User Guide

ReportLab Version 1.09

Lombard Business Park
8 Lombard Road
Wimbledon
London, ENGLAND SW19 3TZ

103 Bayard Street
New Brunswick
New Jersey, 08904)
USA

Chapter 1 Introduction

1.1 About this document

This document is intended to be a conversational introduction to the use of the ReportLab packages. Some previous programming experience is presumed and familiarity with the Python Programming language is recommended. If you are new to Python, we tell you in the next section where to go for orientation.

After working your way throught this, you should be ready to begin writing programs to produce sophisticated reports.

In this chapter, we will cover the groundwork:

- What is ReportLab all about, and why should I use it?
- What is Python?
- How do I get everything set up and running?

Be warned! This document is in a very preliminary form. We need your help to make sure it is complete and helpful. Please send any feedback to our user mailing list, reportlab-users@yahoogroups.com.

1.2 What is ReportLab?

ReportLab is a software library that lets you directly create documents in Adobe's Portabe Document Format (PDF) using the Python programming language.

PDF is the global standard for electronic documents. It supports high-quality printing yet is totally portable across platforms, thanks to the freely available Acrobat Reader. Any application which previously generated hard copy reports can benefit from making PDF documents instead; these can be archived, emailed, placed on the web, or printed out the old-fashioned way. However, the PDF file format is a complex indexed binary format which is impossible to type directly. The PDF format specification is more than 600 pages long and PDF files must provide precise byte offsets -- a single extra character placed anywhere in a valid PDF document can render it invalid. Until now, most of the world's PDF documents have been produced by Adobe's Acrobat tools, which act as a 'print driver'.

The ReportLab library directly creates PDF based on your graphics commands. There are no intervening steps. Your applications can generate reports extremely fast - sometimes orders of magnitude faster than traditional report-writing tools.

By contrast, many other methods for generating PDF documents involve "pipelines" of several processes, which make the generation process slow, and very difficult to manage and maintain.

In addition, because you are writing a program in a powerful general purpose language, there are no restrictions at all on where you get your data from, how you transform it, and the the kind of output you can create. And you can reuse code across whole families of reports.

The ReportLab library is expected to be useful in at least the following contexts:

- Dynamic PDF generation on the web
- High-volume corporate reporting and database publishing
- An embeddable print engine for other applications, including a 'report language' so that users can customize their own reports. *This is particularly relevant to cross-platform apps which cannot rely on a consistent printing or previewing API on each operating system*.
- A 'build system' for complex documents with charts, tables and text such as management accounts, statistical reports and scientific papers
- Going from XML to PDF in one step!

1.3 What is Python?

> **python**, (*Gr. Myth.* An enormous serpent that lurked in the cave of Mount Parnassus and was slain by Apollo) **1.** any of a genus of large, non-poisonous snakes of Asia, Africa and Australia that suffocate their prey to death. **2.** popularly, any large snake that crushes its prey. **3.** totally awesome,

> bitchin' very high level programming language (which in *our* exceedingly humble opinions (for what they are worth) whallops the snot out of all the other contenders (but your mileage may vary real soon now, as far as we know).

Python is an *interpreted, interactive, object-oriented* programming language. It is often compared to Tcl, Perl, Scheme or Java.

Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface.

The Python implementation is portable: it runs on most brands of UNIX (including clones such as Linux), on Windows, DOS, OS/2, Mac, Amiga, DEC/VMS, IBM operating systems, VxWorks, PSOS, ... If your favorite system isn't listed here, it may still be supported, if there's a C programming language compiler for it. Ask around on comp.lang.python -- or just try compiling Python yourself.

Python is copyrighted but **freely usable and distributable, even for commercial use**. The ReportLab core modules share the same copyright with the name of the copyright holder modified. Both packages use the "Berkeley Standard Distribution (BSD) style" free software copyright.

1.4 Installation and Setup

Below we provide an abbreviated setup procedure for Python experts and a more verbose procedure for people who are new to Python.

Installation for experts

First of all, we'll give you the high-speed version for experienced Python developers:

1. Install Python 1.5.1 or later
2. If you want to produce compressed PDF files (recommended), check that zlib is installed.
3. If you want to work with bitmap images, install and test the Python Imaging Library
4. Unpack the reportlab package (reportlab.zip or reportlab.tgz) into a directory on your path
5. `cd` to *`reportlab/pdfgen/test`* and execute `testpdfgen.py`, which will create a file 'testpdfgen.pdf'.

If you have any problems, check the 'Detailed Instructions' section below.

A note on available versions

The `reportlab` library can be found at `ftp.reportlab.com` in the top-level directory. Each successive version is stored in both zip and tgz format, but the contents are identical. Versions are numbered: `ReportLab_0_85.zip`, `ReportLab_0_86.zip` and so on. The latest stable version is also available as just `reportlab.zip` (or `reportlab.tgz`), which is actually a symbolic link to the latest numbered version.

We also make nightly snapshots of our CVS (version control) tree available. In general, these are very stable because we have a comprehensive test suite that all developers can run at any time. New modules and functions within the overall package may be in a state of flux, but stable features can be assumed to be stable. If a bug is reported and fixed, we assume people who need the fix in a hurry will get `current.zip`

Instructions for novices: Windows

This section assumes you don't know much about Python. We cover all of the steps for three common platforms, including how to verify that each one is complete. While this may seem like a long list, everything takes 5 minutes if you have the binaries at hand.

1. Get and install Python from `http://www.python.org/.` Follow the links to 'Download' and get the latest official version. Currently this is Python 1.5.2 in the file `py152.exe`. It will prompt you for a directory location, which by default is `C:\Program Files\Python`. This works, but we recommend entering `C:\Python15`. Python 1.6 will be out shortly and will adopt `C:\Python16` as its default; and quite often one wants to change directory into the Python directory from a command prompt, so a path without spaces saves a lot of typing! After installing, you should be able to run the 'Python (command line)' option from the Start Menu.
2. If on Win9x, we recommend either copying python.exe to a location on your path, or adding your Python directory to the path, so that you can execute Python from any directory.
3. If you want a nice editing environment or might need to access Microsoft applications, get the Pythonwin add-on package from the same page. Once this is installed, you can start Pythonwin from the Start Menu and get a GUI application.

The next step is optional and only necessary if you want to include images in your reports; it can also be carried out later.

4. Install the Python Imaging Library (`PIL`). (todo: make up a bundle that works)
5. Add the `DLL`s in `PIL` to your `Python\DLLs` directory
6. To verify, start the Python interpreter (command line) and type `from PIL import Image`, followed by `import _imaging`. If you see no error messages, all is well.

Now you are ready to install reportlab itself.

7. Unzip the archive straight into your Python directory; it creates a subdirectory named `reportlab`. You should now be able to go to a Python command line interpreter and type `import reportlab` without getting an error message.
8. Open up a `MS-DOS` command prompt and CD to "..\reportlab\pdfgen\test". On NT, enter "testpdfgen.py"; on Win9x, enter "python testpdfgen.py". After a couple of seconds, the script completes and the file testpdfgen.pdf should be ready for viewing. If PIL is installed, there should be a "Python Powered" image on the last page. You're done!

[Note: the "couple of seconds" delay is mainly due to compilation of the python scripts in the ReportLab package. The next time the ReportLab modules are used the execution will be noticably faster because the `pyc` compiled python files will be used in place of the `py` python source files.]

Instructions for Python novices: Unix

1. First you need to decide if you want to install the Python sources and compile these yourself or if you only want to install a binary package for one of the many variants of Linux or Unix. If you want to compile from source download the latest sources from http://www.python.org (currently the latest source is in http://www.python.org/ftp/python/src/py152.tgz). If you wish to use binaries get the latest RPM or DEB or whatever package and install (or get your super user (system administrator) to do the work).
2. If you are building Python yourself, unpack the sources into a temporary directory using a tar command e.g. `tar xzvf py152.tgz`; this will create a subdirectory called Python-1.5.2 (or whatever) cd into this directory. Then read the file `README`! It contains the latest information on how to install Python.
3. If your system has the gzip libz library installed check that the zlib extension will be installed by default by editing the file Modules/Setup.in and ensuring that (near line 405) the line containing zlib zlibmodule.c is uncommented i.e. has no hash '#' character at the beginning. You also need to decide if you will be installing in the default location (/usr/local/) or in some other place. The zlib module is needed if you want compressed PDF and for some images.
4. Invoke the command `./configure --prefix=/usr/local` this should configure the source directory for building. Then you can build the binaries with a `make` command. If your `make` command is not up to it try building with `make MAKE=make`. If all goes well install with `make install`.
5. If all has gone well and python is in the execution search path you should now be able to type `python` and see a **Python** prompt. Once you can do that it's time to try and install ReportLab. First get the latest reportlab.tgz. If ReportLab is to be available to all then the reportlab archive should be unpacked in the lib/site-python directory (typically /usr/local/lib/site-python) if neccessary by a

superuser. Otherwise unpack in a directory of your choice and arrange for that directory to be on your `PYTHONPATH` variable.

```
#put something like this in your
#shell rcfile
PYTHONPATH=$HOME/mypythonpackages
export PYTHONPATH
```

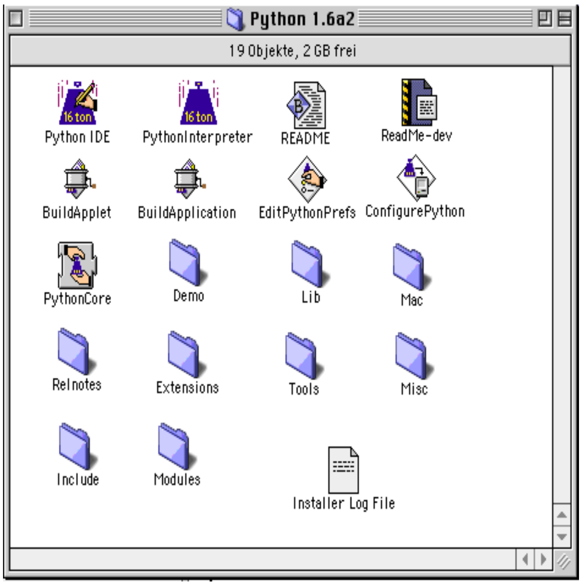
You should now be able to run python and execute the python statement

```
import reportlab
```

6. If you want to use images you should certainly consider getting & installing the Python Imaging Library from http://www.pythonware.com/products/pil.

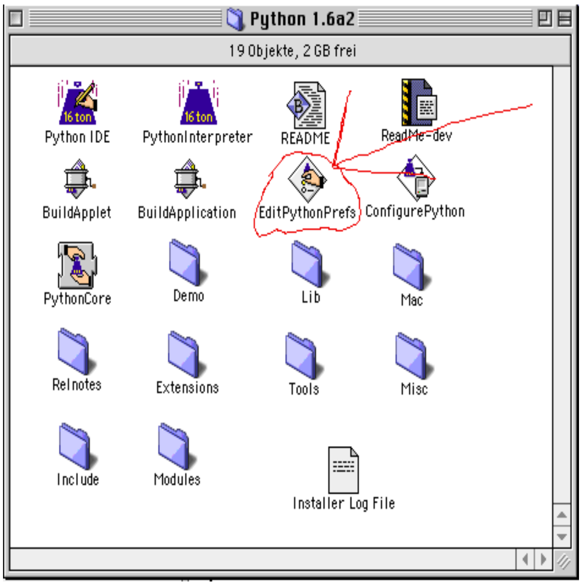
Instructions for Python novices: Mac

First install Python, the latest stable release is 1.52, but it is also possible to run Reportlab with 1.6a2 and probably with 1.6b1/b2. You get the software (ready to run) at font color=blue>http://www.python.org When this is successful done you should have the following folder structure.

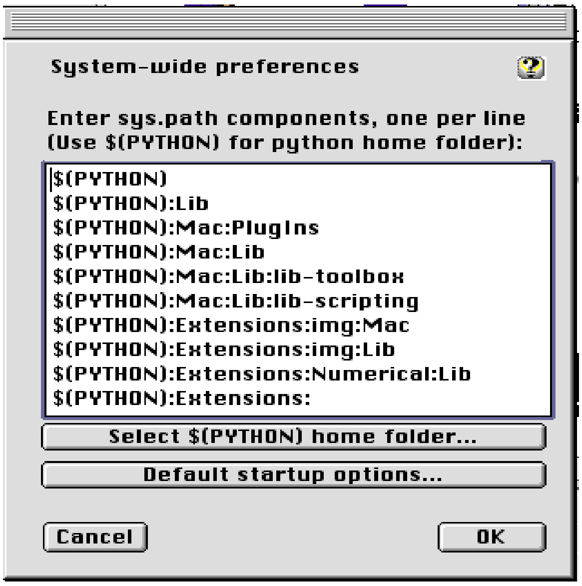


Now you can put Extensions in the Extensions-Folder; which is where you should unpack the **reportlab.zip** with your favorite unpack-utility. You'll get a subfolder named **reportlab**.

After this step, you have to tell the PythonInterpreter, where to look for extensions. Start EditPythonPrefs (by double-clicking the icon).



You should get the following modal dialog. This is the point, where your special data goes in. Reportlab is on the path in Extensions. So all you have to do is add the last line **$(PYTHON):Extensions**.



Now you should test one or more of the demo scripts include with with the sources; eg **reportlab:demos:pythonpoint:pythonpoint.py**. One Problem on the Mac is solved gracefully in Python: if you want a script that takes some arguments, hold down the alt or option-key, while activating Python.

1.5 Getting Involved

ReportLab is an Open Source project. Although we are a commercial company we provide the core PDF generation sources freely, even for commercial purposes, and we make no income directly from these modules. We also welcome help from the community as much as any other Open Source project. There are many ways in which you can help:

- General feedback on the core API. Does it work for you? Are there any rough edges? Does anything feel clunky and awkward?

- New objects to put in reports, or useful utilities for the library. We have an open standard for report objects, so if you have written a nice chart or table class, why not contribute it?
- Demonstrations and Case Studies: If you have produced some nice output, send it to us (with or without scripts). If ReportLab solved a problem for you at work, write a little 'case study' and send it in. And if your web site uses our tools to make reports, let us link to it. We will be happy to display your work (and credit it with your name and company) on our site!
- Working on the core code: we have a long list of things to refine or to implement. If you are missing some features or just want to help out, let us know!

The first step for anyone wanting to learn more or get involved is to join the mailing list. Just send an email with the subject "Subscribe" to `reportlab-users-subscribe@yahoogroups.com`. You can also browse through the group's archives and contributions at `http://www.yahoogroups.com/group/reportlab-users`. This list is the place to report bugs and get support.

1.6 Site Configuration

There are a number of options which most likely need to be configured globally for a site. The python script module `reportlab/rl_config.py` may be edited to change the values of several important sitewide properties.

- shapeChecking: set this to zero to turn off a lot of error checking in the graphis modules
- defaultEncoding: set this to WinAnsiEncoding or MacRomanEncoding.
- defaultPageSize: set this to one of the values defined in reportlab/lib/pagesizes.py; as delivered it is set to pagesizes.A4; other values are pagesizes.letter etc.
- defaultImageCaching: set to zero to inhibit the creation of .a85 files on your hard-drive. The default is to create these preprocessed PDF compatible image files for faster loading
- T1SearchPathPath: this is a python list of strings representing directories that may be queried for information on Type 1 fonts

Chapter 2 General Concepts

Now, we present some more fundamental principles of the graphics library, that will show-up later in various places.

2.1 Drawings and Renderers

A *Drawing* is a platform-independent description of a collection of shapes. It is not directly associated with PDF, Postscript or any other output format. Fortunately, most vector graphics systems have followed the Postscript model and it is possible to describe shapes unambiguously.

A drawing contains a number of primitive *Shapes*. Normal shapes are those widely known as rectangles, circles, lines, etc. One special (logic) shape is a *Group*, which can hold other shapes and apply a transformation to them. Groups represent composites of shapes and allow to treat the composite as if it were a single shape. Just about anything can be built up from a small number of basic shapes.

The package provides several *Renderers* which know how to draw a drawing into different formats. These include PDF (of course), Postscript, and bitmap output. The bitmap renderer uses Raph Levien's *libart* rasterizer and Fredrik Lundh's *Python Imaging Library* (PIL). If you have the right extensions installed, you can generate drawings in bitmap form for the web as well as vector form for PDF documents, and get "identical output".

The PDF renderer has special "privileges" - a Drawing object is also a *Flowable* and, hence, can be placed directly in the story of any Platypus document, or drawn directly on a *Canvas* with one line of code. In addition, the PDF renderer has a utility function to make a one-page PDF document quickly.

We expect to add both input and output filters for many vector graphics formats in future, SVG being a key one. GUIs will be able to obtain screen images from the bitmap output filter working with PIL, so a chart could appear in a Tkinter GUI window.

2.2 Coordinate System

The Y-direction in our X-Y coordinate system points from the bottom *up*. This is consistent with PDF, Postscript and mathematical notation. It also appears to be more natural for people, especially when working with charts. Note that in other graphics models (such as SVG) the Y-coordinate points *down*.

The X-coordinate points, as usual, from left to right. So far there doesn't seem to be any model advocating the opposite direction - at least not yet (with interesting exceptions, as it seems, for Arabs looking at time series charts...).

2.3 Getting Started

Let's create a simple drawing containing the string "Hello World", displayed on top of a coloured rectangle. After creating it we will save the drawing to a standalone PDF file.

```
from reportlab.lib import colors
from reportlab.graphics.shapes import *

d = Drawing(400, 200)
d.add(Rect(50, 50, 300, 100, fillColor=colors.yellow))
d.add(String(150,100, 'Hello World',
             fontSize=18, fillColor=colors.red))

from reportlab.graphics import renderPDF
renderPDF.drawToFile(d, 'example1.pdf', 'My First Drawing')
```

This will produce a PDF file containing the following graphic:

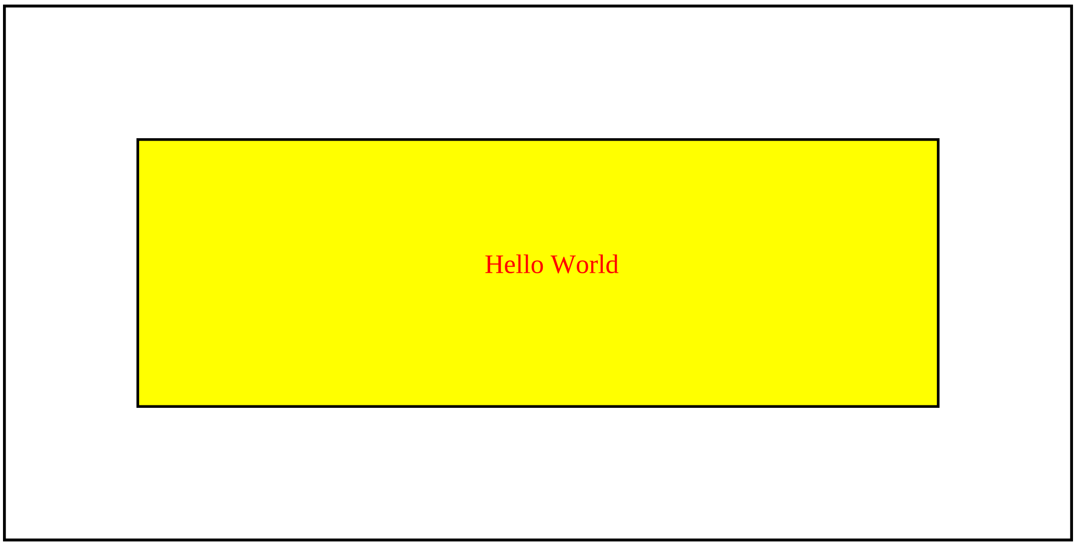


Figure 2-1: 'Hello World'

Each renderer is allowed to do whatever is appropriate for its format, and may have whatever API is needed. If it refers to a file format, it usually has a `drawToFile` function, and that's all you need to know about the renderer. Let's save the same drawing in Encapsulated Postscript format:

```
from reportlab.graphics import renderPS
renderPS.drawToFile(d, 'example1.eps')
```

This will produce an EPS file with the identical drawing, which may be imported into publishing tools such as Quark Express. If we wanted to generate the same drawing as a bitmap file for a website, say, all we need to do is write code like this:

```
from reportlab.graphics import renderPM
renderPM.saveToFile(d, 'example1.png', 'PNG')
```

Many other bitmap formats, like GIF, JPG, TIFF, BMP and PPN are genuinely available, making it unlikely you'll need to add external postprocessing steps to convert to the final format you need.

2.4 Attribute Verification

Python is very dynamic and lets us execute statements at run time that can easily be the source for unexpected behaviour. One subtle 'error' is when assigning to an attribute that the framework doesn't know about because the used attribute's name contains a typo. Python lets you get away with it (adding a new attribute to an object, say), but the graphics framework will not detect this 'typo' without taking special counter-measures.

There are two verification techniques to avoid this situation. The default is for every object to check every assignment at run time, such that you can only assign to 'legal' attributes. This is what happens by default. As this imposes a small performance penalty, this behaviour can be turned off when you need it to be.

```
>>> r = Rect(10,10,200,100, fillColor=colors.red)
>>>
>>> r.fullColor = colors.green # note the typo
>>> r.x = 'not a number'       # illegal argument type
>>> del r.width                # that should confuse it
```

These statements would be caught by the compiler in a statically typed language, but Python lets you get away with it. The first error could leave you staring at the picture trying to figure out why the colors were wrong. The second error would probably become clear only later, when some back-end tries to draw the rectangle. The third, though less likely, results in an invalid object that would not know how to draw itself.

```
>>> r = shapes.Rect(10,10,200,80)
>>> r.fullColor = colors.green
```

```
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "C:\code\users ndy\graphics\shapes.py", line 254, in __setattr__
    validateSetattr(self,attr,value)    #from reportlab.lib.attrmap
  File "C:\code\users ndy\lib ttrmap.py", line 74, in validateSetattr
    raise AttributeError, "Illegal attribute '%s' in class %s" % (name, obj.__class__.__name__)
AttributeError: Illegal attribute 'fullColor' in class Rect
>>>
```

This imposes a performance penalty, so this behaviour can be turned off when you need it to be. To do this, you should use the following lines of code before you first import reportlab.graphics.shapes:

```
>>> import reportlab.rl_config
>>> reportlab.rl_config.shapeChecking = 0
>>> from reportlab.graphics import shapes
>>>
```

Once you turn off `shapeChecking`, the classes are actually built without the verification hook; code should get faster, then. Currently the penalty seems to be about 25% on batches of charts, so it is hardly worth disabling. However, if we move the renderers to C in future (which is eminently possible), the remaining 75% would shrink to almost nothing and the saving from verification would be significant.

Each object, including the drawing itself, has a `verify()` method. This either succeeds, or raises an exception. If you turn off automatic verification, then you should explicitly call `verify()` in testing when developing the code, or perhaps once in a batch process.

2.5 Property Editing

A cornerstone of the reportlab/graphics which we will cover below is that you can automatically document widgets. This means getting hold of all of their editable properties, including those of their subcomponents.

Another goal is to be able to create GUIs and config files for drawings. A generic GUI can be built to show all editable properties of a drawing, and let you modify them and see the results. The Visual Basic or Delphi development environment are good examples of this kind of thing. In a batch charting application, a file could list all the properties of all the components in a chart, and be merged with a database query to make a batch of charts.

To support these applications we have two interfaces, `getProperties` and `setProperties`, as well as a convenience method `dumpProperties`. The first returns a dictionary of the editable properties of an object; the second sets them en masse. If an object has publicly exposed 'children' then one can recursively set and get their properties too. This will make much more sense when we look at *Widgets* later on, but we need to put the support into the base of the framework.

```
>>> r = shapes.Rect(0,0,200,100)
>>> import pprint
>>> pprint.pprint(r.getProperties())
{'fillColor': Color(0.00,0.00,0.00),
 'height': 100,
 'rx': 0,
 'ry': 0,
 'strokeColor': Color(0.00,0.00,0.00),
 'strokeDashArray': None,
 'strokeLineCap': 0,
 'strokeLineJoin': 0,
 'strokeMiterLimit': 0,
 'strokeWidth': 1,
 'width': 200,
 'x': 0,
 'y': 0}
>>> r.setProperties({'x':20, 'y':30, 'strokeColor': colors.red})
>>> r.dumpProperties()
fillColor = Color(0.00,0.00,0.00)
height = 100
rx = 0
ry = 0
strokeColor = Color(1.00,0.00,0.00)
strokeDashArray = None
strokeLineCap = 0
strokeLineJoin = 0
```

```
strokeMiterLimit = 0
strokeWidth = 1
width = 200
x = 20
y = 30
>>>
```

Note: `pprint` *is the standard Python library module that allows you to 'pretty print' output over multiple lines rather than having one very long line.*

These three methods don't seem to do much here, but as we will see they make our widgets framework much more powerful when dealing with non-primitive objects.

2.6 Naming Children

You can add objects to the `Drawing` and `Group` objects. These normally go into a list of contents. However, you may also give objects a name when adding them. This allows you to refer to and possibly change any element of a drawing after constructing it.

```
>>> d = shapes.Drawing(400, 200)
>>> s = shapes.String(10, 10, 'Hello World')
>>> d.add(s, 'caption')
>>> s.caption.text
'Hello World'
>>>
```

Note that you can use the same shape instance in several contexts in a drawing; if you choose to use the same `Circle` object in many locations (e.g. a scatter plot) and use different names to access it, it will still be a shared object and the changes will be global.

This provides one paradigm for creating and modifying interactive drawings.

Chapter 3 Shapes

This chapter describes the concept of shapes and their importance as building blocks for all output generated by the graphics library. Some properties of existing shapes and their relationship to diagrams are presented and the notion of having different renderers for different output formats is briefly introduced.

3.1 Available Shapes

Drawings are made up of Shapes. Absolutely anything can be built up by combining the same set of primitive shapes. The module `shapes.py` supplies a number of primitive shapes and constructs which can be added to a drawing. They are:

- Rect
- Circle
- Ellipse
- Wedge (a pie slice)
- Polygon
- Line
- PolyLine
- String
- Group
- Path (*not implemented yet, but will be added in the future*)

The following drawing, taken from our test suite, shows most of the basic shapes (except for groups). Those with a filled green surface are also called *solid shapes* (these are `Rect`, `Circle`, `Ellipse`, `Wedge` and `Polygon`).

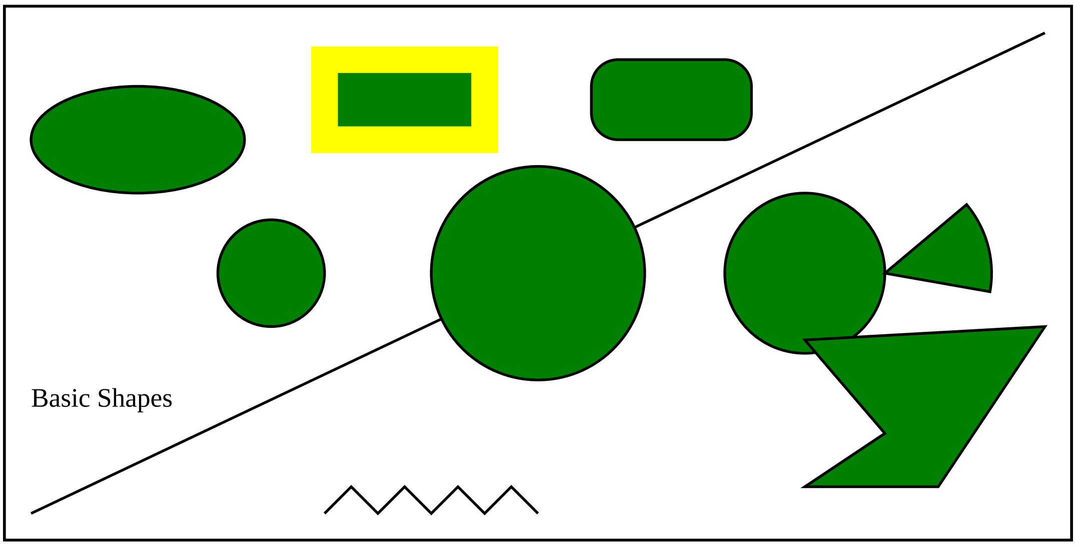


Figure 3-1: Basic shapes

3.2 Shape Properties

Shapes have two kinds of properties - some to define their geometry and some to define their style. Let's create a red rectangle with 3-point thick green borders:

```
>>> from reportlab.graphics.shapes import Rect
>>> from reportlab.lib.colors import red, green
>>> r = Rect(5, 5, 200, 100)
>>> r.fillColor = red
>>> r.strokeColor = green
>>> r.strokeWidth = 3
>>>
```

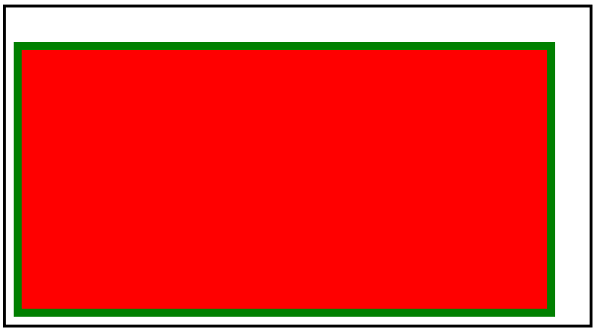


Figure 3-2: red rectangle with green border

Note: In future examples we will omit the import statements.

All shapes have a number of properties which can be set. At an interactive prompt, we can use their *dumpProperties()* method to list these. Here's what you can use to configure a Rect:

```
>>> r.dumpProperties()
fillColor = Color(1.00,0.00,0.00)
height = 100
rx = 0
ry = 0
strokeColor = Color(0.00,0.50,0.00)
strokeDashArray = None
strokeLineCap = 0
strokeLineJoin = 0
strokeMiterLimit = 0
strokeWidth = 3
width = 200
x = 5
y = 5
>>>
```

Shapes generally have *style properties* and *geometry properties*. `x`, `y`, `width` and `height` are part of the geometry and must be provided when creating the rectangle, since it does not make much sense without those properties. The others are optional and come with sensible defaults.

You may set other properties on subsequent lines, or by passing them as optional arguments to the constructor. We could also have created our rectangle this way:

```
>>> r = Rect(5, 5, 200, 100,
             fillColor=red,
             strokeColor=green,
             strokeWidth=3)
```

Let's run through the style properties. `fillColor` is obvious. `stroke` is publishing terminology for the edge of a shape; the stroke has a color, width, possibly a dash pattern, and some (rarely used) features for what happens when a line turns a corner. `rx` and `ry` are optional geometric properties and are used to define the corner radius for a rounded rectangle.

All the other solid shapes share the same style properties.

3.3 Lines

We provide single straight lines, PolyLines and curves. Lines have all the `stroke*` properties, but no `fillColor`. Here are a few Line and PolyLine examples and the corresponding graphics output:

```
Line(50,50, 300,100,
     strokeColor=colors.blue, strokeWidth=5)
Line(50,100, 300,50,
     strokeColor=colors.red,
     strokeWidth=10,
     strokeDashArray=[10, 20])
PolyLine([120,110, 130,150, 140,110, 150,150, 160,110,
          170,150, 180,110, 190,150, 200,110],
```

```
strokeWidth=2,
strokeColor=colors.purple)
```

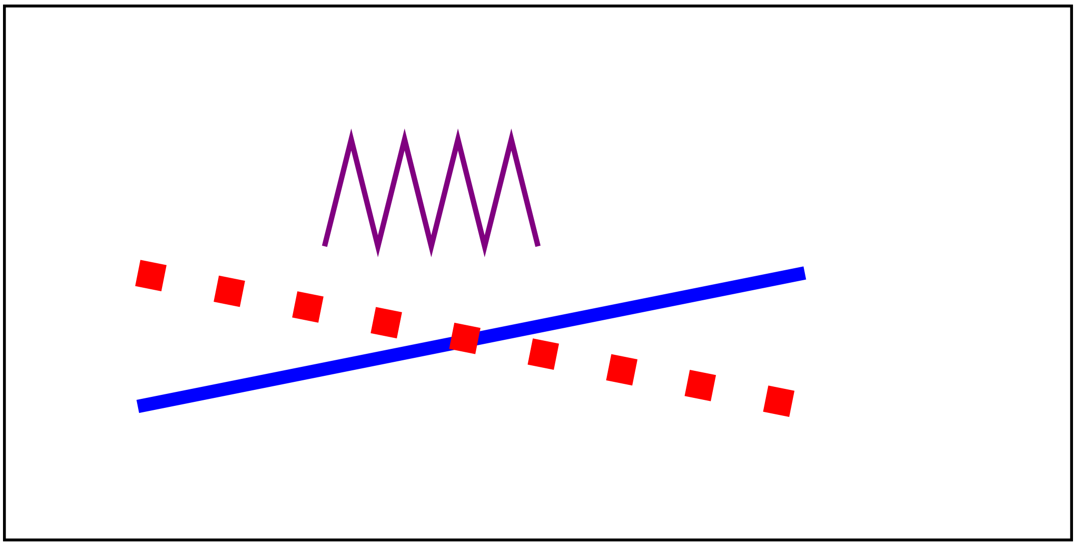


Figure 3-3: Line and PolyLine examples

3.4 Strings

The ReportLab Graphics package is not designed for fancy text layout, but it can place strings at desired locations and with left/right/center alignment. Let's specify a `String` object and look at its properties:

```
>>> s = String(10, 50, 'Hello World')
>>> s.dumpProperties()
fillColor = Color(0.00,0.00,0.00)
fontName = Times-Roman
fontSize = 10
text = Hello World
textAnchor = start
x = 10
y = 50
>>>
```

Strings have a textAnchor property, which may have one of the values 'start', 'middle', 'end'. If this is set to 'start', x and y relate to the start of the string, and so on. This provides an easy way to align text.

Strings use a common font standard: the Type 1 Postscript fonts present in Acrobat Reader. We can thus use the basic 14 fonts in ReportLab and get accurate metrics for them. We have recently also added support for extra Type 1 fonts and the renderers all know how to render Type 1 fonts.

Here is a more fancy example using the code snippet below. Please consult the ReportLab User Guide to see how non-standard like 'LettErrorRobot-Chrome' fonts are being registered!

```
d = Drawing(400, 200)
for size in range(12, 36, 4):
    d.add(String(10+size*2, 10+size*2, 'Hello World',
                 fontName='Times-Roman',
                 fontSize=size))

d.add(String(130, 120, 'Hello World',
             fontName='Courier',
             fontSize=36))

d.add(String(150, 160, 'Hello World',
             fontName='LettErrorRobot-Chrome',
             fontSize=36))
```

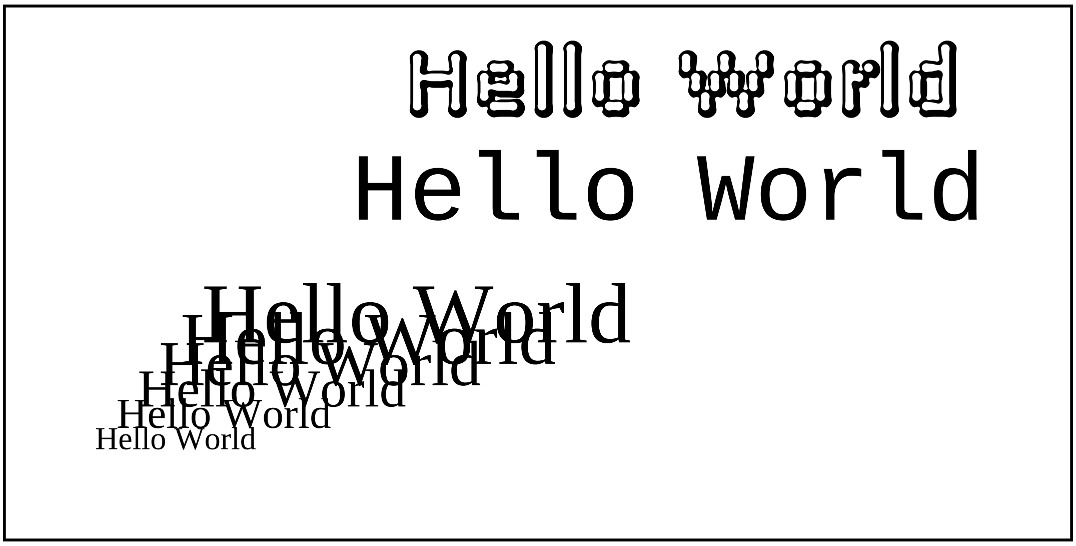


Figure 3-4: fancy font example

3.5 Paths

Postscript paths are a widely understood concept in graphics. They are not implemented in `reportlab/graphics` as yet, but they will be, soon.

3.6 Groups

Finally, we have Group objects. A group has a list of contents, which are other nodes. It can also apply a transformation - its contents can be rotated, scaled or shifted. If you know the math, you can set the transform directly. Otherwise it provides methods to rotate, scale and so on. Here we make a group which is rotated and translated:

```
>>> g =Group(shape1, shape2, shape3)
>>> g.rotate(30)
>>> g.translate(50, 200)
```

Groups provide a tool for reuse. You can make a bunch of shapes to represent some component - say, a coordinate system - and put them in one group called "Axis". You can then put that group into other groups, each with a different translation and rotation, and you get a bunch of axis. It is still the same group, being drawn in different places.

Let's do this with some only slightly more code:

```
d = Drawing(400, 200)

Axis = Group(
    Line(0,0,100,0),  # x axis
    Line(0,0,0,50),   # y axis
    Line(0,10,10,10), # ticks on y axis
    Line(0,20,10,20),
    Line(0,30,10,30),
    Line(0,40,10,40),
    Line(10,0,10,10), # ticks on x axis
    Line(20,0,20,10),
    Line(30,0,30,10),
    Line(40,0,40,10),
    Line(50,0,50,10),
    Line(60,0,60,10),
    Line(70,0,70,10),
    Line(80,0,80,10),
    Line(90,0,90,10),
    String(20, 35, 'Axes', fill=colors.black)
    )
```

```
firstAxisGroup = Group(Axis)
firstAxisGroup.translate(10,10)
d.add(firstAxisGroup)

secondAxisGroup = Group(Axis)
secondAxisGroup.translate(150,10)
secondAxisGroup.rotate(15)

d.add(secondAxisGroup)

thirdAxisGroup = Group(Axis,
                       transform=mmult(translate(300,10),
                                       rotate(30)))
d.add(thirdAxisGroup)
```

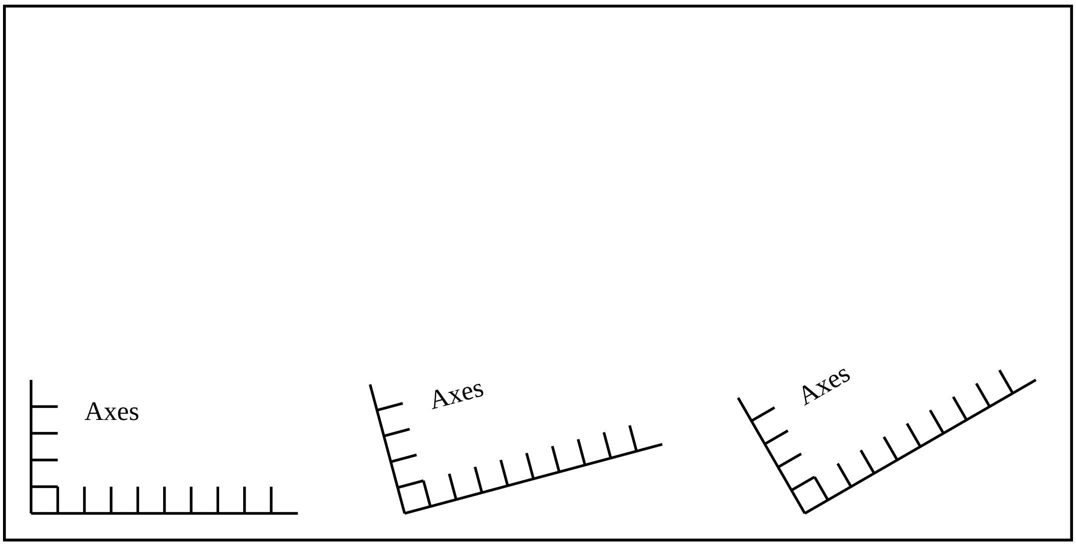


Figure 3-5: Groups examples

Chapter 4 Widgets

We now describe widgets and how they relate to shapes. Using many examples it is shown how widgets make reusable graphics components.

4.1 Shapes vs. Widgets

Up until now, Drawings have been 'pure data'. There is no code in them to actually do anything, except assist the programmer in checking and inspecting the drawing. In fact, that's the cornerstone of the whole concept and is what lets us achieve portability - a renderer only needs to implement the primitive shapes.

We want to build reusable graphic objects, including a powerful chart library. To do this we need to reuse more tangible things than rectangles and circles. We should be able to write objects for other to reuse - arrows, gears, text boxes, UML diagram nodes, even fully fledged charts.

The Widget standard is a standard built on top of the shapes module. Anyone can write new widgets, and we can build up libraries of them. Widgets support the `getProperties()` and `setProperties()` methods, so you can inspect and modify as well as document them in a uniform way.

- A widget is a reusable shape
- it can be initialized with no arguments when its `draw()` method is called it creates a primitive Shape or a Group to represent itself
- It can have any parameters you want, and they can drive the way it is drawn
- it has a `demo()` method which should return an attractively drawn example of itself in a 200x100 rectangle. This is the cornerstone of the automatic documentation tools. The `demo()` method should also have a well written docstring, since that is printed too!

Widgets run contrary to the idea that a drawing is just a bundle of shapes; surely they have their own code? The way they work is that a widget can convert itself to a group of primitive shapes. If some of its components are themselves widgets, they will get converted too. This happens automatically during rendering; the renderer will not see your chart widget, but just a collection of rectangles, lines and strings. You can also explicitly 'flatten out' a drawing, causing all widgets to be converted to primitives.

4.2 Using a Widget

Let's imagine a simple new widget. We will use a widget to draw a face, then show how it was implemented.

```
>>> from reportlab.lib import colors
>>> from reportlab.graphics import shapes
>>> from reportlab.graphics import widgetbase
>>> from reportlab.graphics import renderPDF
>>> d = shapes.Drawing(200, 100)
>>> f = widgetbase.Face()
>>> f.skinColor = colors.yellow
>>> f.mood = "sad"
>>> d.add(f)
>>> renderPDF.drawToFile(d, 'face.pdf', 'A Face')
```

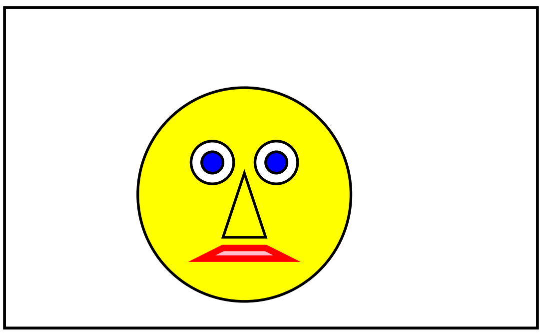


Figure 4-1: A sample widget

Let's see what properties it has available, using the `setProperties()` method we have seen earlier:

```
>>> f.dumpProperties()
eyeColor = Color(0.00,0.00,1.00)
mood = sad
size = 80
skinColor = Color(1.00,1.00,0.00)
x = 10
y = 10
>>>
```

One thing which seems strange about the above code is that we did not set the size or position when we made the face. This is a necessary trade-off to allow a uniform interface for constructing widgets and documenting them - they cannot require arguments in their `__init__()` method. Instead, they are generally designed to fit in a 200 x 100 window, and you move or resize them by setting properties such as x, y, width and so on after creation.

In addition, a widget always provides a `demo()` method. Simple ones like this always do something sensible before setting properties, but more complex ones like a chart would not have any data to plot. The documentation tool calls `demo()` so that your fancy new chart class can create a drawing showing what it can do.

Here are a handful of simple widgets available in the module *signsandsymbols.py*:

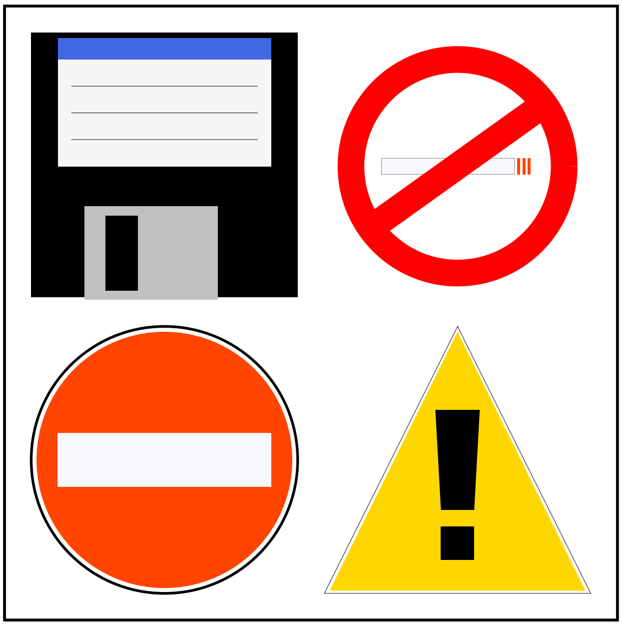


Figure 4-2: A few samples from signsandsymbols.py

And this is the code needed to generate them as seen in the drawing above:

```
from reportlab.graphics.shapes import Drawing
from reportlab.graphics.widgets import signsandsymbols

d = Drawing(230, 230)

ne = signsandsymbols.NoEntry0()
ds = signsandsymbols.DangerSign0()
fd = signsandsymbols.FloppyDisk0()
ns = signsandsymbols.NoSmoking0()

ne.x, ne.y = 10, 10
ds.x, ds.y = 120, 10
fd.x, fd.y = 10, 120
ns.x, ns.y = 120, 120

d.add(ne)
```

```
d.add(ds)
d.add(fd)
d.add(ns)
```

4.3 Compound Widgets

Let's imagine a compound widget which draws two faces side by side. This is easy to build when you have the Face widget.

```
>>> tf = widgetbase.TwoFaces()
>>> tf.faceOne.mood
'happy'
>>> tf.faceTwo.mood
'sad'
>>> tf.dumpProperties()
faceOne.eyeColor = Color(0.00,0.00,1.00)
faceOne.mood = happy
faceOne.size = 80
faceOne.skinColor = None
faceOne.x = 10
faceOne.y = 10
faceTwo.eyeColor = Color(0.00,0.00,1.00)
faceTwo.mood = sad
faceTwo.size = 80
faceTwo.skinColor = None
faceTwo.x = 100
faceTwo.y = 10
>>>
```

The attributes 'faceOne' and 'faceTwo' are deliberately exposed so you can get at them directly. There could also be top-level attributes, but there aren't in this case.

4.4 Verifying Widgets

The widget designer decides the policy on verification, but by default they work like shapes - checking every assignment - if the designer has provided the checking information.

4.5 Implementing Widgets

We tried to make it as easy to implement widgets as possible. Here's the code for a Face widget which does not do any type checking:

```
class Face(Widget):
    """This draws a face with two eyes, mouth and nose."""

    def __init__(self):
        self.x = 10
        self.y = 10
        self.size = 80
        self.skinColor = None
        self.eyeColor = colors.blue
        self.mood = 'happy'

    def draw(self):
        s = self.size  # abbreviate as we will use this a lot
        g = shapes.Group()
        g.transform = [1,0,0,1,self.x, self.y]
        # background
        g.add(shapes.Circle(s * 0.5, s * 0.5, s * 0.5,
                            fillColor=self.skinColor))
        # CODE OMITTED TO MAKE MORE SHAPES
        return g
```

We left out all the code to draw the shapes in this document, but you can find it in the distribution in `widgetbase.py`.

By default, any attribute without a leading underscore is returned by setProperties. This is a deliberate policy to encourage consistent coding conventions.

Once your widget works, you probably want to add support for verification. This involves adding a dictionary to the class called `_verifyMap`, which map from attribute names to 'checking functions'. The `widgetbase.py` module defines a bunch of checking functions with names like `isNumber`, `isListOfShapes` and so on. You can also simply use `None`, which means that the attribute must be present but can have any type. And you can and should write your own checking functions. We want to restrict the "mood" custom attribute to the values "happy", "sad" or "ok". So we do this:

```
class Face(Widget):
    """This draws a face with two eyes.  It exposes a
    couple of properties to configure itself and hides
    all other details"""
    def checkMood(moodName):
        return (moodName in ('happy','sad','ok'))
    _verifyMap = {
        'x': shapes.isNumber,
        'y': shapes.isNumber,
        'size': shapes.isNumber,
        'skinColor':shapes.isColorOrNone,
        'eyeColor': shapes.isColorOrNone,
        'mood': checkMood
        }
```

This checking will be performed on every attribute assignment; or, if `config.shapeChecking` is off, whenever you call `myFace.verify()`.

4.6 Documenting Widgets

We are working on a generic tool to document any Python package or module; this is already checked into ReportLab and will be used to generate a reference for the ReportLab package. When it encounters widgets, it adds extra sections to the manual including:

- the doc string for your widget class
- the code snippet from your *demo()* method, so people can see how to use it
- the drawing produced by the *demo()* method
- the property dump for the widget in the drawing.

This tool will mean that we can have guaranteed up-to-date documentation on our widgets and charts, both on the web site and in print; and that you can do the same for your own widgets, too!

4.7 Widget Design Strategies

We could not come up with a consistent architecture for designing widgets, so we are leaving that problem to the authors! If you do not like the default verification strategy, or the way `setProperties/getProperties` works, you can override them yourself.

For simple widgets it is recommended that you do what we did above: select non-overlapping properties, initialize every property on `__init__` and construct everything when `draw()` is called. You can instead have `__setattr__` hooks and have things updated when certain attributes are set. Consider a pie chart. If you want to expose the individual wedges, you might write code like this:

```
from reportlab.graphics.charts import piecharts
pc = piecharts.Pie()
pc.defaultColors = [navy, blue, skyblue] #used in rotation
pc.data = [10,30,50,25]
pc.slices[7].strokeWidth = 5
```

The last line is problematic as we have only created four wedges - in fact we might not have created them yet. Does `pc.wedges[7]` raise an error? Is it a prescription for what should happen if a seventh wedge is defined, used to override the default settings? We dump this problem squarely on the widget author for now, and recommend that you get a simple one working before exposing 'child objects' whose existence depends on other properties' values :-)

We also discussed rules by which parent widgets could pass properties to their children. There seems to be a general desire for a global way to say that 'all wedges get their lineWidth from the lineWidth of their parent' without a lot of repetitive coding. We do not have a universal solution, so again leave that to widget authors. We hope people will experiment with push-down, pull-down and pattern-matching approaches and come up with something nice. In the meantime, we certainly can write monolithic chart widgets which work like the ones in, say, Visual Basic and Delphi.

For now have a look at the following sample code using an early version of a pie chart widget and the output it generates:

```
from reportlab.lib.colors import *
from reportlab.graphics import shapes,renderPDF
from reportlab.graphics.charts.piecharts import Pie

d = Drawing(400,200)
d.add(String(100,175,"Without labels", textAnchor="middle"))
d.add(String(300,175,"With labels", textAnchor="middle"))

pc = Pie()
pc.x = 25
pc.y = 50
pc.data = [10,20,30,40,50,60]
pc.slices[0].popout = 5
d.add(pc, 'pie1')

pc2 = Pie()
pc2.x = 150
pc2.y = 50
pc2.data = [10,20,30,40,50,60]
pc2.labels = ['a','b','c','d','e','f']
d.add(pc2, 'pie2')

pc3 = Pie()
pc3.x = 275
pc3.y = 50
pc3.data = [10,20,30,40,50,60]
pc3.labels = ['a','b','c','d','e','f']
pc3.wedges.labelRadius = 0.65
pc3.wedges.fontName = "Helvetica-Bold"
pc3.wedges.fontSize = 16
pc3.wedges.fontColor = colors.yellow
d.add(pc3, 'pie3')
```

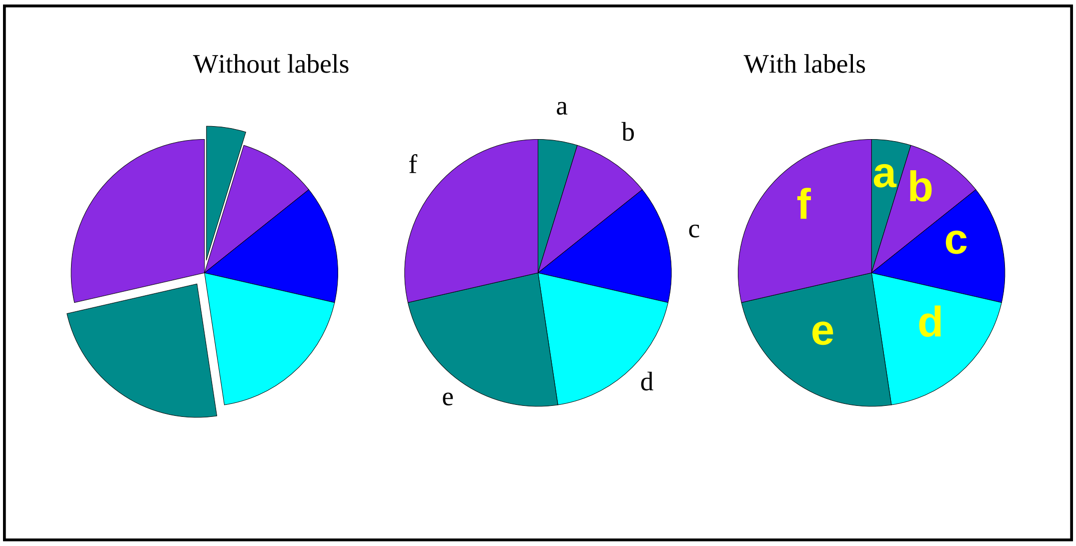


Figure 4-3: Some sample Pies

Chapter 5 Charts

The motivation for much of this is to create a flexible chart package. This chapter presents a treatment of the ideas behind our charting model, what the design goals are and what components of the chart package already exist.

5.1 Design Goals

Here are some of the design goals:

Make simple top-level use really simple

It should be possible to create a simple chart with minimum lines of code, yet have it 'do the right things' with sensible automatic settings. The pie chart snippets above do this. If a real chart has many subcomponents, you still should not need to interact with them unless you want to customize what they do.

Allow precise positioning

An absolute requirement in publishing and graphic design is to control the placing and style of every element. We will try to have properties that specify things in fixed sizes and proportions of the drawing, rather than having automatic resizing. Thus, the 'inner plot rectangle' will not magically change when you make the font size of the y labels bigger, even if this means your labels can spill out of the left edge of the chart rectangle. It is your job to preview the chart and choose sizes and spaces which will work.

Some things do need to be automatic. For example, if you want to fit N bars into a 200 point space and don't know N in advance, we specify bar separation as a percentage of the width of a bar rather than a point size, and let the chart work it out. This is still deterministic and controllable.

Control child elements individually or as a group

We use smart collection classes that let you customize a group of things, or just one of them. For example you can do this in our experimental pie chart:

```
d = Drawing(400,200)
pc = Pie()
pc.x = 150
pc.y = 50
pc.data = [10,20,30,40,50,60]
pc.labels = ['a','b','c','d','e','f']
pc.slices.strokeWidth=0.5
pc.slices[3].popout = 20
pc.slices[3].strokeWidth = 2
pc.slices[3].strokeDashArray = [2,2]
pc.slices[3].labelRadius = 1.75
pc.slices[3].fontColor = colors.red
d.add(pc, '')
```

pc.slices[3] actually lazily creates a little object which holds information about the slice in question; this will be used to format a fourth slice at draw-time if there is one.

Only expose things you should change

It would be wrong from a statistical viewpoint to let you directly adjust the angle of one of the pie wedges in the above example, since that is determined by the data. So not everything will be exposed through the public properties. There may be 'back doors' to let you violate this when you really need to, or methods to provide advanced functionality, but in general properties will be orthogonal.

Composition and component based

Charts are built out of reusable child widgets. A Legend is an easy-to-grasp example. If you need a specialized type of legend (e.g. circular colour swatches), you should subclass the standard Legend widget. Then you could either do something like...

```
c = MyChartWithLegend()
c.legend = MyNewLegendClass()    # just change it
c.legend.swatchRadius = 5    # set a property only relevant to the new one
c.data = [10,20,30]   #   and then configure as usual...
```

...or create/modify your own chart or drawing class which creates one of these by default. This is also very relevant for time series charts, where there can be many styles of x axis.

Top level chart classes will create a number of such components, and then either call methods or set private properties to tell them their height and position - all the stuff which should be done for you and which you cannot customise. We are working on modelling what the components should be and will publish their APIs here as a consensus emerges.

Multiples

A corollary of the component approach is that you can create diagrams with multiple charts, or custom data graphics. Our favourite example of what we are aiming for is the weather report in our gallery contributed by a user; we'd like to make it easy to create such drawings, hook the building blocks up to their legends, and feed that data in a consistent way.

(If you want to see the image, it is available on our website at http://www.reportlab.com/demos/provencio.pdf)

5.2 Overview

A chart or plot is an object which is placed on a drawing; it is not itself a drawing. You can thus control where it goes, put several on the same drawing, or add annotations.

Charts have two axes; axes may be Value or Category axes. Axes in turn have a Labels property which lets you configure all text labels or each one individually. Most of the configuration details which vary from chart to chart relate to axis properties, or axis labels.

Objects expose properties through the interfaces discussed in the previous section; these are all optional and are there to let the end user configure the appearance. Things which must be set for a chart to work, and essential communication between a chart and its components, are handled through methods.

You can subclass any chart component and use your replacement instead of the original provided you implement the essential methods and properties.

5.3 Labels

A label is a string of text attached to some chart element. They are used on axes, for titles or alongside axes, or attached to individual data points. Labels may contain newline characters, but only one font.

The text and 'origin' of a label are typically set by its parent object. They are accessed by methods rather than properties. Thus, the X axis decides the 'reference point' for each tickmark label and the numeric or date text for each label. However, the end user can set properties of the label (or collection of labels) directly to affect its position relative to this origin and all of its formatting.

```
from reportlab.graphics import shapes
from reportlab.graphics.charts.textlabels import Label

d = Drawing(200, 100)

# mark the origin of the label
d.add(Circle(100,90, 5, fillColor=colors.green))

lab = Label()
lab.setOrigin(100,90)
lab.boxAnchor = 'ne'
lab.angle = 45
lab.dx = 0
lab.dy = -20
lab.boxStrokeColor = colors.green
```

```
lab.setText('Some
Multi-Line
Label')

d.add(lab)
```

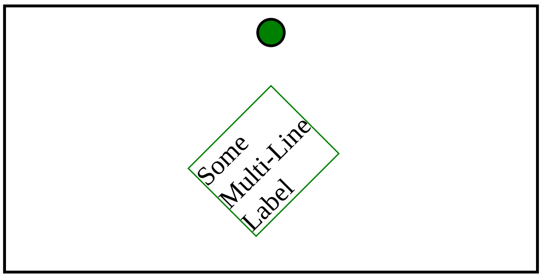


Figure 5-1: Label example

In the drawing above, the label is defined relative to the green blob. The text box should have its north-east corner ten points down from the origin, and be rotated by 45 degrees about that corner.

At present labels have the following properties, which we believe are sufficient for all charts we have seen to date:

Property	Meaning
dx	The label's x displacement.
dy	The label's y displacement.
angle	The angle of rotation (counterclockwise) applied to the label.
boxAnchor	The label's box anchor, one of 'n', 'e', 'w', 's', 'ne', 'nw', 'se', 'sw'.
textAnchor	The place where to anchor the label's text, one of 'start', 'middle', 'end'.
boxFillColor	The fill color used in the label's box.
boxStrokeColor	The stroke color used in the label's box.
boxStrokeWidth	The line width of the label's box.
fontName	The label's font name.
fontSize	The label's font size.
leading	The leading value of the label's text lines.
x	The X-coordinate of the reference point.
y	The Y-coordinate of the reference point.
width	The label's width.
height	The label's height.

Table 5-1 - Label properties

To see many more examples of `Label` objects with different combinations of properties, please have a look into the ReportLab test suite in the folder `reportlab/test`, run the script `test_charts_textlabels.py` and look at the PDF document it generates!

5.4 Axes

We identify two basic kinds of axes - *Value* and *Category* ones. Both come in horizontal and vertical flavors. Both can be subclassed to make very specific kinds of axis. For example, if you have complex rules for which dates to display in a time series application, or want irregular scaling, you override the axis and make a new one.

Axes are responsible for determining the mapping from data to image coordinates; transforming points on request from the chart; drawing themselves and their tickmarks, gridlines and axis labels.

This drawing shows two axes, one of each kind, which have been created directly without reference to any chart:

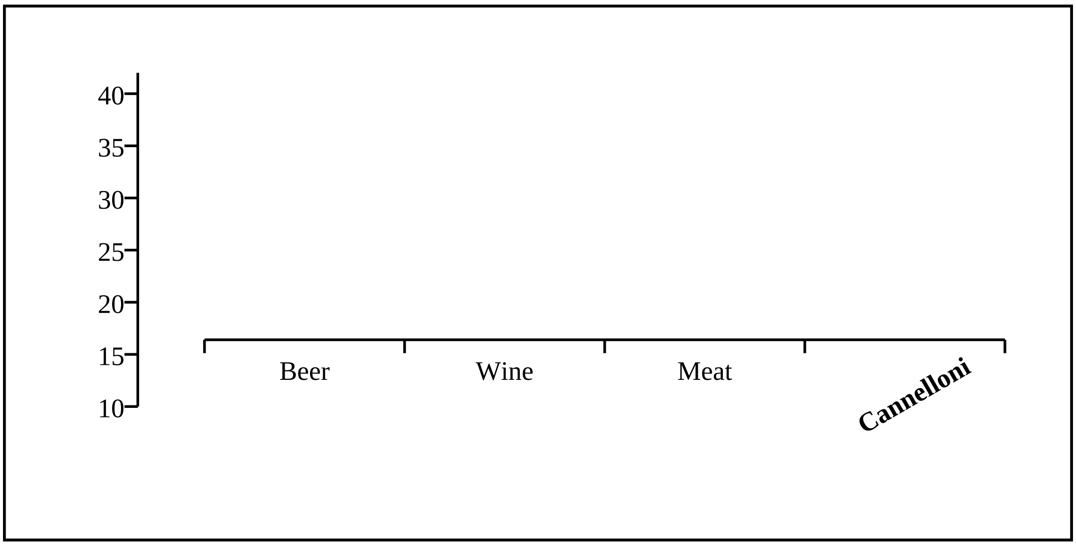


Figure 5-2: Two isolated axes

Here is the code that created them:

```
from reportlab.graphics import shapes
from reportlab.graphics.charts.axes import XCategoryAxis,YValueAxis

drawing = Drawing(400, 200)

data = [(10, 20, 30, 40), (15, 22, 37, 42)]

xAxis = XCategoryAxis()
xAxis.setPosition(75, 75, 300)
xAxis.configure(data)
xAxis.categoryNames = ['Beer', 'Wine', 'Meat', 'Cannelloni']
xAxis.labels.boxAnchor = 'n'
xAxis.labels[3].dy = -15
xAxis.labels[3].angle = 30
xAxis.labels[3].fontName = 'Times-Bold'

yAxis = YValueAxis()
yAxis.setPosition(50, 50, 125)
yAxis.configure(data)

drawing.add(xAxis)
drawing.add(yAxis)
```

Remember that, usually, you won't have to create axes directly; when using a standard chart, it comes with ready-made axes. The methods are what the chart uses to configure it and take care of the geometry. However, we will talk through them in detail below. The orthogonally dual axes to those we describe have essentially the same properties, except for those refering to ticks.

XCategoryAxis class

A Category Axis doesn't really have a scale; it just divides itself into equal-sized buckets. It is simpler than a value axis. The chart (or programmer) sets its location with the method `setPosition(x, y, length)`. The next stage is to show it the data so that it can configure itself. This is easy for a category axis - it just counts the number of data points in one of the data series. When the drawing is drawn, the axis can provide some help to the chart with its `scale()` method, which tells the chart where a given category begins and ends on the page. We have not yet seen any need to let people override the widths or positions of categories.

An XCategoryAxis has the following editable properties:

Property	Meaning
`visible`	Should the axis be drawn at all? Sometimes you don't want to display one or both axes, but they still need to be there as they manage the scaling of points.
`strokeColor`	Color of the axis
`strokeDashArray`	Whether to draw axis with a dash and, if so, what kind. Defaults to None
`strokeWidth`	Width of axis in points
`tickUp`	How far above the axis should the tick marks protrude? (Note that making this equal to chart height gives you a gridline)
`tickDown`	How far below the axis should the tick mark protrude?
`categoryNames`	Either None, or a list of strings. This should have the same length as each data series.
`labels`	A collection of labels for the tick marks. By default the 'north' of each text label (i.e top centre) is positioned 5 points down from the centre of each category on the axis. You may redefine any property of the whole label group or of any one label. If categoryNames=None, no labels are drawn.
`title`	Not Implemented Yet. This needs to be like a label, but also lets you set the text directly. It would have a default location below the axis.

Table 5-2 - XCategoryAxis properties

YValueAxis

The left axis in the diagram is a YValueAxis. A Value Axis differs from a Category Axis in that each point along its length corresponds to a y value in chart space. It is the job of the axis to configure itself, and to convert Y values from chart space to points on demand to assist the parent chart in plotting.

`setPosition(x, y, length)` and `configure(data)` work exactly as for a category axis. If you have not fully specified the maximum, minimum and tick interval, then `configure()` results in the axis choosing suitable values. Once configured, the value axis can convert y data values to drawing space with the `scale()` method. Thus:

```
>>> yAxis = YValueAxis()
>>> yAxis.setPosition(50, 50, 125)
>>> data = [(10, 20, 30, 40),(15, 22, 37, 42)]
>>> yAxis.configure(data)
>>> yAxis.scale(10)  # should be bottom of chart
50.0
>>> yAxis.scale(40)  # should be near the top
167.1875
>>>
```

By default, the highest data point is aligned with the top of the axis, the lowest with the bottom of the axis, and the axis choose 'nice round numbers' for its tickmark points. You may override these settings with the properties below.

Property	Meaning

`visible`	Should the axis be drawn at all? Sometimes you don't want to display one or both axes, but they still need to be there as they manage the scaling of points.
`strokeColor`	Color of the axis
`strokeDashArray`	Whether to draw axis with a dash and, if so, what kind. Defaults to None
`strokeWidth`	Width of axis in points
`tickLeft`	How far to the left of the axis should the tick marks protrude? (Note that making this equal to chart height gives you a gridline)
`tickRight`	How far to the right of the axis should the tick mark protrude?
`valueMin`	The y value to which the bottom of the axis should correspond. Default value is None in which case the axis sets it to the lowest actual data point (e.g. 10 in the example above). It is common to set this to zero to avoid misleading the eye.
`valueMax`	The y value to which the top of the axis should correspond. Default value is None in which case the axis sets it to the highest actual data point (e.g. 42 in the example above). It is common to set this to a 'round number' so data bars do not quite reach the top.
`valueStep`	The y change between tick intervals. By default this is None, and the chart tries to pick 'nice round numbers' which are just wider than the minimumTickSpacing below.
`valueSteps`	A list of numbers at which to place ticks.
`minimumTickSpacing`	This is used when valueStep is set to None, and ignored otherwise. The designer specified that tick marks should be no closer than X points apart (based, presumably, on considerations of the label font size and angle). The chart tries values of the type 1,2,5,10,20,50,100... (going down below 1 if necessary) until it finds an interval which is greater than the desired spacing, and uses this for the step.
`labelTextFormat`	This determines what goes in the labels. Unlike a category axis which accepts fixed strings, the labels on a ValueAxis are supposed to be numbers. You may provide either a 'format string' like '%0.2f' (show two decimal places), or an arbitrary function which accepts a number and returns a string. One use for the latter is to convert a timestamp to a readable year-month-day format.
`title`	Not Implemented Yet. This needs to be like a label, but also lets you set the text directly. It would have a default location below the axis.

Table 5-3 - YValueAxis properties

The `valueSteps` property lets you explicitly specify the tick mark locations, so you don't have to follow regular intervals. Hence, you can plot month ends and month end dates with a couple of helper functions, and without needing special time series chart classes. The following code show how to create a simple `XValueAxis` with special tick intervals. Make sure to set the `valueSteps` attribute before calling the configure method!

```
from reportlab.graphics.shapes import Drawing
from reportlab.graphics.charts.axes import XValueAxis

drawing = Drawing(400, 100)
```

```
data = [(10, 20, 30, 40)]

xAxis = XValueAxis()
xAxis.setPosition(75, 50, 300)
xAxis.valueSteps = [10, 15, 20, 30, 35, 40]
xAxis.configure(data)
xAxis.labels.boxAnchor = 'n'

drawing.add(xAxis)
```

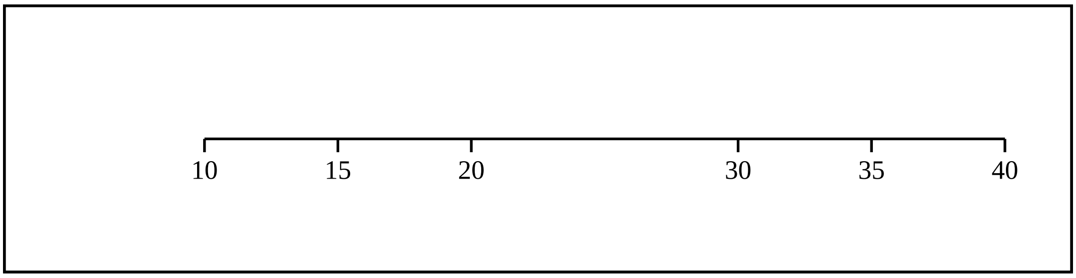


Figure 5-3: An axis with non-equidistant tick marks

In addition to these properties, all axes classes have three properties describing how to join two of them to each other. Again, this is interesting only if you define your own charts or want to modify the appearance of an existing chart using such axes. These properties are listed here only very briefly for now, but you can find a host of sample functions in the module `reportlab/graphics/axes.py` which you can examine...

One axis is joined to another, by calling the method `joinToAxis(otherAxis, mode, pos)` on the first axis, with `mode` and `pos` being the properties described by `joinAxisMode` and `joinAxisPos`, respectively. `'points'` means to use an absolute value, and `'value'` to use a relative value (both indicated by the the `joinAxisPos` property) along the axis.

Property	Meaning
`joinAxis`	Join both axes if true.
`joinAxisMode`	Mode used for connecting axis ('bottom', 'top', 'left', 'right', 'value', 'points', None).
`joinAxisPos`	Position at which to join with other axis.

Table 5-4 - Axes joining properties

5.5 Bar Charts

This describes our current `VerticalBarChart` class, which uses the axes and labels above. We think it is step in the right direction but is is far from final. Note that people we speak to are divided about 50/50 on whether to call this a 'Vertical' or 'Horizontal' bar chart. We chose this name because 'Vertical' appears next to 'Bar', so we take it to mean that the bars rather than the category axis are vertical.

As usual, we will start with an example:

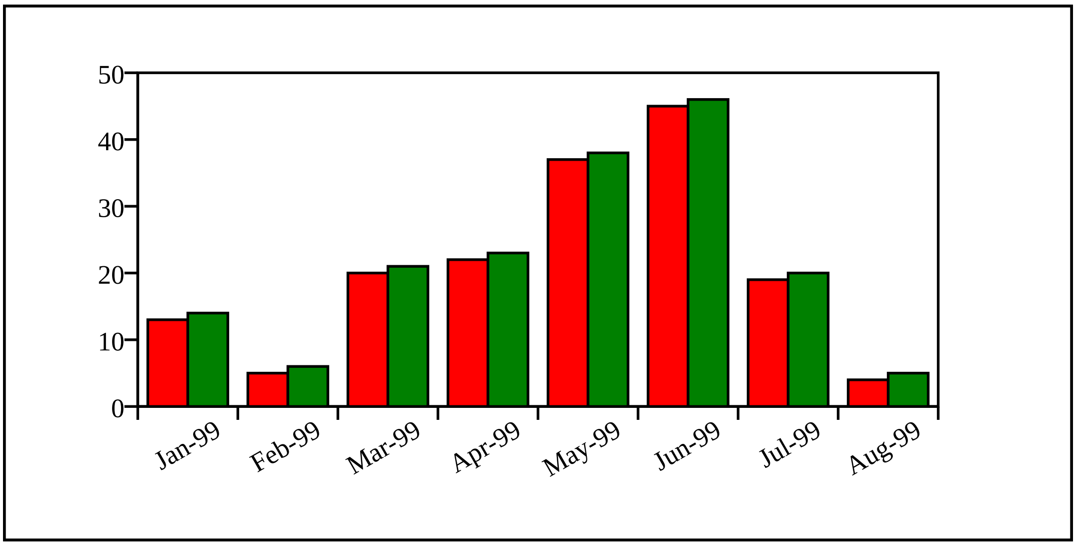


Figure 5-4: Simple bar chart with two data series

```
# code to produce the above chart

from reportlab.graphics.shapes import Drawing
from reportlab.graphics.charts.barcharts import VerticalBarChart

drawing = Drawing(400, 200)

data = [
        (13, 5, 20, 22, 37, 45, 19, 4),
        (14, 6, 21, 23, 38, 46, 20, 5)
        ]

bc = VerticalBarChart()
bc.x = 50
bc.y = 50
bc.height = 125
bc.width = 300
bc.data = data
bc.strokeColor = colors.black

bc.valueAxis.valueMin = 0
bc.valueAxis.valueMax = 50
bc.valueAxis.valueStep = 10

bc.categoryAxis.labels.boxAnchor = 'ne'
bc.categoryAxis.labels.dx = 8
bc.categoryAxis.labels.dy = -2
bc.categoryAxis.labels.angle = 30
bc.categoryAxis.categoryNames = ['Jan-99','Feb-99','Mar-99',
       'Apr-99','May-99','Jun-99','Jul-99','Aug-99']

drawing.add(bc)
```

Most of this code is concerned with setting up the axes and labels, which we have already covered. Here are the top-level properties of the `VerticalBarChart` class:

Property	**Meaning**
`data`	This should be a "list of lists of numbers" or "list of tuples of numbers". If you have just one series, write it as data = [(10,20,30,42),]

`x, y, width, height`	These define the inner 'plot rectangle'. We highlighted this with a yellow border above. Note that it is your job to place the chart on the drawing in a way which leaves room for all the axis labels and tickmarks. We specify this 'inner rectangle' because it makes it very easy to lay out multiple charts in a consistent manner.
`strokeColor`	Defaults to None. This will draw a border around the plot rectangle, which may be useful in debugging. Axes will overwrite this.
`fillColor`	Defaults to None. This will fill the plot rectangle with a solid color. (Note that we could implement dashArray etc. as for any other solid shape)
`barLabelFormat`	This is a format string or function used for displaying labels above each bar. They are positioned automatically above the bar for positive values and below for negative ones.
`useAbsolute`	Defaults to 0. If 1, the three properties below are absolute values in points (which means you can make a chart where the bars stick out from the plot rectangle); if 0, they are relative quantities and indicate the proportional widths of the elements involved.
`barWidth`	As it says. Defaults to 10.
`groupSpacing`	Defaults to 5. This is the space between each group of bars. If you have only one series, use groupSpacing and not barSpacing to split them up. Half of the groupSpacing is used before the first bar in the chart, and another half at the end.
`barSpacing`	Defaults to 0. This is the spacing between bars in each group. If you wanted a little gap between green and red bars in the example above, you would make this non-zero.
`barLabelFormat`	Defaults to None. As with the YValueAxis, if you supply a function or format string then labels will be drawn next to each bar showing the numeric value.
`barLabels`	A collection of labels used to format all bar labels. Since this is a two-dimensional array, you may explicitly format the third label of the second series using this syntax: chart.barLabels[(1,2)].fontSize = 12
`valueAxis`	The value axis, which may be formatted as described previously.
`categoryAxis`	The category axis, which may be formatted as described previously.
`title`	Not Implemented Yet. This needs to be like a label, but also lets you set the text directly. It would have a default location below the axis.

Table 5-5 - VerticalBarChart properties

From this table we deduce that adding the following lines to our code above should double the spacing between bar groups (the `groupSpacing` attribute has a default value of five points) and we should also see some tiny space between bars of the same group (`barSpacing`).

```
bc.groupSpacing = 10
bc.barSpacing = 2.5
```

And, in fact, this is exactly what we can see after adding these lines to the code above. Notice how the width of the individual bars has changed as well. This is because the space added between the bars has to be 'taken' from somewhere as the total chart width stays unchanged.

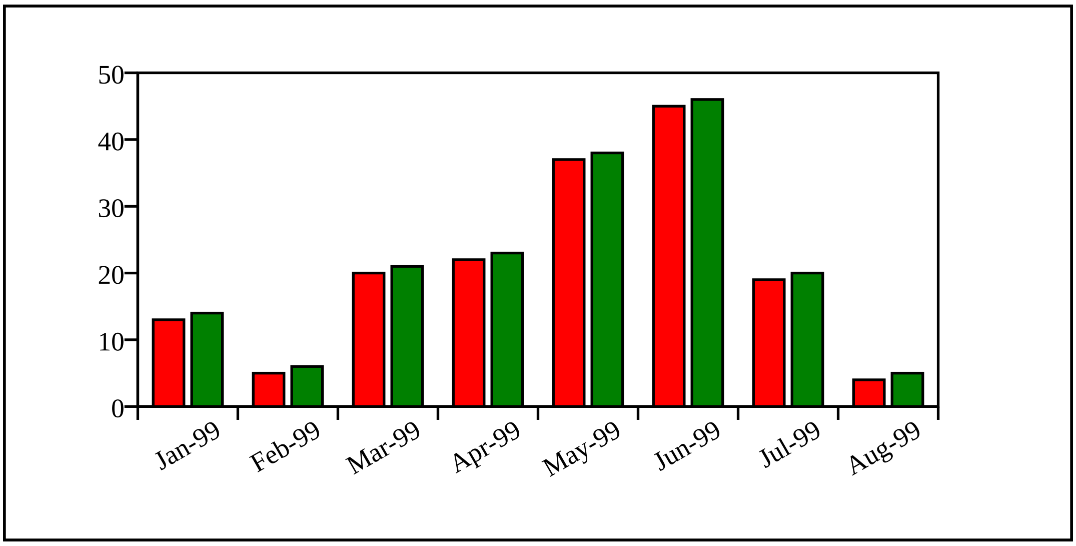


Figure 5-5: Like before, but with modified spacing

Bars labels are automatically displayed for negative values *below* the lower end of the bar for positive values *above* the upper end of the other ones.

5.6 Line Charts

We consider "Line Charts" to be essentially the same as "Bar Charts", but with lines instead of bars. Both share the same pair of Category/Value axes pairs. This is in contrast to "Line Plots", where both axes are *Value* axes.

The following code and its output shall serve as a simple example. More explanation will follow. For the time being you can also study the output of running the tool `reportlab/lib/graphdocpy.py` withough any arguments and search the generated PDF document for examples of Line Charts.

```
from reportlab.graphics.charts.linecharts import HorizontalLineChart

drawing = Drawing(400, 200)

data = [
    (13, 5, 20, 22, 37, 45, 19, 4),
    (5, 20, 46, 38, 23, 21, 6, 14)
]

lc = HorizontalLineChart()
lc.x = 50
lc.y = 50
lc.height = 125
lc.width = 300
lc.data = data
lc.joinedLines = 1
catNames = string.split('Jan Feb Mar Apr May Jun Jul Aug', ' ')
lc.categoryAxis.categoryNames = catNames
lc.categoryAxis.labels.boxAnchor = 'n'
lc.valueAxis.valueMin = 0
lc.valueAxis.valueMax = 60
lc.valueAxis.valueStep = 15
lc.lines[0].strokeWidth = 2
lc.lines[1].strokeWidth = 1.5
drawing.add(lc)
```

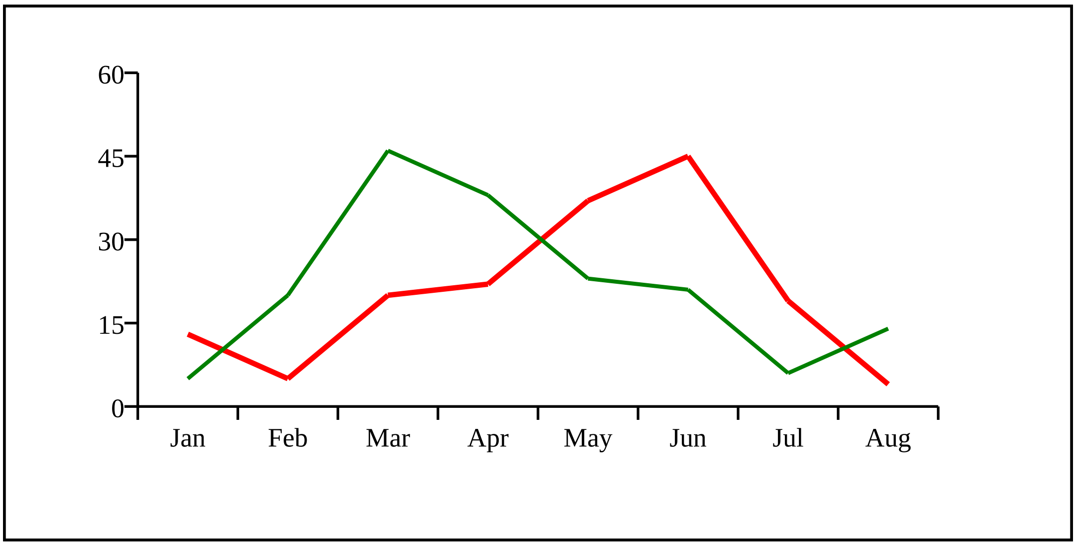


Figure 5-6: HorizontalLineChart sample

Add properties table.

5.7 Line Plots

Below we show a more complex example of a Line Plot that also uses some experimental features like line markers placed at each data point.

```
from reportlab.graphics.charts.lineplots import LinePlot
from reportlab.graphics.charts.markers import makeFilledCircle, makeEmptyCircle

drawing = Drawing(400, 200)

data = [
    ((1,1), (2,2), (2.5,1), (3,3), (4,5)),
    ((1,2), (2,3), (2.5,2), (3.5,5), (4,6))
]

lp = LinePlot()
lp.x = 50
lp.y = 50
lp.height = 125
lp.width = 300
lp.data = data
lp.joinedLines = 1
lp.lines[0].symbol = makeFilledCircle
lp.lines[1].symbol = makeEmptyCircle
lp.lineLabelFormat = '%2.0f'
lp.strokeColor = colors.black
lp.xValueAxis.valueMin = 0
lp.xValueAxis.valueMax = 5
lp.xValueAxis.valueSteps = [1, 2, 2.5, 3, 4, 5]
lp.xValueAxis.labelTextFormat = '%2.1f'
lp.yValueAxis.valueMin = 0
lp.yValueAxis.valueMax = 7
lp.yValueAxis.valueSteps = [1, 2, 3, 5, 6]

drawing.add(lp)
```

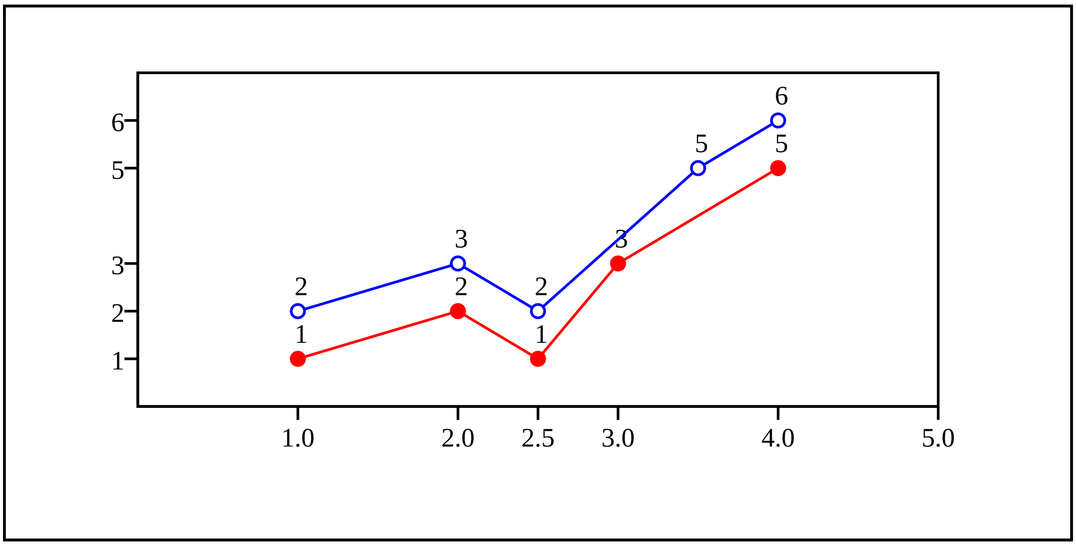


Figure 5-7: LinePlot sample

Add properties table.

5.8 Pie Charts

We've already seen a pie chart example above. This is provisional but seems to do most things. At the very least we need to change the name. For completeness we will cover it here.

```
from reportlab.graphics.charts.piecharts import Pie

d = Drawing(200, 100)

pc = Pie()
pc.x = 65
pc.y = 15
pc.width = 70
pc.height = 70
pc.data = [10,20,30,40,50,60]
pc.labels = ['a','b','c','d','e','f']

pc.slices.strokeWidth=0.5
pc.slices[3].popout = 10
pc.slices[3].strokeWidth = 2
pc.slices[3].strokeDashArray = [2,2]
pc.slices[3].labelRadius = 1.75
pc.slices[3].fontColor = colors.red

d.add(pc)
```

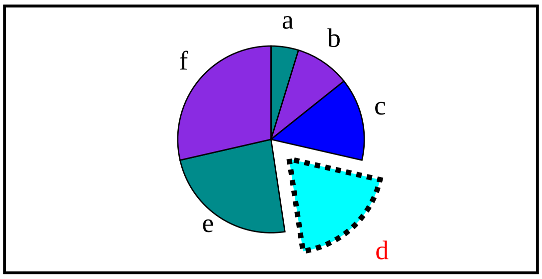


Figure 5-8: A bare bones pie chart

Properties are covered below. The pie has a 'wedges' collection and we document wedge properties in the same table. This was invented before we finished the `Label` class and will probably be reworked to use such labels shortly.

Add properties table.

5.9 Legends

Various preliminary legend classes can be found but need a cleanup to be consistent with the rest of the charting model. Legends are the natural place to specify the colors and line styles of charts; we propose that each chart is created with a `legend` attribute which is invisible. One would then do the following to specify colors:

```
myChart.legend.defaultColors = [red, green, blue]
```

One could also define a group of charts sharing the same legend:

```
myLegend = Legend()
myLegend.defaultColor = [red, green.....] #yuck!
myLegend.columns = 2
# etc.
chart1.legend = myLegend
chart2.legend = myLegend
chart3.legend = myLegend
```

Does this work? Is it an acceptable complication over specifying chart colors directly?

5.10 Remaining Issues

There are several issues that are *almost* solved, but for which is is a bit too early to start making them really public. Nevertheless, here is a list of things that are under way:

7. Color specification - right now the chart has an undocumented property `defaultColors`, which provides a list of colors to cycle through, such that each data series gets its own color. Right now, if you introduce a legend, you need to make sure it shares the same list of colors. Most likely, this will be replaced with a scheme to specify a kind of legend containing attributes with different values for each data series. This legend can then also be shared by several charts, but need not be visible itself.
8. Additional chart types - when the current design will have become more stable, we expect to add variants of bar charts to deal with stacked and percentile bars as well as the side-by-side variant seen here.

5.11 Outlook

It will take some time to deal with the full range of chart types. We expect to finalize bars and pies first and to produce trial implementations of more general plots, thereafter.

X-Y Plots

Most other plots involve two value axes and directly plotting x-y data in some form. The series can be plotted as lines, marker symbols, both, or custom graphics such as open-high-low-close graphics. All share the concepts of scaling and axis/title formatting. At a certain point, a routine will loop over the data series and 'do something' with the data points at given x-y locations. Given a basic line plot, it should be very easy to derive a custom chart type just by overriding a single method - say, `drawSeries()`.

Marker customisation and custom shapes

Well known plotting packages such as excel, Mathematica and Excel offer ranges of marker types to add to charts. We can do better - you can write any kind of chart widget you want and just tell the chart to use it as an example.

Combination plots

Combining multiple plot types is really easy. You can just draw several charts (bar, line or whatever) in the same rectangle, suppressing axes as needed. So a chart could correlate a line with Scottish typhoid cases over a 15 year period on the left axis with a set of bars showing inflation rates on the right axis. If anyone can remind us where this example came from we'll attribute it, and happily show the well-known graph as an example.

Interactive editors

One principle of the Graphics package is to make all 'interesting' properties of its graphic components accessible and changeable by setting apropriate values of corresponding public attributes. This makes it very tempting to build a tool like a GUI editor that that helps you with doing that interactively.

ReportLab has built such a tool using the Tkinter toolkit that loads pure Python code describing a drawing and records your property editing operations. This "change history" is then used to create code for a subclass of that chart, say, that can be saved and used instantly just like any other chart or as a new starting point for another interactive editing session.

This is still work in progress, though, and the conditions for releasing this need to be further elaborated.

Misc.

This has not been an exhaustive look at all the chart classes. Those classes are constantly being worked on. To see exactly what is in the current distribution, use the `graphdocpy.py` utility. By default, it will run on reportlab/graphics, and produce a full report. (If you want to run it on other modules or packages, `graphdocpy.py -h` prints a help message that will tell you how.)

This is the tool that was mentioned in the section on 'Documenting Widgets'.