBCDataSource()
```

You should then see it change in the attributes window. You can then double click on that line to set the various attributes of the datasource. Once you have done this, the attribute window should look something like this:

```
associations   = Array(0, DataAssociation)
groupingColumn = 0
name           = 'samplechartdata'
password       = ''
sql            = 'SELECT * FROM generic_pie'
user           = ''
```

The *name* in this case is 'samplechartdata', though it could be any name you have previously set up as an ODBC data source with the OS.

If your ODBC datasource connection requires a username and password (for instance if you're using SQL server without Windows Authentication), you can append them to the datasource name separated by backslashes. So to connect to dsn 'mysqlserver' with username 'john' and password 'secret', set the dataSource.name property to 'mysqlserver/john/secret'

The *sql* attribute is the line of SQL(structured query language) to be used to pull the data from the database. In most cases you can either use

```
'SELECT * FROM yourdatasource'
```

which gives your chart access to everything in the database (where your datasource is the name of the datasource as given in samplechartdata), or

```
'SELECT columnname, columnname, columnname FROM yourdatasource'
```

which will only give you the specifically named columns from the datasource - whether there are any other columns in the database or not.

5.3 Setting the Data Associations

Once you have the data from the database, then you must define what to do with it in your chart. This is what the data associations are for.

When you first start a data source, there are no data associations set up. Double-clicking on the 'dataSource = ODBCDataSource(DataSource)' line brings up a set of attributes in the attribute window which includes:

```
associations   = Array(0, DataAssociation)
```

This shows that you have no data associations set-up yet. You can either edit that line in place, and make the zero into the number of data associations you will need, or double click on it, which will show you an attribute that says:

```
size = 0
```

where you can then edit the zero in the same way.

Whichever way you do it, the size line then changes (e.g. to 'size = 4'), and you will see a number of lines in the pattern of:

```
elementXX = DataAssociation(column=n, target=None, assocType='vector')
```

The *column* is the position in the list of data retrieved by the SQL statement that we saw under the dataSource. Notice how the count of columns starts a 0 rather than 1, and that column 0 is usually the chartId.

So, for example, if the SQL was

```
'SELECT chartId,numberOfBoxes,label,value FROM generic_slidebox'
```

then:

- column 0 would be the data in the column of the database called 'chartId',
- column 1 would be that of 'numberOfBoxes',
- column 2 would be 'label' and,
- column 3 would be 'value'.

Typically, column 0 is held by chartID which is used in setting the filename for the chart.

If you use a SQL statement which explicitly names the database columns you want to retrieve in this way, it is less fragile and likely to break than one where you pull everything using a '*'. If you use a 'SELECT * FROM' type SQL statement, it will break if anyone inserts another column between two of yours and changes the order

of the columns you are retrieving. Naming them saves you from this risk.

The *target* is the name of the variable in your chart that this data should be plugged into. This should always be qualified by using the name of the widget (or other object) where that variable is set. So you would use 'PieChart.data' rather than just data (if you wanted to set the data attribute in the widget PieChart), or 'chart.xValueAxis.labeltextFormat' rather than just 'labeltextFormat' if you wanted to set the format of the label text in the X-Axis of a linechart called 'chart'.

And lastly, the *assocType* is the type of association used to retrieve the data. You can pull single or multiple columns or rows from a database or just a single cell. Valid values are 'scalar','vector','matrix', 'tmatrix' (from transformed matrix) and 'rowmap'. For more info on these, look in individual examples and the following chapter ('More on Data Sources').

5.4 A practical example: Slidebox.py

The previous descriptions were probably pretty confusing without any solid examples to compare the against, so this section will rectify that. We will work through 6 examples (which should be available in rlextra/examples/graphics directory - those that work off the example database use the have the suffix of _db.py). In all of these examples you will see an example of what the finished chart looks like, along with the data in the sample database and the Data Source and Data Association lines.

Example Chart

Database Layout

Table: generic_slidebox

chartId	numberOfBoxes	label	value	title
1	7	source: Guinness	6	Beer Sales
2	7	source: Hoegaarden	5	Beer Sales
3	5	source: Youngs	4	Beer Sales

Drawing Data Attributes

```
#Autogenerated by ReportLab guiedit do not edit
from rlextra.graphics.guiedit.datacharts import ODBCDataSource, DataAssociation, DataAwareDrawing
from reportlab.graphics.shapes import _DrawingEditorMixin
from reportlab.graphics.charts.slidebox import SlideBox

class SlideBoxDrawing(_DrawingEditorMixin,DataAwareDrawing):
■def __init__(self,width=400,height=200,*args,**kw):
■■apply(DataAwareDrawing.__init__,(self,width,height)+args,kw)
■■self._add(self,SlideBox(),name='SlideBox',validate=None,desc='The main chart')
■■self.height         = 40
■■self.width          = 168
■■self.dataSource     = ODBCDataSource()
■■self.dataSource.sql ='SELECT chartId,numberOfBoxes,label,value FROM generic_slidebox'
■■self.dataSource.associations.size = 4
■■self.dataSource.associations.element00 = DataAssociation(column=0, target='chartId', assocType='sca
■■self.dataSource.associations.element01 = DataAssociation(column=1,
                                                       target='SlideBox.numberOfBoxes',
                                                       assocType='scalar')
■■self.dataSource.associations.element02 = DataAssociation(column=2,
                                                       target='SlideBox.sourceLabelText',
                                                       assocType='scalar')
■■self.dataSource.associations.element03 = DataAssociation(column=3,
                                                       target='SlideBox.trianglePosition',
                                                       assocType='scalar')
■■self.verbose            = 1
■■self.formats            = ['eps', 'pdf']
■■self.outDir             = './output/'
■■self.fileNamePattern    = 'slidebox%03d'


if __name__=="__main__": #NORUNTESTS
■SlideBoxDrawing().go()
```

Description

This example can be found in the directory `rlextra/examples/graphics`, with the filename of `slidebox_db.py`. `slidebox.py` is similar, but runs off a CSV file rather than a database.

In this first real example, we have given you the whole code that the Drawing Editor produces. For all of the other charts you will only be shown the self.dataSource attributes (and you should never need to look at the raw code if you are using the Drawing Editor). This chart is the simplest (from a data standpoint), so we can both show you what the code looks like in its raw state, and point out a few other pieces of information at the same

time.

The are only three pieces of data that a slidebox chart requires:
numberOfBoxes: the number of coloured and numbered boxes that make up this widget (in the example graphic above, it is 7).
label: the text that is displayed at the bottom right of the widget (in this case 'source: ReportLab')
value: the number where the triangle pointers should be displayed.

Looking at the code of slidebox.py (in the 'Drawing Data Attributes' section above), the first line we need to consider is

```
self._add(self,SlideBox(),name='SlideBox',validate=None,desc='The main chart')
```

This is line that adds the widget 'SlideBox' to the basic DataAwareDrawing. If we had been making this up from scratch, we could have achieved the same thing by selecting 'Add New Widget' from the Action menu, clicking on SlideBox in the list of widgets, and typing in the name 'SlideBox' in the 'New Widget Name?' text box.

The 'self.height' and 'self.width' lines, as you would expect, set the height and width of the resulting drawing. The 'self.dataSource' line sets things up so that we can use a database as the source for our data.

The 'self.dataSource.sql' line shows the line of SQL we use to pull this data. This looks in the table generic_slidebox in the database we have already specified and retrieves, for each chart items from the columns named 'chartId', 'numberOfBoxes', 'label' and 'value'.

The four lines that start 'self.dataSource.associations' set up what to do with this data. Each of these has an assocType of 'scalar'. Scalars are the simplest type of data association. They retrieve a single cell from the database (actually a 1x1 matrix). So for each chart, we retrieve the single element 'chartId' (which is not inserted into the widget that we use to create the slidebox since it is used by the DataAwareDrawing in constructing the filename), the single element 'numberOfBoxes' which is inserted into the 'numberOfBoxes' attribute of the widget SlideBox, the element 'label' which is inserted into sourceLabelText attribute of SlideBox, and the element 'value' which is inserted into the attribute trianglePosition of the SlideBox widget. You can see all these attributes when you double-click on the 'SlideBox = SlideBox(Widget)' line in the attributes window - it then displays all the attributes for this widget.

When you are editing a drawing using the Drawing Editor, everything is static. How can we tell if the data associations are correct? This is where the 'sampleData' and the 'test' attributes come into action. If you set the test attribute to be '1' (rather than the default '0'), it will test if the data-aware parts of your drawing work. It connects to your data source, fetches the data and applies the first block of data to the drawing (i.e. in this case, uses the first row in the table). If it fails, it gives you an error message in the Log window which should explain what went wrong. So for example, if you have the dataSource name wrong (or have forgotten to set it up with the OS), you would see something like this:

```
OperationalError: ('IM002', 0, '[Microsoft][ODBC Driver Manager] Data
source name not found and no default driver specified', 5896)
```

If everything works correctly, then the information from the first block of data is used, the main window will change to reflect the new status of the SlideBox widget, and the data itself is appears in the 'sampleData' attribute:

```
sampleData      = [(1, 7, 'source: Guinness', 6)]
```

If you want to continue testing, you can use numbers higher than one for the test attribute, and the drawing will fetch the corresponding data and use it in the same way (as long as you don't overshoot the end of the data in the data source).

The last attribute to mention before we talk about running this script and creating the charts is 'verbose'. If verbose is set to 0, then when you run it, the script is silent. It produces no text on your screen confirming what it has done. If verbose is set to 1, then it produces a message telling you when it has written each file. (These messages may be very useful in testing, but less so on a server - you can suppress them at will using this attribute).

The last line of the code is the one that makes the drawing 'active'. DataAwareDrawing has a 'go' method - a piece of code that connects to the specified data source, repeatedly fetches blocks of data, closes the connection, and uses the data in the way you have specified. The Drawing Editor inserts a call to this into your new drawing, so that when you run it from the command line or the desktop it will automatically do this. Guiedit automatically does this if the class has a 'go' method to make it active - DataAwareDrawing has, but Drawing doesn't. You should never have to insert this call by hand - but it's worth knowing about.

5.5 Running the resulting scripts

Once you have made any changes you want to the chart and then saved it to a file, you can then execute it. To run the script that creates the charts from this base chart, you need to drop back into the Operating System. You can use Windows Explorer (or the Finder if you are on a Mac) to find slidebox.py and double click on it.

You should see a window open and the following text appear in it:

```
generating PDF file './output/slidebox001.pdf'
generating EPS file './output/slidebox001.eps'
generating PDF file './output/slidebox002.pdf'
generating EPS file './output/slidebox002.eps'
generating PDF file './output/slidebox003.pdf'
generating EPS file './output/slidebox003.eps'
```

Running slidebox.py has created an both an EPS chart and a PDF chart for each chartID in our sample database. If you inspect them, you will find that the data from the data source has been inserted into the relevant places in the chart to produce the correct behaviour.

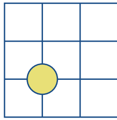
NB This method of running the file with a double-click is fine for Macintosh users, but may cause problems on a Windows system. On a Mac, the dialogue box will stay on screen (with a title of 'terminated') once the program has finished running. On a Windows system, the window appears, stays around for only as long as it takes to run the program and then closes. If the program terminates due to an error, an error message is displayed - but is not on screen long enough to be visible. This is why for Windows users, we strongly recommend starting the program from an MS-DOS console (available from the start menu e.g. Start->Programs->Accessories->Command Prompt). Once you have the console open, cd to the correct directory and start the python script. For example:

```
C:\python22\rlextra\examples\graphics
python slidebox_db.py
```

This way, if an error occurs you will definitely see the error message, and are not in danger of having a directory full of last month's charts.

5.6 Example 2: Dotbox.py

Example Chart



Database Layout

Table: generic_dotbox

chartId	dotx	doty	notes
1	1	1	Optional
2	2	2	Notes
3	2	1	Not Interesting
4	1	3	Ignore this... unless you want to.
5	0	0	
6	3	3	

Drawing Data Attributes

```
self.dataSource                      = ODBCDataSource()
self.dataSource.sql                  = 'SELECT chartId,dotx,doty FROM generic_dotbox'
self.dataSource.groupingColumn       = 0
self.dataSource.associations.size = 3
self.dataSource.associations.element00 = DataAssociation(column=0, target='chartId',
                                                        assocType='scalar')
self.dataSource.associations.element01 = DataAssociation(column=1, target='DotBox.dotXPosition',
                                                        assocType='scalar')
self.dataSource.associations.element02 = DataAssociation(column=2, target='DotBox.dotYPosition',
                                                        assocType='scalar')
```

Description

The data to be fetched from the dataSource in this case is also all scalars. So the single datum chartID is fetched and used by DataAwareDrawing, dotx is retrieved from the dataSource and inserted into the attribute dotXPosition in the widget DotBox, and doty is inserted into DotBox's dotYPosition attribute.

Two other things to notice are that the dataSource can have columns that are *not* referenced by any of the DataAssociations. You can have data in the database that is used by other programs or departments which doesn't affect the production of your charts in any way (such as the 'notes' column in this example).

Conversely, the widget that constructs the charts in your DataAwareDrawing can have attributes that are not data-aware, and still have to be set manually using the Drawing Editor. In DotBox, the labels are held in the xlabels and ylabels attributes. These are fixed and not set by the dataSource.

In this specific chart, the number of divisions inside the actual 'dot box' are set by the length of the list of labels. Using

```
xlabels        = ['Value', 'Blend', 'Growth']
```

produced a dotbox where there are three boxes along the x axis, but

```
xlabels        = ['Value', 'Blend', 'Growth', 'Something else']
```

will produce one with four along this axis. If you set either of these attributes to 'None' you will get an error, so if you want to remove the labels for one of these DotBoxes, you need to do something like:

```
xlabels        = [None, None, None]
```

which removes the labels but keeps the rest of the structure.

Other things that we can mention about Dotbox include previews and borders.

- **Preview**

  This controls whether your EPS files have a (cross-platform) TIFF image embedded as a preview. The default is 1 - they are on. If you need small file sizes set this to 0. If the number is greater than 1, the previews are enlarged. This makes the file sizes a lot bigger. This has no effect on output formats other than EPS.

- **showBorder**

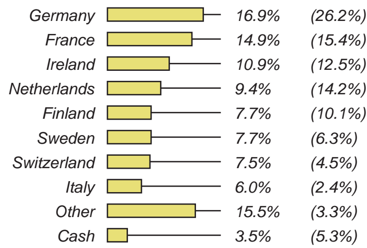
  If you are only using PDF output, you can use the showBorder attribute. Set this to 1, and it will add a border around the edge of your widget. For historical reasons, this doesn't work with EPS.

  One way to add a border that works in both PDF and EPS is to add the following:

```
background = Rect(0,0,self.width, self.height, fillColor=None, strokeColor=black, strokeWidth=1)
```

5.7 Example 3: Horizontalbarchart.py

Example Chart



Database Layout

Table: generic_bar

chartId	rowId	name	value1	value2	value3
1	1	Widgets	17	25	32
1	2	Sprockets	19	28	35
1	3	Thingummies	13	20	30
2	1	Doodahs	20	29	21
2	2	Dohickies	11	12	11
2	3	Thingamajigs	32	17	12
2	4	Oojamaflips	14	15	21
3	1	Type1	12		
3	2	Type2	6		
3	3	Type3	2		
3	4	Type4	20		
3	5	TypeX	12		
3	6	TypeXI	15		
3	7	TypeXII	25		
3	8	Unknown	8		

Drawing Data Attributes

```
self.dataSource = ODBCDataSource()
self.dataSource.sql = 'SELECT chartId, name, value1, value2 FROM generic_bar'
self.dataSource.associations = Array(4, DataAssociation)
self.dataSource.groupingColumn = 0
self.dataSource.associations.element00 = DataAssociation(column=0,
                                                          target='chartId',
                                                          assocType='scalar')
self.dataSource.associations.element01 = DataAssociation(column=1,
                                                          target='HorizontalBarChart.categoryNames',
                                                          assocType='vector')
self.dataSource.associations.element02 = DataAssociation(column=[2],
                                                          target='HorizontalBarChart.data',
                                                          assocType='tmatrix')
self.dataSource.associations.element03 = DataAssociation(column=3,
                                                          target='HorizontalBarChart.labelRow2',
                                                          assocType='vector')
```

To Remove the Labels

```
HorizontalBarChart.labelRow2      = None
HorizontalBarChart.labelData      = None
HorizontalBarChart.categoryNames = None
```

change element01 and element03 in the dataSource.associations so that the target=None:

```
element01 = DataAssociation(column=1, target='None', assocType='vector')
element03 = DataAssociation(column=3, target='None', assocType='vector')
```

Description

A number of new things are introduced in this chart. The first one is the groupingColumn. Since each chart of this type uses data from more than one row in this table, our DataAwareDrawing needs to know which rows go together. The groupingColumn attribute is what provides this information. In this case (and most cases), the grouping column is '0' - the first one in the row (like many of the other attributes of our chart classes, counting these starts at 0). So, all the rows which have a chartID of 1 go into the same chart (Widgets, Sprockets, and Thingummies), all those whose chartID is 2 go into another and so on.

Unlike the previous two chart types, we have more than one assocType in this chart. We can now see both 'vector' and 'tmatrix' Data Associations in action.

vector

self.dataSource.associations.element01 takes a vector of column 1 and places this in the categoryNames attribute of the HorizontalBarChart widget.

Since this vector is specified as a single number (rather than a list - it's not surrounded by square brackets), it returns a column of data for that chart. So, for a chartId of '1', it returns:

```
['Widgets', 'Sprockets', 'Thingummies']
```

which become the categoryNames (the labels on the far left).

Shown as a diagram, this looks like this - the column outlined in red is the what is being placed into categoryNames. Note that we are only showing the columns that the SQL retrieved from the database, not all the columns that actually are in the database as above.

chartId	name	value1	value2
1	Widgets	17	25
1	Sprockets	19	28
1	Thingummies	13	20

self.dataSource.associations.element03 (the other vector) also does this, but for column 3 in the data rather than for column 1).

tmatrix

self.dataSource.associations.element02 takes a tmatrix ('transform matrix') of column 2 and places it into the 'data' attribute of the widget HorizontalBarChart.

The column specified is a list with one item in it ('[2]'), so the tmatrix returns:

```
((17.0, 19.0, 13.0),)
```

This can be imagined as the Data Association taking this red column:

chartId	name	value1	value2
1	Widgets	17	25
1	Sprockets	19	28
1	Thingummies	13	20

and converting it into this before passing it on:

17	19	13

It is also possible to specify more than one column by using a list with more than one item in it. If we used ('[2,3]'), the tmatrix would return:

```
((17, 19, 13), (25, 28, 20))
```

This can be thought of as taking these columns:

chartId	name	value1	value2
1	Widgets	17	25
1	Sprockets	19	28
1	Thingummies	13	20

and converting them to:

17	19	13
25	28	20

Both vectors and tmatrices can be a lot more complex than this - see chapter 6 for more details on that.

Other things of note include:

- **Axes**

  Most charts will be implemented using axes - whether they are visible or not.

  The chart HorizontalBarChart has two axes - the valueAxis (the one along the bottom), and the categoryAxis (the one up the side). In this chart, they are both invisible. Both these axes have the following attributes:

  ```
  visible
  visibleAxis
  visibleTicks
  ```

  The *visibleAxis* attribute controls whether the actual axis is drawn - i.e. the main line that the ticks hang off. The *visibleTicks* controls whether those ticks are displayed. The other attributes that control the ticks are *tickUp*/*tickDown* in the valueAxis, and *tickLeft*/*tickRight* in the categoryAxis. Unlike the 'visible' attributes we have just mentioned, these tick-related attributes are not Boolean. These 'tick' attributes take a number which is the length of the tick. An easy mistake to make is to set visibleTicks to 1, but have both tickLeft and tickRight set to 0. In this case, the ticks are displayed, but you can't see them due to them having a zero length.

  The *visible* attribute controls whether anything axis-related is drawn at all. This over-rides the other attributes - you might have visibleAxis set to 1, but if visible is set to 0, then it won't be displayed.

  (For more information about how to use Axes, look in section 5 of the ReportLab Graphics Guide, available on our web site at http://www.reportlab.com/download.html)

- **reverseDirection**

  This attribute is under the categoryAxis (i.e. it's HorizontalBarChart.categoryAxis.reverseDirection). It is available to all bar charts. As its name implies, if this is set to 1, it reverses the direction of the plot. The bars and their barLabels are then drawn in the order opposite to the way they are laid out in the data (e.g. [1,2,3] is reversed to [3,2,1]). This allows you to change if the bars are drawn from the bottom up or the top down (inside a chart).

- **labelRow2**

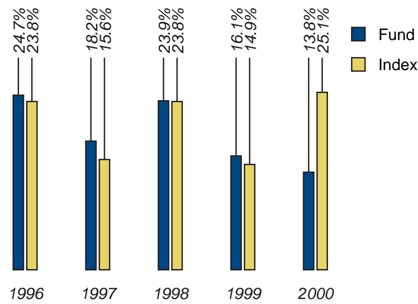
  This is an attribute available only to this particular chart (i.e. *rlextra/examples/graphics/horizontalbarchart_db.py*). What is does is allows you to have an optional second row of labels appearing to the right of the main labels. If labelRow2 is None or an empty list then no these labels do not appear.

  labelRow2 uses the same format as the main labels (the attribute *labelFormat*). For information on using format strings, look in section 4.5 (Formats and Formatters) in the chapter on 'Working with Charts'.

Like the other bar charts, the horizontalbarchart also has a number of attributes relating to the 'line'. The attributes *lineLength*, *lineStrokeWidth* and *lineColor* all refer to it. This line is the thin, horizontal line which goes from the tip of the actual bar to the labels on the right hand side. LineLength controls how long it is, lineColor controls what colour it is and lineStrokeWidth controls its width.

5.8 Example 4: Verticalbarchart.py

Example Chart



Database Layout

Table: generic_bar

chartId	rowId	name	value1	value2	value3
1	1	Widgets	17	25	32
1	2	Sprockets	19	28	35
1	3	Thingummies	13	20	30
2	1	Doodahs	20	29	21
2	2	Dohickies	11	12	11
2	3	Thingamajigs	32	17	12
2	4	Oojamaflips	14	15	21
3	1	Type1	12		
3	2	Type2	6		
3	3	Type3	2		
3	4	Type4	20		
3	5	TypeX	12		
3	6	TypeXI	15		
3	7	TypeXII	25		
3	8	Unknown	8		

Drawing Data Attributes

```
self.dataSource = ODBCDataSource()
self.dataSource.sql     = 'SELECT chartId, name, value1, value2 FROM generic_bar'
self.dataSource.groupingColumn = 0
self.dataSource.associations   = Array(3, DataAssociation)
self.dataSource.associations.element00 = DataAssociation(column=0,
                                          target='chartId',
                                          assocType='scalar')
self.dataSource.associations.element01 = DataAssociation(column=[2,3],
                                          target='VerticalBarChart.data',
                                          assocType='tmatrix')
self.dataSource.associations.element02 = DataAssociation(column=1,
                                          target='VerticalBarChart.categoryAxis.categoryNames',
                                          assocType='vector')
```

To Remove the Labels

```
VerticalBarChart.categoryAxis.categoryNames   = None
VerticalBarChart.barLabelFormat               = None
```

to remove the labels in the legend - change this line (in VerticalBarChart.legend):

```
colorNamePairs = [(PCMYKColor(100,65,0,30,spotName='PANTONE 288 CV'), 'Fund'),
                  (PCMYKColor(11,11,72,0,spotName='PANTONE 458 CV'), 'Index')]
to:
colorNamePairs = [(PCMYKColor(100,65,0,30,spotName='PANTONE 288 CV'), None),
                  (PCMYKColor(11,11,72,0,spotName='PANTONE 458 CV'), None)]
(i.e. change the strings inside the single quotes into 'None'(No quotes))
```

change element02 in the dataSource.associations so that the target=None:

```
element02 = DataAssociation(column=1, target=None, assocType='vector')
```

Description

Like the other bar charts, VerticalBarChart has a *reverseDirection* attribute. In this case it is under self.VerticalBarChart.categoryAxis.reverseDirection.

Under VerticalBarChart.barLabels are a number of attributes to do with the labels at the tops of the bars (the ones that the 'lines' lead to). VerticalBarChart.barLabels.angle is the angle that these labels are printed at - in this case 90°. The *dx* and *dy* attributes refer to the horizontal and vertical shifts that are applied to these labels away from the bars they refer to. A negative number moves it down/to the left of its bar, and a positive number moves it up/to the right. (For more details on these, refer to section 5.3 ('Labels') of the Graphics Guide and consult any autogenerated reference that you have).

While we are discussing barcharts, this is the place to mention three attributes of the valueAxis:

- *valueMin*
- *valueMax*
- *forceZero*

valueMin and ValueMax

If you *know* that your data will *always* be in a certain range, you can use *valueMin* and *valueMax*. ValueMin sets as an absolute the lowest value that will appear on the chart, and this always appears at the bottom of the chart. ValueMax does the same thing at the top of the chart (Of course, if you are using a horizontal barchart then top and bottom become left and right). If you don't absolutely know the boundaries of your data, then it is best to avoid using these. If you use them anyway, you may well find yourself in a situation where the bars flow outside the chart and its axes.

Given the above warning, *valueMin* and *valueMax* can be used to fix the position of an axis. If the values to be used are a mixture of positive and negative numbers, and you have not set the *valueMin* and *valueMax*, the axis will 'float' up and down - i.e. it will appear in different places in different charts, as dictated by the data. Setting these attributes can avoid this.

As an example, for the above chart if you decided to set the *valueMin* and *valueMax* to 40, you would see the following lines appear in the source window:

```
self.VerticalBarChart.valueAxis.valueMax          = 40
self.VerticalBarChart.valueAxis.valueMin          = 40
```

forceZero

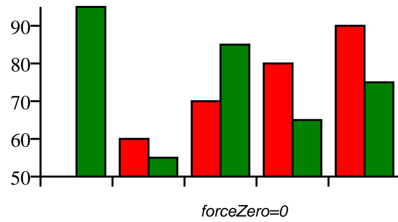
When the *valueMin* and *valueMax* haven't been set, the chart will set its own lower and upper limits. For example, if your data ranges from 50 to 100, the lower limit the chart chooses will be around 50. This can be misleading. ForceZero forces the lower limit to be 0, so the heights of the bars and the scaling are in a more straightforward proportion to each other. If you start getting unusual behaviour in your charts, try setting forceZero to 1 and seeing what effect this has.

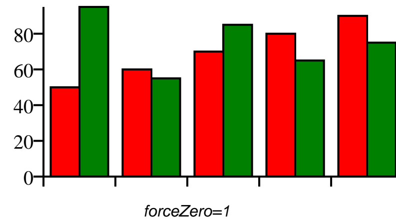
ForceZero takes a boolean argument - 1 sets it to 'on' and forces zero to appear on the value axis, 0, sets it to 'off' and allows the chart to omit zero. You can right click on the *forceZero* attribute in the attributes window of the Drawing Editor to set this.

As an example, if you decided to use *forceZero* for the above chart you would see this appear in the source window:

```
self.VerticalBarChart.valueAxis.forceZero          = 1
```

These two examples both have the same data set, but the one on the left has forceZero off, and the one on the right has it on. Other than this forceZero attribute, these are the same chart.





5.9 Example 5: Linechart.py

Example Chart

```
Line chart example would go here
```

Database Layout

Table: generic_time_series

chartId	date	value1	value3	value2
119	02/12/1997	1.00		1.00
119	31/12/1997	1.00		1.00
119	31/01/1998	1.00		1.01
119	28/02/1998	1.02		1.02
119	31/03/1998	1.01		1.01
119	30/04/1998	1.03		1.02
119	31/05/1998	1.03		1.02
119	30/06/1998	1.03		1.03
119	31/07/1998	1.03		1.03
119	31/08/1998	1.02		1.05
119	30/09/1998	1.05		1.09
...				
666	31/07/1992	1.03		1.13
666	31/08/1992	1.04		1.14
666	30/09/1992	1.04		1.14
666	31/10/1992	0.99		1.19
666	30/11/1992	1.05		1.25
666	31/12/1992	1.06		1.16

Drawing Data Attributes

```
self.dataSource      = ODBCDataSource()
self.dataSource.sql  = 'SELECT chartId, date, value1*100, value2*100, value3*100 FROM generic_time_ser
self.dataSource.associations   = Array(2, DataAssociation)
self.dataSource.associations.element00 = DataAssociation(column=0,
                                                         target='chartId',
                                                         assocType='scalar')
self.dataSource.associations.element01 = DataAssociation(column=[[1, 2], [1,3], [1,4]],
                                                         target='chart.data',
                                                         assocType='tmatrix')
```

Description

As standard in this chart class, there is the facility to take up to three lines from the data and plot them. If the column in the data is empty, it is ignored. To see an example of this, type in 'test=3' into the entry field - the chart graphic in the main the Drawing Editor window will change to one that uses the three columns of data from the table generic_time_series in sampledata.mdb

If you look under dataSource.associations, you will see the following line:

```
element01 = DataAssociation(column=[[1, 2], [1, 3], [1, 4]],
                            target='chart.data',
                            assocType='tmatrix')
```

In the chart data, the date (column 1 - the second column in the database) is used to provide the x-coordinate in sets of x-y pairs, which are then plotted as a line. Here is an example of what the data looks like when it is

inserted into chart.data - this is from test three again:

```
(
    ((19971202, 100.0), (19971231, 100.3), ... (20000630, 114.6),  (20000731, 113.6)),
    ((19971202, 100.0), (19971231, 101.0), ... (20000630, 133.58), (20000731, 132.60)),
    ((19971202, 100.0), (19971231, 107.0), ... (20000630, 123.58), (20000731, 122.60))
)
```

Sections omitted for brevity - indicated by '...'

Adding more lines

If three lines are not enough, you can add extra lines to this chart. To add them, first change the SQL under self.dataSource from:

```
sql = 'SELECT chartId, date, value1*100, value2*100, value3*100
                                          FROM generic_time_series'
```

to

```
sql = 'SELECT chartId, date, value1*100, value2*100, value3*100, value4*100
                                          FROM generic_time_series'
```

adding as many extra as you need. Then change the element under dataSource.associations from :

```
element01 = DataAssociation(column=[[1, 2], [1, 3], [1, 4]],
                            target='chart.data',
                            assocType='tmatrix')
```

to

```
element01 = DataAssociation(column=[[1, 2], [1, 3], [1, 4], [1,5]],
                            target='chart.data',
                            assocType='tmatrix')
```

Make sure you have enough columns in your database, otherwise you will get an error message when you try to test or run it:

```
Traceback (most recent call last):
File "c:\Python22\rlextra\graphics\guiedit\datacharts.py", line 572, in testConnect
  self.applyDataSet(sample)
File "c:\Python22\rlextra\graphics\guiedit\datacharts.py", line 646, in applyDataSet
  y = row[c]
dexError: tuple index out of range
```

Removing lines

The first and most obvious approach is to remove them from the database. If a column in the database is empty, the chart will ignore it.

A second way to change the data association so that the data isn't being used. E.g., to plot only one line, no matter what else is in the database, change:

```
element01 = DataAssociation(column=[[1, 2], [1, 3], [1, 4], [1,5]],
                            target='chart.data',
                            assocType='tmatrix')
```

to

```
element01 = DataAssociation(column=[[1, 2]], target='chart.data', assocType='tmatrix')
```

yet another way of doing it is to change the SQL. For example, change:

```
sql = 'SELECT chartId, date, value1*100, value2*100, value3*100 FROM generic_time_series'
```

to

```
sql = 'SELECT chartId, date, value1*100, null, value3*100 FROM generic_time_series'
```

The *null* in the SQL acts as a placeholder - it sits in your SELECT statement but is ignored.

Changing the colour of the lines (and other line attributes)

All the attributes to do with the lines in this chart are found under chart.lines. This is a collection of attributes - see section 4.4 Working with Collections for more general info about Collections.

If a line begins with the actual attribute name, then it applies to all the lines in the chart. So

```
strokeWidth = 1
```

sets the width of every line (or at least every line that is plotting data) to 1 point. If a line starts with a number inside square brackets, then it only applies to that one line. So

```
[1].strokeColor = ChbkYellow
```

changes the colour of the *second* line to the colour we have previously defined as ChbkYellow (since we count from 0). This happens to be the chartbook yellow we have imported from the file xxxcolors.py, but it could be any color that reportlab.graphics recognises.

If you set colours for a number of lines, and the actual number of lines on the chart is greater than that, the colours will 'rollover'. If you set chart.lines.[0]strokeColor to red and chart.lines.[1]strokeColor to blue, then proceed to actually have four lines plotted on your chart, then the first line will be red, the second blue, the third red again, and the fourth blue again.

One attribute that is a slightly harder to explain is the strokeDashArray. If you want to have a dotted or dashed line on your chart, you do this using the strokeDashArray. You give this an argument of a list of numbers (e.g. '[5,2]' or '[4,2,8,2]'). The first number is the number of points that that dash will be on for, the second is the number of points it will be off for. More complex patterns of numbers will produce more complex sequences of dots dashes.

The attributes under chart.lines are:

```
strokeDashArray
strokeWidth
strokeColor
```

Other attributes

- reversePlotOrder

  This controls the order in which lines are plotted over each other. If the line which is most important in the data is appearing underneath all the others change this from 0 to 1 (or vice versa). This appears under the attributes for the chart.

The following attributes are found under chart.yValueAxis:

- requiredRange

  The *requiredRange* is the least range that should be displayed. For example, if you have data which ranges from -1.2 to 0.75, if the chart is left to set its own limits these will look like very big swings. Giving a required range of 30 would (in this case) give you a range of -15 to +15 and put the numbers into perspective. The default value for requiredRange for this chart is 30.

- leftAxisPercent

  leftAxisPercent controls whether or not a percentage sign ('%') is added to the end of labels. It can be set to 0 or 1.

- leftAxisOrigShiftIPC and leftAxisOrigShiftMin

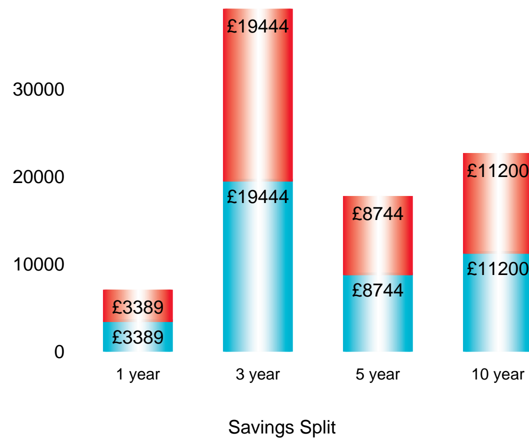
  These both refer to the lowest tick on the left axis of the chart. If it is 100 (or whatever your baseline is), it can require a 'shift' so that it doesn't appear on the bottom line of the grid. If you want this shift to take place, you need to set one of these attributes. Both perform the shift, but they differ on how it is specified. *leftAxisOrigShiftIPC* specifies it as a ration (IPC = 'In Per Cent'), and *leftAxisOrigShiftMin* specifies it an as amount.

- SkipLL0

  In some situations, you don't want the first tick label on the axis to appear. Setting *SkipLL0* to 1 is used to skip this first tick label.

5.10 Example 6: SectorCylinderChart

Example Chart



Description

This example is mainly here to show you how to use a couple of attributes that haven't been mentioned so far.

All barcharts have the attribute of *Symbol*. This is used where you want a bar to be represented by a graphic rather than the default rectangle that bars use. You can use:

```
Symbol = ShadedRect(Widget)
```

This then allows you access to the attributes under Symbol and so allows you to use various shading effects. Initially, you can set the start and end colours, which does pretty much as you would expect - the shading starts with the start colour, shades through the number of gradations set in the *numShades* attribute and then finishes with the end colour.

You can then go on to set the *cylinderMode* attribute. Set to 0, this gives you a simple shade (as we've just described). Set it to 1, and it shades from the start colour to the end colour and then back to the start colour again, giving the illusion of a cylinder.

You can set the *orientation* of the shading using the attribute of the same name. This can be set to 'horizontal' or 'vertical'. Horizontal has the gradation going from top to bottom, vertical has it going from side to side.

Another attribute you can set is *style* (under chart.categoryAxis). If you set this to 'parallel', having more than one series in the chart data produces bars which are side by side (as we saw in the verticalbarchart.py in example 5). If you set this to 'stacked', the behaviour is different. Multiple series in the data lead to having bars which are stacked on top of each other (or different coloured divisions of a single bar, if that's the way you see it).

Putting all these attributes together allows you to create a chart like the example above.

6 More about Using Axes

This section provides a pseudo-code description of the order in which various attributes are used in the most commonly used axis - 'ValueAxis'. This can be useful when trying to debug a new chart class or when figuring out how one works.

It also gives brief descriptions of some attributes which only appear in these axes (such as leftAxisOrigShiftIPC, and leftAxisSkipLL0).

Don't feel that you have to understand everything in this section. Feel free to skip it. The best way to understand how these work is by using them, or by experimenting with them in the Drawing Editor. This is more for use when you can't figure out why something is going wrong, rather than just figuring out how it works at all.

6.1 ValueAxis

```
ValueAxis:
setPosition (called by the containing chart)
    set the _x, _y & _length attributes passed from
    above
configure(data)  (called by the containing chart)
    _setRange
        compute _valueMin and _valueMax
        either from valueMin, valueMax or directly from the data.
        if _valueMin == _valueMax:
            force in a fake range of unit length.
        if forceZero:
            ensure _valueMin <= 0 <= _valueMax
        _rangeAdjust:
            do nothing method to allow hooking at this point
    _calcScaleFactor
        _scaleFactor=(_valueMax-_valueMin)/_length
    _calcTickmarkPositions
        if self.valueSteps is set:
            _tickValues = valueSteps
        else
            _calcValueStep
                if valueStep is set
                    _valueStep = valueStep
                else
                    rawRange = _ValueMax-_valueMin
                    n = min(maximumTicks-1, _length/minimumTickSpacing
                    rawInterval = rawRange/n
                    _valueStep = nextRoundNumber(rawInterval)
            ticks = []
            t = int(_valueMin/_valueStep)*_valueStep
            if t>=_ValueMin: append tick to ticks
            t += _valueStep
            while t<=_valuemax:
                append t to ticks
                t += _valueStep
            _tickValues = ticks
        _adjustAxisTicks
            in the base class this is a null hook method
            client classes can override to adjust the tick mark
            positions
    _configured = 1

draw
    if visible
        makeAxis
            if visibleAxis
                determine join from joinAxisMode, joinAxisPos and joinAxis
                draw the axis line

        makeTicks
            if visibleTicks
                for tick in _tickValues:
                    v = scaled tick
                    draw tick line using up/down and the scaled value

        makeTickLabels
            for tick _tickValues
                v = scaled tick
                if labelTextFormat is a string use as a format
                    txt = labelTextFormat % tick
```

```
elif labelTextFormat is a list
    txt = labelTextFormat[i] (i = tick sequence num)
elif labelTextFormat is callable
    txt = labelTextFormat(tick)
else arise value error
draw the label using the scaled value v
and using the labels[i] styles
```

NB: in the above if *_valueMin* and *_valueMax* are calculated from the data nothing forces them to be nice numbers so that then the axis may not start or end on tick values.

7 More on Data Sources

7.1 Making a Simple Chart Data Aware

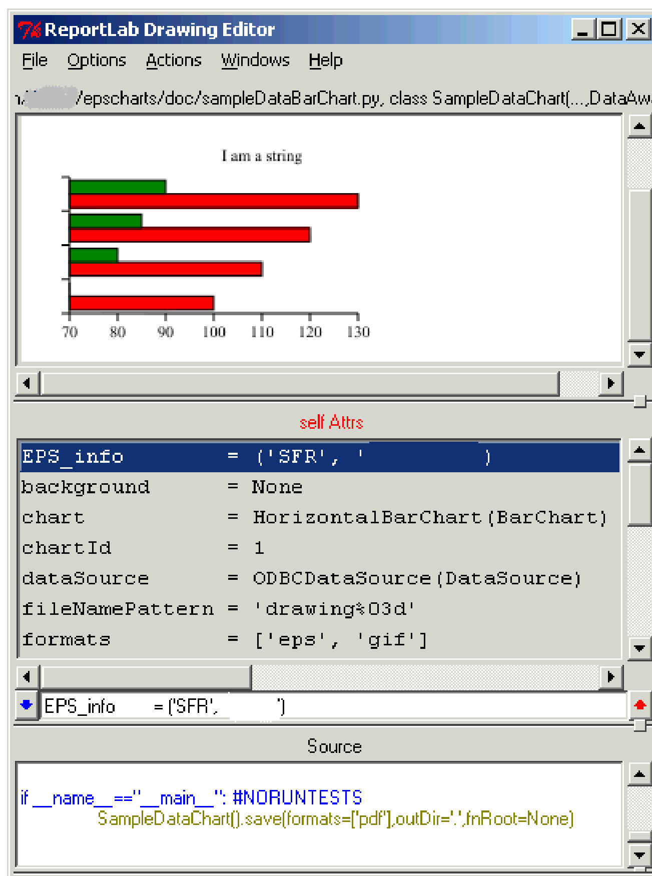
As an example, we are going to create a simple bar chart and make it pull data from the example Access Database (unpack sampledata.mdb from the file rlextra/examples/graphics/sampledata.zip). We'll assume you have set up the ODBC connection and that the name is 'samplechartdata'.

Start the Drawing Editor, and select 'New' from the File menu. When the 'New Project parameters' dialog box appears, select 'DataAwareDrawing', give it a name ('SampleDataChart'), and click on OK. Select 'Add New Widget', click on 'HorizontalBarChart', give it a name ('chart') and click on OK.

Click on `chart = HorizontalBarChart(BarChart)`, and change the x to be 75 and y to be 30 so it's a bit further from the edge.

Select 'Add New Widget' again, type in "String(125, 124, 'I am a string')" (since String has to be given its x and y coordinates and some text), give it a name ('title') and click on OK.

So far we have just set up a basic static bar chart. The rest of this example will be concerned with making it data aware.



Find the line that says

```
dataSource = DataSource(Widget)
```

Click on it, and change it so that it reads

```
dataSource = ODBCDataSource()
```

In the attributes window it will change so that it now reads

```
dataSource = ODBCDataSource(DataSource)
```

Double click on the data source line, and the attributes window will change to show you the dataSource attributes. Normally, you would have to change the 'name' line to point to the correct ODBC data source, but in this example file the default name is the correct one.

If you were using a stored procedure, you would also have to set the 'password' and 'user' attributes while you were here as well.

Change the sql line to read

```
sql = 'SELECT chartId, rowId, name, value1, value2, value3 FROM
generic_bar'
```

This line tells the chart where to get the data from. In this case it is selecting the columns called chartId, rowId (and so on) from the table called generic_bar in our Access database. We can get these names by looking at the table in the database and choosing which columns we need to use.

The groupingColumn is the column that we will use to decide which data will be in a chart. It might typically a fund name or ID, but could be anything. What matters is that it is the same for each element that will go in a single chart, but will not be repeated in other charts.

We do not have to change this, since it will remain as 0. When counting the columns, we start counting at 0 (as programmers do). Therefore, column 0 is the first one we retrieve in the SQL - chartId.

For this to actually allow us to do anything useful, we must associate it with the chartId of the chart. And to do this, you must change the number of Data Associations to something greater than 0.

Double-click on the line that says

```
associations = Array(0, DataAssociation)
```

you should now see a line that says

```
size = 0
```

Change this 0 to a number 4, and you will see a number of new lines appear, each one looking something like this:

```
element00 = DataAssociation(column=0, target=None,
assocType='vector')
```

Now we can make our first attribute data aware. Click on the first line 'element' line, and change it to read

```
element00 = DataAssociation(column=0, target='chartId',
assocType='scalar')
```

You can either change it in the text edit box below the attributes window, or you can double click on it and change each element separately.

The column number is the number of the column in the SQL statement that we set earlier. As already mentioned, 0 is the first column in the list in the SQL statement - chartId. Target is where that item of data is going to be 'plugged-into' in our chart. In this case, it's at the top level, so we can just type in 'chartId' without having to say which part of the chart it refers to.

This line should now read:

```
element00 = DataAssociation(column=0, target='chartId',
assocType='scalar')
```

If you go back to the top level of the chart, you can type 'test=1' into the entry field and hit enter. The chartId will change to 1. Nothing on the chart itself will change yet, since we haven't plumbed any of it in yet, but it's

enough to show you that the chart is pulling data from the database.

Now we can make the title data aware. Normally, you would have a separate field for the title, but in this example we are just going to use the name of the first column. To do this, double click on DataSource, then on associations to get you back into the attribute for self.dataSource.associations. Click on `element01`, and edit it so it reads

```
element01 = DataAssociation(column=2, target='title.text',
assocType='scalar')
```

The pulls the data from column 2 (the third column) in the SQL statement and fits it in as a single item into the attribute text of title (the String we added earlier).

Sometimes the best way of finding out what kind of data association is required for a chart is to look at the sample data it uses as a default, and see which kind of data association produces that kind of output. Then you can figure out which columns (or rows) from the database you need to specify.

We have seen scalars in use - here is a quick run down of the other data associations:

scalar

We have already seen what a scalar is - it's basically just a single variable. You can think of it as a 1x1 matrix if you want, but basically it's just a single box which is pulled from the database and fitted into a single slot in the chart.

Examples of something we can use a scalar for:

```
text        = 'Widgets'
x           = 125
```

Typical use for a scalar:
String.text, Label.text, chartId

vector

Vectors are useful when you need to pull a column of data from a database, and use all of it (or at least all that is in that set, as defined by the groupingColumn).

If you give a single number for the column, then one column will be returned as a list (it will be surrounded by brackets). So, for something like this:

```
DataAssociation(column=1, target='HorizontalBarChart.categoryNames',
                assocType='vector')
```

you might get something back looking like this:

```
['Widgets', 'Sprockets', 'Thingummies']
```

Examples of something we can use a vector with a single number for:

```
categoryNames     = ('Cash', 'Other', 'Italy', 'Switzerland')
labelRow2         = [5.3, 3.3, 2.4, 4.5]
```

If you give a more than one number for the column, they must be separated by a comma and enclosed in square brackets. Then the data association will return a list (enclosed by square brackets) containing tuples of the columns (enclosed by parentheses).

For something like this:

```
DataAssociation(column=[1, 2], target='data', assocType='vector')
```

you might get something back looking like this:

```
[('a01', 'a02'), ('a11', 'a12'), ('a21', 'a12'), ('a11', 'a12')]
```

Examples of something we can use a vector with a numbers in square brackets for:

```
data         = [('a01', 'a02'), ('a11', 'a12'), ('a21', 'a12'), ('a11', 'a12')]
```

Typical use for a vector:
chart categoryNames, horizontal barchart data, pie chart data and legend names

matrix

Matrices aren't very often used. More often you will be using a tmatrix (see below).

For a description of a matrix, look in section 7.3 (Data Association Types)

tmatrix

A tmatrix is a 'transformed matrix'. It is used to convert a vertical column (or columns) into a horizontal list.

The most common use is where the column is given as a single number inside square brackets (a list), the tmatrix returns a tuple inside another tuple (parentheses inside other parentheses). So for something like this:

```
column=[13], target='data', assocType='tmatrix'
```

You might get back something looking like this:

```
((0.6, -1.3, -0.7, -0.4),)
```

Examples of something we can use a tmatrix with a single number for (as seen in the default data):

```
data            = ((0.0, -0.10, 0.0, 0.3, 0.3),)
data            = [(3.5, 15.5, 6.0, 7.5)]
```

For charts where you need two data sets (e.g., where you are plotting one set of bars against another), the column specification would be two numbers inside the square brackets:

```
column=[8, 7], target='data', assocType='tmatrix'
```

would give you something like:

```
data           = ((-10.9, -7.1, 2.9), (9.6, 13.2, 29.5, -6.2))
```

Typical use for a tmatrix:
chart.data (for various barcharts)

rowmap

Rowmap is seldom used. It used for picking out individual cells in a row and stringing them together, rather than selecting a whole column.

Typical use for a rowmap:
chart.data

If you need more information and examples on any one of these data associations, read section 7.3.

The sample data in our example barchart looks like this:

```
data            = [(100, 110, 120, 130), (70, 80, 85, 90)]
```

So, it is a possibility that it is either a vector or a tmatrix. Since tmatrices are used for the chart data for barcharts, we can use one of them.

Looking at the SQL statement will tell us what we need to plot and what positions it is in.

```
sql = 'SELECT chartId, rowId, name, value1, value2, value3 FROM
generic_bar'
```

We can see that we have value1, value2 and value3 to plot on this chart, in positions 3, 4 and 5.

So we need to change a data association to read

```
element02 = DataAssociation(column=[3,4,5], target='chart.data',
                            assocType='tmatrix')
```

The last thing we will do is to will do is set up a data association for the bar category names (the names at the left of each group of bars).

The names appear the third column of the data, so (since we count from 0) this becomes a column specification of 2:

```
element03 = DataAssociation(column=2,
                            target='chart.categoryAxis.categoryNames',
                            assocType='vector')
```

Going to the top level and doing test=1 (or 2 or 3) shows us that the it works and is pulling data for the chart, the title and the category labels from the database.

This example is saved as the file `rlextra/examples/graphics/sampleDataBarChart.py` if you wish to play with it further.

There are a number of other things you can do to make it better - making it verbose so you can see it saving the files when you run it from the command line, changing the fileNamePattern, changing the bar colours so they look better and so on. However, it is data aware and works, so anything else doesn't really belong in this section.

7.2 Changing the Data Associations on an Existing Chart

There may be a number of ways you need to change the data associations on an existing chart. You may need to:

- Add a new column and data association,
- Remove an existing column and data association
- Change an existing column and data association in place
- Change the input to a stored procedure

Adding a new column and data Association

If someone has added an extra column to the database (or altered the stored procedure to return an extra column), and you need to make us of it, you need to do the following.

- Edit the SQL to add in the new item,
- Add a new Data Association,
- Change the new Data Association to retrieve the correct data and target it to the correct chart attribute

Removing an existing column and data association

If you no longer need to use something stored in the database (or being returned by a stored procedure), you need to

- Edit the SQL to remove the item you no longer require,
- Change the Data Association
- Either make sure the target has something sensible in it or remove it altogether

To edit the SQL, double-click on the line '`dataSource = ODBCDataSource(DataSource)`' in the attributes window, then double-click on the line that begins with '`sql = `'. You can then remove the name of the item you no longer need. This will no longer be retrieved from the data base.

To edit the data associations, find the line in the source window that refers to the item you want to remove, right click on it and click on the button labelled 'Remove'. This data association then disappears completely. You will notice that in the attributes window, it changes to something like this:

```
element04 = DataAssociation(column=0, target=None, assocType='vector')
```

If no other data associations below it are actually being used, you can change the size attribute to remove them as well.

To do this, go back to the top level (so the header of the Attributes window says 'self.Attr'), then double-click on the line that says something like '`associations = Array(3, DataAssociation)`', click on the line that says '`size =`' and edit the number.

A simpler, 'quick and dirty' way of doing this is to edit the Data Association so that the target reads 'None'. The data is still retrieved from the database, but is not used. This is only suitable if you don't want something to appear in your chart any longer but the structure of the data in the database will not change.

If you are removing the data association because you want that element to be static (for example, a title which will now be the same on every chart rather than being data aware), you should edit it in the attributes window so that it contains what you want. If you want to remove it completely, find any lines referring to it in the source window, right click on them and click on the 'Remove' button. Once all the lines that refer to it have been removed, you can remove the line that added it to your chart in the first place (the one that begins with '`self._add(self, `' and then the name of that element and its attributes).

Changing an existing column and data association in place

Usually this will consist of either adding or removing data associations. If you use the column names (rather than a '*') in the SQL, changing the order of the columns in the database should not break anything.

7.3 Data Association Types

This section gives a more formalized explanation of how Data Associations work, with more varied examples. We covered how most of them work in the various worked examples and the previous section. You can skip this section unless you need it.

Assume we have a bunch of columns:

1	a01	a02	a03	a04
1	a11	a12	a13	a14
1	a21	a12	a13	a24
1	a11	a12	a13	a34
2	b01	b02	b03	b04
2	b11	b12	b13	b14
2	b21	b12	b13	b24
2	b11	b12	b13	b34
3	c01	c02	c03	c04
3	c11	c12	c13	c14
3	c21	c12	c13	c24
3	c11	c12	c13	c34

scalar:

the column specification must be a scalar. The associated target is set to the value of that column in the first row of the data block. Typically the chartId is a scalar, it normally has a data association like

```
DataAssociation(column=0, target='chartId', assocType='scalar')
```

Assuming that the grouping column is the default=1 then the row grouping will mean that chartId will receive the values i1, i2 and i2 with each data block. Note that although we see more than one value in column 0 corresponding to the rows i1 only the first of such rows is used for a scalar. So another scalar defined by

```
DataAssociation(column=1, target='bongo', assocType='scalar')
```

would mean that self.bongo received the values a01, b01 and c01.

vector:

the column specification can be a scalar or a sequence. With a scalar the target receives a row vector corresponding to the selected column i.e. all the rows in the group. When a list is used the elements of the output vector become lists themselves.

```
SELECT Id, C1, C2 FROM example
column=0, target='chartId', assocType='scalar'
column=1, target='data', assocType='vector'
chartId:1
    data:['a01', 'a11', 'a21', 'a11']

chartId:2
    data:['b01', 'b11', 'b21', 'b11']

chartId:3
    data:['c01', 'c11', 'c21', 'c11']

SELECT Id, C1, C2 FROM example
column=0, target='chartId', assocType='scalar'
column=[1, 2], target='data', assocType='vector'
chartId:1
    data:[('a01', 'a02'), ('a11', 'a12'), ('a21', 'a12'), ('a11', 'a12')]

chartId:2
    data:[('b01', 'b02'), ('b11', 'b12'), ('b21', 'b12'), ('b11', 'b12')]

chartId:3
```

```
    data:[('c01', 'c02'), ('c11', 'c12'), ('c21', 'c12'), ('c11', 'c12')]
```

matrix:

the column specification can be a scalar, a list or a list of lists. A scalar is treated as a single element list.

as an example:

```
SELECT Id, C1, C2, C3  FROM example
column=0, target='chartId', assocType='scalar'
column=1, target='cat', assocType='matrix'
column=[2, 3], target='dog', assocType='matrix'
column=[[1, 2], [2, 3]], target='data', assocType='matrix'
chartId:1
cat:(('a01',), ('a11',), ('a21',), ('a11',))
dog:(('a02', 'a03'), ('a12', 'a13'), ('a12', 'a13'), ('a12', 'a13'))
data:((('a01', 'a02'), ('a02', 'a03')),
 (('a11', 'a12'), ('a12', 'a13')),
 (('a21', 'a12'), ('a12', 'a13')),
 (('a11', 'a12'), ('a12', 'a13')))

chartId:2
cat:(('b01',), ('b11',), ('b21',), ('b11',))
dog:(('b02', 'b03'), ('b12', 'b13'), ('b12', 'b13'), ('b12', 'b13'))
data:((('b01', 'b02'), ('b02', 'b03')),
 (('b11', 'b12'), ('b12', 'b13')),
 (('b21', 'b12'), ('b12', 'b13')),
 (('b11', 'b12'), ('b12', 'b13')))

chartId:3
cat:(('c01',), ('c11',), ('c21',), ('c11',))
dog:(('c02', 'c03'), ('c12', 'c13'), ('c12', 'c13'), ('c12', 'c13'))
data:((('c01', 'c02'), ('c02', 'c03')),
 (('c11', 'c12'), ('c12', 'c13')),
 (('c21', 'c12'), ('c12', 'c13')),
 (('c11', 'c12'), ('c12', 'c13')))
```

tmatrix:

the column specification can be a scalar, a list or a list of lists. A scalar is treated as a single element list.

When the column spec is a simple list as in HorizontalBarChart we get

```
SELECT Id, C1, C2, C3, C4 FROM example
column=0, target='chartId', assocType='scalar'
column=1, target='cat', assocType='vector'
column=[2], target='data', assocType='tmatrix'
column=4, target='label', assocType='vector'
chartId:1
    cat:['a01', 'a11', 'a21', 'a11']
    data:(('a02', 'a12', 'a12', 'a12'),)
    label:['a04', 'a14', 'a24', 'a34']

chartId:2
    cat:['b01', 'b11', 'b21', 'b11']
    data:(('b02', 'b12', 'b12', 'b12'),)
    label:['b04', 'b14', 'b24', 'b34']

chartId:3
    cat:['c01', 'c11', 'c21', 'c11']
    data:(('c02', 'c12', 'c12', 'c12'),)
    label:['c04', 'c14', 'c24', 'c34']
```

As an example consider treating the above data as a time series with dates in column 1 i.e. a01 is a date. The data are in columns 2, 3, 4 so we use pairs [1,2], [1,3] and [1,4], this is used in linechart.py. For a single chart we should have a list of lists of pairs.

```
SELECT Id, C1, C2, C3, C4 FROM example
column=0, target='chartId', assocType='scalar'
column=[[1, 2], [1, 3], [1, 4]], target='data', assocType='tmatrix'
chartId:1
data:
    (((a01, a02), (a11, a12), (a21, a12), (a11, a12)),
    ((a01, a03), (a11, a13), (a21, a13), (a11, a13)),
    ((a01, a04), (a11, a14), (a21, a24), (a11, a34)))

chartId:2
data:
```

```
    (((b01, b02), (b11, b12), (b21, b12), (b11, b12)),
    ((b01, b03), (b11, b13), (b21, b13), (b11, b13)),
    ((b01, b04), (b11, b14), (b21, b24), (b11, b34)))

chartId:3
data:
    (((c01, c02), (c11, c12), (c21, c12), (c11, c12)),
    ((c01, c03), (c11, c13), (c21, c13), (c11, c13)),
    ((c01, c04), (c11, c14), (c21, c24), (c11, c34)))
```

rowmap:

the column specification can be a list or a list of lists.

Rowmaps are used to pick individual elements out of a row of data and convert them into a list or list of lists.

```
SELECT Id, C1, C2, C3, C4  FROM example
column=0, target='chartId', assocType='scalar'
column=[1, 2, 3, 4], target='cat', assocType='rowmap'
column=[1, 3], target='dog', assocType='rowmap'
column=[[2, 3], [1, 4]], target='data', assocType='rowmap'
chartId:1
cat:['a01', 'a02', 'a03', 'a04']
dog:['a01', 'a03']
data:[['a02', 'a03'], ['a01', 'a04']]

chartId:2
cat:['b01', 'b02', 'b03', 'b04']
dog:['b01', 'b03']
data:[['b02', 'b03'], ['b01', 'b04']]

chartId:3
cat:['c01', 'c02', 'c03', 'c04']
dog:['c01', 'c03']
data:[['c02', 'c03'], ['c01', 'c04']]
```

The following is an example of one in use, this is from the QRI chart drawing2_returns.py:

```
dataSource.associations.element01.assocType='rowmap'
dataSource.associations.element01.column    = [4,7,10,13,16]
dataSource.associations.element01.target    = 'chart.categoryAxis.categoryNames'
```

8 New Drawing Editor Features

8.1 Test Mode

This isn't actually a new feature, but it has been expanded slightly. It is also mentioned here since it works well with the new diagnostic features mentioned below. Setting the chart attribute 'test' to 1 will fetch the first set of data from the database in a data aware chart. Setting it to 2 will fetch the second, and so on. This is very useful when making sure that charts actually work.

One new feature of the test mode is that any 'test' attributes which are set can be automatically stripped out of the chart code. If you have a number of tests set saved in the code it can both bloat the size of the code, and fix the results of the chart rather than having them be dynamic. For example, if test was set to three then all charts produced with that chart class would always be from the third set of data pulled from the database.

To turn this on or off, select 'Filter test attr' from the Options menu.

8.2 Errors

If you have the 'log' window minimized to conserve space, even having error messages appearing in red text won't draw them to your attention. The Drawing Editor now pops-up a requester telling you both that you have had an error and what it is. If you want more details on it, you can then look in the log window.

When one of these pops-up, the Drawing Editor will also beep to get your attention. If you are finding this annoying, you can toggle it off or on using the 'Error bells' item in the Options menu.

8.3 Diagnostics

Two new diagnostic features have been added to Guiedit: Debug_memo and a diagnostic chart.

Debug_Memo

If a data aware chart is crashing midway through a chart run, we need a way of gathering as much information as possible about the cause. Debug_memo is a way of doing this.

To switch on DebugMemo, edit the line in the chart class (from the source window, by right clicking on it and changing it in the Alter Edit dialog box) that has the 'go' method. This will be the last line in the file (or near it), and will look something like this:

```
SampleDataChart().go()
```

Add in 'debug=1', so it looks something like this:

```
SampleDataChart().go(debug=1)
```

The next time the chart has a problem, it should save a debug file. It's file name will be in the form of 'RL_classname_pickle.dbg', so for the above example you would have a file called 'RL_SampleDataChart_pickle.dbg'.

The following information will probably not be suitable for users, but may be of assistance to anyone supporting the chart classes with problems.

To see the output from a debug memo, start up Python and do the following (where yourDebugFileName is the name of the file produced by DebugMemo):

```
from reportlab.lib.utils import DebugMemo
dbg = DebugMemo(fn=yourDebugFileName,mode='r')
dbg.load()
dbg.show()
```

This will show information including the traceback produced, the data which produced the error, any command line arguments, the version numbers for various files and so on.

The Diagnostic chart

We have produced a chart which will show a number of useful pieces of information about your system. These include the current working directory, any arguments used from the sql of that chart and the path.

This chart is called `diagnostic.py` and is located in the `rlextra\graphics\guiedit` directory (under your Python directory).

If you ever need to send it (either to us or whoever is providing support for your charts), make sure you export it to a suitable format (e.g. GIF) and send that image.

9 Using 'Knockout'

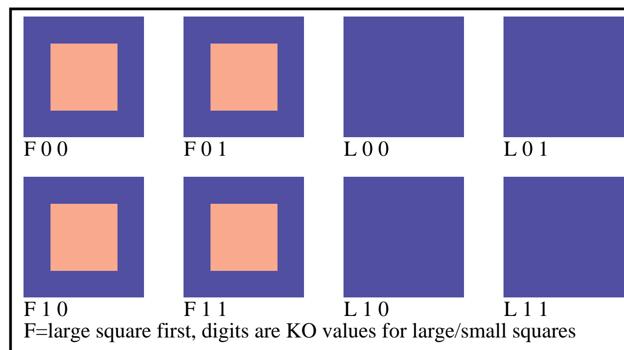
9.1 'Knockout' - what it is, and why you might want to use it

Simply put, 'knockout' is when a colour which overlays another colour 'knocks out' anything below it. It's the removal of colour in one area to accommodate another colour (i.e. it's an area of a background colour that does not print in that colour).

Using knockout adds the complication that the version of document (or graph or diagram) that you see in guiedit will not necessarily match the one you get when you print it out. What are the advantages? If you are using a printer who prints the black ('key') plate last, any black lines or grids used in construction of graphs and charts will appear *over* the bars or lines that should be the focus of the chart. Knockout removes any lines under the bars (or pie etc) so that they won't 'show through'.

Note that knockout is renderer specific. The whole concept is only useful in a renderer that supports it. Currently it is only supported in renderPS_SEP - our Postscript renderer. This means that it isn't possible to use knockout with bitmaps (and isn't currently supported in PDF, though that may change in the future).

Consider the following example, which illustrates all aspects of knockout with two squares. It uses non-spot colours with the two colours overlapping, providing 8 combinations. This is how it would look in Guiedit.



Knockout example

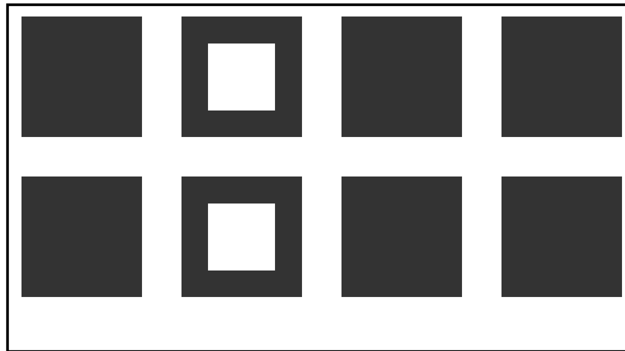
Going from left to right, top to bottom, we can see that from the labelling that:

- F00 has the large square drawn first and no knockout,
- F01 has the large square drawn first and the knockout on for the small square,
- L00 has the small square drawn first and no knockout,
- L01 has the small square drawn first and the knockout on for the small square,
- F10 has the large square drawn first and the knockout on for the large square,
- F11 has the large square drawn first and the knockout on for both squares,
- L10 has the small square drawn first and the knockout on for the large square,
- L11 has the small square drawn first and the knockout on for both squares.

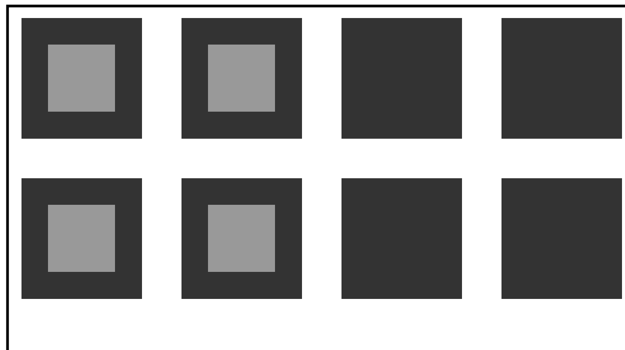
Notice how in Guiedit, when the large square is drawn first, the smaller square always appears on top of it, and when the smaller square is drawn first the larger square always overwrites it.

The four illustrations on the next page show how it looks when split into its constituent colour separations.

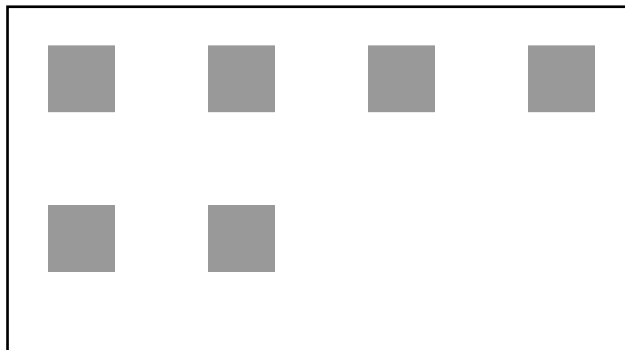
Notice how where the little square has the knockout set, it 'punches out' a hole in anything below it. The cyan separation of the squares labelled F01 and F11 show this quite well, where the small squares have knocked out areas of the large squares.



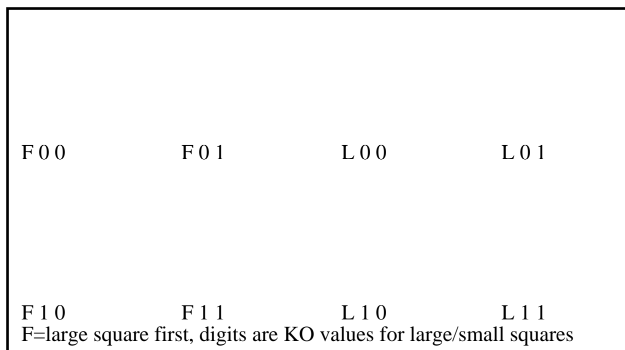
Knockout example: Cyan separation



Knockout example: Magenta separation



Knockout example: Yellow separation



Knockout example: Black ('Key') separation

L01 and L11 also have the knockout set on the small squares, but nothing is punched out of the large square for these. This is because our implementation of knockout is dependant on the *ordering* of objects.

When knockout is set on an object, it will knockout anything which has already been drawn (i.e. 'below' it in the ordering of items on the canvas). In these L01 and L11, the large square hasn't been drawn yet when the small one is created, so there is nothing there to knockout.

This is also illustrated by the squares labelled L10 and L11 in the Yellow separation. Because the large square has knockout set and is drawn second, it knocks out the smaller square - and since the smaller square is smaller, it is removed completely.

9.2 How to use knockout

OK, so now we know what knockout is, how do we actually use it?

We saw about the *spotName* and *density* attributes for colours (CMYK and PCMYK colors) in section 4.2 ("Working with Colours"). There is another optional attribute called 'knockout'. This parameter is optional, if it isn't specified it defaults to None. None in this case means 'use the default' (see below).

This means that knockout is an attribute of a *colour* rather than a shape or object. If knockout is set to 1, then all objects using that colour will knockout anything underneath them in the way we've already seen.

As an example, this is how the colour used for the big squares in the example is defined:

```
PCMYKColor(100,100,0,0,density=80,knockout=0)
```

The default is to have knockout set to 1 - so everything 'knocks out'. This can be changed by adding a line to the file rl_config.py. If you add the line :

```
knockout = 0
```

to the rl_config file, knockout will be turned off (and 'knockout = 1' turns it back on again). Omitting it altogether also turns it back on by reverting to the default.