ReportLab PDF Library

User Guide

ReportLab Version 3.0

Document generated on 2014/03/23 15:59:18

Thornton House
Thornton Road
Wimbledon
London SW19 4NG, UK

Table of contents

Table of contents

Chapter 1 Introduction

1.1 About this document
1.2 What is the ReportLab PDF Library?
1.3 ReportLab's commercial software
1.4 What is Python?
1.5 Acknowledgements
1.6 Installation and Setup
1.7 Getting Involved
1.8 Site Configuration
1.9 Learning More About Python
1.10 Goals for the 3.x release series

Chapter 2 Graphics and Text with pdfgen

2.1 Basic Concepts
2.2 More about the Canvas
2.3 Drawing Operations
2.4 The tools: the "draw" operations
2.5 The toolbox: the "state change" operations
2.6 Other canvas methods.
2.7 Coordinates (default user space)
2.8 Colors
2.9 Color space checking
2.10 Color Overprinting
2.11 Standard fonts and text objects
2.12 Text object methods
2.13 Paths and Lines
2.14 Rectangles, circles, ellipses
2.15 Bezier curves
2.16 Path object methods
2.17 Further Reading: The ReportLab Graphics Library

Chapter 3 Fonts and encodings

3.1 Unicode and UTF8 are the default input encodings
3.2 Automatic output font substitution

3.3 Using non-standard Type 1 fonts
3.4 Standard Single-Byte Font Encodings
3.5 TrueType Font Support
3.6 Asian Font Support
3.7 RenderPM tests

Chapter 4 Exposing PDF Special Capabilities

4.1 Forms
4.2 Links and Destinations
4.3 Outline Trees
4.4 Page Transition Effects
4.5 Internal File Annotations
4.6 Encryption

Chapter 5 PLATYPUS - Page Layout and Typography Using Scripts

5.1 Design Goals
5.2 Getting started
5.3 Flowables
5.4 Guidelines for flowable positioning
5.5 Frames
5.6 Documents and Templates

Chapter 6 Paragraphs

6.1 Using Paragraph Styles
6.2 Paragraph XML Markup Tags
6.3 Intra-paragraph markup
6.4 Bullets and Paragraph Numbering

Chapter 7 Tables and TableStyles

7.1 Table User Methods
7.2 TableStyle
7.3 TableStyle User Methods
7.4 TableStyle Commands

Chapter 8 Programming Flowables

8.1 DocAssign(self, var, expr, life='forever')
8.2 DocExec(self, stmt, lifetime='forever')
8.3 DocPara(self, expr, format=None, style=None, klass=None, escape=True)

8.4 DocAssert(self, cond, format=None)
8.5 DocIf(self, cond, thenBlock, elseBlock=[])
8.6 DocWhile(self, cond, whileBlock)

Chapter 9 Other Useful Flowables

9.1 Preformatted(text, style, bulletText=None, dedent=0, maxLineLength=None, splitChars=None, newLineChars=None)
9.2 XPreformatted(text, style, bulletText=None, dedent=0, frags=None)
9.3 Image(filename, width=None, height=None)
9.4 Spacer(width, height)
9.5 PageBreak()
9.6 CondPageBreak(height)
9.7 KeepTogether(flowables)
9.8 TableOfContents()
9.9 SimpleIndex()
9.10 ListFlowable(),ListItem()

Chapter 10 Writing your own Flowable Objects

10.1 A very simple Flowable
10.2 Modifying a Built in Flowable

Chapter 11 Graphics

11.1 Introduction
11.2 General Concepts
11.3 Charts
11.4 Labels
11.5 Axes
11.6 Bar Charts
11.7 Line Charts
11.8 Line Plots
11.9 Pie Charts
11.10 Legends
11.11 Shapes
11.12 Widgets

Appendix A ReportLab Demos

A.1 Odyssey
A.2 Standard Fonts and Colors
A.3 Py2pdf

A.4 Gadflypaper
A.5 Pythonpoint

Chapter 1 Introduction

1.1 About this document

This document is an introduction to the ReportLab PDF library. 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.

This manual does not cover 100% of the features, but should explain all the main concepts and help you get started, and point you at other learning resources. After working your way through 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?

We need your help to make sure this manual is complete and helpful. Please send any feedback to our user mailing list, which is signposted from www.reportlab.com.

1.2 What is the ReportLab PDF Library?

This is a software library that lets you directly create documents in Adobe's Portable Document Format (PDF) using the Python programming language. It also creates charts and data graphics in various bitmap and vector formats as well as PDF.

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 or driving a printer 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. This makes it harder to generate than HTML.

Most of the world's PDF documents have been produced by Adobe's Acrobat tools, or rivals such as JAWS PDF Creator, which act as 'print drivers'. Anyone wanting to automate PDF production would typically use a product like Quark, Word or Framemaker running in a loop with macros or plugins, connected to Acrobat. Pipelines of several languages and products can be slow and somewhat unwieldy.

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. This approach is shared by several other libraries - PDFlib for C, iText for Java, iTextSharp for .NET and others. However, The ReportLab library differs in that it can work at much higher levels, with a full featured engine for laying out documents complete with tables and charts.

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 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 ReportLab's commercial software

The ReportLab library forms the foundation of our commercial solution for PDF generation, Report Markup Language (RML). This is available for evaluation on our web site with full documentation. We believe that RML is the fastest and easiest way to develop rich PDF workflows. You work in a markup language at a similar level to HTML, using your favorite templating system to populate an RML document; then call our rml2pdf API function to generate a PDF. It's what ReportLab staff use to build all of the solutions you can see on reportlab.com. Key differences:

- ■■2022 Fully documented with two manuals, a formal specification (the DTD) and extensive self-documenting tests. (By contrast, we try to make sure the open source documentation isn't wrong, but we don't always keep up with the code)
- ■■2022 Work in high-level markup rather than constructing graphs of Python objects
- ■■2022 Requires no Python expertise - your colleagues may thank you after you've left!'
- ■■2022 Support for vector graphics and inclusion of other PDF documents
- ■■2022 Many more useful features expressed with a single tag, which would need a lot of coding in the open source package
- ■■2022 Commercial support is included

We ask open source developers to consider trying out RML where it is appropriate. You can register on our site and try out a copy before buying. The costs are reasonable and linked to the volume of the project, and the revenue helps us spend more time developing this software.

1.4 What is Python?

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.

Python is as old as Java and has been growing steadily in popularity for years; since our library first came out it has entered the mainstream. Many ReportLab library users are already Python devotees, but if you are not, we feel that the language is an excellent choice for document-generation apps because of its expressiveness and ability to get data from anywhere.

Python is copyrighted but **freely usable and distributable, even for commercial use**.

1.5 Acknowledgements

Many people have contributed to ReportLab. We would like to thank in particular (in alphabetical order): Albertas Agejevas, Alex Buck, Andre Reitz, Andrew Cutler, Andrew Mercer, Ben Echols, Benjamin Dumke, Benn B, Chad Miller, Chris Lee, Christian Jacobs, Dinu Gherman, Eric Johnson, Felix Labrecque, Gary Poster, Germán M. Bravo, Guillaume Francois, Hans Brand, Henning Vonbargen, Hosam Aly, Ian Stevens, James Martin-Collar, Jeff Bauer, Jerome Alet, Jerry Casiano, Jorge Godoy, Keven D Smith, Magnus Lie Hetland, Marcel Tromp, Marius Gedminas, Matthew Duggan, Matthias Kirst, Matthias Klose, Max M, Michael Egorov, Mike Folwell, Mirko Dziadzka, Moshe Wagner, Nate Silva, Paul McNett, Peter Johnson, PJACock, Publio da Costa Melo, Randolph Bentson, Robert Alsina, Robert Hölzl, Robert Kern, Ron Peleg, Simon King, Stephan Richter, Steve Halasz, T Blatter, Tim Roberts, Tomasz Swiderski, Ty Sarna, Volker Haas, Yoann Roman, and many more.

Special thanks go to Just van Rossum for his valuable assistance with font technicalities.

Moshe Wagner and Hosam Aly deserve a huge thanks for contributing to the RTL patch, which is not yet on thr trunk.

Marius Gedminas deserves a big hand for contributing the work on TrueType fonts and we are glad to include these in the toolkit. Finally we thank Michal Kosmulski for the DarkGarden font for and Bitstream

Inc. for the Vera fonts.

1.6 Installation and Setup

To avoid duplication, the installation instructions are kept in the README file in our distribution, which can be viewed online at *http://bitbucket.org/rptlab/reportlab/*

This release (3.0) of ReportLab requires Python versions 2.7, 3.3 or higher. If you need to use Python 2.5 or 2.6, please use the latest ReportLab 2.x package.

1.7 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:

- ■■2022 General feedback on the core API. Does it work for you? Are there any rough edges? Does anything feel clunky and awkward?
- ■■2022 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?
- ■■2022 Snippets and Case Studies: If you have produced some nice output, register online on *http://www.reportlab.com* and submit a snippet of your output (with or without scripts). If ReportLab solved a problem for you at work, write a little 'case study' and submit it. 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!
- ■■2022 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. To Subscribe visit `http://two.pairlist.net/mailman/listinfo/reportlab-users`. From there you can also browse through the group's archives and contributions. The mailing list is the place to report bugs and get support.

The code now lives on BitBucket (`http://bitbucket.org/rptlab/reportlab/`) in a Mercurial repository, along with an issue tracker and wiki. Everyone should feel free to contribute, but if you are working actively on some improvements or want to draw attention to an issue, please use the mailing list to let us know.

1.8 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.

- ■■2022 verbose: set to integer values to control diagnostic output.
- ■■2022 shapeChecking: set this to zero to turn off a lot of error checking in the graphics modules
- ■■2022 defaultEncoding: set this to WinAnsiEncoding or MacRomanEncoding.
- ■■2022 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.
- ■■2022 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
- ■■2022 T1SearchPath: this is a python list of strings representing directories that may be queried for information on Type 1 fonts
- ■■2022 TTFSearchPath: this is a python list of strings representing directories that may be queried for information on TrueType fonts
- ■■2022 CMapSearchPath: this is a python list of strings representing directories that may be queried for information on font code maps.
- ■■2022 showBoundary: set to non-zero to get boundary lines drawn.

- ■■2022 ZLIB_WARNINGS: set to non-zero to get warnings if the Python compression extension is not found.
- ■■2022 pageComression: set to non-zero to try and get compressed PDF.
- ■■2022 allowtableBoundsErrors: set to 0 to force an error on very large Platypus table elements
- ■■2022 emptyTableAction: Controls behaviour for empty tables, can be 'error' (default), 'indicate' or 'ignore'.

1.9 Learning More About Python

If you are a total beginner to Python, you should check out one or more from the growing number of resources on Python programming. The following are freely available on the web:

- ■■2022 **Python Documentation.** A list of documentation on the Python.org web site.
  `http://www.python.org/doc/`
- ■■2022 **Python Tutorial.** The official Python Tutorial , originally written by Guido van Rossum himself. `http://docs.python.org/tutorial/`
- ■■2022 **Learning to Program.** A tutorial on programming by Alan Gauld. Has a heavy emphasis on Python, but also uses other languages.
  `http://www.freenetpages.co.uk/hp/alan.gauld/`
- ■■2022 **Instant Python**. A 6-page minimal crash course by Magnus Lie Hetland.
  `http://www.hetland.org/python/instant-python.php`
- ■■2022 **Dive Into Python**. A free Python tutorial for experienced programmers.
  `http://www.diveintopython.net/`

1.10 Goals for the 3.x release series

ReportLab 3.0 has been produced to help in the migration to Python 3.x. Python 3.x will be standard in future Ubuntu releases and is gaining popularity, and a good proportion of major Python packages now run on Python 3.

- ■■2022 Python 3.x compatibility. A single line of code should run on 2.7 and 3.3
- ■■2022 __init__.py restricts to 2.7 or >=3.3
- ■■2022 __init__.py allow the import of on optional reportlab.local_rl_mods to allow monkey patching etc.
- ■■2022 rl_config now imports rl_settings & optionally local_rl_settings
- ■■2022 ReportLab C extensions now live inside reportlab; _rl_accel is no longer required. All _rl_accel imports now pass through reportlab.lib.rl_accel
- ■■2022 xmllib is gone, alongside the paraparser stuff that caused issues in favour of HTMLParser.
- ■■2022 some obsolete C extensions (sgmlop and pyHnj) are gone
- ■■2022 Improved support for multi-threaded systems to the _rl_accel C extension module.
- ■■2022 Removed reportlab/lib/ para.py & pycanvas.py. These would better belong in third party packages, which can make use of the monkeypatching feature above.
- ■■2022 Add ability to output greyscale and 1-bit PIL images without conversion to RGB. (contributed by Matthew Duggan)
- ■■2022 highlight annotation (contributed by Ben Echols)
- ■■2022 full compliance with pip, easy_install, wheels etc

Detailed release notes are available at
`http://www.reportlab.com/software/documentation/relnotes/30/`

Chapter 2 Graphics and Text with pdfgen

2.1 Basic Concepts

The pdfgen package is the lowest level interface for generating PDF documents. A pdfgen program is essentially a sequence of instructions for "painting" a document onto a sequence of pages. The interface object which provides the painting operations is the pdfgen canvas.

The canvas should be thought of as a sheet of white paper with points on the sheet identified using Cartesian *(X,Y)* coordinates which by default have the *(0,0)* origin point at the lower left corner of the page. Furthermore the first coordinate *x* goes to the right and the second coordinate *y* goes up, by default.

A simple example program that uses a canvas follows.

```
from reportlab.pdfgen import canvas
def hello(c):
    c.drawString(100,100,"Hello World")
c = canvas.Canvas("hello.pdf")
hello(c)
c.showPage()
c.save()
```

The above code creates a canvas object which will generate a PDF file named hello.pdf in the current working directory. It then calls the hello function passing the canvas as an argument. Finally the showPage method saves the current page of the canvas and the save method stores the file and closes the canvas.

The showPage method causes the canvas to stop drawing on the current page and any further operations will draw on a subsequent page (if there are any further operations -- if not no new page is created). The save method must be called after the construction of the document is complete -- it generates the PDF document, which is the whole purpose of the canvas object.

2.2 More about the Canvas

Before describing the drawing operations, we will digress to cover some of the things which can be done to configure a canvas. There are many different settings available. If you are new to Python or can't wait to produce some output, you can skip ahead, but come back later and read this!

First of all, we will look at the constructor arguments for the canvas:

```
def __init__(self,filename,
             pagesize=(595.27,841.89),
             bottomup = 1,
             pageCompression=0,
             encoding=rl_config.defaultEncoding,
             verbosity=0
             encrypt=None):
```

The filename argument controls the name of the final PDF file. You may also pass in any open file object (such as sys.stdout, the python process standard output) and the PDF document will be written to that. Since PDF is a binary format, you should take care when writing other stuff before or after it; you can't deliver PDF documents inline in the middle of an HTML page!

The pagesize argument is a tuple of two numbers in points (1/72 of an inch). The canvas defaults to A4 (an international standard page size which differs from the American standard page size of letter), but it is better to explicitly specify it. Most common page sizes are found in the library module reportlab.lib.pagesizes, so you can use expressions like

```
from reportlab.lib.pagesizes import letter, A4
myCanvas = Canvas('myfile.pdf', pagesize=letter)
width, height = letter  #keep for later
```

If you have problems printing your document make sure you are using the right page size (usually either `A4` or `letter`). Some printers do not work well with pages that are too large or too small.

Very often, you will want to calculate things based on the page size. In the example above we extracted the width and height. Later in the program we may use the `width` variable to define a right margin as `width - inch` rather than using a constant. By using variables the margin will still make sense even if the page size changes.

The `bottomup` argument switches coordinate systems. Some graphics systems (like PDF and PostScript) place (0,0) at the bottom left of the page others (like many graphical user interfaces [GUI's]) place the origen at the top left. The `bottomup` argument is deprecated and may be dropped in future
Need to see if it really works for all tasks, and if not then get rid of it

The `pageCompression` option determines whether the stream of PDF operations for each page is compressed. By default page streams are not compressed, because the compression slows the file generation process. If output size is important set `pageCompression=1`, but remember that, compressed documents will be smaller, but slower to generate. Note that images are *always* compressed, and this option will only save space if you have a very large amount of text and vector graphics on each page.

The `encoding` argument is largely obsolete in version 2.0 and can probably be omitted by 99% of users. Its default value is fine unless you very specifically need to use one of the 25 or so characters which are present in MacRoman and not in Winansi. A useful reference to these is here: http://www.alanwood.net/demos/charsetdiffs.html. The parameter determines which font encoding is used for the standard Type 1 fonts; this should correspond to the encoding on your system. Note that this is the encoding used *internally by the font*; text you pass to the ReportLab toolkit for rendering should always either be a Python unicode string object or a UTF-8 encoded byte string (see the next chapter)! The font encoding has two values at present: `'WinAnsiEncoding'` or `'MacRomanEncoding'`. The variable `rl_config.defaultEncoding` above points to the former, which is standard on Windows, Mac OS X and many Unices (including Linux). If you are Mac user and don't have OS X, you may want to make a global change: modify the line at the top of *reportlab/pdfbase/pdfdoc.py* to switch it over. Otherwise, you can probably just ignore this argument completely and never pass it. For all TTF and the commonly-used CID fonts, the encoding you pass in here is ignored, since the reportlab library itself knows the right encodings in those cases.

The demo script `reportlab/demos/stdfonts.py` will print out two test documents showing all code points in all fonts, so you can look up characters. Special characters can be inserted into string commands with the usual Python escape sequences; for example \101 = 'A'.

The `verbosity` argument determines how much log information is printed. By default, it is zero to assist applications which want to capture PDF from standard output. With a value of 1, you will get a confirmation message each time a document is generated. Higher numbers may give more output in future.

The `encrypt` argument determines if and how the document is encrypted. By default, the document is not encrypted. If `encrypt` is a string object, it is used as the user password for the pdf. If `encrypt` is an instance of `reportlab.lib.pdfencrypt.StandardEncryption`, this object is used to encrypt the pdf. This allows more finegrained control over the encryption settings. Encryption is covered in more detail in Chapter 4.
to do - all the info functions and other non-drawing stuff
Cover all constructor arguments, and setAuthor etc.

2.3 Drawing Operations

Suppose the `hello` function referenced above is implemented as follows (we will not explain each of the operations in detail yet).

```
def hello(c):
    from reportlab.lib.units import inch
    # move the origin up and to the left
    c.translate(inch,inch)
    # define a large font
    c.setFont("Helvetica", 14)
    # choose some colors
```

```
c.setStrokeColorRGB(0.2,0.5,0.3)
c.setFillColorRGB(1,0,1)
# draw some lines
c.line(0,0,0,1.7*inch)
c.line(0,0,1*inch,0)
# draw a rectangle
c.rect(0.2*inch,0.2*inch,1*inch,1.5*inch, fill=1)
# make text go straight up
c.rotate(90)
# change color
c.setFillColorRGB(0,0,0.77)
# say hello (note after rotate the y coord needs to be negative!)
c.drawString(0.3*inch, -inch, "Hello World")
```

Examining this code notice that there are essentially two types of operations performed using a canvas. The first type draws something on the page such as a text string or a rectangle or a line. The second type changes the state of the canvas such as changing the current fill or stroke color or changing the current font type and size.

If we imagine the program as a painter working on the canvas the "draw" operations apply paint to the canvas using the current set of tools (colors, line styles, fonts, etcetera) and the "state change" operations change one of the current tools (changing the fill color from whatever it was to blue, or changing the current font to `Times-Roman` in 15 points, for example).

The document generated by the "hello world" program listed above would contain the following graphics.

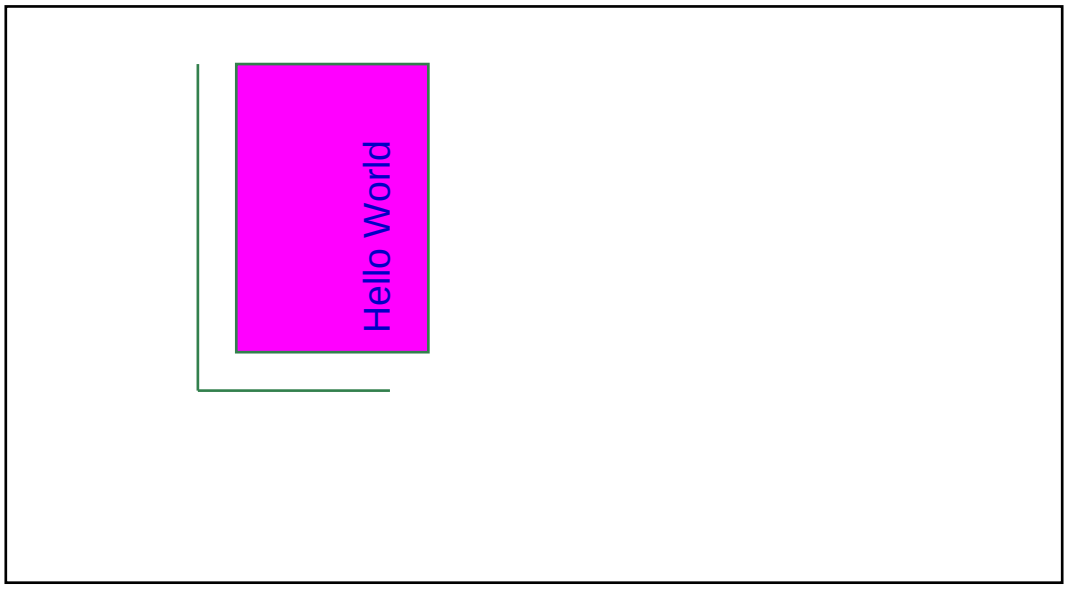


Figure 2-1: "Hello World" in pdfgen

About the demos in this document

This document contains demonstrations of the code discussed like the one shown in the rectangle above. These demos are drawn on a "tiny page" embedded within the real pages of the guide. The tiny pages are 5.5 inches wide and 3 inches tall. The demo displays show the actual output of the demo code. For convenience the size of the output has been reduced slightly.

2.4 The tools: the "draw" operations

This section briefly lists the tools available to the program for painting information onto a page using the canvas interface. These will be discussed in detail in later sections. They are listed here for easy reference and for summary purposes.

Line methods

```
canvas.line(x1,y1,x2,y2)

canvas.lines(linelist)
```

The line methods draw straight line segments on the canvas.

Shape methods

```
canvas.grid(xlist, ylist)

canvas.bezier(x1, y1, x2, y2, x3, y3, x4, y4)

canvas.arc(x1,y1,x2,y2)

canvas.rect(x, y, width, height, stroke=1, fill=0)

canvas.ellipse(x1,y1, x2,y2, stroke=1, fill=0)

canvas.wedge(x1,y1, x2,y2, startAng, extent, stroke=1, fill=0)

canvas.circle(x_cen, y_cen, r, stroke=1, fill=0)

canvas.roundRect(x, y, width, height, radius, stroke=1, fill=0)
```

The shape methods draw common complex shapes on the canvas.

String drawing methods

```
canvas.drawString(x, y, text):

canvas.drawRightString(x, y, text)

canvas.drawCentredString(x, y, text)
```

The draw string methods draw single lines of text on the canvas.

The text object methods

```
textobject = canvas.beginText(x, y)

canvas.drawText(textobject)
```

Text objects are used to format text in ways that are not supported directly by the `canvas` interface. A program creates a text object from the `canvas` using `beginText` and then formats text by invoking `textobject` methods. Finally the `textobject` is drawn onto the canvas using `drawText`.

The path object methods

```
path = canvas.beginPath()

canvas.drawPath(path, stroke=1, fill=0)

canvas.clipPath(path, stroke=1, fill=0)
```

Path objects are similar to text objects: they provide dedicated control for performing complex graphical drawing not directly provided by the canvas interface. A program creates a path object using `beginPath` populates the path with graphics using the methods of the path object and then draws the path on the canvas

using drawPath.

It is also possible to use a path as a "clipping region" using the clipPath method -- for example a circular path can be used to clip away the outer parts of a rectangular image leaving only a circular part of the image visible on the page.

Image methods

You need the Python Imaging Library (PIL) to use images with the ReportLab package. Examnples of the techniques below can be found by running the script test_pdfgen_general.py in our tests subdirectory and looking at page 7 of the output.

There are two similar-sounding ways to draw images. The preferred one is the drawImage method. This implements a caching system so you can define an image once and draw it many times; it will only be stored once in the PDF file. drawImage also exposes one advanced parameter, a transparency mask, and will expose more in future. The older technique, drawInlineImage, stores bitmaps within the page stream and is thus very inefficient if you use the same image more than once in a document; but can result in PDFs which render faster if the images are very small and not repeated. We'll discuss the oldest one first:

```
canvas.drawInlineImage(self, image, x,y, width=None,height=None)
```

The drawInlineImage method places an image on the canvas. The image parameter may be either a PIL Image object or an image filename. Many common file formats are accepted including GIF and JPEG. It returns the size of the actual image in pixels as a (width, height) tuple.

```
canvas.drawImage(self, image, x,y, width=None,height=None,mask=None)
```

The arguments and return value work as for drawInlineImage. However, we use a caching system; a given image will only be stored the first time it is used, and just referenced on subsequent use. If you supply a filename, it assumes that the same filename means the same image. If you supply a PIL image, it tests if the content has actually changed before re-embedding.

The mask parameter lets you create transparent images. It takes 6 numbers and defines the range of RGB values which will be masked out or treated as transparent. For example with [0,2,40,42,136,139], it will mask out any pixels with a Red value from 0 or 1, Green from 40 or 41 and Blue of 136, 137 or 138 (on a scale of 0-255). It's currently your job to know which color is the 'transparent' or background one.

PDF allows for many image features and we will expose more of the over time, probably with extra keyword arguments to drawImage.

Ending a page

```
canvas.showPage()
```

The showPage method finishes the current page. All additional drawing will be done on another page.

Warning! All state changes (font changes, color settings, geometry transforms, etcetera) are FORGOTTEN when you advance to a new page in pdfgen. Any state settings you wish to preserve must be set up again before the program proceeds with drawing!

2.5 The toolbox: the "state change" operations

This section briefly lists the ways to switch the tools used by the program for painting information onto a page using the canvas interface. These too will be discussed in detail in later sections.

Changing Colors

```
canvas.setFillColorCMYK(c, m, y, k)

canvas.setStrikeColorCMYK(c, m, y, k)

canvas.setFillColorRGB(r, g, b)

canvas.setStrokeColorRGB(r, g, b)

canvas.setFillColor(acolor)

canvas.setStrokeColor(acolor)

canvas.setFillGray(gray)

canvas.setStrokeGray(gray)
```

PDF supports three different color models: gray level, additive (red/green/blue or RGB), and subtractive with darkness parameter (cyan/magenta/yellow/darkness or CMYK). The ReportLab packages also provide named colors such as `lawngreen`. There are two basic color parameters in the graphics state: the `Fill` color for the interior of graphic figures and the `Stroke` color for the boundary of graphic figures. The above methods support setting the fill or stroke color using any of the four color specifications.

Changing Fonts

```
canvas.setFont(psfontname, size, leading = None)
```

The `setFont` method changes the current text font to a given type and size. The `leading` parameter specifies the distance down to move when advancing from one text line to the next.

Changing Graphical Line Styles

```
canvas.setLineWidth(width)

canvas.setLineCap(mode)

canvas.setLineJoin(mode)

canvas.setMiterLimit(limit)

canvas.setDash(self, array=[], phase=0)
```

Lines drawn in PDF can be presented in a number of graphical styles. Lines can have different widths, they can end in differing cap styles, they can meet in different join styles, and they can be continuous or they can be dotted or dashed. The above methods adjust these various parameters.

Changing Geometry

```
canvas.setPageSize(pair)

canvas.transform(a,b,c,d,e,f):

canvas.translate(dx, dy)

canvas.scale(x, y)

canvas.rotate(theta)
```

```
canvas.skew(alpha, beta)
```

All PDF drawings fit into a specified page size. Elements drawn outside of the specified page size are not visible. Furthermore all drawn elements are passed through an affine transformation which may adjust their location and/or distort their appearence. The `setPageSize` method adjusts the current page size. The `transform`, `translate`, `scale`, `rotate`, and `skew` methods add additional transformations to the current transformation. It is important to remember that these transformations are *incremental* -- a new transform modifies the current transform (but does not replace it).

State control

```
canvas.saveState()

canvas.restoreState()
```

Very often it is important to save the current font, graphics transform, line styles and other graphics state in order to restore them later. The `saveState` method marks the current graphics state for later restoration by a matching `restoreState`. Note that the save and restore method invokation must match -- a restore call restores the state to the most recently saved state which hasn't been restored yet. You cannot save the state on one page and restore it on the next, however -- no state is preserved between pages.

2.6 Other `canvas` methods.

Not all methods of the `canvas` object fit into the "tool" or "toolbox" categories. Below are some of the misfits, included here for completeness.

```
canvas.setAuthor()
canvas.addOutlineEntry(title, key, level=0, closed=None)
canvas.setTitle(title)
canvas.setSubject(subj)
canvas.pageHasData()
canvas.showOutline()
canvas.bookmarkPage(name)
canvas.bookmarkHorizontalAbsolute(name, yhorizontal)
canvas.doForm()
canvas.beginForm(name, lowerx=0, lowery=0, upperx=None, uppery=None)
canvas.endForm()
canvas.linkAbsolute(contents, destinationname, Rect=None, addtopage=1, name=None, **kw)
canvas.linkRect(contents, destinationname, Rect=None, addtopage=1, relative=1, name=None, **kw)
canvas.getPageNumber()
canvas.addLiteral()
canvas.getAvailableFonts()
canvas.stringWidth(self, text, fontName, fontSize, encoding=None)
canvas.setPageCompression(onoff=1)
canvas.setPageTransition(self, effectname=None, duration=1,
                         direction=0,dimension='H',motion='I')
```

2.7 Coordinates (default user space)

By default locations on a page are identified by a pair of numbers. For example the pair `(4.5*inch, 1*inch)` identifies the location found on the page by starting at the lower left corner and moving to the right 4.5 inches and up one inch.

For example, the following function draws a number of elements on a `canvas`.

```
def coords(canvas):
    from reportlab.lib.units import inch
    from reportlab.lib.colors import pink, black, red, blue, green
    c = canvas
    c.setStrokeColor(pink)
    c.grid([inch, 2*inch, 3*inch, 4*inch], [0.5*inch, inch, 1.5*inch, 2*inch, 2.5*inch])
    c.setStrokeColor(black)
    c.setFont("Times-Roman", 20)
    c.drawString(0,0, "(0,0) the Origin")
    c.drawString(2.5*inch, inch, "(2.5,1) in inches")
    c.drawString(4*inch, 2.5*inch, "(4, 2.5)")
    c.setFillColor(red)
```

```
c.rect(0,2*inch,0.2*inch,0.3*inch, fill=1)
c.setFillColor(green)
c.circle(4.5*inch, 0.4*inch, 0.2*inch, fill=1)
```

In the default user space the "origin" *(0,0)* point is at the lower left corner. Executing the `coords` function in the default user space (for the "demo minipage") we obtain the following.

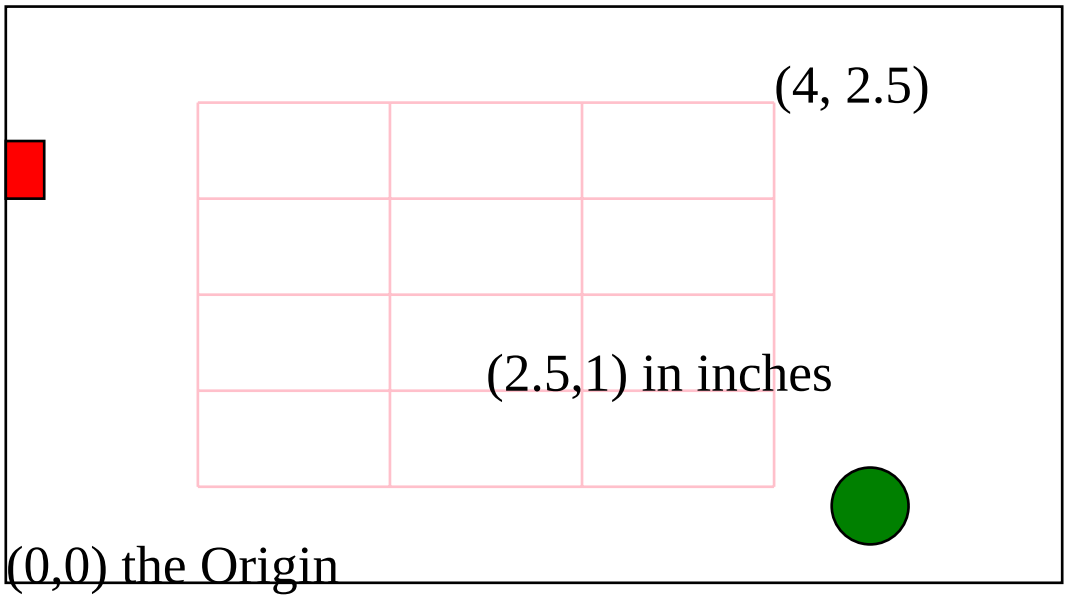


Figure 2-2: The Coordinate System

Moving the origin: the `translate` method

Often it is useful to "move the origin" to a new point off the lower left corner. The `canvas.translate(x,y)` method moves the origin for the current page to the point currently identified by *(x,y)*.

For example the following translate function first moves the origin before drawing the same objects as shown above.

```
def translate(canvas):
    from reportlab.lib.units import cm
    canvas.translate(2.3*cm, 0.3*cm)
    coords(canvas)
```

This produces the following.

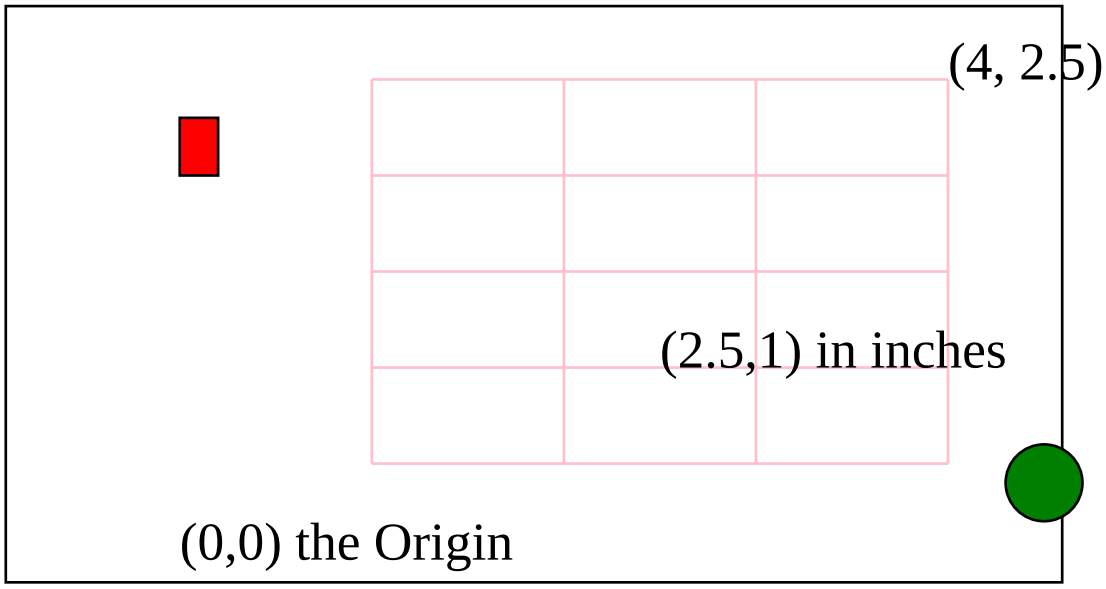


Figure 2-3: Moving the origin: the `translate` method

Note: As illustrated in the example it is perfectly possible to draw objects or parts of objects "off the page". In particular a common confusing bug is a translation operation that translates the entire drawing off the visible area of the page. If a program produces a blank page it is possible that all the drawn objects are off the page.

Shrinking and growing: the scale operation

Another important operation is scaling. The scaling operation `canvas.scale(dx,dy)` stretches or shrinks the *x* and *y* dimensions by the *dx*, *dy* factors respectively. Often *dx* and *dy* are the same -- for example to reduce a drawing by half in all dimensions use `dx = dy = 0.5`. However for the purposes of illustration we show an example where `dx` and `dy` are different.

```
def scale(canvas):
    canvas.scale(0.75, 0.5)
    coords(canvas)
```

This produces a "short and fat" reduced version of the previously displayed operations.

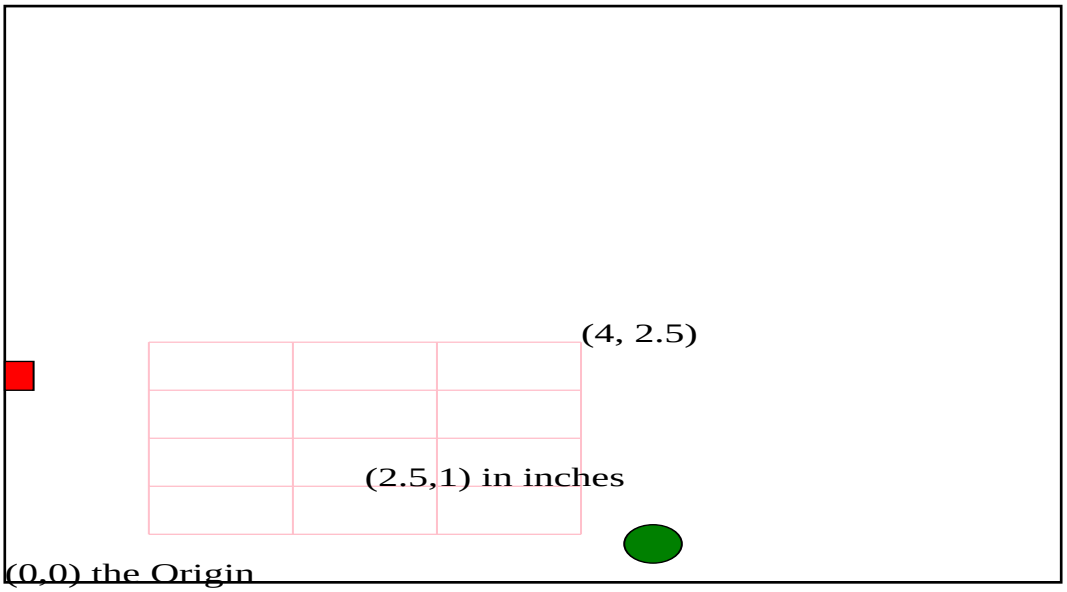


Figure 2-4: Scaling the coordinate system

Note: scaling may also move objects or parts of objects off the page, or may cause objects to "shrink to nothing."

Scaling and translation can be combined, but the order of the operations are important.

```
def scaletranslate(canvas):
    from reportlab.lib.units import inch
    canvas.setFont("Courier-BoldOblique", 12)
    # save the state
    canvas.saveState()
    # scale then translate
    canvas.scale(0.3, 0.5)
    canvas.translate(2.4*inch, 1.5*inch)
    canvas.drawString(0, 2.7*inch, "Scale then translate")
    coords(canvas)
    # forget the scale and translate...
    canvas.restoreState()
    # translate then scale
    canvas.translate(2.4*inch, 1.5*inch)
    canvas.scale(0.3, 0.5)
    canvas.drawString(0, 2.7*inch, "Translate then scale")
    coords(canvas)
```

This example function first saves the current `canvas` state and then does a `scale` followed by a `translate`. Afterward the function restores the state (effectively removing the effects of the scaling and translation) and then does the *same* operations in a different order. Observe the effect below.

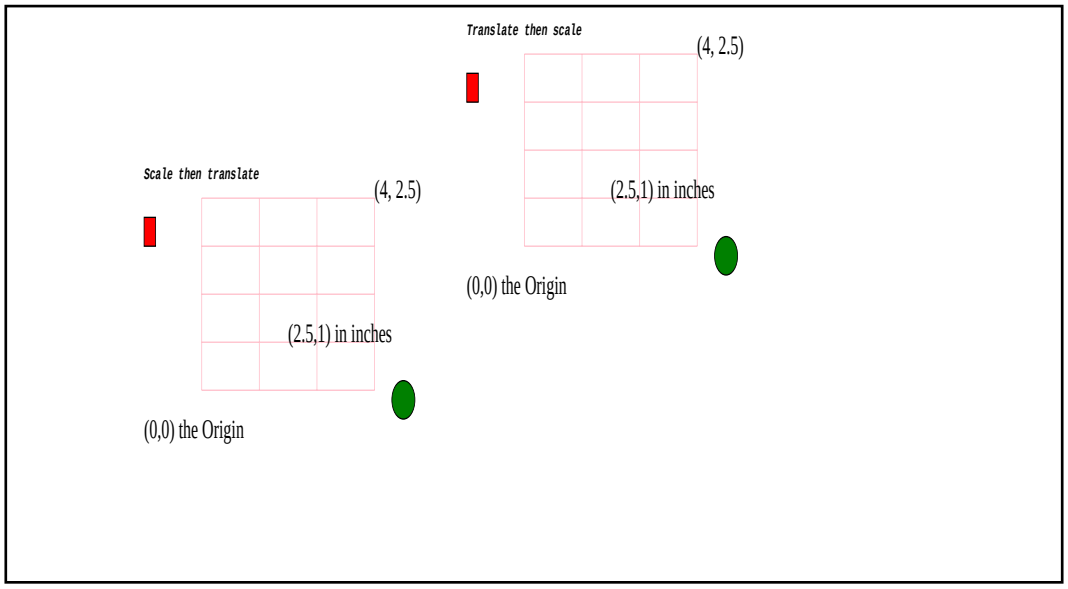


Figure 2-5: Scaling and Translating

Note: scaling shrinks or grows everything including line widths so using the canvas.scale method to render a microscopic drawing in scaled microscopic units may produce a blob (because all line widths will get expanded a huge amount). Also rendering an aircraft wing in meters scaled to centimeters may cause the lines to shrink to the point where they disappear. For engineering or scientific purposes such as these scale and translate the units externally before rendering them using the canvas.

Saving and restoring the `canvas` state: `saveState` and `restoreState`

The `scaletranslate` function used an important feature of the `canvas` object: the ability to save and restore the current parameters of the `canvas`. By enclosing a sequence of operations in a matching pair of `canvas.saveState()` an `canvas.restoreState()` operations all changes of font, color, line style, scaling, translation, or other aspects of the `canvas` graphics state can be restored to the state at the point of the `saveState()`. Remember that the save/restore calls must match: a stray save or restore operation may cause unexpected and undesirable behavior. Also, remember that *no* `canvas` state is preserved across page breaks, and the save/restore mechanism does not work across page breaks.

Mirror image

It is interesting although perhaps not terribly useful to note that scale factors can be negative. For example the following function

```
def mirror(canvas):
    from reportlab.lib.units import inch
    canvas.translate(5.5*inch, 0)
    canvas.scale(-1.0, 1.0)
    coords(canvas)
```

creates a mirror image of the elements drawn by the `coord` function.

Figure 2-6: Mirror Images

Notice that the text strings are painted backwards.

2.8 Colors

There are generally two types of colors used in PDF depending on the media where the PDF will be used. The most commonly known screen colors model RGB can be used in PDF, however in professional printing another color model CMYK is mainly used which gives more control over how inks are applied to paper. More on these color models below.

RGB Colors

The `RGB` or additive color representation follows the way a computer screen adds different levels of the red, green, and blue light to make any color in between, where white is formed by turning all three lights on full (`1,1,1`).

There are three ways to specify RGB colors in `pdfgen`: by name (using the `color` module, by red/green/blue (additive, `RGB`) value, or by gray level. The `colors` function below exercises each of the four methods.

```
def colorsRGB(canvas):
    from reportlab.lib import colors
    from reportlab.lib.units import inch
    black = colors.black
    y = x = 0; dy=inch*3/4.0; dx=inch*5.5/5; w=h=dy/2; rdx=(dx-w)/2
    rdy=h/5.0; texty=h+2*rdy
    canvas.setFont("Helvetica",10)
    for [namedcolor, name] in (
           [colors.lavenderblush, "lavenderblush"],
           [colors.lawngreen, "lawngreen"],
           [colors.lemonchiffon, "lemonchiffon"],
           [colors.lightblue, "lightblue"],
           [colors.lightcoral, "lightcoral"]):
        canvas.setFillColor(namedcolor)
        canvas.rect(x+rdx, y+rdy, w, h, fill=1)
        canvas.setFillColor(black)
        canvas.drawCentredString(x+dx/2, y+texty, name)
        x = x+dx
    y = y + dy; x = 0
    for rgb in [(1,0,0), (0,1,0), (0,0,1), (0.5,0.3,0.1), (0.4,0.5,0.3)]:
        r,g,b = rgb
        canvas.setFillColorRGB(r,g,b)
        canvas.rect(x+rdx, y+rdy, w, h, fill=1)
        canvas.setFillColor(black)
```

```
            canvas.drawCentredString(x+dx/2, y+texty, "r%s g%s b%s"%rgb)
            x = x+dx
        y = y + dy; x = 0
        for gray in (0.0, 0.25, 0.50, 0.75, 1.0):
            canvas.setFillGray(gray)
            canvas.rect(x+rdx, y+rdy, w, h, fill=1)
            canvas.setFillColor(black)
            canvas.drawCentredString(x+dx/2, y+texty, "gray: %s"%gray)
            x = x+dx
```

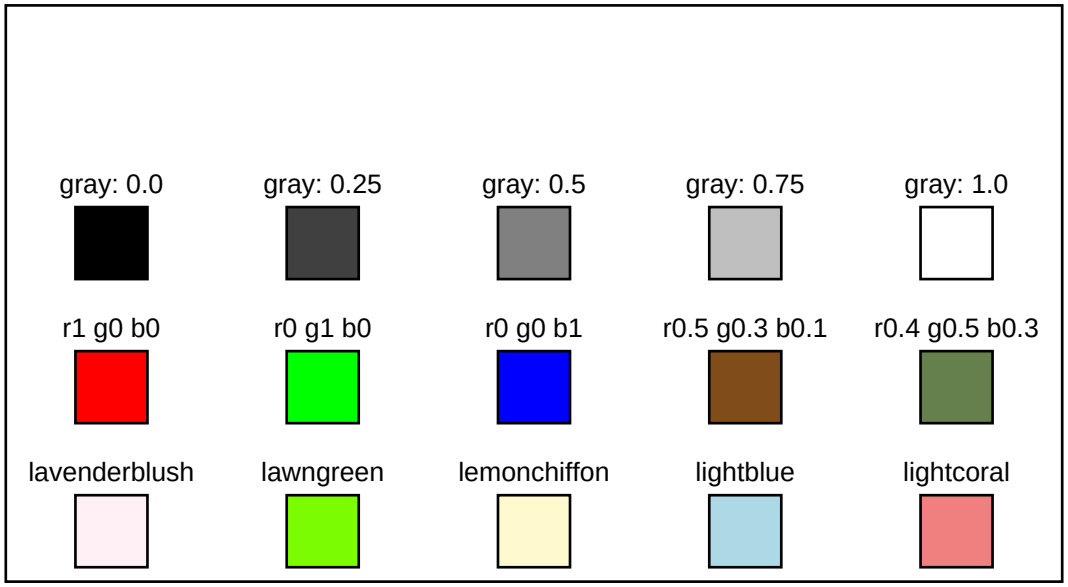


Figure 2-7: RGB Color Models

RGB Color Transparency

Objects may be painted over other objects to good effect in `pdfgen`. Generally There are two modes of handling objects that overlap in space, the default objects in the top layer will hide any part of other objects that falls underneath it. If you need transparency you got two choices:

1. If your document is intended to be printed in a professional way and you are working in CMYK color space then you can use overPrint. In overPrinting the colors physically mix in the printer and thus a new color is obtained. By default a knockout will be applied and only top object appears. Read the CMYK section if this is what you intend to use.

2. If your document is intended for screen output and you are using RGB colors then you can set an alpha value, where alpha is the opacity value of the color. The default alpha value is `1` (fully opaque) and you can use any real number value in the range 0-1.

Alpha transparency (`alpha`) is similar to overprint but works in RGB color space this example below demonstrates the alpha funtionality. Refer to our website http://www.reportlab.com/snippets/ and look for snippets of overPrint and alpha to see the code that generates the graph below.

```
def alpha(canvas):
    from reportlab.graphics.shapes import Rect
    from reportlab.lib.colors import Color, black, blue, red
    red50transparent = Color( 100, 0, 0, alpha=0.5)
    c = canvas
    c.setFillColor(black)
    c.setFont('Helvetica', 10)
    c.drawString(25,180, 'solid')
    c.setFillColor(blue)
    c.rect(25,25,100,100, fill=True, stroke=False)
    c.setFillColor(red)
    c.rect(100,75,100,100, fill=True, stroke=False)
    c.setFillColor(black)
    c.drawString(225,180, 'transparent')
```

```
c.setFillColor(blue)
c.rect(225,25,100,100, fill=True, stroke=False)
c.setFillColor(red50transparent)
c.rect(300,75,100,100, fill=True, stroke=False)
```

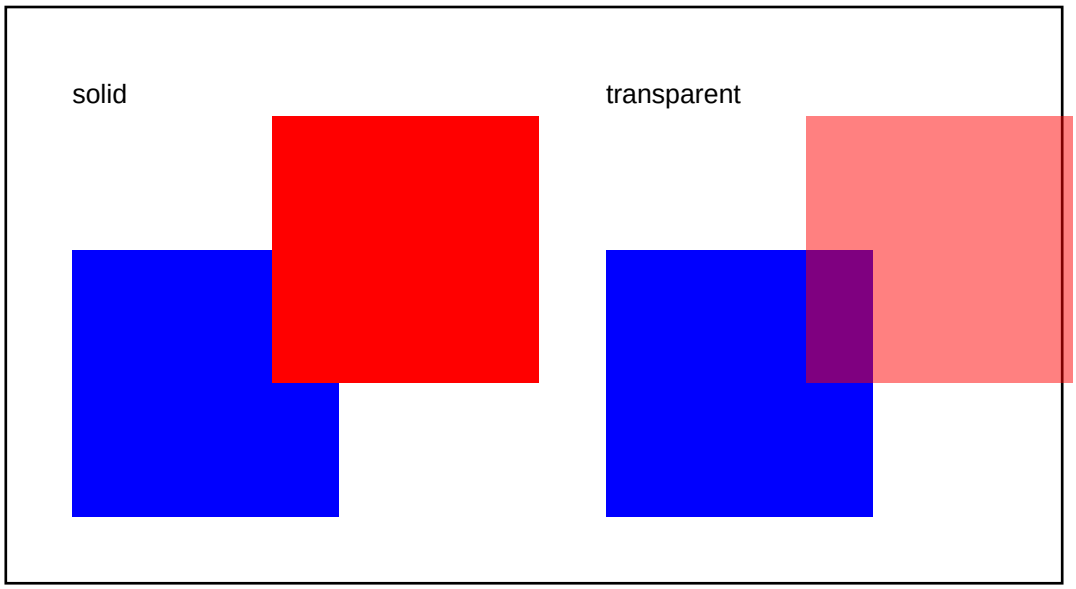


Figure 2-8: Alpha example

CMYK Colors

The `CMYK` or subtractive method follows the way a printer mixes three pigments (cyan, magenta, and yellow) to form colors. Because mixing chemicals is more difficult than combining light there is a fourth parameter for darkness. For example a chemical combination of the `CMY` pigments generally never makes a perfect black -- instead producing a muddy color -- so, to get black printers don not use the `CMY` pigments but use a direct black ink. Because `CMYK` maps more directly to the way printer hardware works it may be the case that colors specified in `CMYK` will provide better fidelity and better control when printed.

There are two ways of representing CMYK Color: each color can be represented either by a real value between 0 and 1, or integer value between 0 and 100. Depending on your preference you can either use CMYKColor (for real values) or PCMYKColor (for integer values). 0 means 'no ink', so printing on white papers gives you white. 1 (or 100 if you use PCMYKColor) means 'the maximum amount of ink'. e.g. CMYKColor(0,0,0,1) is black, CMYKColor(0,0,0,0) means 'no ink', and CMYKColor(0.5,0,0,0) means 50 percent cyan color.

```
def colorsCMYK(canvas):
    from reportlab.lib.colors import CMYKColor, PCMYKColor
    from reportlab.lib.units import inch
    # creates a black CMYK ; CMYKColor use real values
    black = CMYKColor(0,0,0,1)
    # creates a cyan CMYK ; PCMYKColor use integer values
    cyan = PCMYKColor(100,0,0,0)
    y = x = 0; dy=inch*3/4.0; dx=inch*5.5/5; w=h=dy/2; rdx=(dx-w)/2
    rdy=h/5.0; texty=h+2*rdy
    canvas.setFont("Helvetica",10)
    y = y + dy; x = 0
    for cmyk in [(1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1), (0,0,0,0)]:
        c,m,y1,k = cmyk
        canvas.setFillColorCMYK(c,m,y1,k)
        canvas.rect(x+rdx, y+rdy, w, h, fill=1)
        canvas.setFillColor(black)
        canvas.drawCentredString(x+dx/2, y+texty, "c%s m%s y%s k%s"%cmyk)
        x = x+dx
```

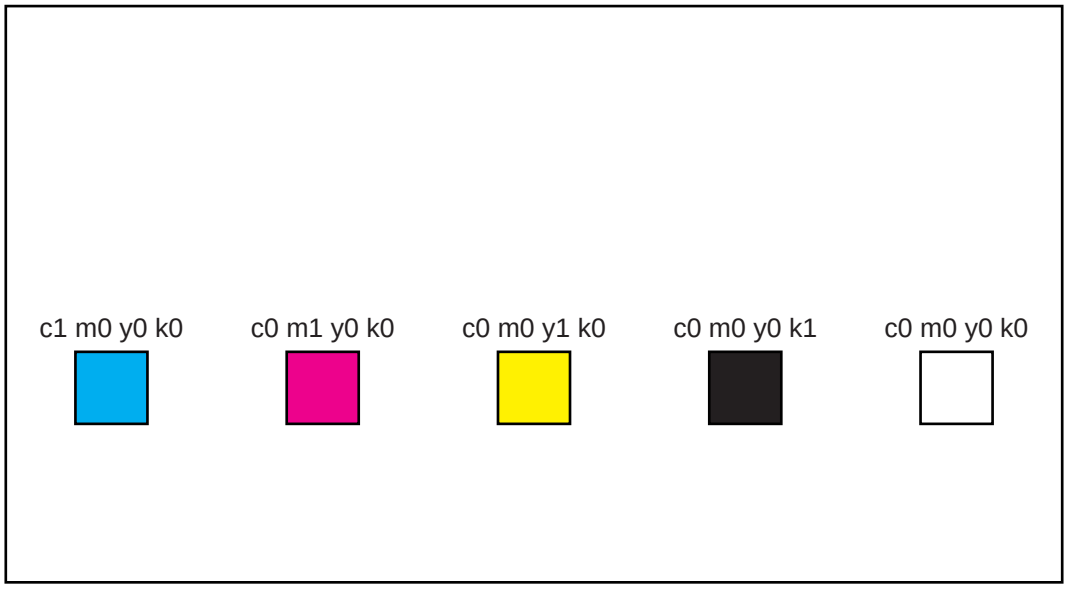


Figure 2-9: CMYK Color Models

2.9 Color space checking

The `enforceColorSpace` argument of the canvas is used to enforce the consistency of the colour model used in a document. It accepts these values: CMYK, RGB, SEP, SEP_BLACK, SEP_CMYK. 'SEP' refers to named color separations such as Pantone spot colors - these can be mixed with CMYK or RGB according to the parameter used. The default is 'MIXED' which allows you to use colors from any color space. An exception is raised if any colors used are not convertible to the specified model, e.g. rgb and cmyk (more information in test_pdfgen_general). This approach doesn't check external images included in document.

2.10 Color Overprinting

When two CMYK colored objects overlap in printing, then either the object 'on top' will knock out the color of the the one underneath it, or the colors of the two objects will mix in the overlapped area. This behaviour can be set using the property `overPrint`.

The `overPrint` function will cause ovelapping areas of color to mix. In the example below, the colors of the rectangles on the left should appear mixed where they overlap - If you can't see this effect then you may need to enable the 'overprint preview' option in your PDF viewing software. Some PDF viewers such as `evince` do not support overPrint; however Adobe Acrobat Reader does support it.

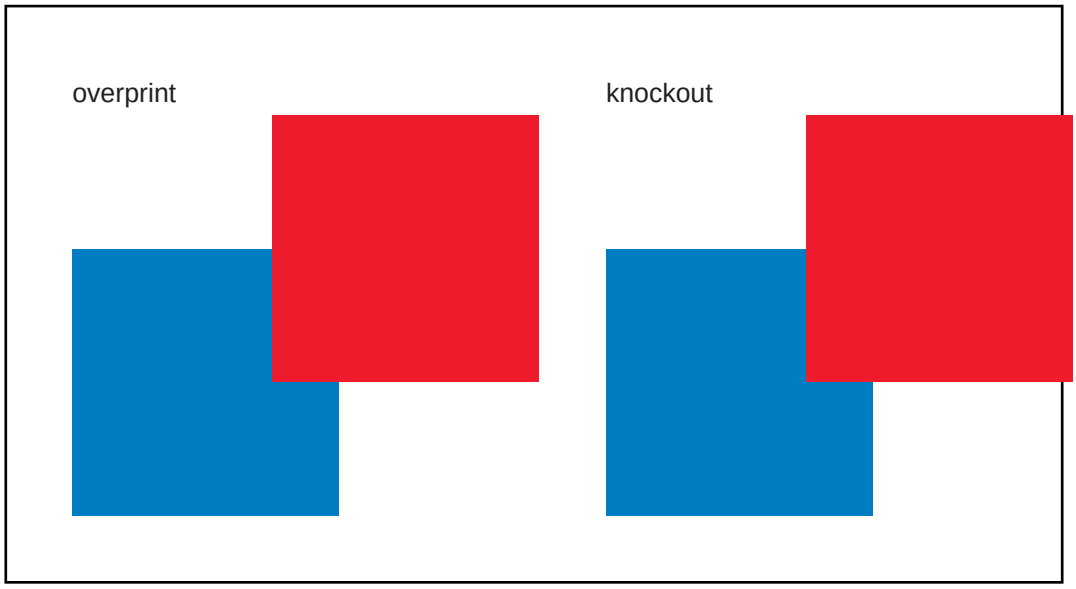


Figure 2-10: overPrint example

Other Object Order of Printing Examples

The word "SPUMONI" is painted in white over the colored rectangles, with the apparent effect of "removing" the color inside the body of the word.

```
def spumoni(canvas):
    from reportlab.lib.units import inch
    from reportlab.lib.colors import pink, green, brown, white
    x = 0; dx = 0.4*inch
    for i in range(4):
        for color in (pink, green, brown):
            canvas.setFillColor(color)
            canvas.rect(x,0,dx,3*inch,stroke=0,fill=1)
            x = x+dx
    canvas.setFillColor(white)
    canvas.setStrokeColor(white)
    canvas.setFont("Helvetica-Bold", 85)
    canvas.drawCentredString(2.75*inch, 1.3*inch, "SPUMONI")
```

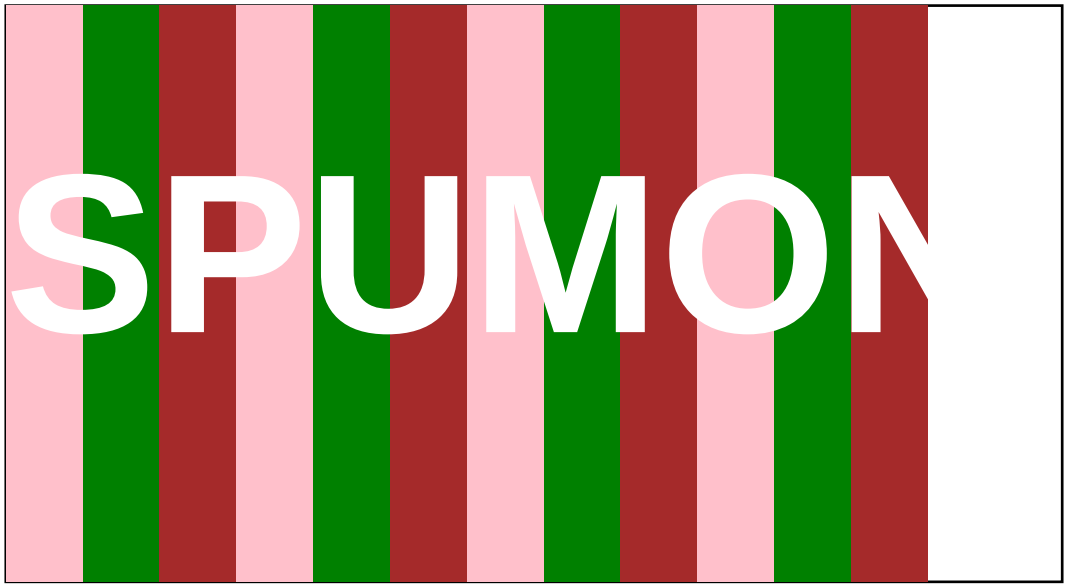


Figure 2-11: Painting over colors

The last letters of the word are not visible because the default `canvas` background is white and painting white letters over a white background leaves no visible effect.

This method of building up complex paintings in layers can be done in very many layers in `pdfgen` -- there are fewer physical limitations than there are when dealing with physical paints.

```
def spumoni2(canvas):
    from reportlab.lib.units import inch
    from reportlab.lib.colors import pink, green, brown, white, black
    # draw the previous drawing
    spumoni(canvas)
    # now put an ice cream cone on top of it:
    # first draw a triangle (ice cream cone)
    p = canvas.beginPath()
    xcenter = 2.75*inch
    radius = 0.45*inch
    p.moveTo(xcenter-radius, 1.5*inch)
    p.lineTo(xcenter+radius, 1.5*inch)
    p.lineTo(xcenter, 0)
    canvas.setFillColor(brown)
    canvas.setStrokeColor(black)
    canvas.drawPath(p, fill=1)
    # draw some circles (scoops)
    y = 1.5*inch
    for color in (pink, green, brown):
        canvas.setFillColor(color)
        canvas.circle(xcenter, y, radius, fill=1)
        y = y+radius
```

The `spumoni2` function layers an ice cream cone over the `spumoni` drawing. Note that different parts of the cone and scoops layer over eachother as well.

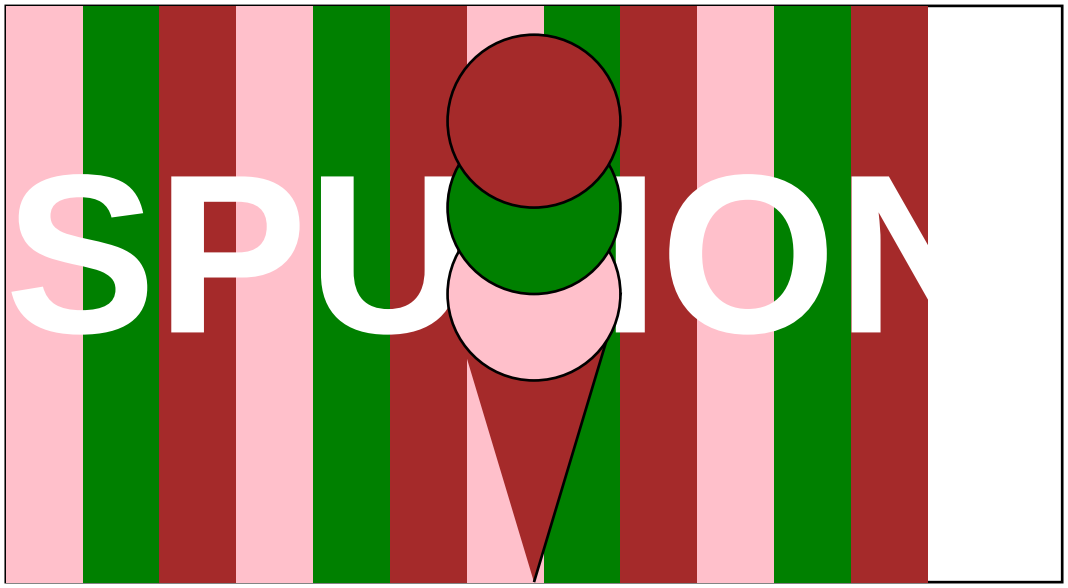


Figure 2-12: building up a drawing in layers

2.11 Standard fonts and text objects

Text may be drawn in many different colors, fonts, and sizes in `pdfgen`. The `textsize` function demonstrates how to change the color and font and size of text and how to place text on the page.

```
def textsize(canvas):
    from reportlab.lib.units import inch
    from reportlab.lib.colors import magenta, red
    canvas.setFont("Times-Roman", 20)
    canvas.setFillColor(red)
    canvas.drawCentredString(2.75*inch, 2.5*inch, "Font size examples")
    canvas.setFillColor(magenta)
```

```
        size = 7
        y = 2.3*inch
        x = 1.3*inch
        for line in lyrics:
            canvas.setFont("Helvetica", size)
            canvas.drawRightString(x,y,"%s points: " % size)
            canvas.drawString(x,y, line)
            y = y-size*1.2
            size = size+1.5
```

The `textsize` function generates the following page.

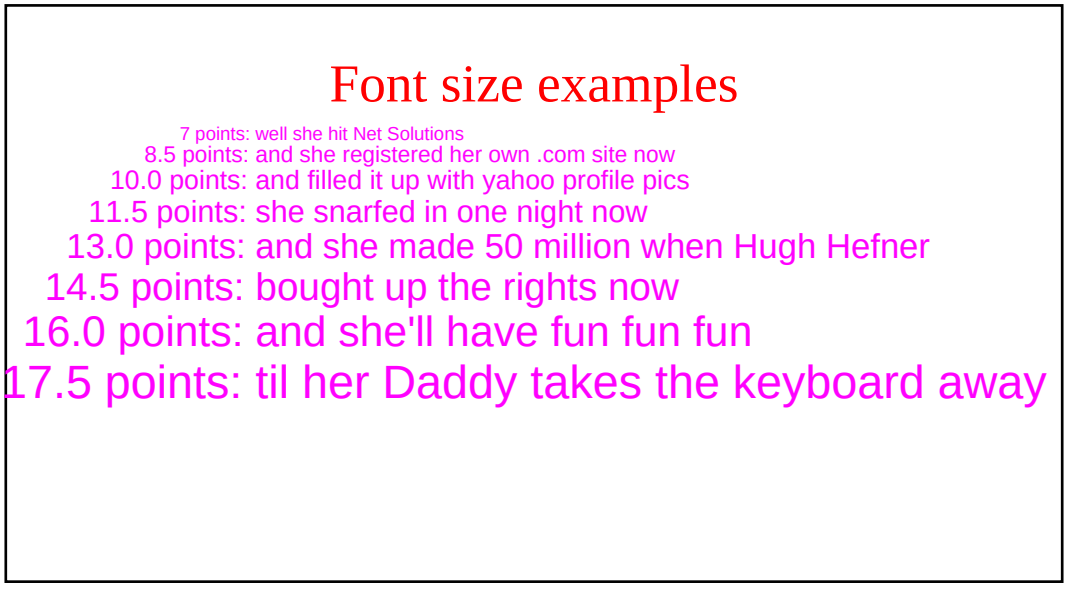


Figure 2-13: text in different fonts and sizes

A number of different fonts are always available in `pdfgen`.

```
    def fonts(canvas):
        from reportlab.lib.units import inch
        text = "Now is the time for all good men to..."
        x = 1.8*inch
        y = 2.7*inch
        for font in canvas.getAvailableFonts():
            canvas.setFont(font, 10)
            canvas.drawString(x,y,text)
            canvas.setFont("Helvetica", 10)
            canvas.drawRightString(x-10,y, font+":")
            y = y-13
```

The `fonts` function lists the fonts that are always available. These don't need to be stored in a PDF document, since they are guaranteed to be present in Acrobat Reader.

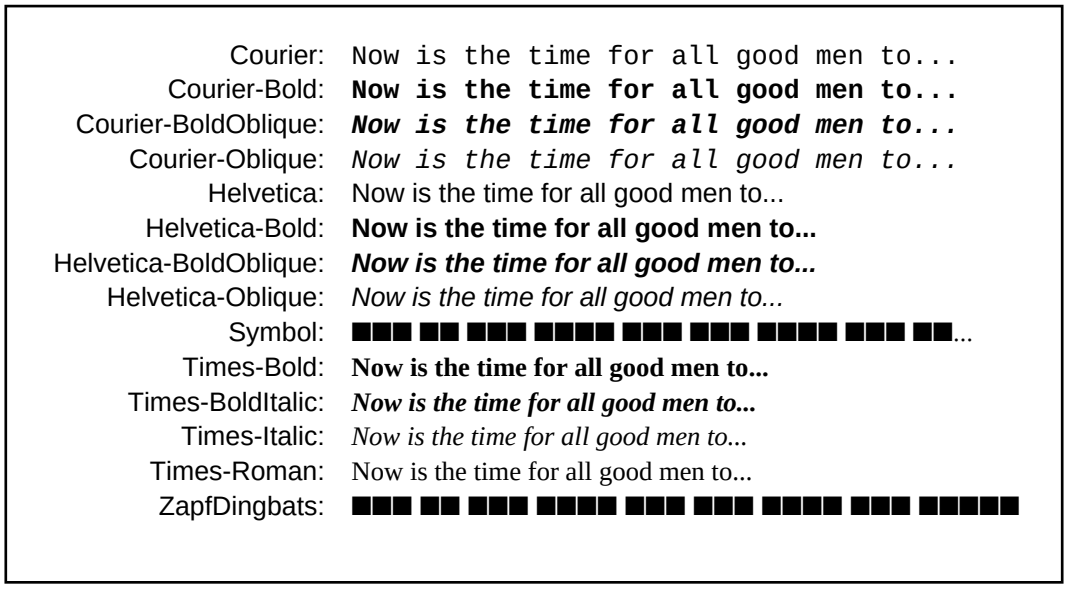


Figure 2-14: the 14 standard fonts

The Symbol and ZapfDingbats fonts cannot display properly because the required glyphs are not present in those fonts.

For information on how to use arbitrary fonts, see the next chapter.

2.12 Text object methods

For the dedicated presentation of text in a PDF document, use a text object. The text object interface provides detailed control of text layout parameters not available directly at the `canvas` level. In addition, it results in smaller PDF that will render faster than many separate calls to the `drawString` methods.

```
textobject.setTextOrigin(x,y)

textobject.setTextTransform(a,b,c,d,e,f)

textobject.moveCursor(dx, dy) # from start of current LINE

(x,y) = textobject.getCursor()

x = textobject.getX(); y = textobject.getY()

textobject.setFont(psfontname, size, leading = None)

textobject.textOut(text)

textobject.textLine(text='')

textobject.textLines(stuff, trim=1)
```

The text object methods shown above relate to basic text geometry.

A text object maintains a text cursor which moves about the page when text is drawn. For example the `setTextOrigin` places the cursor in a known position and the `textLine` and `textLines` methods move the text cursor down past the lines that have been missing.

```
def cursormoves1(canvas):
    from reportlab.lib.units import inch
    textobject = canvas.beginText()
    textobject.setTextOrigin(inch, 2.5*inch)
    textobject.setFont("Helvetica-Oblique", 14)
    for line in lyrics:
```

```
            textobject.textLine(line)
        textobject.setFillGray(0.4)
        textobject.textLines('''
        With many apologies to the Beach Boys
        and anyone else who finds this objectionable
        ''')
        canvas.drawText(textobject)
```

The `cursormoves` function relies on the automatic movement of the text cursor for placing text after the origin has been set.

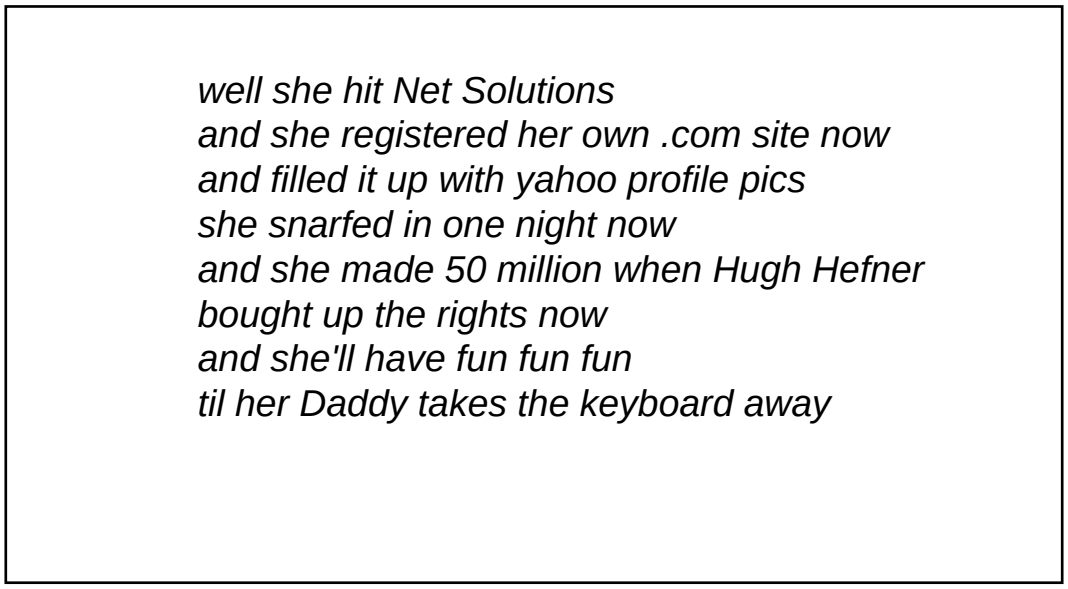


Figure 2-15: How the text cursor moves

It is also possible to control the movement of the cursor more explicitly by using the `moveCursor` method (which moves the cursor as an offset from the start of the current *line* NOT the current cursor, and which also has positive *y* offsets move *down* (in contrast to the normal geometry where positive *y* usually moves up.

```
def cursormoves2(canvas):
    from reportlab.lib.units import inch
    textobject = canvas.beginText()
    textobject.setTextOrigin(2, 2.5*inch)
    textobject.setFont("Helvetica-Oblique", 14)
    for line in lyrics:
        textobject.textOut(line)
        textobject.moveCursor(14,14) # POSITIVE Y moves down!!!
    textobject.setFillColorRGB(0.4,0,1)
    textobject.textLines('''
    With many apologies to the Beach Boys
    and anyone else who finds this objectionable
    ''')
    canvas.drawText(textobject)
```

Here the `textOut` does not move the down a line in contrast to the `textLine` function which does move down.

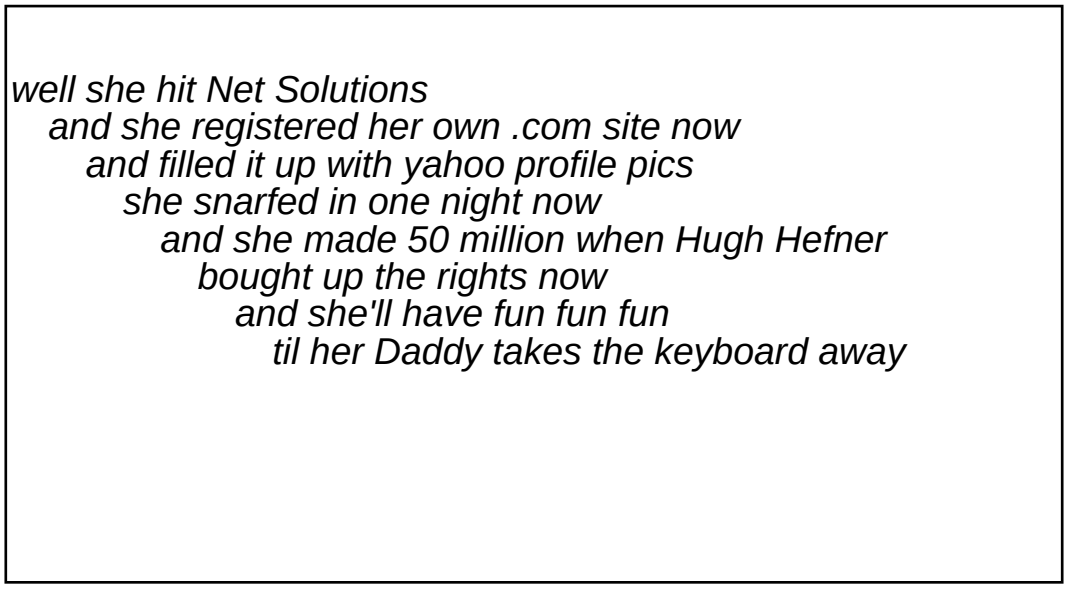


Figure 2-16: How the text cursor moves again

Character Spacing

```
textobject.setCharSpace(charSpace)
```

The `setCharSpace` method adjusts one of the parameters of text -- the inter-character spacing.

```
def charspace(canvas):
    from reportlab.lib.units import inch
    textobject = canvas.beginText()
    textobject.setTextOrigin(3, 2.5*inch)
    textobject.setFont("Helvetica-Oblique", 10)
    charspace = 0
    for line in lyrics:
        textobject.setCharSpace(charspace)
        textobject.textLine("%s: %s" %(charspace,line))
        charspace = charspace+0.5
    textobject.setFillGray(0.4)
    textobject.textLines('''
    With many apologies to the Beach Boys
    and anyone else who finds this objectionable
    ''')
    canvas.drawText(textobject)
```

The `charspace` function exercises various spacing settings. It produces the following page.

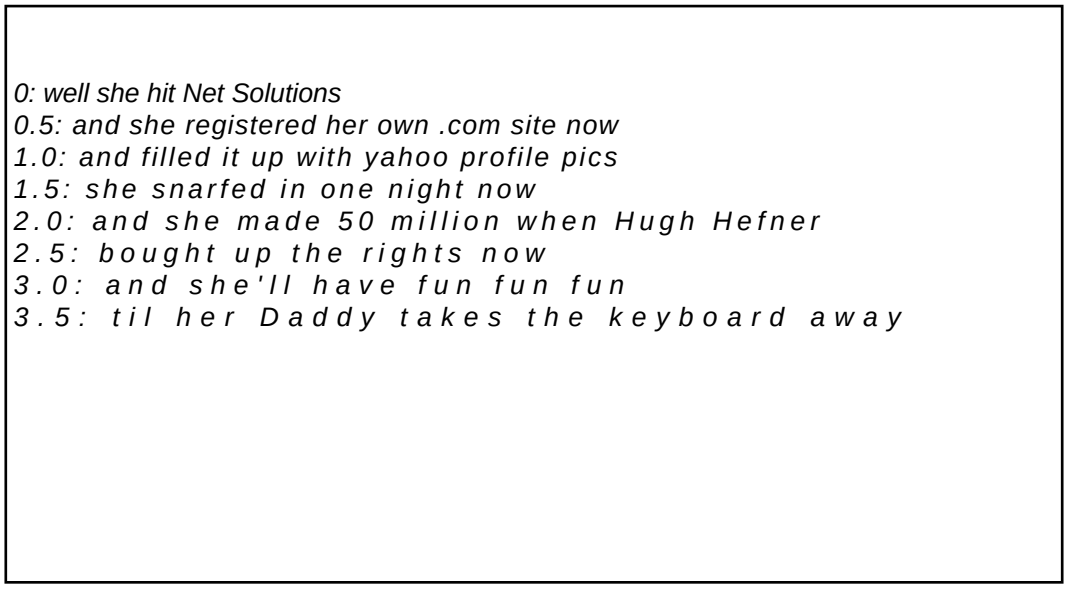


Figure 2-17: Adjusting inter-character spacing

Word Spacing

```
textobject.setWordSpace(wordSpace)
```

The `setWordSpace` method adjusts the space between words.

```
def wordspace(canvas):
    from reportlab.lib.units import inch
    textobject = canvas.beginText()
    textobject.setTextOrigin(3, 2.5*inch)
    textobject.setFont("Helvetica-Oblique", 12)
    wordspace = 0
    for line in lyrics:
        textobject.setWordSpace(wordspace)
        textobject.textLine("%s: %s" %(wordspace,line))
        wordspace = wordspace+2.5
    textobject.setFillColorCMYK(0.4,0,0.4,0.2)
    textobject.textLines('''
    With many apologies to the Beach Boys
    and anyone else who finds this objectionable
    ''')
    canvas.drawText(textobject)
```

The `wordspace` function shows what various word space settings look like below.

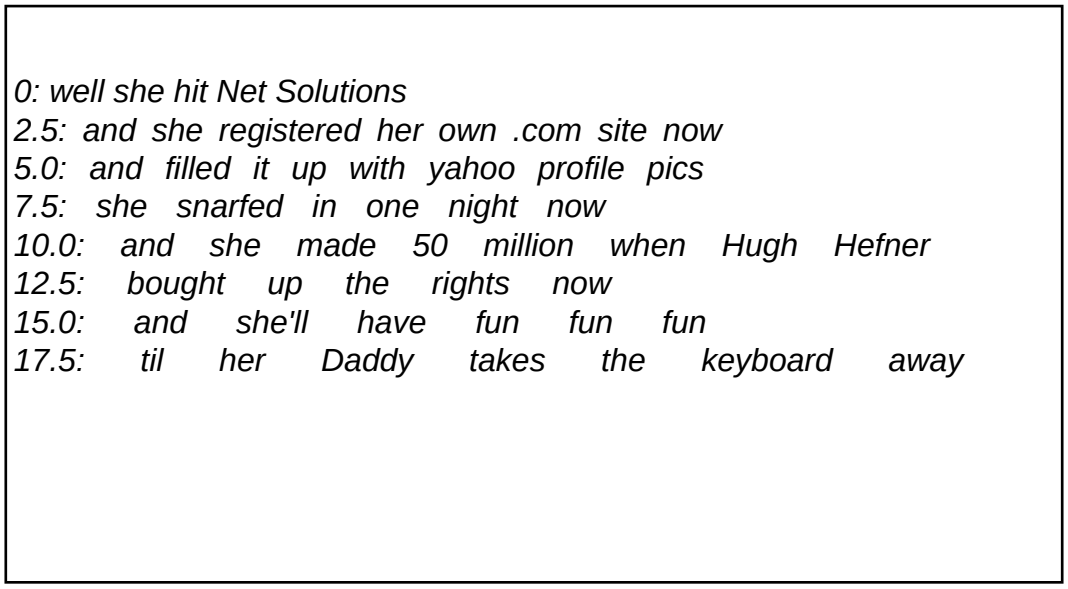


Figure 2-18: Adjusting word spacing

Horizontal Scaling

```
textobject.setHorizScale(horizScale)
```

Lines of text can be stretched or shrunken horizontally by the `setHorizScale` method.

```
def horizontalscale(canvas):
    from reportlab.lib.units import inch
    textobject = canvas.beginText()
    textobject.setTextOrigin(3, 2.5*inch)
    textobject.setFont("Helvetica-Oblique", 12)
    horizontalscale = 80 # 100 is default
    for line in lyrics:
        textobject.setHorizScale(horizontalscale)
        textobject.textLine("%s: %s" %(horizontalscale,line))
        horizontalscale = horizontalscale+10
    textobject.setFillColorCMYK(0.0,0.4,0.4,0.2)
    textobject.textLines('''
    With many apologies to the Beach Boys
    and anyone else who finds this objectionable
    ''')
    canvas.drawText(textobject)
```

The horizontal scaling parameter *`horizScale`* is given in percentages (with 100 as the default), so the 80 setting shown below looks skinny.

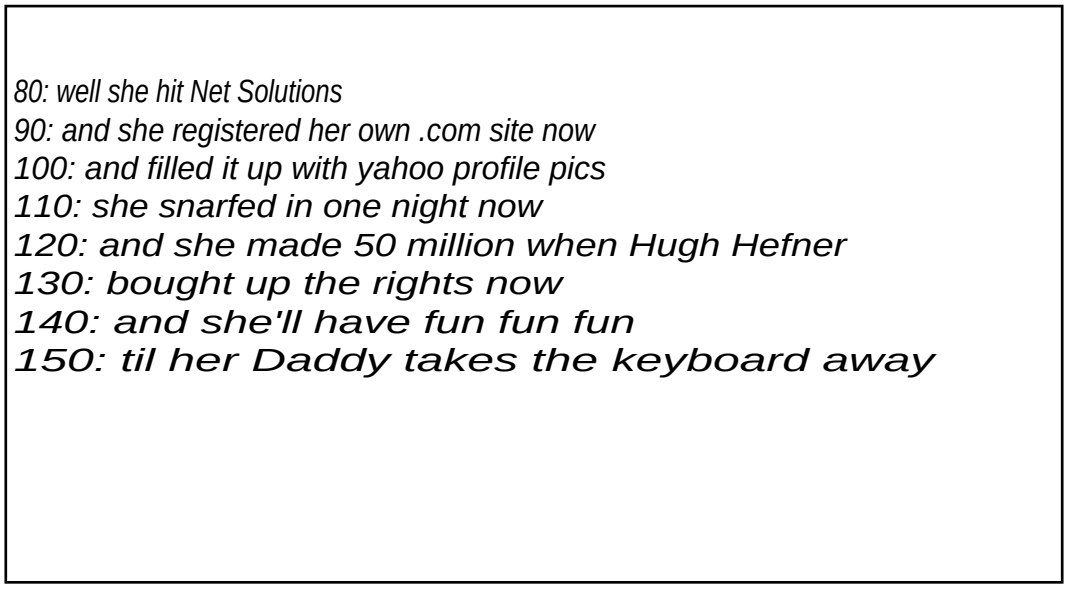


Figure 2-19: adjusting horizontal text scaling

Interline spacing (Leading)

```
textobject.setLeading(leading)
```

The vertical offset between the point at which one line starts and where the next starts is called the leading offset. The `setLeading` method adjusts the leading offset.

```
def leading(canvas):
    from reportlab.lib.units import inch
    textobject = canvas.beginText()
    textobject.setTextOrigin(3, 2.5*inch)
    textobject.setFont("Helvetica-Oblique", 14)
    leading = 8
    for line in lyrics:
        textobject.setLeading(leading)
        textobject.textLine("%s: %s" %(leading,line))
        leading = leading+2.5
    textobject.setFillColorCMYK(0.8,0,0,0.3)
    textobject.textLines('''
    With many apologies to the Beach Boys
    and anyone else who finds this objectionable
    ''')
    canvas.drawText(textobject)
```

As shown below if the leading offset is set too small characters of one line my write over the bottom parts of characters in the previous line.

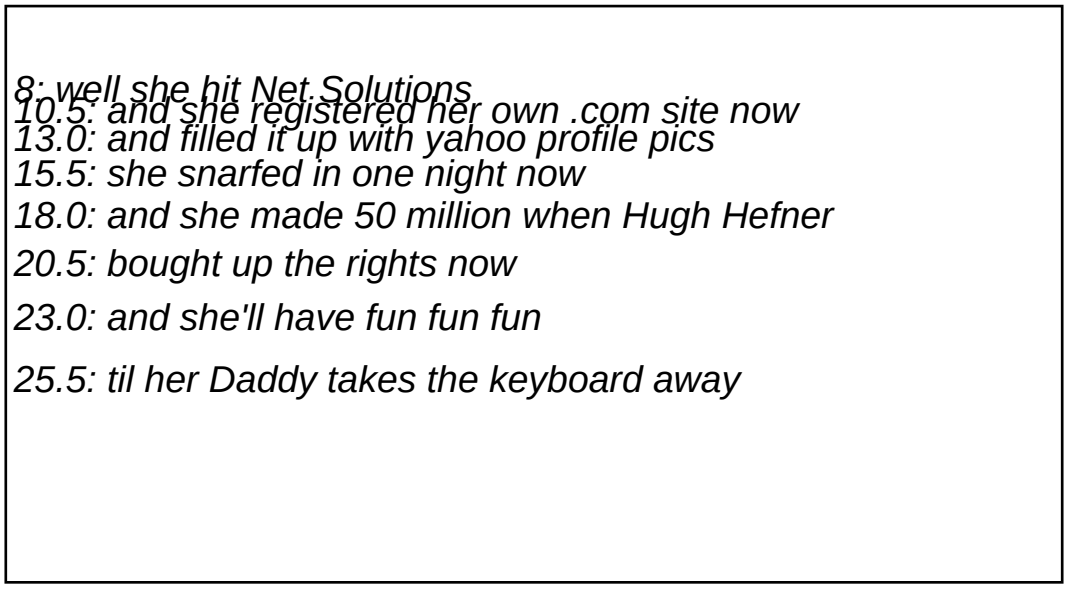


Figure 2-20: adjusting the leading

Other text object methods

```
textobject.setTextRenderMode(mode)
```

The `setTextRenderMode` method allows text to be used as a forground for clipping background drawings, for example.

```
textobject.setRise(rise)
```

The `setRise` method raises or lowers text on the line (for creating superscripts or subscripts, for example).

```
textobject.setFillColor(aColor);
textobject.setStrokeColor(self, aColor)
# and similar
```

These color change operations change the color of the text and are otherwise similar to the color methods for the `canvas` object.

2.13 Paths and Lines

Just as textobjects are designed for the dedicated presentation of text, path objects are designed for the dedicated construction of graphical figures. When path objects are drawn onto a `canvas` they are drawn as one figure (like a rectangle) and the mode of drawing for the entire figure can be adjusted: the lines of the figure can be drawn (stroked) or not; the interior of the figure can be filled or not; and so forth.

For example the `star` function uses a path object to draw a star

```
def star(canvas, title="Title Here", aka="Comment here.",
         xcenter=None, ycenter=None, nvertices=5):
    from math import pi
    from reportlab.lib.units import inch
    radius=inch/3.0
    if xcenter is None: xcenter=2.75*inch
    if ycenter is None: ycenter=1.5*inch
    canvas.drawCentredString(xcenter, ycenter+1.3*radius, title)
    canvas.drawCentredString(xcenter, ycenter-1.4*radius, aka)
    p = canvas.beginPath()
    p.moveTo(xcenter,ycenter+radius)
    from math import pi, cos, sin
    angle = (2*pi)*2/5.0
    startangle = pi/2.0
```

```
        for vertex in range(nvertices-1):
            nextangle = angle*(vertex+1)+startangle
            x = xcenter + radius*cos(nextangle)
            y = ycenter + radius*sin(nextangle)
            p.lineTo(x,y)
        if nvertices==5:
           p.close()
        canvas.drawPath(p)
```

The `star` function has been designed to be useful in illustrating various line style parameters supported by `pdfgen`.

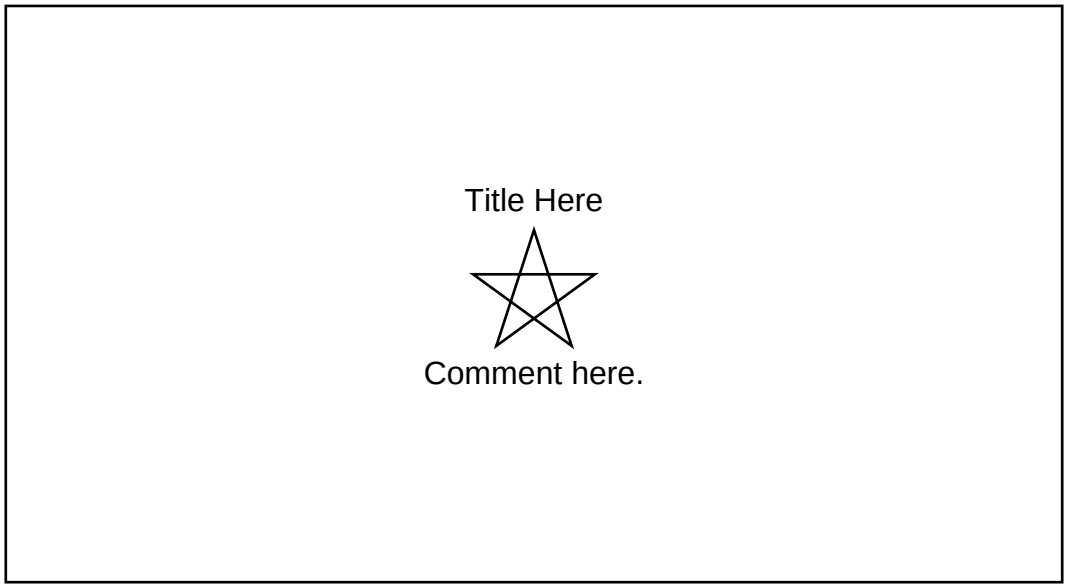


Figure 2-21: line style parameters

Line join settings

The `setLineJoin` method can adjust whether line segments meet in a point a square or a rounded vertex.

```
def joins(canvas):
    from reportlab.lib.units import inch
    # make lines big
    canvas.setLineWidth(5)
    star(canvas, "Default: mitered join", "0: pointed", xcenter = 1*inch)
    canvas.setLineJoin(1)
    star(canvas, "Round join", "1: rounded")
    canvas.setLineJoin(2)
    star(canvas, "Bevelled join", "2: square", xcenter=4.5*inch)
```

The line join setting is only really of interest for thick lines because it cannot be seen clearly for thin lines.

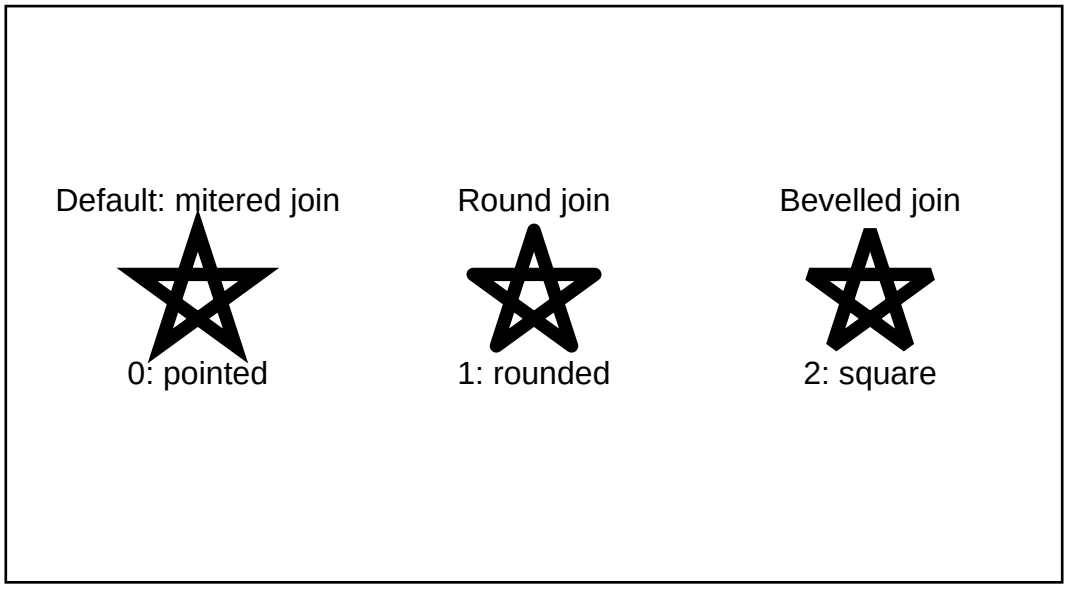


Figure 2-22: different line join styles

Line cap settings

The line cap setting, adjusted using the `setLineCap` method, determines whether a terminating line ends in a square exactly at the vertex, a square over the vertex or a half circle over the vertex.

```
def caps(canvas):
    from reportlab.lib.units import inch
    # make lines big
    canvas.setLineWidth(5)
    star(canvas, "Default", "no projection",xcenter = 1*inch,
         nvertices=4)
    canvas.setLineCap(1)
    star(canvas, "Round cap", "1: ends in half circle", nvertices=4)
    canvas.setLineCap(2)
    star(canvas, "Square cap", "2: projects out half a width", xcenter=4.5*inch,
       nvertices=4)
```

The line cap setting, like the line join setting, is only clearly visible when the lines are thick.

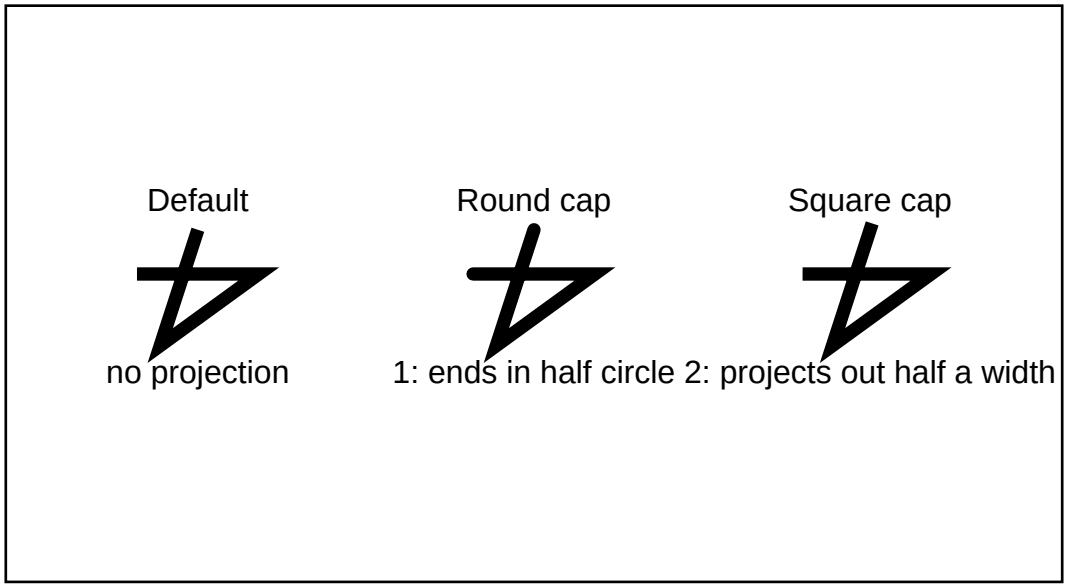


Figure 2-23: line cap settings

Dashes and broken lines

The `setDash` method allows lines to be broken into dots or dashes.

```
def dashes(canvas):
    from reportlab.lib.units import inch
    # make lines big
    canvas.setDash(6,3)
    star(canvas, "Simple dashes", "6 points on, 3 off", xcenter = 1*inch)
    canvas.setDash(1,2)
    star(canvas, "Dots", "One on, two off")
    canvas.setDash([1,1,3,3,1,4,4,1], 0)
    star(canvas, "Complex Pattern", "[1,1,3,3,1,4,4,1]", xcenter=4.5*inch)
```

The patterns for the dashes or dots can be in a simple on/off repeating pattern or they can be specified in a complex repeating pattern.

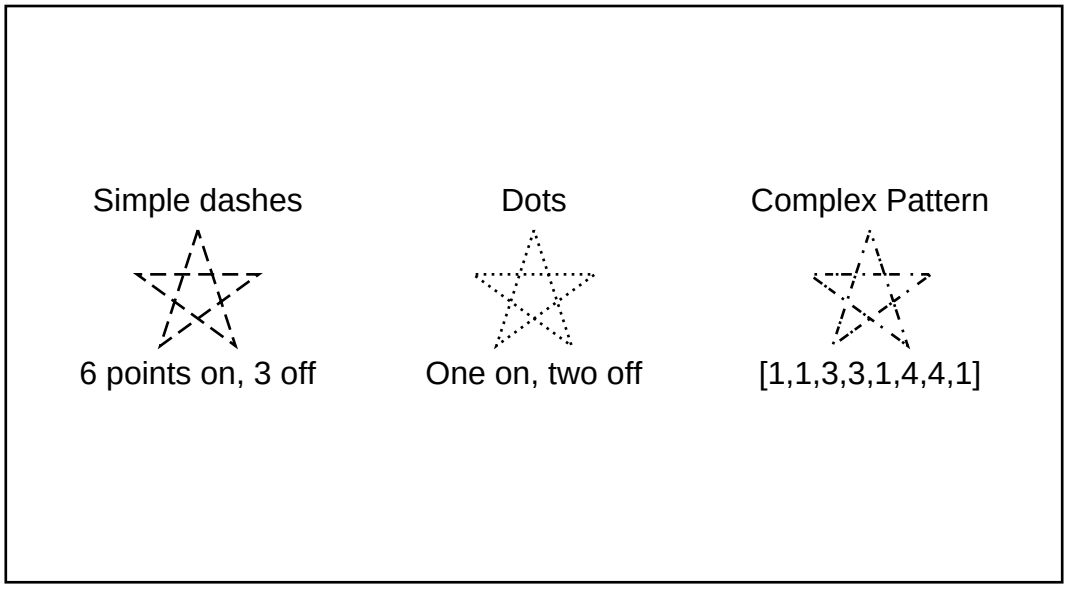


Figure 2-24: some dash patterns

Creating complex figures with path objects

Combinations of lines, curves, arcs and other figures can be combined into a single figure using path objects. For example the function shown below constructs two path objects using lines and curves. This function will be used later on as part of a pencil icon construction.

```
def penciltip(canvas, debug=1):
    from reportlab.lib.colors import tan, black, green
    from reportlab.lib.units import inch
    u = inch/10.0
    canvas.setLineWidth(4)
    if debug:
        canvas.scale(2.8,2.8) # make it big
        canvas.setLineWidth(1) # small lines
    canvas.setStrokeColor(black)
    canvas.setFillColor(tan)
    p = canvas.beginPath()
    p.moveTo(10*u,0)
    p.lineTo(0,5*u)
    p.lineTo(10*u,10*u)
    p.curveTo(11.5*u,10*u, 11.5*u,7.5*u, 10*u,7.5*u)
    p.curveTo(12*u,7.5*u, 11*u,2.5*u, 9.7*u,2.5*u)
    p.curveTo(10.5*u,2.5*u, 11*u,0, 10*u,0)
    canvas.drawPath(p, stroke=1, fill=1)
    canvas.setFillColor(black)
    p = canvas.beginPath()
    p.moveTo(0,5*u)
```

```
    p.lineTo(4*u,3*u)
    p.lineTo(5*u,4.5*u)
    p.lineTo(3*u,6.5*u)
    canvas.drawPath(p, stroke=1, fill=1)
    if debug:
        canvas.setStrokeColor(green) # put in a frame of reference
        canvas.grid([0,5*u,10*u,15*u], [0,5*u,10*u])
```

Note that the interior of the pencil tip is filled as one object even though it is constructed from several lines and curves. The pencil lead is then drawn over it using a new path object.

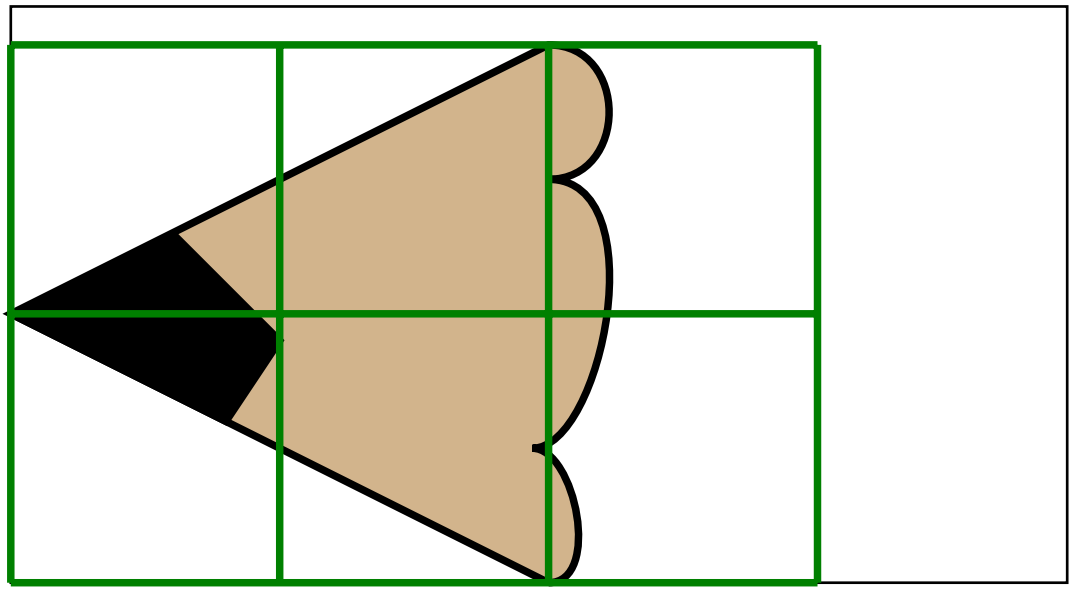


Figure 2-25: a pencil tip

2.14 Rectangles, circles, ellipses

The `pdfgen` module supports a number of generally useful shapes such as rectangles, rounded rectangles, ellipses, and circles. Each of these figures can be used in path objects or can be drawn directly on a `canvas`. For example the `pencil` function below draws a pencil icon using rectangles and rounded rectangles with various fill colors and a few other annotations.

```
def pencil(canvas, text="No.2"):
    from reportlab.lib.colors import yellow, red, black,white
    from reportlab.lib.units import inch
    u = inch/10.0
    canvas.setStrokeColor(black)
    canvas.setLineWidth(4)
    # draw erasor
    canvas.setFillColor(red)
    canvas.circle(30*u, 5*u, 5*u, stroke=1, fill=1)
    # draw all else but the tip (mainly rectangles with different fills)
    canvas.setFillColor(yellow)
    canvas.rect(10*u,0,20*u,10*u, stroke=1, fill=1)
    canvas.setFillColor(black)
    canvas.rect(23*u,0,8*u,10*u,fill=1)
    canvas.roundRect(14*u, 3.5*u, 8*u, 3*u, 1.5*u, stroke=1, fill=1)
    canvas.setFillColor(white)
    canvas.rect(25*u,u,1.2*u,8*u, fill=1,stroke=0)
    canvas.rect(27.5*u,u,1.2*u,8*u, fill=1, stroke=0)
    canvas.setFont("Times-Roman", 3*u)
    canvas.drawCentredString(18*u, 4*u, text)
    # now draw the tip
    penciltip(canvas,debug=0)
    # draw broken lines across the body.
    canvas.setDash([10,5,16,10],0)
    canvas.line(11*u,2.5*u,22*u,2.5*u)
    canvas.line(22*u,7.5*u,12*u,7.5*u)
```

Note that this function is used to create the "margin pencil" to the left. Also note that the order in which the elements are drawn are important because, for example, the white rectangles "erase" parts of a black rectangle and the "tip" paints over part of the yellow rectangle.

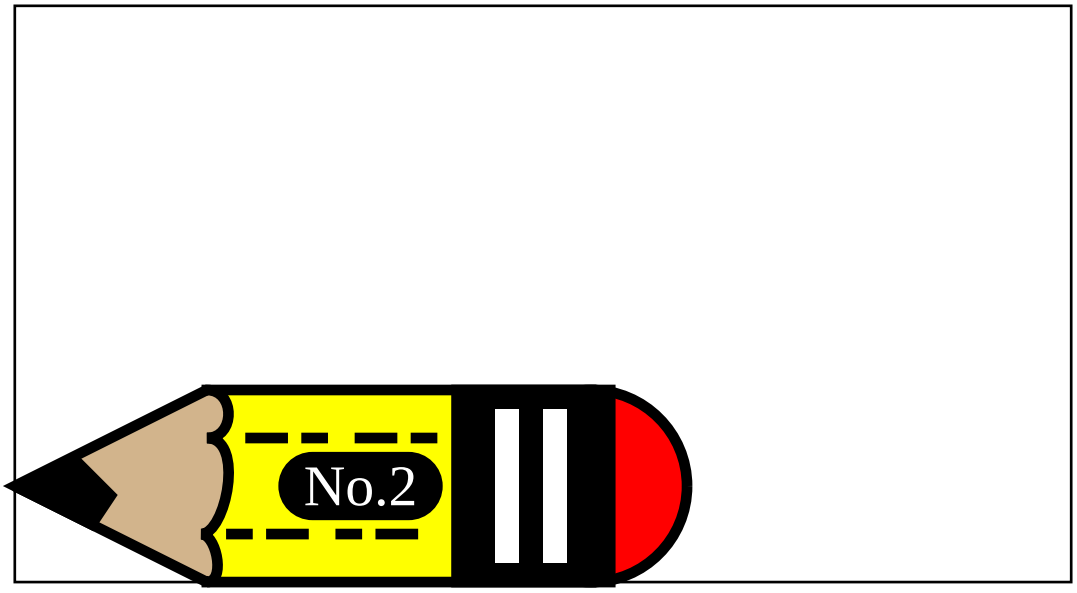


Figure 2-26: a whole pencil

2.15 Bezier curves

Programs that wish to construct figures with curving borders generally use Bezier curves to form the borders.

```
def bezier(canvas):
    from reportlab.lib.colors import yellow, green, red, black
    from reportlab.lib.units import inch
    i = inch
    d = i/4
    # define the bezier curve control points
    x1,y1, x2,y2, x3,y3, x4,y4 = d,1.5*i, 1.5*i,d, 3*i,d, 5.5*i-d,3*i-d
    # draw a figure enclosing the control points
    canvas.setFillColor(yellow)
    p = canvas.beginPath()
    p.moveTo(x1,y1)
    for (x,y) in [(x2,y2), (x3,y3), (x4,y4)]:
        p.lineTo(x,y)
    canvas.drawPath(p, fill=1, stroke=0)
    # draw the tangent lines
    canvas.setLineWidth(inch*0.1)
    canvas.setStrokeColor(green)
    canvas.line(x1,y1,x2,y2)
    canvas.setStrokeColor(red)
    canvas.line(x3,y3,x4,y4)
    # finally draw the curve
    canvas.setStrokeColor(black)
    canvas.bezier(x1,y1, x2,y2, x3,y3, x4,y4)
```

A Bezier curve is specified by four control points `(x1,y1)`, `(x2,y2)`, `(x3,y3)`, `(x4,y4)`. The curve starts at `(x1,y1)` and ends at `(x4,y4)` and the line segment from `(x1,y1)` to `(x2,y2)` and the line segment from `(x3,y3)` to `(x4,y4)` both form tangents to the curve. Furthermore the curve is entirely contained in the convex figure with vertices at the control points.

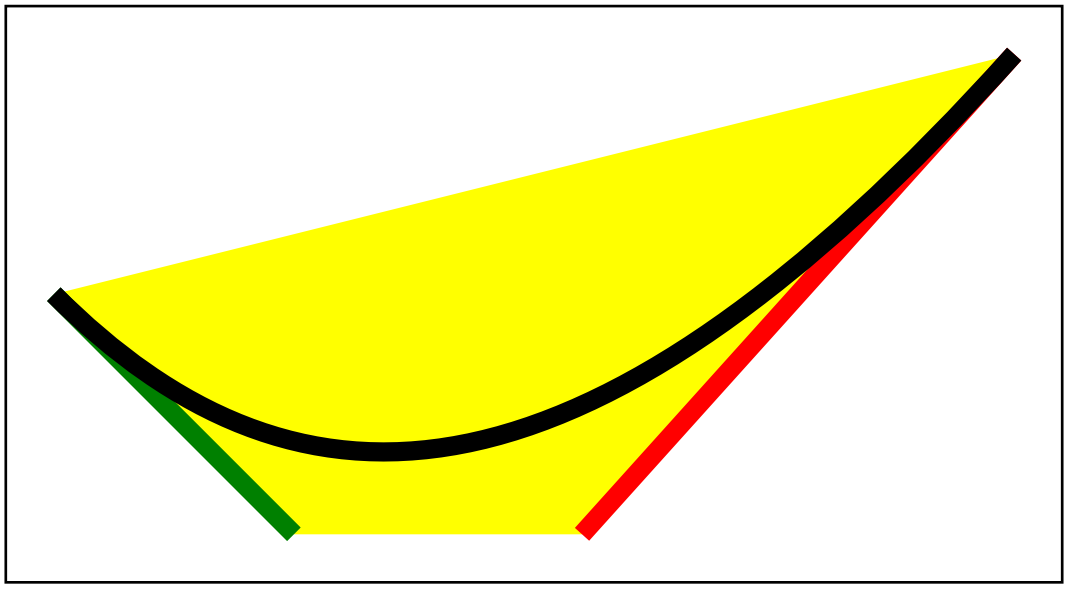


Figure 2-27: basic bezier curves

The drawing above (the output of `testbezier`) shows a bezier curves, the tangent lines defined by the control points and the convex figure with vertices at the control points.

Smoothly joining bezier curve sequences

It is often useful to join several bezier curves to form a single smooth curve. To construct a larger smooth curve from several bezier curves make sure that the tangent lines to adjacent bezier curves that join at a control point lie on the same line.

```
def bezier2(canvas):
    from reportlab.lib.colors import yellow, green, red, black
    from reportlab.lib.units import inch
    # make a sequence of control points
    xd,yd = 5.5*inch/2, 3*inch/2
    xc,yc = xd,yd
    dxdy = [(0,0.33), (0.33,0.33), (0.75,1), (0.875,0.875),
            (0.875,0.875), (1,0.75), (0.33,0.33), (0.33,0)]
    pointlist = []
    for xoffset in (1,-1):
        yoffset = xoffset
        for (dx,dy) in dxdy:
            px = xc + xd*xoffset*dx
            py = yc + yd*yoffset*dy
            pointlist.append((px,py))
        yoffset = -xoffset
        for (dy,dx) in dxdy:
            px = xc + xd*xoffset*dx
            py = yc + yd*yoffset*dy
            pointlist.append((px,py))
    # draw tangent lines and curves
    canvas.setLineWidth(inch*0.1)
    while pointlist:
        [(x1,y1),(x2,y2),(x3,y3),(x4,y4)] = pointlist[:4]
        del pointlist[:4]
        canvas.setLineWidth(inch*0.1)
        canvas.setStrokeColor(green)
        canvas.line(x1,y1,x2,y2)
        canvas.setStrokeColor(red)
        canvas.line(x3,y3,x4,y4)
        # finally draw the curve
        canvas.setStrokeColor(black)
        canvas.bezier(x1,y1, x2,y2, x3,y3, x4,y4)
```

The figure created by `testbezier2` describes a smooth complex curve because adjacent tangent lines "line up" as illustrated below.

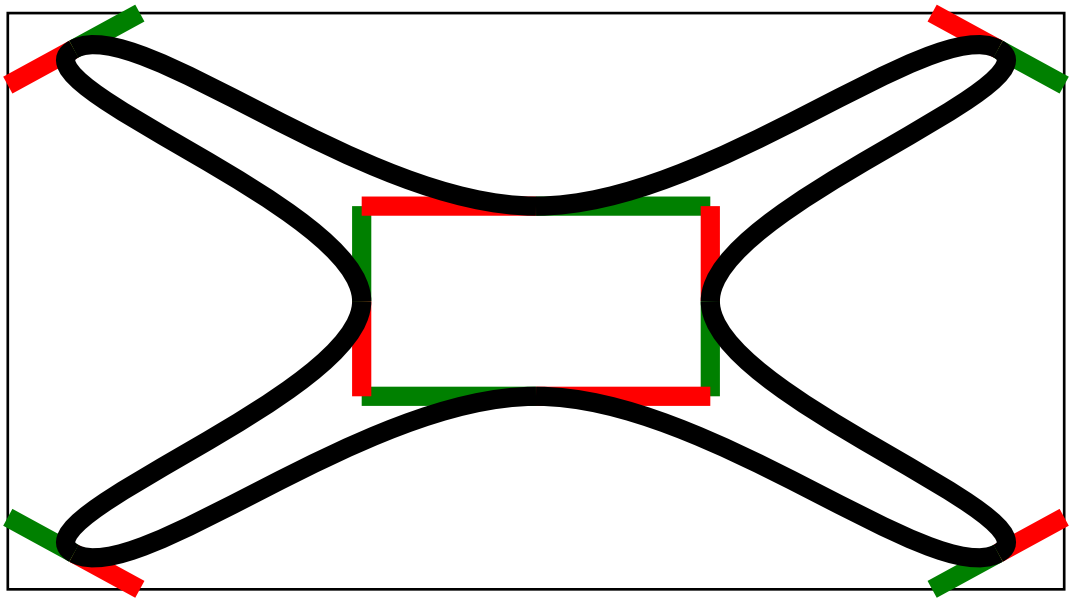


Figure 2-28: bezier curves

2.16 Path object methods

Path objects build complex graphical figures by setting the "pen" or "brush" at a start point on the canvas and drawing lines or curves to additional points on the canvas. Most operations apply paint on the canvas starting at the end point of the last operation and leave the brush at a new end point.

```
pathobject.moveTo(x,y)
```

The `moveTo` method lifts the brush (ending any current sequence of lines or curves if there is one) and replaces the brush at the new *`(x,y)`* location on the canvas to start a new path sequence.

```
pathobject.lineTo(x,y)
```

The `lineTo` method paints straight line segment from the current brush location to the new *`(x,y)`* location.

```
pathobject.curveTo(x1, y1, x2, y2, x3, y3)
```

The `curveTo` method starts painting a Bezier curve beginning at the current brush location, using *`(x1,y1)`*, *`(x2,y2)`*, and *`(x3,y3)`* as the other three control points, leaving the brush on *`(x3,y3)`*.

```
pathobject.arc(x1,y1, x2,y2, startAng=0, extent=90)

pathobject.arcTo(x1,y1, x2,y2, startAng=0, extent=90)
```

The `arc` and `arcTo` methods paint partial ellipses. The `arc` method first "lifts the brush" and starts a new shape sequence. The `arcTo` method joins the start of the partial ellipse to the current shape sequence by line segment before drawing the partial ellipse. The points *`(x1,y1)`* and *`(x2,y2)`* define opposite corner points of a rectangle enclosing the ellipse. The `startAng` is an angle (in degrees) specifying where to begin the partial ellipse where the 0 angle is the midpoint of the right border of the enclosing rectangle (when *`(x1,y1)`* is the lower left corner and *`(x2,y2)`* is the upper right corner). The `extent` is the angle in degrees to traverse on the ellipse.

```
def arcs(canvas):
    from reportlab.lib.units import inch
    canvas.setLineWidth(4)
    canvas.setStrokeColorRGB(0.8, 1, 0.6)
    # draw rectangles enclosing the arcs
    canvas.rect(inch, inch, 1.5*inch, inch)
    canvas.rect(3*inch, inch, inch, 1.5*inch)
```

```
    canvas.setStrokeColorRGB(0, 0.2, 0.4)
    canvas.setFillColorRGB(1, 0.6, 0.8)
    p = canvas.beginPath()
    p.moveTo(0.2*inch, 0.2*inch)
    p.arcTo(inch, inch, 2.5*inch,2*inch, startAng=-30, extent=135)
    p.arc(3*inch, inch, 4*inch, 2.5*inch, startAng=-45, extent=270)
    canvas.drawPath(p, fill=1, stroke=1)
```

The `arcs` function above exercises the two partial ellipse methods. It produces the following drawing.

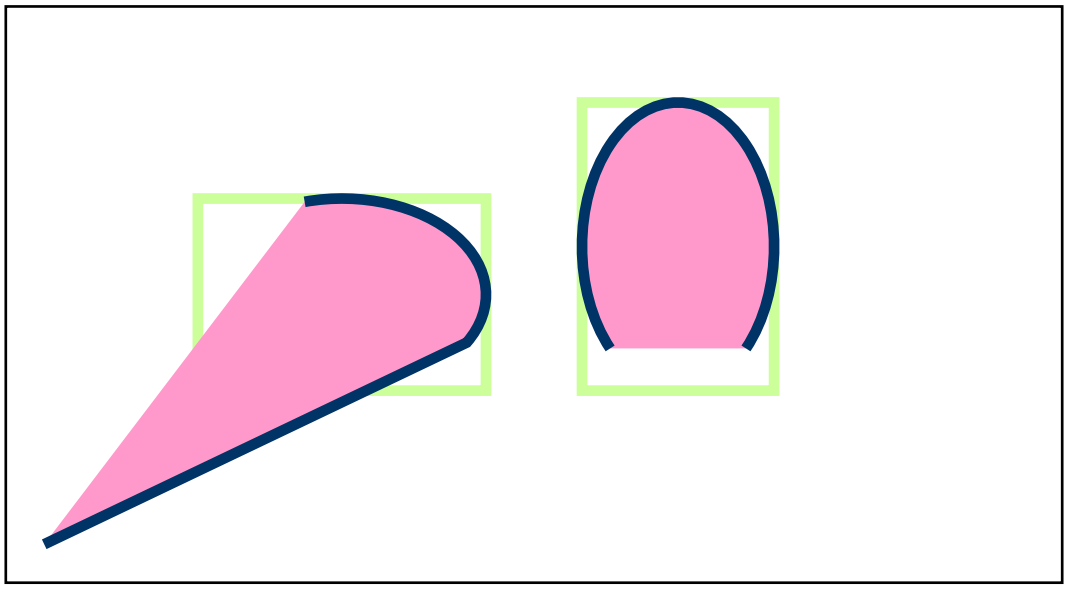


Figure 2-29: arcs in path objects

```
pathobject.rect(x, y, width, height)
```

The `rect` method draws a rectangle with lower left corner at *`(x,y)`* of the specified *`width`* and *`height`*.

```
pathobject.ellipse(x, y, width, height)
```

The `ellipse` method draws an ellipse enclosed in the rectange with lower left corner at *`(x,y)`* of the specified *`width`* and *`height`*.

```
pathobject.circle(x_cen, y_cen, r)
```

The `circle` method draws a circle centered at *`(x_cen, y_cen)`* with radius *`r`*.

```
def variousshapes(canvas):
    from reportlab.lib.units import inch
    inch = int(inch)
    canvas.setStrokeGray(0.5)
    canvas.grid(range(0,int(11*inch/2),int(inch/2)), range(0,int(7*inch/2),int(inch/2)))
    canvas.setLineWidth(4)
    canvas.setStrokeColorRGB(0, 0.2, 0.7)
    canvas.setFillColorRGB(1, 0.6, 0.8)
    p = canvas.beginPath()
    p.rect(0.5*inch, 0.5*inch, 0.5*inch, 2*inch)
    p.circle(2.75*inch, 1.5*inch, 0.3*inch)
    p.ellipse(3.5*inch, 0.5*inch, 1.2*inch, 2*inch)
    canvas.drawPath(p, fill=1, stroke=1)
```

The `variousshapes` function above shows a rectangle, circle and ellipse placed in a frame of reference grid.

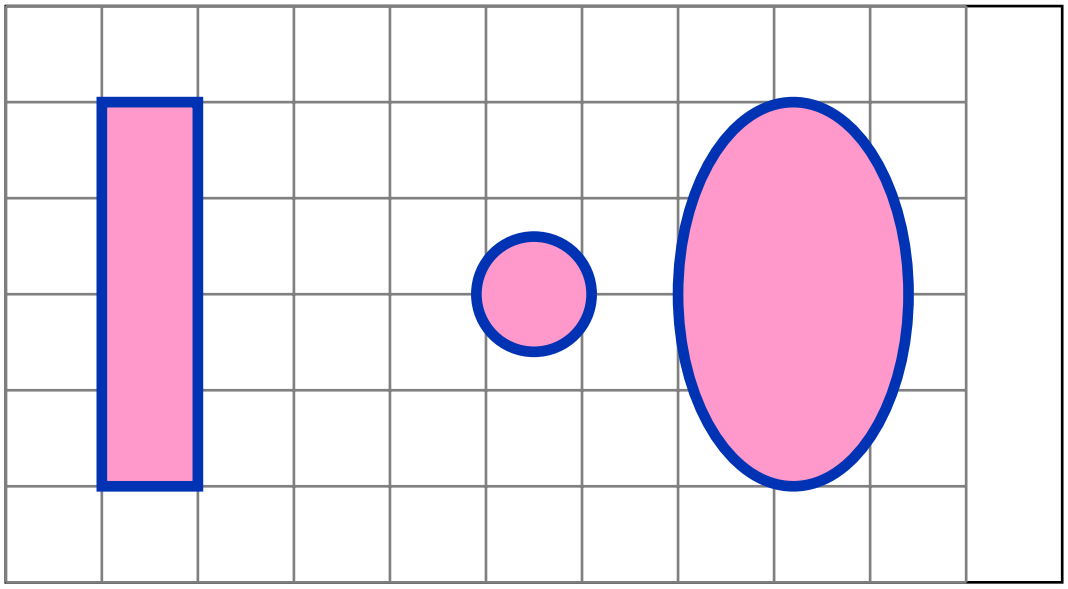


Figure 2-30: rectangles, circles, ellipses in path objects

```
pathobject.close()
```

The `close` method closes the current graphical figure by painting a line segment from the last point of the figure to the starting point of the figure (the the most recent point where the brush was placed on the paper by `moveTo` or `arc` or other placement operations).

```
def closingfigures(canvas):
    from reportlab.lib.units import inch
    h = inch/3.0; k = inch/2.0
    canvas.setStrokeColorRGB(0.2,0.3,0.5)
    canvas.setFillColorRGB(0.8,0.6,0.2)
    canvas.setLineWidth(4)
    p = canvas.beginPath()
    for i in (1,2,3,4):
        for j in (1,2):
            xc,yc = inch*i, inch*j
            p.moveTo(xc,yc)
            p.arcTo(xc-h, yc-k, xc+h, yc+k, startAng=0, extent=60*i)
            # close only the first one, not the second one
            if j==1:
                p.close()
    canvas.drawPath(p, fill=1, stroke=1)
```

The `closingfigures` function illustrates the effect of closing or not closing figures including a line segment and a partial ellipse.

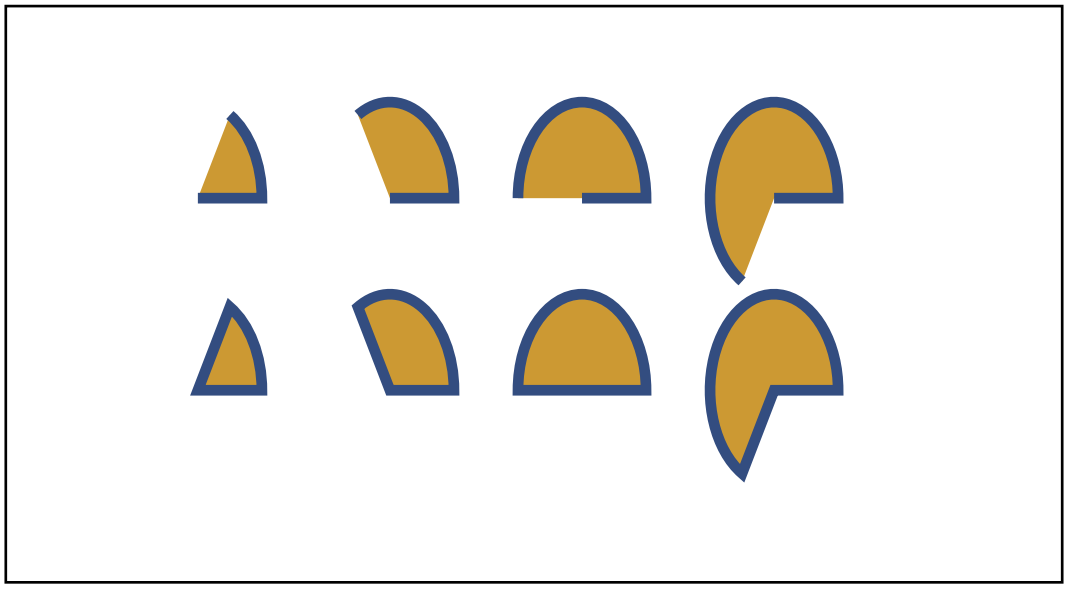


Figure 2-31: closing and not closing pathobject figures

Closing or not closing graphical figures effects only the stroked outline of a figure, not the filling of the figure as illustrated above.

For a more extensive example of drawing using a path object examine the `hand` function.

```
def hand(canvas, debug=1, fill=0):
    (startx, starty) = (0,0)
    curves = [
      ( 0, 2), ( 0, 4), ( 0, 8), # back of hand
      ( 5, 8), ( 7,10), ( 7,14),
      (10,14), (10,13), ( 7.5, 8), # thumb
      (13, 8), (14, 8), (17, 8),
      (19, 8), (19, 6), (17, 6),
      (15, 6), (13, 6), (11, 6), # index, pointing
      (12, 6), (13, 6), (14, 6),
      (16, 6), (16, 4), (14, 4),
      (13, 4), (12, 4), (11, 4), # middle
      (11.5, 4), (12, 4), (13, 4),
      (15, 4), (15, 2), (13, 2),
      (12.5, 2), (11.5, 2), (11, 2), # ring
      (11.5, 2), (12, 2), (12.5, 2),
      (14, 2), (14, 0), (12.5, 0),
      (10, 0), (8, 0), (6, 0), # pinky, then close
      ]
    from reportlab.lib.units import inch
    if debug: canvas.setLineWidth(6)
    u = inch*0.2
    p = canvas.beginPath()
    p.moveTo(startx, starty)
    ccopy = list(curves)
    while ccopy:
        [(x1,y1), (x2,y2), (x3,y3)] = ccopy[:3]
        del ccopy[:3]
        p.curveTo(x1*u,y1*u,x2*u,y2*u,x3*u,y3*u)
    p.close()
    canvas.drawPath(p, fill=fill)
    if debug:
        from reportlab.lib.colors import red, green
        (lastx, lasty) = (startx, starty)
        ccopy = list(curves)
        while ccopy:
            [(x1,y1), (x2,y2), (x3,y3)] = ccopy[:3]
            del ccopy[:3]
            canvas.setStrokeColor(red)
            canvas.line(lastx*u,lasty*u, x1*u,y1*u)
            canvas.setStrokeColor(green)
            canvas.line(x2*u,y2*u, x3*u,y3*u)
            (lastx,lasty) = (x3,y3)
```

In debug mode (the default) the `hand` function shows the tangent line segments to the bezier curves used to compose the figure. Note that where the segments line up the curves join smoothly, but where they do not line up the curves show a "sharp edge".

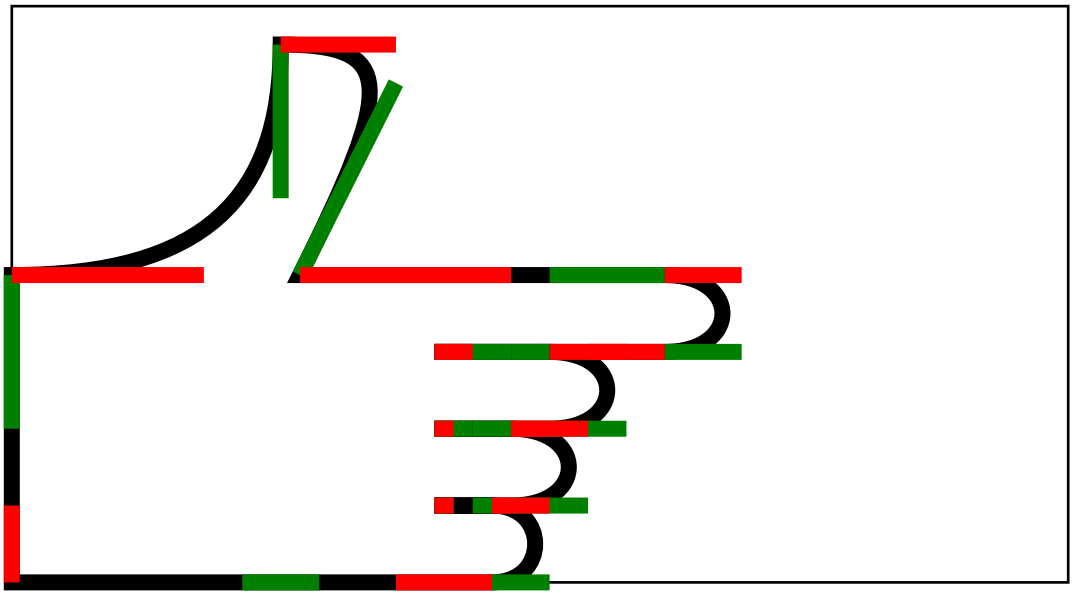


Figure 2-32: an outline of a hand using bezier curves

Used in non-debug mode the `hand` function only shows the Bezier curves. With the `fill` parameter set the figure is filled using the current fill color.

```
def hand2(canvas):
    canvas.translate(20,10)
    canvas.setLineWidth(3)
    canvas.setFillColorRGB(0.1, 0.3, 0.9)
    canvas.setStrokeGray(0.5)
    hand(canvas, debug=0, fill=1)
```

Note that the "stroking" of the border draws over the interior fill where they overlap.

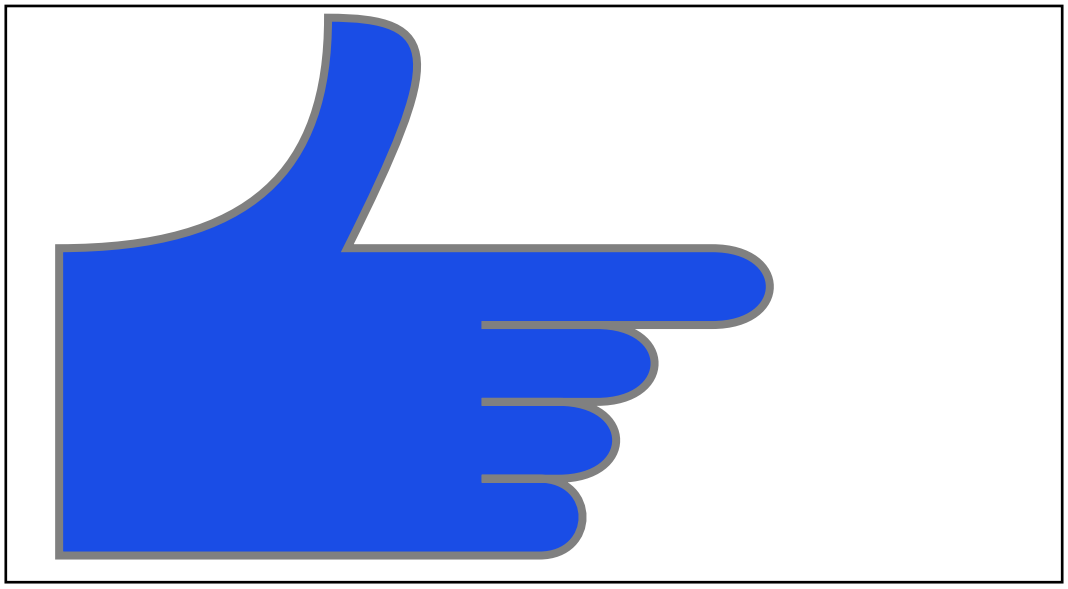


Figure 2-33: the finished hand, filled

2.17 Further Reading: The ReportLab Graphics Library

So far the graphics we have seen were created on a fairly low level. It should be noted, though, that there is another way of creating much more sophisticated graphics using the dedicated high-level *ReportLab Graphics Library*.

It can be used to produce high-quality, platform-independant, reusable graphics for different output formats (vector and bitmap) like PDF, EPS, SVG, JPG and PNG.

A more thorough description of its philsophy and features is now covered in Chapter 11 of this document, *Graphics*, which contains information about the existing components and how to create customized ones.

Chapter 11 also contains details of the ReportLab charting package and its components (labels, axes, legends and different types of charts like bar, line and pie charts) that builds directly on the graphics library.

Chapter 3 Fonts and encodings

This chapter covers fonts, encodings and Asian language capabilities. If you are purely concerned with generating PDFs for Western European languages, you can just read the "Unicode is the default" section below and skip the rest on a first reading. We expect this section to grow considerably over time. We hope that Open Source will enable us to give better support for more of the world's languages than other tools, and we welcome feedback and help in this area.

3.1 Unicode and UTF8 are the default input encodings

Starting with reportlab Version 2.0 (May 2006), all text input you provide to our APIs should be in UTF8 or as Python Unicode objects. This applies to arguments to canvas.drawString and related APIs, table cell content, drawing object parameters, and paragraph source text.

We considered making the input encoding configurable or even locale-dependent, but decided that "explicit is better than implicit".

This simplifies many things we used to do previously regarding greek letters, symbols and so on. To display any character, find out its unicode code point, and make sure the font you are using is able to display it.

If you are adapting a ReportLab 1.x application, or reading data from another source which contains single-byte data (e.g. latin-1 or WinAnsi), you need to do a conversion into Unicode. The Python codecs package now includes converters for all the common encodings, including Asian ones.

If your data is not encoded as UTF8, you will get a UnicodeDecodeError as soon as you feed in a non-ASCII character. For example, this snippet below is attempting to read in and print a series of names, including one with a French accent: *`Marc-Andr\u00e9 Lemburg`*. The standard error is quite helpful and tells you what character it doesn't like:

```
>>> from reportlab.pdfgen.canvas import Canvas
>>> c = Canvas('temp.pdf')
>>> y = 700
>>> for line in file('latin_python_gurus.txt','r'):
...     c.drawString(100, y, line.strip())
...
Traceback (most recent call last):
...
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 9-11: invalid data
-->\u00e9 L<--emburg
>>>
```

The simplest fix is just to convert your data to unicode, saying which encoding it comes from, like this:

```
>>> for line in file('latin_input.txt','r'):
...     uniLine = unicode(line, 'latin-1')
...     c.drawString(100, y, uniLine.strip())
>>>
>>> c.save()
```

3.2 Automatic output font substitution

There are still a number of places in the code, including the rl_config defaultEncoding parameter, and arguments passed to various Font constructors, which refer to encodings. These were useful in the past when people needed to use glyphs in the Symbol and ZapfDingbats fonts which are supported by PDF viewing devices. By default the standard fonts (Helvetica, Courier, Times Roman) will offer the glyphs available in Latin-1. However, if our engine detects a character not in the font, it will attempt to switch to Symbol or ZapfDingbats to display these. For example, if you include the Unicode character for a pair of right-facing scissors, \u2702, in a call to *`drawString`*, you should see them (there is an example in *`test_pdfgen_general.py/pdf`*). It is not necessary to switch fonts in your code.

3.3 Using non-standard Type 1 fonts

As discussed in the previous chapter, every copy of Acrobat Reader comes with 14 standard fonts built in. Therefore, the ReportLab PDF Library only needs to refer to these by name. If you want to use other fonts, they must be available to your code and will be embedded in the PDF document.

You can use the mechanism described below to include arbitrary fonts in your documents. We have an open source font named *DarkGardenMK* which we may use for testing and/or documenting purposes (and which you may use as well). It comes bundled with the ReportLab distribution in the directory `reportlab/fonts`.

Right now font-embedding relies on font description files in the Adobe AFM ('Adobe Font Metrics') and PFB ('Printer Font Binary') format. The former is an ASCII file and contains information about the characters ('glyphs') in the font such as height, width, bounding box info and other 'metrics', while the latter is a binary file that describes the shapes of the font. The `reportlab/fonts` directory contains the files `'DarkGardenMK.afm'` and `'DarkGardenMK.pfb'` that are used as an example font.

In the following example locate the folder containing the test font and register it for future use with the `pdfmetrics` module, after which we can use it like any other standard font.

```
import os
import reportlab
folder = os.path.dirname(reportlab.__file__) + os.sep + 'fonts'
afmFile = os.path.join(folder, 'DarkGardenMK.afm')
pfbFile = os.path.join(folder, 'DarkGardenMK.pfb')

from reportlab.pdfbase import pdfmetrics
justFace = pdfmetrics.EmbeddedType1Face(afmFile, pfbFile)
faceName = 'DarkGardenMK' # pulled from AFM file
pdfmetrics.registerTypeFace(justFace)
justFont = pdfmetrics.Font('DarkGardenMK',
                           faceName,
                           'WinAnsiEncoding')
pdfmetrics.registerFont(justFont)

canvas.setFont('DarkGardenMK', 32)
canvas.drawString(10, 150, 'This should be in')
canvas.drawString(10, 100, 'DarkGardenMK')
```

Note that the argument "WinAnsiEncoding" has nothing to do with the input; it's to say which set of characters within the font file will be active and available.

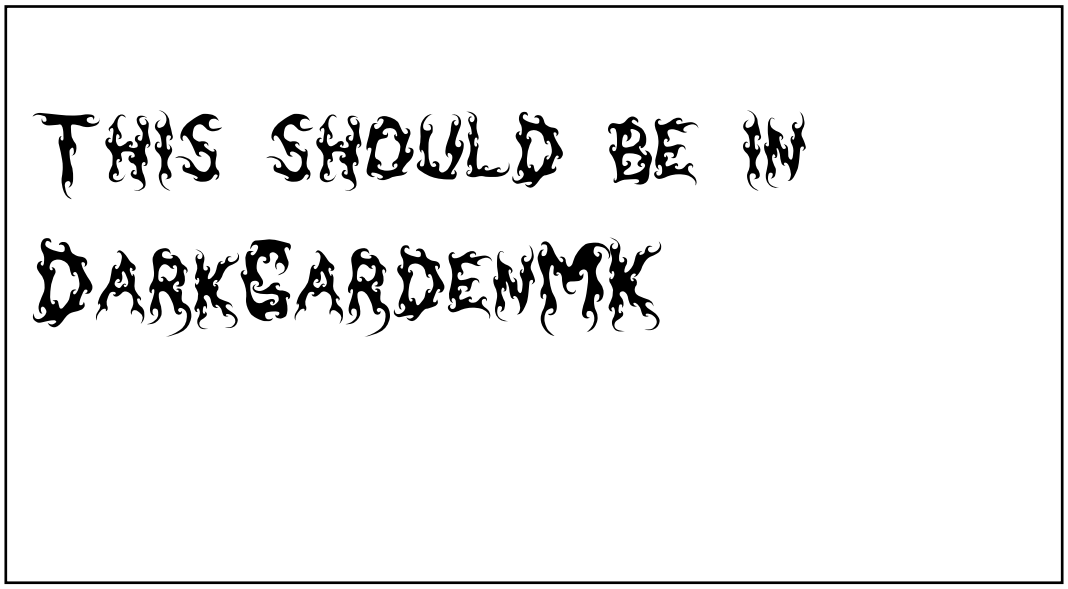


Figure 3-1: Using a very non-standard font

The font's facename comes from the AFM file's `FontName` field. In the example above we knew the name in advance, but quite often the names of font description files are pretty cryptic and then you might want to retrieve the name from an AFM file automatically. When lacking a more sophisticated method you can use some code as simple as this:

```
class FontNameNotFoundError(Exception):
    pass


def findFontName(path):
    "Extract a font name from an AFM file."

    f = open(path)

    found = 0
    while not found:
        line = f.readline()[:-1]
        if not found and line[:16] == 'StartCharMetrics':
            raise FontNameNotFoundError, path
        if line[:8] == 'FontName':
            fontName = line[9:]
            found = 1

    return fontName
```

In the *DarkGardenMK* example we explicitely specified the place of the font description files to be loaded. In general, you'll prefer to store your fonts in some canonic locations and make the embedding mechanism aware of them. Using the same configuration mechanism we've already seen at the beginning of this section we can indicate a default search path for Type-1 fonts.

Unfortunately, there is no reliable standard yet for such locations (not even on the same platform) and, hence, you might have to edit the file `reportlab/rl_config.py` to modify the value of the `T1SearchPath` identifier to contain additional directories. Our own recommendation is to use the *`reportlab/fonts`* folder in development; and to have any needed fonts as packaged parts of your application in any kind of controlled server deployment. This insulates you from fonts being installed and uninstalled by other software or system administrator.

Warnings about missing glyphs

If you specify an encoding, it is generally assumed that the font designer has provided all the needed glyphs. However, this is not always true. In the case of our example font, the letters of the alphabet are present, but many symbols and accents are missing. The default behaviour is for the font to print a 'notdef' character - typically a blob, dot or space - when passed a character it cannot draw. However, you can ask the library to warn you instead; the code below (executed before loading a font) will cause warnings to be generated for any glyphs not in the font when you register it.

```
import reportlab.rl_config
reportlab.rl_config.warnOnMissingFontGlyphs = 0
```

3.4 Standard Single-Byte Font Encodings

This section shows you the glyphs available in the common encodings.

The code chart below shows the characters in the `WinAnsiEncoding`. This is the standard encoding on Windows and many Unix systems in America and Western Europe. It is also knows as Code Page 1252, and is practically identical to ISO-Latin-1 (it contains one or two extra characters). This is the default encoding used by the Reportlab PDF Library. It was generated from a standard routine in `reportlab/lib`, `codecharts.py`, which can be used to display the contents of fonts. The index numbers along the edges are in hex.

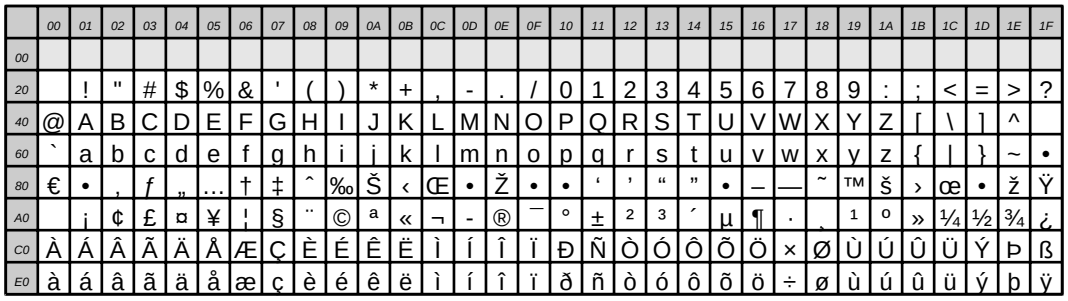


Figure 3-2: WinAnsi Encoding

The code chart below shows the characters in the `MacRomanEncoding`. as it sounds, this is the standard encoding on Macintosh computers in America and Western Europe. As usual with non-unicode encodings, the first 128 code points (top 4 rows in this case) are the ASCII standard and agree with the WinAnsi code chart above; but the bottom 4 rows differ.

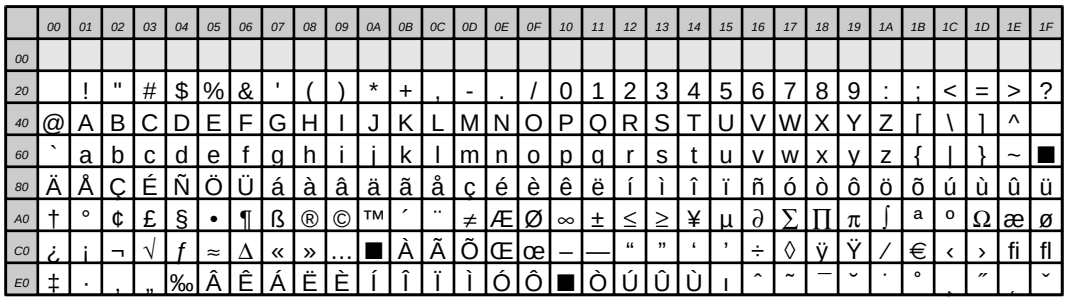


Figure 3-3: MacRoman Encoding

These two encodings are available for the standard fonts (Helvetica, Times-Roman and Courier and their variants) and will be available for most commercial fonts including those from Adobe. However, some fonts contain non- text glyphs and the concept does not really apply. For example, ZapfDingbats and Symbol can each be treated as having their own encoding.

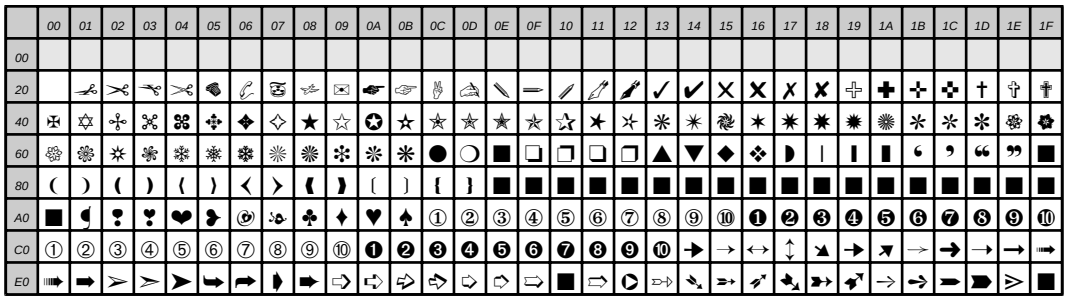


Figure 3-4: ZapfDingbats and its one and only encoding

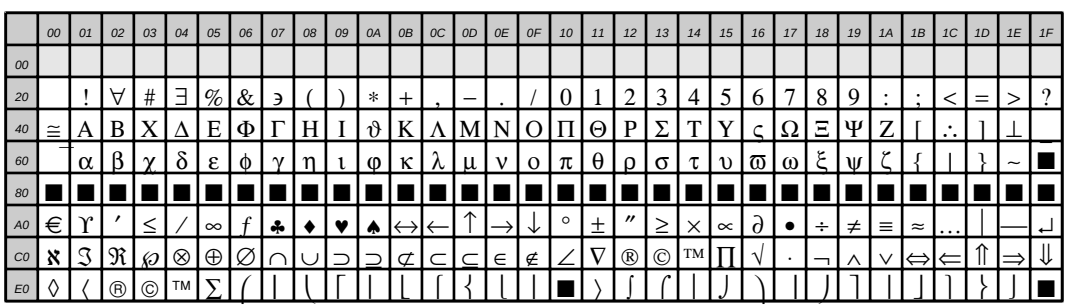


Figure 3-5: Symbol and its one and only encoding

3.5 TrueType Font Support

Marius Gedminas (mgedmin@delfi.lt) with the help of Viktorija Zaksiene (vika@pov.lt) have contributed support for embedded TrueType fonts. TrueType fonts work in Unicode/UTF8 and are not limited to 256 characters.

We use **reportlab.pdfbase.ttfonts.TTFont** to create a true type font object and register using **reportlab.pdfbase.pdfmetrics.registerFont**. In pdfgen drawing directly to the canvas we can do

```
# we know some glyphs are missing, suppress warnings
import reportlab.rl_config
reportlab.rl_config.warnOnMissingFontGlyphs = 0

from reportlab.pdfbase import pdfmetrics
from reportlab.pdfbase.ttfonts import TTFont
pdfmetrics.registerFont(TTFont('Vera', 'Vera.ttf'))
pdfmetrics.registerFont(TTFont('VeraBd', 'VeraBd.ttf'))
pdfmetrics.registerFont(TTFont('VeraIt', 'VeraIt.ttf'))
pdfmetrics.registerFont(TTFont('VeraBI', 'VeraBI.ttf'))
canvas.setFont('Vera', 32)
canvas.drawString(10, 150, "Some text encoded in UTF-8")
canvas.drawString(10, 100, "In the Vera TT Font!")
```

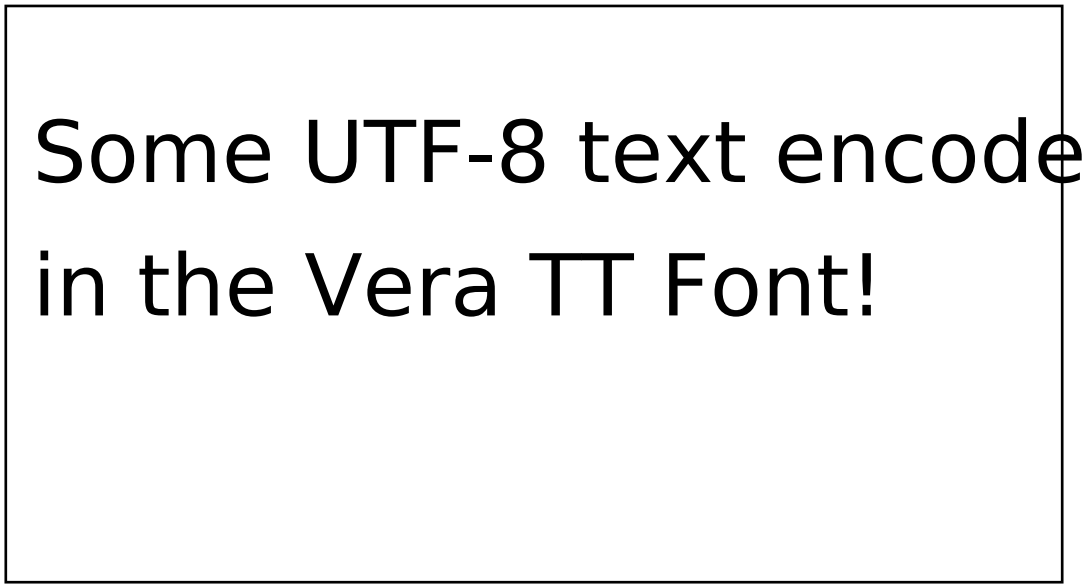


Figure 3-6: Using a the Vera TrueType Font

In the above example the true type font object is created using

```
TTFont(name,filename)
```

so that the ReportLab internal name is given by the first argument and the second argument is a string(or file like object) denoting the font's TTF file. In Marius' original patch the filename was supposed to be exactly correct, but we have modified things so that if the filename is relative then a search for the corresponding file is done in the current directory and then in directories specified by `reportlab.rl_config.TTFSearchpath`!

Before using the TT Fonts in Platypus we should add a mapping from the family name to the individual font names that describe the behaviour under the `<b>` and `<i>` attributes.

```
from reportlab.pdfbase.pdfmetrics import registerFontFamily
registerFontFamily('Vera',normal='Vera',bold='VeraBd',italic='VeraIt',boldItalic='VeraBI')
```

If we only have a Vera regular font, no bold or italic then we must map all to the same internal fontname. `<b>` and `<i>` tags may now be used safely, but have no effect. After registering and mapping the Vera font

as above we can use paragraph text like

Figure 3-7: Using TTF fonts in paragraphs

3.6 Asian Font Support

The Reportlab PDF Library aims to expose full support for Asian fonts. PDF is the first really portable solution for Asian text handling. There are two main approaches for this: Adobe's Asian Language Packs, or TrueType fonts.

Asian Language Packs

This approach offers the best performance since nothing needs embedding in the PDF file; as with the standard fonts, everything is on the reader.

Adobe makes available add-ons for each main language. In Adobe Reader 6.0 and 7.0, you will be prompted to download and install these as soon as you try to open a document using them. In earlier versions, you would see an error message on opening an Asian document and had to know what to do.

Japanese, Traditional Chinese (Taiwan/Hong Kong), Simplified Chinese (mainland China) and Korean are all supported and our software knows about the following fonts:

- ■■2022 `chs` = Chinese Simplified (mainland): '`STSong-Light`'
- ■■2022 `cht` = Chinese Traditional (Taiwan): '`MSung-Light`', '`MHei-Medium`'
- ■■2022 `kor` = Korean: '`HYSMyeongJoStd-Medium`','`HYGothic-Medium`'
- ■■2022 `jpn` = Japanese: '`HeiseiMin-W3`', '`HeiseiKakuGo-W5`'

Since many users will not have the font packs installed, we have included a rather grainy *`bitmap`* of some Japanese characters. We will discuss below what is needed to generate them.

An image should have appeared here.

Prior to Version 2.0, you had to specify one of many native encodings when registering a CID Font. In version 2.0 you should a new UnicodeCIDFont class.

```
from reportlab.pdfbase import pdfmetrics
from reportlab.pdfbase.cidfonts import UnicodeCIDFont
pdfmetrics.registerFont(UnicodeCIDFont('HeiseiMin-W3'))
canvas.setFont('HeiseiMin-W3', 16)

# the two unicode characters below are "Tokyo"
msg = u'\u6771\u4EAC : Unicode font, unicode input'
canvas.drawString(100, 675, msg)
```

The old coding style with explicit encodings should still work, but is now only relevant if you need to construct vertical text. We aim to add more readable options for horizontal and vertical text to the UnicodeCIDFont constructor in future. The following four test scripts generate samples in the corresponding languages:

```
tests/test_multibyte_jpn.py
tests/test_multibyte_kor.py
tests/test_multibyte_chs.py
tests/test_multibyte_cht.py
```

In previous versions of the ReportLab PDF Library, we had to make use of Adobe's CMap files (located near Acrobat Reader if the Asian Language packs were installed). Now that we only have one encoding to deal with, the character width data is embedded in the package, and CMap files are not needed for generation. The CMap search path in *`rl_config.py`* is now deprecated and has no effect if you restrict yourself to UnicodeCIDFont.

TrueType fonts with Asian characters

This is the easy way to do it. No special handling at all is needed to work with Asian TrueType fonts. Windows users who have installed, for example, Japanese as an option in Control Panel, will have a font "msmincho.ttf" which can be used. However, be aware that it takes time to parse the fonts, and that quite large subsets may need to be embedded in your PDFs. We can also now parse files ending in .ttc, which are a slight variation of .ttf.

To Do

We expect to be developing this area of the package for some time.accept2dyear Here is an outline of the main priorities. We welcome help!

- ■■2022 Ensure that we have accurate character metrics for all encodings in horizontal and vertical writing.
- ■■2022 Add options to `UnicodeCIDFont` to allow vertical and proportional variants where the font permits it.
- ■■2022 Improve the word wrapping code in paragraphs and allow vertical writing.

3.7 RenderPM tests

This may also be the best place to mention the test function of `reportlab/graphics/renderPM.py`, which can be considered the cannonical place for tests which exercise renderPM (the "PixMap Renderer", as opposed to renderPDF, renderPS or renderSVG).

If you run this from the command line, you should see lots of output like the following.

```
C:\code\reportlab\graphics>renderPM.py
wrote pmout\renderPM0.gif
wrote pmout\renderPM0.tif
wrote pmout\renderPM0.png
wrote pmout\renderPM0.jpg
wrote pmout\renderPM0.pct
...
wrote pmout\renderPM12.gif
wrote pmout\renderPM12.tif
wrote pmout\renderPM12.png
wrote pmout\renderPM12.jpg
wrote pmout\renderPM12.pct
wrote pmout\index.html
```

This runs a number of tests progressing from a "Hello World" test, through various tests of Lines; text strings in a number of sizes, fonts, colours and alignments; the basic shapes; translated and rotated groups; scaled coordinates; rotated strings; nested groups; anchoring and non-standard fonts.

It creates a subdirectory called `pmout`, writes the image files into it, and writes an `index.html` page which makes it easy to refer to all the results.

The font-related tests which you may wish to look at are test #11 ('Text strings in a non-standard font') and test #12 ('Test Various Fonts').

Chapter 4 Exposing PDF Special Capabilities

PDF provides a number of features to make electronic document viewing more efficient and comfortable, and our library exposes a number of these.

4.1 Forms

The Form feature lets you create a block of graphics and text once near the start of a PDF file, and then simply refer to it on subsequent pages. If you are dealing with a run of 5000 repetitive business forms - for example, one-page invoices or payslips - you only need to store the backdrop once and simply draw the changing text on each page. Used correctly, forms can dramatically cut file size and production time, and apparently even speed things up on the printer.

Forms do not need to refer to a whole page; anything which might be repeated often should be placed in a form.

The example below shows the basic sequence used. A real program would probably define the forms up front and refer to them from another location.

```
def forms(canvas):
    #first create a form...
    canvas.beginForm("SpumoniForm")
    #re-use some drawing functions from earlier
    spumoni(canvas)
    canvas.endForm()

    #then draw it
    canvas.doForm("SpumoniForm")
```

4.2 Links and Destinations

PDF supports internal hyperlinks. There is a very wide range of link types, destination types and events which can be triggered by a click. At the moment we just support the basic ability to jump from one part of a document to another, and to control the zoom level of the window after the jump. The bookmarkPage method defines a destination that is the endpoint of a jump.

```
canvas.bookmarkPage(name,
                    fit="Fit",
                    left=None,
                    top=None,
                    bottom=None,
                    right=None,
                    zoom=None
                    )
```

By default the `bookmarkPage` method defines the page itself as the destination. After jumping to an endpoint defined by bookmarkPage, the PDF browser will display the whole page, scaling it to fit the screen:

```
canvas.bookmarkPage(name)
```

The `bookmarkPage` method can be instructed to display the page in a number of different ways by providing a `fit` parameter.

fit	Parameters Required	Meaning
Fit		**Entire page fits in window (the default)**
FitH	top	Top coord at top of window, width scaled to fit
FitV	left	Left coord at left of window, height scaled to fit
FitR	left bottom right top	Scale window to fit the specified rectangle
XYZ	left top zoom	Fine grained control. If you omit a parameter the PDF browser interprets it as "leave as is"

Table 4-1 - Required attributes for different fit types

Note : `fit` settings are case-sensitive so `fit="FIT"` is invalid

Sometimes you want the destination of a jump to be some part of a page. The `FitR` fit allows you to identify a particular rectangle, scaling the area to fit the entire page.

To set the display to a particular x and y coordinate of the page and to control the zoom directly use fit="XYZ".

```
canvas.bookmarkPage('my_bookmark',fit="XYZ",left=0,top=200)
```

This destination is at the leftmost of the page with the top of the screen at position 200. Because `zoom` was not set the zoom remains at whatever the user had it set to.

```
canvas.bookmarkPage('my_bookmark',fit="XYZ",left=0,top=200,zoom=2)
```

This time zoom is set to expand the page 2X its normal size.

Note : Both `XYZ` and `FitR` fit types require that their positional parameters (`top, bottom, left, right`) be specified in terms of the default user space. They ignore any geometric transform in effect in the canvas graphic state.

Note: Two previous bookmark methods are supported but deprecated now that bookmarkPage is so general. These are `bookmarkHorizontalAbsolute` and `bookmarkHorizontal`.

Defining internal links

```
canvas.linkAbsolute(contents, destinationname, Rect=None, addtopage=1, name=None,
thickness=0, color=None, dashArray=None, **kw)
```

The `linkAbsolute` method defines a starting point for a jump. When the user is browsing the generated document using a dynamic viewer (such as Acrobat Reader) when the mouse is clicked when the pointer is within the rectangle specified by `Rect` the viewer will jump to the endpoint associated with `destinationname`. As in the case with `bookmarkHorizontalAbsolute` the rectangle `Rect` must be specified in terms of the default user space. The `contents` parameter specifies a chunk of text which displays in the viewer if the user left-clicks on the region.

The rectangle `Rect` must be specified in terms of a tuple `(x1,y1,x2,y2)` identifying the lower left and upper right points of the rectangle in default user space.

For example the code

```
canvas.bookmarkPage("Meaning_of_life")
```

defines a location as the whole of the current page with the identifier `Meaning_of_life`. To create a rectangular link to it while drawing a possibly different page, we would use this code:

```
canvas.linkAbsolute("Find the Meaning of Life", "Meaning_of_life",
                    (inch, inch, 6*inch, 2*inch))
```

By default during interactive viewing a rectangle appears around the link. Use the keyword argument `Border='[0 0 0]'` to suppress the visible rectangle around the during viewing link. For example

```
canvas.linkAbsolute("Meaning of Life", "Meaning_of_life",
                    (inch, inch, 6*inch, 2*inch), Border='[0 0 0]')
```

The `thickness`, `color` and `dashArray` arguments may be used alternately to specify a border if no Border argument is specified. If Border is specified it must be either a string representation of a PDF array or a `PDFArray` (see the pdfdoc module). The `color` argument (which should be a `Color` instance) is equivalent to a keyword argument `C` which should resolve to a PDF color definition (Normally a three entry PDF array).

The `canvas.linkRect` method is similar in intent to the `linkAbsolute` method, but has an extra argument `relative=1` so is intended to obey the local userspace transformation.

4.3 Outline Trees

Acrobat Reader has a navigation page which can hold a document outline; it should normally be visible when you open this guide. We provide some simple methods to add outline entries. Typically, a program to make a document (such as this user guide) will call the method `canvas.addOutlineEntry(`*`self, title, key, level=0, closed=None`*`)` as it reaches each heading in the document.

`title` is the caption which will be displayed in the left pane. The *`key`* must be a string which is unique within the document and which names a bookmark, as with the hyperlinks. The *`level`* is zero - the uppermost level - unless otherwise specified, and it is an error to go down more than one level at a time (for example to follow a level 0 heading by a level 2 heading). Finally, the *`closed`* argument specifies whether the node in the outline pane is closed or opened by default.

The snippet below is taken from the document template that formats this user guide. A central processor looks at each paragraph in turn, and makes a new outline entry when a new chapter occurs, taking the chapter heading text as the caption text. The key is obtained from the chapter number (not shown here), so Chapter 2 has the key 'ch2'. The bookmark to which the outline entry points aims at the whole page, but it could as easily have been an individual paragraph.

```
#abridged code from our document template
if paragraph.style == 'Heading1':
    self.chapter = paragraph.getPlainText()
    key = 'ch%d' % self.chapterNo
    self.canv.bookmarkPage(key)
    self.canv.addOutlineEntry(paragraph.getPlainText(),
                                          key, 0, 0)
```

4.4 Page Transition Effects

```
canvas.setPageTransition(self, effectname=None, duration=1,
                         direction=0,dimension='H',motion='I')
```

The `setPageTransition` method specifies how one page will be replaced with the next. By setting the page transition effect to "dissolve" for example the current page will appear to melt away when it is replaced by the next page during interactive viewing. These effects are useful in spicing up slide presentations, among other places. Please see the reference manual for more detail on how to use this method.

4.5 Internal File Annotations

```
canvas.setAuthor(name)
canvas.setTitle(title)
canvas.setSubject(subj)
```

These methods have no automatically seen visible effect on the document. They add internal annotations to the document. These annotations can be viewed using the "Document Info" menu item of the browser and they also can be used as a simple standard way of providing basic information about the document to archiving software which need not parse the entire file. To find the annotations view the `*.pdf` output file using a standard text editor (such as `notepad` on MS/Windows or `vi` or `emacs` on unix) and look for the string `/Author` in the file contents.

```
def annotations(canvas):
    from reportlab.lib.units import inch
    canvas.drawString(inch, 2.5*inch,
       "setAuthor, setTitle, setSubject have no visible effect")
    canvas.drawString(inch, inch, "But if you are viewing this document dynamically")
    canvas.drawString(inch, 0.5*inch, "please look at File/Document Info")
    canvas.setAuthor("the ReportLab Team")
    canvas.setTitle("ReportLab PDF Generation User Guide")
    canvas.setSubject("How to Generate PDF files using the ReportLab modules")
```

If you want the subject, title, and author to automatically display in the document when viewed and printed you must paint them onto the document like any other text.

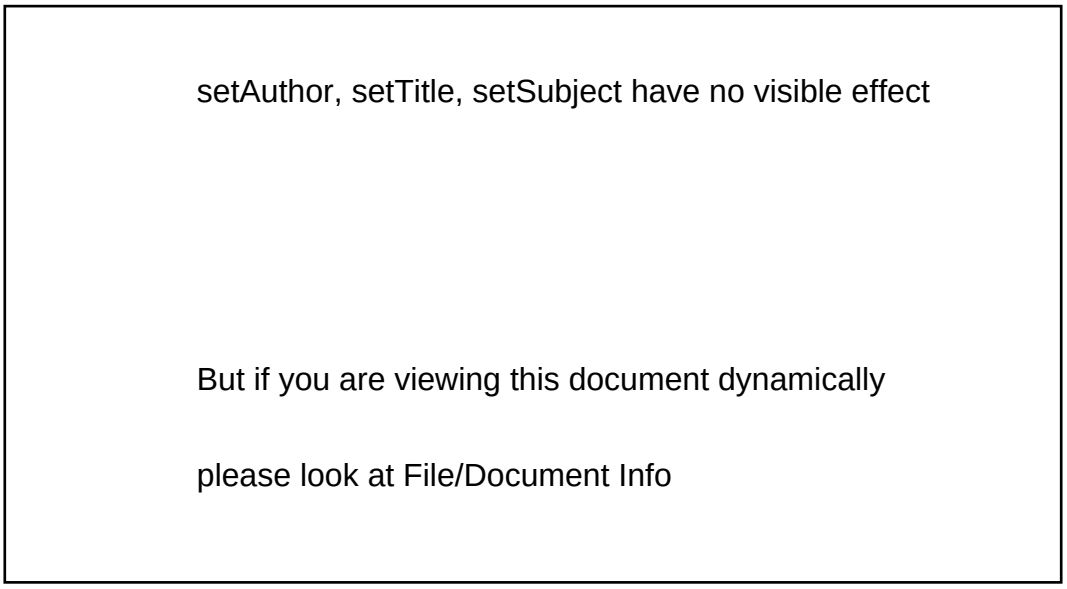


Figure 4-1: Setting document internal annotations

4.6 Encryption

About encrypting PDF files

Adobe's PDF standard allows you to do three related things to a PDF file when you encrypt it:

- ■■2022 Apply password protection to it, so a user must supply a valid password before being able to read it,
- ■■2022 Encrypt the contents of the file to make it useless until it is decrypted, and
- ■■2022 Control whether the user can print, copy and paste or modify the document while viewing it.

The PDF security handler allows two different passwords to be specified for a document:

- ■■2022 The 'owner' password (aka the 'security password' or 'master password')
- ■■2022 The 'user' password (aka the 'open password')

When a user supplies either one of these passwords, the PDF file will be opened, decrypted and displayed on screen.

If the owner password is supplied, then the file is opened with full control - you can do anything to it, including changing the security settings and passwords, or re-encrypting it with a new password.

If the user password was the one that was supplied, you open it up in a more restricted mode. The restrictions were put in place when the file was encrypted, and will either allow or deny the user permission to do the following:

- ■■2022 Modifying the document's contents
- ■■2022 Copying text and graphics from the document
- ■■2022 Adding or modifying text annotations and interactive form fields
- ■■2022 Printing the document

Note that all password protected PDF files are encrypted, but not all encrypted PDFs are password protected. If a document's user password is an empty string, there will be no prompt for the password when the file is opened. If you only secure a document with the owner password, there will also not be a prompt for the password when you open the file. If the owner and user passwords are set to the same string when encrypting the PDF file, the document will always open with the user access privileges. This means that it is possible to create a file which, for example, is impossible for anyone to print out, even the person who created it.

Owner Password set?	User Password set?	Result
Y	-	No password required when opening file. Restrictions apply to everyone.
-	Y	User password required when opening file. Restrictions apply to everyone.
Y	Y	A password required when opening file. Restrictions apply only if user password supplied.

When a PDF file is encrypted, encryption is applied to all the strings and streams in the file. This prevents people who don't have the password from simply removing the password from the PDF file to gain access to it - it renders the file useless unless you actually have the password.

PDF's standard encryption methods use the MD5 message digest algorithm (as described in RFC 1321, The MD5 Message-Digest Algorithm) and an encryption algorithm known as RC4. RC4 is a symmetric stream cipher - the same algorithm is used both for encryption and decryption, and the algorithm does not change the length of the data.

How To Use Encryption

Documents can be encrypted by passing an argument to the canvas object.

If the argument is a string object, it is used as the User password to the PDF.

The argument can also be an instance of the class `reportlab.lib.pdfencrypt.StandardEncryption`, which allows more finegrained control over encryption settings.

The `StandardEncryption` constructor takes the following arguments:

```
def __init__(self, userPassword,
        ownerPassword=None,
        canPrint=1,
        canModify=1,
        canCopy=1,
        canAnnotate=1,
        strength=40):
```

The `userPassword` and `ownerPassword` parameters set the relevant password on the encrypted PDF.

The boolean flags `canPrint`, `canModify`, `canCopy`, `canAnnotate` determine wether a user can perform the corresponding actions on the PDF when only a user password has been supplied.

If the user supplies the owner password while opening the PDF, all actions can be performed regardless of the flags.

Example

To create a document named hello.pdf with a user password of 'rptlab' on which printing is not allowed, use the following code:

```
from reportlab.pdfgen import canvas
from reportlab.lib import pdfencrypt

enc=pdfencrypt.StandardEncryption("rptlab",canPrint=0)

def hello(c):
    c.drawString(100,100,"Hello World")
c = canvas.Canvas("hello.pdf",encrypt=enc)
hello(c)
c.showPage()
c.save()
```

Chapter 5 PLATYPUS - Page Layout and Typography Using Scripts

5.1 Design Goals

Platypus stands for "Page Layout and Typography Using Scripts". It is a high level page layout library which lets you programmatically create complex documents with a minimum of effort.

The design of Platypus seeks to separate "high level" layout decisions from the document content as much as possible. Thus, for example, paragraphs are constructed using paragraph styles and pages are constructed using page templates with the intention that hundreds of documents with thousands of pages can be reformatted to different style specifications with the modifications of a few lines in a single shared file which contains the paragraph styles and page layout specifications.

The overall design of Platypus can be thought of has having several layers, top down, these are

`DocTemplates` the outermost container for the document;

`PageTemplates` specifications for layouts of pages of various kinds;

`Frames` specifications of regions in pages that can contain flowing text or graphics.

`Flowables` text or graphic elements that should be "flowed into the document (i.e. things like images, paragraphs and tables, but not things like page footers or fixed page graphics).

`pdfgen.Canvas` the lowest level which ultimately receives the painting of the document from the other layers.

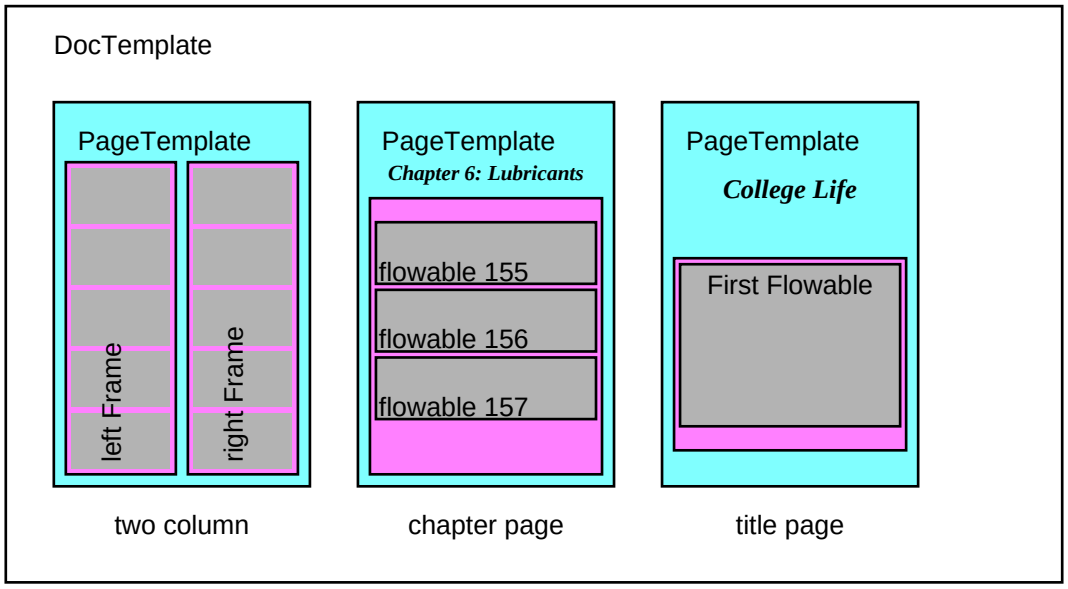


Figure 5-1: Illustration of DocTemplate structure

The illustration above graphically illustrates the concepts of `DocTemplates`, `PageTemplates` and `Flowables`. It is deceptive, however, because each of the `PageTemplates` actually may specify the format for any number of pages (not just one as might be inferred from the diagram).

`DocTemplates` contain one or more `PageTemplates` each of which contain one or more `Frames`. `Flowables` are things which can be *flowed* into a `Frame` e.g. a `Paragraph` or a `Table`.

To use platypus you create a document from a `DocTemplate` class and pass a list of `Flowable`s to its `build` method. The document `build` method knows how to process the list of flowables into something reasonable.

Internally the `DocTemplate` class implements page layout and formatting using various events. Each of the events has a corresponding handler method called `handle_XXX` where `XXX` is the event name. A typical event is `frameBegin` which occurs when the machinery begins to use a frame for the first time.

A Platypus story consists of a sequence of basic elements called `Flowables` and these elements drive the data driven Platypus formatting engine. To modify the behavior of the engine a special kind of flowable, `ActionFlowables`, tell the layout engine to, for example, skip to the next column or change to another `PageTemplate`.

5.2 Getting started

Consider the following code sequence which provides a very simple "hello world" example for Platypus.

```
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.rl_config import defaultPageSize
from reportlab.lib.units import inch
PAGE_HEIGHT=defaultPageSize[1]; PAGE_WIDTH=defaultPageSize[0]
styles = getSampleStyleSheet()
```

First we import some constructors, some paragraph styles and other conveniences from other modules.

```
Title = "Hello world"
pageinfo = "platypus example"
def myFirstPage(canvas, doc):
    canvas.saveState()
    canvas.setFont('Times-Bold',16)
    canvas.drawCentredString(PAGE_WIDTH/2.0, PAGE_HEIGHT-108, Title)
    canvas.setFont('Times-Roman',9)
    canvas.drawString(inch, 0.75 * inch, "First Page / %s" % pageinfo)
    canvas.restoreState()
```

We define the fixed features of the first page of the document with the function above.

```
def myLaterPages(canvas, doc):
    canvas.saveState()
    canvas.setFont('Times-Roman',9)
    canvas.drawString(inch, 0.75 * inch, "Page %d %s" % (doc.page, pageinfo))
    canvas.restoreState()
```

Since we want pages after the first to look different from the first we define an alternate layout for the fixed features of the other pages. Note that the two functions above use the `pdfgen` level canvas operations to paint the annotations for the pages.

```
def go():
    doc = SimpleDocTemplate("phello.pdf")
    Story = [Spacer(1,2*inch)]
    style = styles["Normal"]
    for i in range(100):
        bogustext = ("This is Paragraph number %s.  " % i) *20
        p = Paragraph(bogustext, style)
        Story.append(p)
        Story.append(Spacer(1,0.2*inch))
    doc.build(Story, onFirstPage=myFirstPage, onLaterPages=myLaterPages)
```

Finally, we create a story and build the document. Note that we are using a "canned" document template here which comes pre-built with page templates. We are also using a pre-built paragraph style. We are only using two types of flowables here -- `Spacers` and `Paragraphs`. The first `Spacer` ensures that the Paragraphs skip past the title string.

To see the output of this example program run the module `docs/userguide/examples.py` (from the ReportLab `docs` distribution) as a "top level script". The script interpretation `python examples.py` will generate the Platypus output `phello.pdf`.

5.3 Flowables

Flowables are things which can be drawn and which have wrap, draw and perhaps split methods. Flowable is an abstract base class for things to be drawn and an instance knows its size and draws in its own coordinate system (this requires the base API to provide an absolute coordinate system when the Flowable.draw method is called). To get an instance use f=Flowable().

It should be noted that the Flowable class is an *abstract* class and is normally only used as a base class.

To illustrate the general way in which Flowables are used we show how a derived class Paragraph is used and drawn on a canvas. Paragraphs are so important they will get a whole chapter to themselves.

```
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.platypus import Paragraph
from reportlab.pdfgen.canvas import Canvas
styleSheet = getSampleStyleSheet()
style = styleSheet['BodyText']
P=Paragraph('This is a very silly example',style)
canv = Canvas('doc.pdf')
aW = 460    # available width and height
aH = 800
w,h = P.wrap(aW, aH)    # find required space
if w<=aW and h<=aH:
    P.drawOn(canv,0,aH)
    aH = aH - h         # reduce the available height
    canv.save()
else:
    raise ValueError, "Not enough room"
```

Flowable User Methods

Flowable.draw()

This will be called to ask the flowable to actually render itself. The Flowable class does not implement draw. The calling code should ensure that the flowable has an attribute canv which is the pdfgen.Canvas which should be drawn to an that the Canvas is in an appropriate state (as regards translations rotations, etc). Normally this method will only be called internally by the drawOn method. Derived classes must implement this method.

Flowable.drawOn(canvas,x,y)

This is the method which controlling programs use to render the flowable to a particular canvas. It handles the translation to the canvas coordinate (*x*,*y*) and ensuring that the flowable has a canv attribute so that the draw method (which is not implemented in the base class) can render in an absolute coordinate frame.

Flowable.wrap(availWidth, availHeight)

This will be called by the enclosing frame before objects are asked their size, drawn or whatever. It returns the size actually used.

Flowable.split(self, availWidth, availheight):

This will be called by more sophisticated frames when wrap fails. Stupid flowables should return [] meaning that they are unable to split. Clever flowables should split themselves and return a list of flowables. It is up to the client code to ensure that repeated attempts to split are avoided. If the space is sufficient the split method should return [self]. Otherwise the flowable should rearrange itself and return a list [f0,...] of flowables which will be considered in order. The implemented split method should avoid changing self as this will allow sophisticated layout mechanisms to do multiple passes over a list of flowables.

5.4 Guidelines for flowable positioning

Two methods, which by default return zero, provide guidance on vertical spacing of flowables:

```
Flowable.getSpaceAfter(self):
Flowable.getSpaceBefore(self):
```

These methods return how much space should follow or precede the flowable. The space doesn't belong to the flowable itself i.e. the flowable's `draw` method shouldn't consider it when rendering. Controlling programs will use the values returned in determining how much space is required by a particular flowable in context.

All flowables have an `hAlign` property: `('LEFT', 'RIGHT', 'CENTER' or 'CENTRE')`. For paragraphs, which fill the full width of the frame, this has no effect. For tables, images or other objects which are less than the width of the frame, this determines their horizontal placement.

The chapters which follow will cover the most important specific types of flowables: Paragraphs and Tables.

5.5 Frames

`Frames` are active containers which are themselves contained in `PageTemplates`. `Frames` have a location and size and maintain a concept of remaining drawable space. The command

```
Frame(x1, y1, width,height, leftPadding=6, bottomPadding=6,
        rightPadding=6, topPadding=6, id=None, showBoundary=0)
```

creates a `Frame` instance with lower left hand corner at coordinate `(x1,y1)` (relative to the canvas at use time) and with dimensions `width` x `height`. The `Padding` arguments are positive quantities used to reduce the space available for drawing. The `id` argument is an identifier for use at runtime e.g. 'LeftColumn' or 'RightColumn' etc. If the `showBoundary` argument is non-zero then the boundary of the frame will get drawn at run time (this is useful sometimes).

`Frame` User Methods

```
Frame.addFromList(drawlist, canvas)
```

consumes `Flowables` from the front of `drawlist` until the frame is full. If it cannot fit one object, raises an exception.

```
Frame.split(flowable,canv)
```

Asks the flowable to split using up the available space and return the list of flowables.

```
Frame.drawBoundary(canvas)
```

draws the frame boundary as a rectangle (primarily for debugging).

Using `Frames`

`Frames` can be used directly with canvases and flowables to create documents. The `Frame.addFromList` method handles the `wrap` & `drawOn` calls for you. You don't need all of the Platypus machinery to get something useful into PDF.

```
from reportlab.pdfgen.canvas import Canvas
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.lib.units import inch
from reportlab.platypus import Paragraph, Frame
styles = getSampleStyleSheet()
styleN = styles['Normal']
styleH = styles['Heading1']
story = []

#add some flowables
story.append(Paragraph("This is a Heading",styleH))
story.append(Paragraph("This is a paragraph in <i>Normal</i> style.",
    styleN))
c  = Canvas('mydoc.pdf')
f = Frame(inch, inch, 6*inch, 9*inch, showBoundary=1)
f.addFromList(story,c)
c.save()
```

5.6 Documents and Templates

The `BaseDocTemplate` class implements the basic machinery for document formatting. An instance of the class contains a list of one or more `PageTemplates` that can be used to describe the layout of information on a single page. The `build` method can be used to process a list of `Flowables` to produce a **PDF** document.

The `BaseDocTemplate` class

```
BaseDocTemplate(self, filename,
                pagesize=defaultPageSize,
                pageTemplates=[],
                showBoundary=0,
                leftMargin=inch,
                rightMargin=inch,
                topMargin=inch,
                bottomMargin=inch,
                allowSplitting=1,
                title=None,
                author=None,
                _pageBreakQuick=1,
                encrypt=None)
```

creates a document template suitable for creating a basic document. It comes with quite a lot of internal machinery, but no default page templates. The required `filename` can be a string, the name of a file to receive the created **PDF** document; alternatively it can be an object which has a `write` method such as a `BytesIO` or `file` or `socket`.

The allowed arguments should be self explanatory, but `showBoundary` controls whether or not `Frame` boundaries are drawn which can be useful for debugging purposes. The `allowSplitting` argument determines whether the builtin methods should try to *split* individual `Flowables` across `Frames`. The `_pageBreakQuick` argument determines whether an attempt to do a page break should try to end all the frames on the page or not, before ending the page. The encrypt argument determines wether or not and how the document is encrypted. By default, the document is not encrypted. If `encrypt` is a string object, it is used as the user password for the pdf. If `encrypt` is an instance of `reportlab.lib.pdfencrypt.StandardEncryption`, this object is used to encrypt the pdf. This allows more finegrained control over the encryption settings.

User `BaseDocTemplate` Methods

These are of direct interest to client programmers in that they are normally expected to be used.

```
BaseDocTemplate.addPageTemplates(self,pageTemplates)
```

This method is used to add one or a list of `PageTemplates` to an existing documents.

```
BaseDocTemplate.build(self, flowables, filename=None, canvasmaker=canvas.Canvas)
```

This is the main method which is of interest to the application programmer. Assuming that the document instance is correctly set up the `build` method takes the *story* in the shape of the list of flowables (the `flowables` argument) and loops through the list forcing the flowables one at a time through the formatting machinery. Effectively this causes the `BaseDocTemplate` instance to issue calls to the instance `handle_XXX` methods to process the various events.

User Virtual `BaseDocTemplate` Methods

These have no semantics at all in the base class. They are intended as pure virtual hooks into the layout machinery. Creators of immediately derived classes can override these without worrying about affecting the properties of the layout engine.

```
BaseDocTemplate.afterInit(self)
```

This is called after initialisation of the base class; a derived class could overide the method to add default `PageTemplates`.

BaseDocTemplate.afterPage(self)

This is called after page processing, and immediately after the afterDrawPage method of the current page template. A derived class could use this to do things which are dependent on information in the page such as the first and last word on the page of a dictionary.

BaseDocTemplate.beforeDocument(self)

This is called before any processing is done on the document, but after the processing machinery is ready. It can therefore be used to do things to the instance's `pdfgen.canvas` and the like.

BaseDocTemplate.beforePage(self)

This is called at the beginning of page processing, and immediately before the beforeDrawPage method of the current page template. It could be used to reset page specific information holders.

BaseDocTemplate.filterFlowables(self,flowables)

This is called to filter flowables at the start of the main handle_flowable method. Upon return if flowables[0] has been set to None it is discarded and the main method returns immediately.

BaseDocTemplate.afterFlowable(self, flowable)

Called after a flowable has been rendered. An interested class could use this hook to gather information about what information is present on a particular page or frame.

`BaseDocTemplate` Event handler Methods

These methods constitute the greater part of the layout engine. Programmers shouldn't have to call or override these methods directly unless they are trying to modify the layout engine. Of course, the experienced programmer who wants to intervene at a particular event, XXX, which does not correspond to one of the virtual methods can always override and call the base method from the drived class version. We make this easy by providing a base class synonym for each of the handler methods with the same name prefixed by an underscore '_'.

```
def handle_pageBegin(self):
    doStuff()
    BaseDocTemplate.handle_pageBegin(self)
    doMoreStuff()

#using the synonym
def handle_pageEnd(self):
    doStuff()
    self._handle_pageEnd()
    doMoreStuff()
```

Here we list the methods only as an indication of the events that are being handled. Interested programmers can take a look at the source.

```
handle_currentFrame(self,fx)
handle_documentBegin(self)
handle_flowable(self,flowables)
handle_frameBegin(self,*args)
handle_frameEnd(self)
handle_nextFrame(self,fx)
handle_nextPageTemplate(self,pt)
handle_pageBegin(self)
handle_pageBreak(self)
handle_pageEnd(self)
```

Using document templates can be very easy; `SimpleDoctemplate` is a class derived from `BaseDocTemplate` which provides its own `PageTemplate` and `Frame` setup.

```
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.lib.pagesizes import letter
from reportlab.platypus import Paragraph, SimpleDocTemplate
styles = getSampleStyleSheet()
```

```
styleN = styles['Normal']
styleH = styles['Heading1']
story = []

#add some flowables
story.append(Paragraph("This is a Heading",styleH))
story.append(Paragraph("This is a paragraph in <i>Normal</i> style.",
    styleN))
doc = SimpleDocTemplate('mydoc.pdf',pagesize = letter)
doc.build(story)
```

PageTemplates

The `PageTemplate` class is a container class with fairly minimal semantics. Each instance contains a list of `Frames` and has methods which should be called at the start and end of each page.

```
PageTemplate(id=None,frames=[],onPage=_doNothing,onPageEnd=_doNothing)
```

is used to initialize an instance, the `frames` argument should be a list of `Frames` whilst the optional `onPage` and `onPageEnd` arguments are callables which should have signature `def XXX(canvas,document)` where `canvas` and `document` are the canvas and document being drawn. These routines are intended to be used to paint non-flowing (i.e. standard) parts of pages. These attribute functions are exactly parallel to the pure virtual methods `PageTemplate.beforPage` and `PageTemplate.afterPage` which have signature `beforPage(self,canvas,document)`. The methods allow class derivation to be used to define standard behaviour, whilst the attributes allow instance changes. The `id` argument is used at run time to perform `PageTemplate` switching so `id='FirstPage'` or `id='TwoColumns'` are typical.

Chapter 6 Paragraphs

The `reportlab.platypus.Paragraph` class is one of the most useful of the Platypus `Flowables`; it can format fairly arbitrary text and provides for inline font style and colour changes using an XML style markup. The overall shape of the formatted text can be justified, right or left ragged or centered. The XML markup can even be used to insert greek characters or to do subscripts.

The following text creates an instance of the `Paragraph` class:

```
Paragraph(text, style, bulletText=None)
```

The `text` argument contains the text of the paragraph; excess white space is removed from the text at the ends and internally after linefeeds. This allows easy use of indented triple quoted text in **Python** scripts. The `bulletText` argument provides the text of a default bullet for the paragraph. The font and other properties for the paragraph text and bullet are set using the style argument.

The `style` argument should be an instance of class `ParagraphStyle` obtained typically using

```
from reportlab.lib.styles import ParagraphStyle
```

this container class provides for the setting of multiple default paragraph attributes in a structured way. The styles are arranged in a dictionary style object called a `stylesheet` which allows for the styles to be accessed as `stylesheet['BodyText']`. A sample style sheet is provided.

```
from reportlab.lib.styles import getSampleStyleSheet
stylesheet=getSampleStyleSheet()
normalStyle = stylesheet['Normal']
```

The options which can be set for a `Paragraph` can be seen from the `ParagraphStyle` defaults.

class ParagraphStyle

```
class ParagraphStyle(PropertySet):
    defaults = {
        'fontName':'Times-Roman',
        'fontSize':10,
        'leading':12,
        'leftIndent':0,
        'rightIndent':0,
        'firstLineIndent':0,
        'alignment':TA_LEFT,
        'spaceBefore':0,
        'spaceAfter':0,
        'bulletFontName':'Times-Roman',
        'bulletFontSize':10,
        'bulletIndent':0,
        'textColor': black,
        'backColor':None,
        'wordWrap':None,
        'borderWidth': 0,
        'borderPadding': 0,
        'borderColor': None,
        'borderRadius': None,
        'allowWidows': 1,
        'allowOrphans': 0,
        }
```

6.1 Using Paragraph Styles

The `Paragraph` and `ParagraphStyle` classes together handle most common formatting needs. The following examples draw paragraphs in various styles, and add a bounding box so that you can see exactly what space is taken up.

```
spaceBefore = 0
fontName = Helvetica
bulletFontName = Helvetica
borderRadius = None
firstLineIndent = 0
leftIndent = 0
rightIndent = 0
wordWrap = None
allowWidows = 1
backColor = None
textTransform = None
alignment = 0
borderColor = None
splitLongWords = 1
leading = 12
bulletIndent = 0
allowOrphans = 0
bulletFontSize = 10
fontSize = 10
borderWidth = 0
borderPadding = 0
endDots = None
textColor = Color(0,0,0,1)
spaceAfter = 0
```

You are hereby charged that on the 28th day of May, 1970, you did willfully, unlawfully, and with malice of forethought, publish an alleged English-Hungarian phrase book with intent to cause a breach of the peace. How do you plead?

Figure 6-1: The default `ParagraphStyle`

The two attributes `spaceBefore` and `spaceAfter` do what they say, except at the top or bottom of a frame. At the top of a frame, `spaceBefore` is ignored, and at the bottom, `spaceAfter` is ignored. This means that you could specify that a 'Heading2' style had two inches of space before when it occurs in mid-page, but will not get acres of whitespace at the top of a page. These two attributes should be thought of as 'requests' to the Frame and are not part of the space occupied by the Paragraph itself.

The `fontSize` and `fontName` tags are obvious, but it is important to set the `leading`. This is the spacing between adjacent lines of text; a good rule of thumb is to make this 20% larger than the point size. To get double-spaced text, use a high `leading`. If you set `autoLeading`(default `"off"`) to `"min"`(use observed leading even if smaller than specified) or `"max"`(use the larger of observed and specified) then an attempt is made to determine the leading on a line by line basis. This may be useful if the lines contain different font sizes etc.

The figure below shows space before and after and an increased leading:

```
spaceBefore = 6
fontName = Helvetica
bulletFontName = Helvetica
borderRadius = None
firstLineIndent = 0
leftIndent = 0
rightIndent = 0
wordWrap = None
allowWidows = 1
backColor = None
textTransform = None
alignment = 0
borderColor = None
splitLongWords = 1
leading = 16
bulletIndent = 0
allowOrphans = 0
bulletFontSize = 10
fontSize = 10
borderWidth = 0
borderPadding = 0
endDots = None
textColor = Color(0,0,0,1)
spaceAfter = 6
```

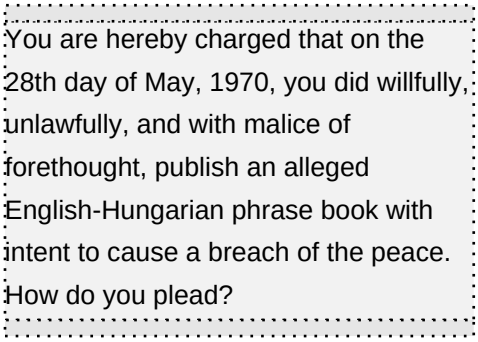


Figure 6-2: Space before and after and increased leading

The attribute `borderPadding` adjusts the padding between the paragraph and the border of its background. This can either be a single value or a tuple containing 2 to 4 values. These values are applied the same way as in Cascading Style Sheets (CSS). If a single value is given, that value is applied to all four sides. If more than one value is given, they are applied in clockwise order to the sides starting at the top. If two or three values are given, the missing values are taken from the opposite side(s). Note that in the following example the yellow box is drawn by the paragraph itself.

```
spaceBefore = 0
fontName = Helvetica
bulletFontName = Helvetica
borderRadius = None
firstLineIndent = 0
leftIndent = 0
rightIndent = 0
wordWrap = None
allowWidows = 1
backColor = #FFFF00
textTransform = None
alignment = 0
borderColor = #000000
splitLongWords = 1
leading = 12
bulletIndent = 0
allowOrphans = 0
bulletFontSize = 10
fontSize = 10
borderWidth = 1
borderPadding = (7, 2, 20)
endDots = None
textColor = Color(0,0,0,1)
spaceAfter = 0
```

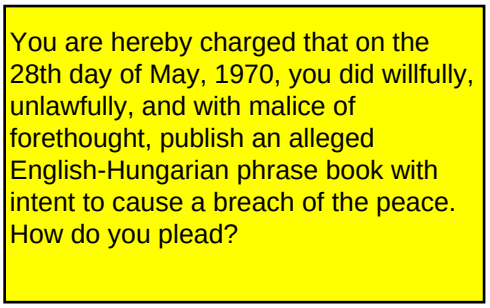


Figure 6-3: Variable padding

The `leftIndent` and `rightIndent` attributes do exactly what you would expect; `firstLineIndent` is added to the `leftIndent` of the first line. If you want a straight left edge,

remember to set `firstLineIndent` equal to 0.

```
spaceBefore = 0
fontName = Helvetica
bulletFontName = Helvetica
borderRadius = None
firstLineIndent = 24
leftIndent = 24
rightIndent = 24
wordWrap = None
allowWidows = 1
backColor = None
textTransform = None
alignment = 0
borderColor = None
splitLongWords = 1
leading = 12
bulletIndent = 0
allowOrphans = 0
bulletFontSize = 10
fontSize = 10
borderWidth = 0
borderPadding = 0
endDots = None
textColor = Color(0,0,0,1)
spaceAfter = 0
```

You are hereby charged that on the 28th day of May, 1970, you did willfully, unlawfully, and with malice of forethought, publish an alleged English-Hungarian phrase book with intent to cause a breach of the peace. How do you plead?

Figure 6-4: one third inch indents at left and right, two thirds on first line

Setting `firstLineIndent` equal to a negative number, `leftIndent` much higher, and using a different font (we'll show you how later!) can give you a definition list:.

```
spaceBefore = 0
fontName = Helvetica
bulletFontName = Helvetica
borderRadius = None
firstLineIndent = 0
leftIndent = 36
rightIndent = 0
wordWrap = None
allowWidows = 1
backColor = None
textTransform = None
alignment = 0
borderColor = None
splitLongWords = 1
leading = 12
bulletIndent = 0
allowOrphans = 0
bulletFontSize = 10
fontSize = 10
borderWidth = 0
borderPadding = 0
endDots = None
textColor = Color(0,0,0,1)
spaceAfter = 0
```

Judge Pickles: You are hereby charged that on the 28th day of May, 1970, you did willfully, unlawfully, and with malice of forethought, publish an alleged English-Hungarian phrase book with intent to cause a breach of the peace. How do you plead?

Figure 6-5: Definition Lists

There are four possible values of `alignment`, defined as constants in the module *reportlab.lib.enums*. These are TA_LEFT, TA_CENTER or TA_CENTRE, TA_RIGHT and TA_JUSTIFY, with values of 0, 1, 2 and 4 respectively. These do exactly what you would expect.

Set `wordWrap` to `'CJK'` to get Asian language linewrapping. For normal western text you can change the way the line breaking algorithm handles *widows* and *orphans* with the `allowWidows` and `allowOrphans` values. Both should normally be set to `0`, but for historical reasons we have allowed *widows*. The default color of the text can be set with `textColor` and the paragraph background colour can be set with `backColor`. The paragraph's border properties may be changed using `borderWidth`, `borderPadding`, `borderColor` and `borderRadius`.

6.2 Paragraph XML Markup Tags

XML markup can be used to modify or specify the overall paragraph style, and also to specify intra-paragraph markup.

The outermost < para > tag

The paragraph text may optionally be surrounded by <para attributes....> </para> tags. The attributes if any of the opening <para> tag affect the style that is used with the `Paragraph` `text` and/or `bulletText`.

Attribute	Synonyms
alignment	alignment, align
autoLeading	autoLeading, autoleading
backColor	backColor, bgcolor, backcolor, bg
bulletColor	bulletColor, bcolor, bulletcolor
bulletFontName	bulletFontName, bulletfontname, bfont
bulletFontSize	bulletfontsize, bulletFontSize, bfontsize
bulletIndent	bindent, bulletindent, bulletIndent
bulletOffsetY	boffsety, bulletoffsety, bulletOffsetY
firstLineIndent	findent, firstlineindent, firstLineIndent
fontName	fontName, font, fontname, face
fontSize	fontSize, size, fontsize
leading	leading
leftIndent	leftindent, leftIndent, lindent
rightIndent	rightIndent, rightindent, rindent
spaceAfter	spacea, spaceafter, spaceAfter
spaceBefore	spaceb, spaceBefore, spacebefore
textColor	color, textcolor, fg, textColor

Table 6-2 - Synonyms for style attributes

Some useful synonyms have been provided for our Python attribute names, including lowercase versions, and the equivalent properties from the HTML standard where they exist. These additions make it much easier to build XML-printing applications, since much intra-paragraph markup may not need translating. The table below shows the allowed attributes and synonyms in the outermost paragraph tag.

6.3 Intra-paragraph markup

'

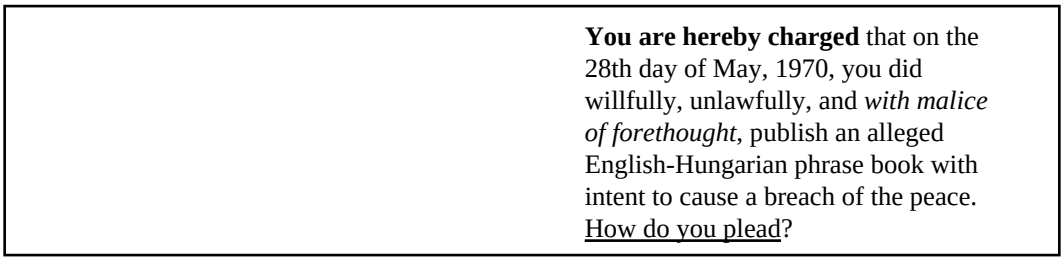


Figure 6-6: Simple bold and italic tags

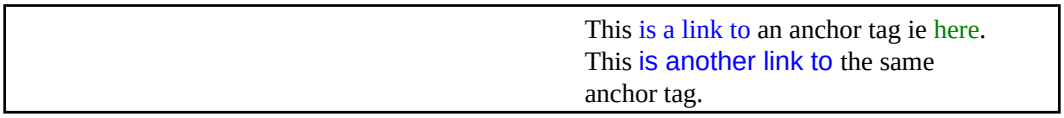


Figure 6-7: anchors and links

The **link** tag can be used as a reference, but not as an anchor. The a and link hyperlink tags have additional attributes *fontName*, *fontSize*, *color* & *backColor* attributes. The hyperlink reference can have a scheme of **http:***(external webpage)*, **pdf:***(different pdf document)* or **document:***(same pdf document)*; a missing scheme is treated as **document** as is the case when the reference starts with # (in which case the anchor should omit it). Any other scheme is treated as some kind of URI.

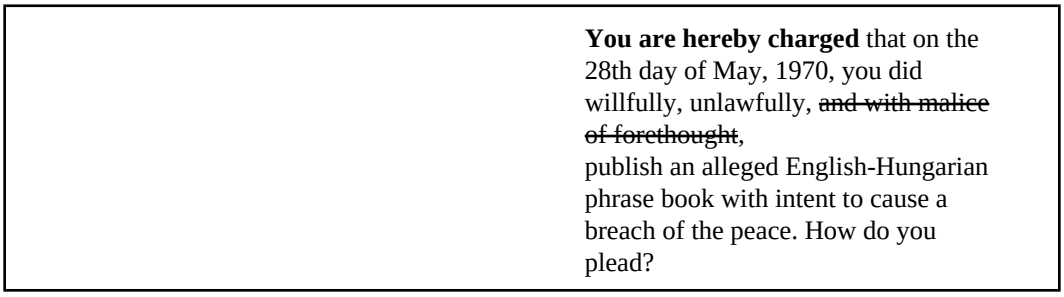


Figure 6-8: Strong, strike, and break tags

The `<font>` tag

The `<font>` tag can be used to change the font name, size and text color for any substring within the paragraph. Legal attributes are `size`, `face`, `name` (which is the same as `face`), `color`, and `fg` (which is the same as `color`). The `name` is the font family name, without any 'bold' or 'italic' suffixes. Colors may be HTML color names or a hex string encoded in a variety of ways; see *`reportlab.lib.colors`* for the formats allowed.

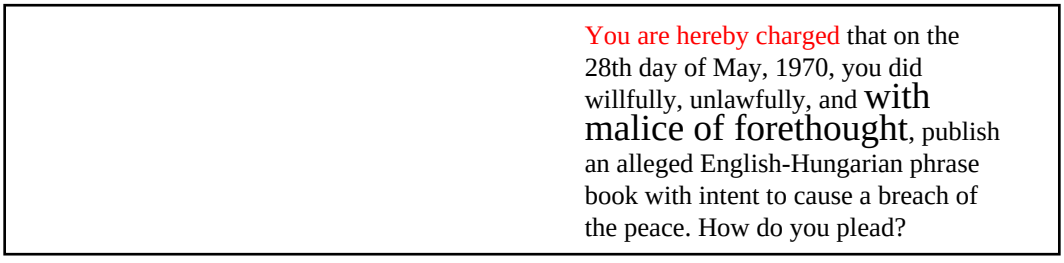


Figure 6-9: The `font` tag

Superscripts and Subscripts

Superscripts and subscripts are supported with the

Equation (α): ε $^{\iota\pi}$ = -1

Figure 6-10: Greek letters and superscripts

Inline Images

We can embed images in a paragraph with the <img/> tag which has attributes `src`, `width`, `height` whose meanings are obvious. The `valign` attribute may be set to a css like value from "baseline", "sub", "super", "top", "text-top", "middle", "bottom", "text-bottom"; the value may also be a numeric percentage or an absolute value.

This <img/> text is aligned **top**.

This <img/> text is aligned **bottom**.

This <img/> text is aligned **middle**.

This <img/> text is aligned **-4**.

This <img/> text is aligned **+4**.

This <img/> text has width **10**.

Figure 6-11: Inline images

Numbering Paragraphs and Lists

The `<seq>` tag provides comprehensive support for numbering lists, chapter headings and so on. It acts as an interface to the `Sequencer` class in *`reportlab.lib.sequencer`*. These are used to number headings and figures throughout this document. You may create as many separate 'counters' as you wish, accessed with the `id` attribute; these will be incremented by one each time they are accessed. The `seqreset` tag resets a counter. If you want it to resume from a number other than 1, use the syntax <seqreset id="mycounter" base="42">. Let's have a go:

1, 2, 3. Reset. 1, 2, 3.

Figure 6-12: Basic sequences

You can save specifying an ID by designating a counter ID as the *default* using the <seqdefault id="Counter"> tag; it will then be used whenever a counter ID is not specified. This saves some typing, especially when doing multi-level lists; you just change counter ID when stepping in or out a level.

Continued... 4, 5, 6, 7, 8, 9, 10.

Figure 6-13: The default sequence

Finally, one can access multi-level sequences using a variation of Python string formatting and the `template` attribute in a <seq> tags. This is used to do the captions in all of the figures, as well as the level two headings. The substring `%(counter)s` extracts the current value of a counter without incrementing it; appending a plus sign as in `%(counter)s` increments the counter. The figure captions use a pattern like the one below:

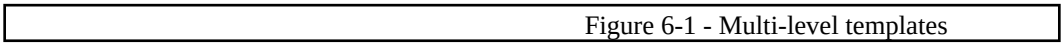


Figure 6-14: Multi-level templates

We cheated a little - the real document used 'Figure', but the text above uses 'FigureNo' - otherwise we would have messed up our numbering!

6.4 Bullets and Paragraph Numbering

In addition to the three indent properties, some other parameters are needed to correctly handle bulleted and numbered lists. We discuss this here because you have now seen how to handle numbering. A paragraph may have an optional *`bulletText`* argument passed to its constructor; alternatively, bullet text may be placed in a tag at its head. This text will be drawn on the first line of the paragraph, with its x origin determined by the `bulletIndent` attribute of the style, and in the font given in the `bulletFontName` attribute. The "bullet" may be a single character such as (doh!) a bullet, or a fragment of text such as a number in some numbering sequence, or even a short title as used in a definition list. Fonts may offer various bullet characters but we suggest first trying the Unicode bullet (•), which may be written as `•`, `•` or (in utf8) `\xe2\x80\xa2`):

Attribute	Synonyms
bulletColor	bulletColor, color, fg, bulletcolor
bulletFontName	bulletFontName, bulletfontname, font, face
bulletFontSize	fontsize, size, bulletfontsize, bulletFontSize
bulletIndent	bulletindent, indent, bulletIndent
bulletOffsetY	offsety, bulletOffsetY, bulletoffsety

Table 6-3 - <bullet> attributes & synonyms

The <bullet> tag is only allowed once in a given paragraph and its use overrides the implied bullet style and *`bulletText`* specified in the *`Paragraph`* creation.

```
spaceBefore = 0
fontName = Times-Roman
bulletFontName = Symbol
borderRadius = None
firstLineIndent = 0
leftIndent = 54
rightIndent = 0
wordWrap = None
allowWidows = 1
backColor = None
textTransform = None
alignment = 0
borderColor = None
splitLongWords = 1
leading = 12
bulletIndent = 18
allowOrphans = 0
bulletFontSize = 10
fontSize = 10
borderWidth = 0
borderPadding = 0
endDots = None
textColor = Color(0,0,0,1)
spaceAfter = 0
```

■■2022 this is a bullet point. Spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam spam

Figure 6-15: Basic use of bullet points

Exactly the same technique is used for numbers, except that a sequence tag is used. It is also possible to put a multi-character string in the bullet; with a deep indent and bold bullet font, you can make a compact definition list.

Chapter 7 Tables and TableStyles

The `Table` and `LongTable` classes derive from the `Flowable` class and are intended as a simple textual gridding mechanisms. The `longTable` class uses a greedy algorithm when calculating column widths and is intended for long tables where speed counts. `Table` cells can hold anything which can be converted to a **Python** `string` or `Flowables` (or lists of `Flowables`).

Our present tables are a trade-off between efficient drawing and specification and functionality. We assume the reader has some familiarity with HTML tables. In brief, they have the following characteristics:

- ■■2022 They can contain anything convertible to a string; flowable objects such as other tables; or entire sub-stories
- ■■2022 They can work out the row heights to fit the data if you don't supply the row height. (They can also work out the widths, but generally it is better for a designer to set the width manually, and it draws faster).
- ■■2022 They can split across pages if needed (see the canSplit attribute). You can specify that a number of rows at the top and bottom should be repeated after the split (e.g. show the headers again on page 2,3,4...)
- ■■2022 They have a simple and powerful notation for specifying shading and gridlines which works well with financial or database tables, where you don't know the number of rows up front. You can easily say 'make the last row bold and put a line above it'
- ■■2022 The style and data are separated, so you can declare a handful of table styles and use them for a family of reports. Styes can also 'inherit', as with paragraphs.

There is however one main limitation compared to an HTML table. They define a simple rectangular grid. There is no simple row or column spanning; if you need to span cells, you must nest tables inside table cells instead or use a more complex scheme in which the lead cell of a span contains the actual contents.

`Tables` are created by passing the constructor an optional sequence of column widths, an optional sequence of row heights, and the data in row order. Drawing of the table can be controlled by using a `TableStyle` instance. This allows control of the color and weight of the lines (if any), and the font, alignment and padding of the text. A primitive automatic row height and or column width calculation mechanism is provided for.

7.1 `Table` User Methods

These are the main methods which are of interest to the client programmer.

`Table(data, colWidths=None, rowHeights=None, style=None, splitByRow=1, repeatRows=0, repeatCols=0)`

The `data` argument is a sequence of sequences of cell values each of which should be convertible to a string value using the `str` function or should be a Flowable instance (such as a `Paragraph`) or a list (or tuple) of such instances. If a cell value is a `Flowable` or list of `Flowables` these must either have a determined width or the containing column must have a fixed width. The first row of cell values is in `data[0]` i.e. the values are in row order. The `i`, `j`th cell value is in `data[i][j]`. Newline characters `'\n'` in cell values are treated as line split characters and are used at *draw* time to format the cell into lines.

The other arguments are fairly obvious, the `colWidths` argument is a sequence of numbers or possibly `None`, representing the widths of the columns. The number of elements in `colWidths` determines the number of columns in the table. A value of `None` means that the corresponding column width should be calculated automatically.

The `rowHeights` argument is a sequence of numbers or possibly `None`, representing the heights of the rows. The number of elements in `rowHeights` determines the number of rows in the table. A value of `None` means that the corresponding row height should be calculated automatically.

The `style` argument can be an initial style for the table.

The `splitByRow` argument is only needed for tables both too tall and too wide to fit in the current context. In this case you must decide whether to 'tile' down and across, or across and then down. This parameter is a Boolean indicating that the `Table` should split itself by row before attempting to split itself by column when too little space is available in the current drawing area and the caller wants the `Table` to split.

Splitting a `Table` by column is currently not implemented, so setting `splitByRow` to `False` will result in a `NotImplementedError`.

The `repeatRows` argument specifies the number of leading rows that should be repeated when the `Table` is asked to split itself. The `repeatCols` argument is currently ignored as a `Table` cannot be split by column.

`Table.setStyle(tblStyle)`

This method applies a particular instance of class `TableStyle` (discussed below) to the `Table` instance. This is the only way to get `tables` to appear in a nicely formatted way.

Successive uses of the `setStyle` method apply the styles in an additive fashion. That is, later applications override earlier ones where they overlap.

7.2 `TableStyle`

This class is created by passing it a sequence of *commands*, each command is a tuple identified by its first element which is a string; the remaining elements of the command tuple represent the start and stop cell coordinates of the command and possibly thickness and colors, etc.

7.3 `TableStyle` User Methods

`TableStyle(commandSequence)`

The creation method initializes the `TableStyle` with the argument command sequence as an example:

```
LIST_STYLE = TableStyle(
    [('LINEABOVE', (0,0), (-1,0), 2, colors.green),
    ('LINEABOVE', (0,1), (-1,-1), 0.25, colors.black),
    ('LINEBELOW', (0,-1), (-1,-1), 2, colors.green),
    ('ALIGN', (1,1), (-1,-1), 'RIGHT')]
    )
```

`TableStyle.add(commandSequence)`

This method allows you to add commands to an existing `TableStyle`, i.e. you can build up `TableStyles` in multiple statements.

```
LIST_STYLE.add('BACKGROUND', (0,0), (-1,0), colors.Color(0,0.7,0.7))
```

`TableStyle.getCommands()`

This method returns the sequence of commands of the instance.

```
cmds = LIST_STYLE.getCommands()
```

7.4 `TableStyle` Commands

The commands passed to `TableStyles` come in three main groups which affect the table background, draw lines, or set cell styles.

The first element of each command is its identifier, the second and third arguments determine the cell coordinates of the box of cells which are affected with negative coordinates counting backwards from the limit values as in **Python** indexing. The coordinates are given as (column, row) which follows the spreadsheet 'A1' model, but not the more natural (for mathematicians) 'RC' ordering. The top left cell is (0, 0) the bottom right is (-1, -1). Depending on the command various extra (???) occur at indices beginning at 3 on.

TableStyle Cell Formatting Commands

The cell formatting commands all begin with an identifier, followed by the start and stop cell definitions and the perhaps other arguments. the cell formatting commands are:

```
FONT                    - takes fontname, optional fontsize and optional leading.
FONTNAME (or FACE)      - takes fontname.
FONTSIZE (or SIZE)      - takes fontsize in points; leading may get out of sync.
LEADING                 - takes leading in points.
TEXTCOLOR               - takes a color name or (R,G,B) tuple.
ALIGNMENT (or ALIGN)    - takes one of LEFT, RIGHT and CENTRE (or CENTER) or DECIMAL.
LEFTPADDING             - takes an integer, defaults to 6.
RIGHTPADDING            - takes an integer, defaults to 6.
BOTTOMPADDING           - takes an integer, defaults to 3.
TOPPADDING              - takes an integer, defaults to 3.
BACKGROUND              - takes a color.
ROWBACKGROUNDS          - takes a list of colors to be used cyclically.
COLBACKGROUNDS          - takes a list of colors to be used cyclically.
VALIGN                  - takes one of TOP, MIDDLE or the default BOTTOM
```

This sets the background cell color in the relevant cells. The following example shows the BACKGROUND, and TEXTCOLOR commands in action:

```
data=  [['00', '01', '02', '03', '04'],
        ['10', '11', '12', '13', '14'],
        ['20', '21', '22', '23', '24'],
        ['30', '31', '32', '33', '34']]
t=Table(data)
t.setStyle(TableStyle([('BACKGROUND',(1,1),(-2,-2),colors.green),
                       ('TEXTCOLOR',(0,0),(1,-1),colors.red)]))
```

produces

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24
30	31	32	33	34

To see the effects of the alignment styles we need some widths and a grid, but it should be easy to see where the styles come from.

```
data=  [['00', '01', '02', '03', '04'],
        ['10', '11', '12', '13', '14'],
        ['20', '21', '22', '23', '24'],
        ['30', '31', '32', '33', '34']]
t=Table(data,5*[0.4*inch], 4*[0.4*inch])
t.setStyle(TableStyle([('ALIGN',(1,1),(-2,-2),'RIGHT'),
                       ('TEXTCOLOR',(1,1),(-2,-2),colors.red),
                       ('VALIGN',(0,0),(0,-1),'TOP'),
                       ('TEXTCOLOR',(0,0),(0,-1),colors.blue),
                       ('ALIGN',(0,-1),(-1,-1),'CENTER'),
                       ('VALIGN',(0,-1),(-1,-1),'MIDDLE'),
                       ('TEXTCOLOR',(0,-1),(-1,-1),colors.green),
                       ('INNERGRID', (0,0), (-1,-1), 0.25, colors.black),
                       ('BOX', (0,0), (-1,-1), 0.25, colors.black),
                       ]))
```

produces

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24
30	31	32	33	34

`TableStyle` *Line Commands*

Line commands begin with the identifier, the start and stop cell coordinates and always follow this with the thickness (in points) and color of the desired lines. Colors can be names, or they can be specified as a (R, G, B) tuple, where R, G and B are floats and (0, 0, 0) is black. The line command names are: GRID, BOX, OUTLINE, INNERGRID, LINEBELOW, LINEABOVE, LINEBEFORE and LINEAFTER. BOX and OUTLINE are equivalent, and GRID is the equivalent of applying both BOX and INNERGRID.

We can see some line commands in action with the following example.

```
data=  [['00', '01', '02', '03', '04'],
        ['10', '11', '12', '13', '14'],
        ['20', '21', '22', '23', '24'],
        ['30', '31', '32', '33', '34']]
t=Table(data,style=[('GRID',(1,1),(-2,-2),1,colors.green),
                    ('BOX',(0,0),(1,-1),2,colors.red),
                    ('LINEABOVE',(1,2),(-2,2),1,colors.blue),
                    ('LINEBEFORE',(2,1),(2,-2),1,colors.pink),
                    ])
```

produces

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24
30	31	32	33	34

Line commands cause problems for tables when they split; the following example shows a table being split in various positions

```
data=  [['00', '01', '02', '03', '04'],
        ['10', '11', '12', '13', '14'],
        ['20', '21', '22', '23', '24'],
        ['30', '31', '32', '33', '34']]
t=Table(data,style=[
                    ('GRID',(0,0),(-1,-1),0.5,colors.grey),
                    ('GRID',(1,1),(-2,-2),1,colors.green),
                    ('BOX',(0,0),(1,-1),2,colors.red),
                    ('BOX',(0,0),(-1,-1),2,colors.black),
                    ('LINEABOVE',(1,2),(-2,2),1,colors.blue),
                    ('LINEBEFORE',(2,1),(2,-2),1,colors.pink),
                    ('BACKGROUND', (0, 0), (0, 1), colors.pink),
                    ('BACKGROUND', (1, 1), (1, 2), colors.lavender),
                    ('BACKGROUND', (2, 2), (2, 3), colors.orange),
                    ])
```

produces

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24
30	31	32	33	34

00	01	02	03	04

10	11	12	13	14
20	21	22	23	24
30	31	32	33	34

00	01	02	03	04
10	11	12	13	14

20	21	22	23	24
30	31	32	33	34

When unsplit and split at the first or second row.

Complex Cell Values

As mentioned above we can have complicated cell values including `Paragraphs`, `Images` and other `Flowables` or lists of the same. To see this in operation consider the following code and the table it produces. Note that the `Image` has a white background which will obscure any background you choose for the cell. To get better results you should use a transparent background.

```
I = Image('../images/replogo.gif')
I.drawHeight = 1.25*inch*I.drawHeight / I.drawWidth
I.drawWidth = 1.25*inch
P0 = Paragraph('''
               <b>A pa<font color=red>r</font>a<i>graph</i></b>
               <super><font color=yellow>1</font></super>''',
               styleSheet["BodyText"])
P = Paragraph('''
       <para align=center spaceb=3>The <b>ReportLab Left
       <font color=red>Logo</font></b>
       Image</para>''',
       styleSheet["BodyText"])
data=  [['A',   'B', 'C',     P0, 'D'],
        ['00', '01', '02', [I,P], '04'],
        ['10', '11', '12', [P,I], '14'],
        ['20', '21', '22',  '23', '24'],
        ['30', '31', '32',  '33', '34']]

t=Table(data,style=[('GRID',(1,1),(-2,-2),1,colors.green),
                    ('BOX',(0,0),(1,-1),2,colors.red),
                    ('LINEABOVE',(1,2),(-2,2),1,colors.blue),
                    ('LINEBEFORE',(2,1),(2,-2),1,colors.pink),
                    ('BACKGROUND', (0, 0), (0, 1), colors.pink),
                    ('BACKGROUND', (1, 1), (1, 2), colors.lavender),
                    ('BACKGROUND', (2, 2), (2, 3), colors.orange),
                    ('BOX',(0,0),(-1,-1),2,colors.black),
                    ('GRID',(0,0),(-1,-1),0.5,colors.black),
                    ('VALIGN',(3,0),(3,0),'BOTTOM'),
                    ('BACKGROUND',(3,0),(3,0),colors.limegreen),
                    ('BACKGROUND',(3,1),(3,1),colors.khaki),
                    ('ALIGN',(3,1),(3,1),'CENTER'),
                    ('BACKGROUND',(3,2),(3,2),colors.beige),
                    ('ALIGN',(3,2),(3,2),'LEFT'),
                    ])

t._argW[3]=1.5*inch
```

produces

A	B	C	A paragraph [1]	D
00	01	02	The **ReportLab Left Logo** Image	04
10	11	12	The **ReportLab Left Logo** Image	14

20	21	22	23	24
30	31	32	33	34

TableStyle Span Commands

Our `Table` classes support the concept of spanning, but it isn't specified in the same way as html. The style specification

```
SPAN, (sc,sr), (ec,er)
```

indicates that the cells in columns `sc -ec` and rows `sr -er` should be combined into a super cell with contents determined by the cell `(sc, sr)`. The other cells should be present, but should contain empty strings or you may get unexpected results.

```
data=  [['Top\nLeft', '', '02', '03', '04'],
        ['', '', '12', '13', '14'],
        ['20', '21', '22', 'Bottom\nRight', ''],
        ['30', '31', '32', '', '']]
t=Table(data,style=[
                ('GRID',(0,0),(-1,-1),0.5,colors.grey),
                ('BACKGROUND',(0,0),(1,1),colors.palegreen),
                ('SPAN',(0,0),(1,1)),
                ('BACKGROUND',(-2,-2),(-1,-1), colors.pink),
                ('SPAN',(-2,-2),(-1,-1)),
                ])
```

produces

Top Left		02	03	04
		12	13	14
20	21	22	Bottom Right	
30	31	32		

notice that we don't need to be conservative with our `GRID` command. The spanned cells are not drawn through.

Special TableStyle Indeces

In any style command the first row index may be set to one of the special strings `'splitlast'` or `'splitfirst'` to indicate that the style should be used only for the last row of a split table, or the first row of a continuation. This allows splitting tables with nicer effects around the split.

Chapter 8 Programming `Flowables`

The following flowables let you conditionally evaluate and execute expressions and statements at wrap time:

8.1 `DocAssign(self, var, expr, life='forever')`

Assigns a variable of name `var` to the expression `expr`. E.g.:

```
DocAssign('i',3)
```

8.2 `DocExec(self, stmt, lifetime='forever')`

Executes the statement `stmt`. E.g.:

```
DocExec('i-=1')
```

8.3 `DocPara(self, expr, format=None, style=None, klass=None, escape=True)`

Creates a paragraph with the value of expr as text. If format is specified it should use %(__expr__)s for string interpolation of the expression expr (if any). It may also use %(name)s interpolations for other variables in the namespace. E.g.:

```
DocPara('i',format='The value of i is %(__expr__)d',style=normal)
```

8.4 `DocAssert(self, cond, format=None)`

Raises an `AssertionError` containing the `format` string if `cond` evaluates as `False`.

```
DocAssert(val, 'val is False')
```

8.5 `DocIf(self, cond, thenBlock, elseBlock=[])`

If `cond` evaluates as `True`, this flowable is replaced by the `thenBlock` elsethe `elseBlock`.

```
DocIf('i>3',Paragraph('The value of i is larger than 3',normal),\
        Paragraph('The value of i is not larger than 3',normal))
```

8.6 `DocWhile(self, cond, whileBlock)`

Runs the `whileBlock` while `cond` evaluates to `True`. E.g.:

```
DocAssign('i',5)
DocWhile('i',[DocPara('i',format='The value of i is %(__expr__)d',style=normal),DocExec('i-=1')])
```

This example produces a set of paragraphs of the form:

```
The value of i is 5
The value of i is 4
The value of i is 3
The value of i is 2
The value of i is 1
```

Chapter 9 Other Useful Flowables

9.1 Preformatted(text, style, bulletText=None, dedent=0, maxLineLength=None, splitChars=None, newLineChars=None)

Creates a preformatted paragraph which does no wrapping, line splitting or other manipulations. No XML style tags are taken account of in the text. If dedent is non zero dedent common leading spaces will be removed from the front of each line.

Defining a maximum line length

You can use the property maxLineLength to define a maximum line length. If a line length exceeds this maximum value, the line will be automatically splitted.

The line will be split on any single character defined in splitChars. If no value is provided for this property, the line will be split on any of the following standard characters: space, colon, full stop, semi-colon, coma, hyphen, forward slash, back slash, left parenthesis, left square bracket and left curly brace

Characters can be automatically inserted at the beginning of each line that has been created. You can set the property newLineChars to the characters you want to use.

```
from reportlab.lib.styles import getSampleStyleSheet
stylesheet=getSampleStyleSheet()
normalStyle = stylesheet['Code']
text='''
class XPreformatted(Paragraph):
    def __init__(self, text, style, bulletText = None, frags=None, caseSensitive=1):
        self.caseSensitive = caseSensitive
        if maximumLineLength and text:
            text = self.stopLine(text, maximumLineLength, splitCharacters)
        cleaner = lambda text, dedent=dedent: ''.join(_dedenter(text or '',dedent))
        self._setup(text, style, bulletText, frags, cleaner)
'''
t=Preformatted(text,normalStyle,maxLineLength=60, newLineChars='> ')
```

produces

```
class XPreformatted(Paragraph):
    def __init__(self, text, style, bulletText = None,
> frags=None, caseSensitive=1):
        self.caseSensitive = caseSensitive
        if maximumLineLength and text:
            text = self.stopLine(text, maximumLineLength,
> splitCharacters)
        cleaner = lambda text, dedent=dedent: ''.join(
> _dedenter(text or '',dedent))
        self._setup(text, style, bulletText, frags, cleaner)
```

9.2 XPreformatted(text, style, bulletText=None, dedent=0, frags=None)

This is a non rearranging form of the Paragraph class; XML tags are allowed in text and have the same meanings as for the Paragraph class. As for Preformatted, if dedent is non zero dedent common leading spaces will be removed from the front of each line.

```
from reportlab.lib.styles import getSampleStyleSheet
stylesheet=getSampleStyleSheet()
normalStyle = stylesheet['Code']
text='''

   This is a non rearranging form of the <b>Paragraph</b> class;
   <b><font color=red>XML</font></b> tags are allowed in <i>text</i> and have the same

      meanings as for the <b>Paragraph</b> class.
   As for <b>Preformatted</b>, if dedent is non zero <font color="red" size="+1">dedent</font>
```

```
            common leading spaces will be removed from the
        front of each line.
        You can have &amp;amp; style entities as well for &amp; &lt; &gt; and &quot;.

    '''
    t=XPreformatted(text,normalStyle,dedent=3)
```

produces

```
    This is a non rearranging form of the Paragraph class;
    XML tags are allowed in text and have the same

       meanings as for the Paragraph class.
    As for Preformatted, if dedent is non zero dedent
        common leading spaces will be removed from the
    front of each line.
    You can have &amp; style entities as well for & < > and ".
```

9.3 Image(filename, width=None, height=None)

Create a flowable which will contain the image defined by the data in file `filename`. The default **PDF** image type *jpeg* is supported and if the **PIL** extension to **Python** is installed the other image types can also be handled. If `width` and or `height` are specified then they determine the dimension of the displayed image in *points*. If either dimension is not specified (or specified as `None`) then the corresponding pixel dimension of the image is assumed to be in *points* and used.

```
    Image("lj8100.jpg")
```

will display as

whereas

```
    im = Image("lj8100.jpg", width=2*inch, height=2*inch)
    im.hAlign = 'CENTER'
```

produces

9.4 Spacer(width, height)

This does exactly as would be expected; it adds a certain amount of space into the story. At present this only works for vertical space.

9.5 PageBreak()

This `Flowable` represents a page break. It works by effectively consuming all vertical space given to it. This is sufficient for a single `Frame` document, but would only be a frame break for multiple frames so the `BaseDocTemplate` mechanism detects `pageBreaks` internally and handles them specially.

9.6 CondPageBreak(height)

This `Flowable` attempts to force a `Frame` break if insufficient vertical space remains in the current `Frame`. It is thus probably wrongly named and should probably be renamed as `CondFrameBreak`.

9.7 KeepTogether(flowables)

This compound `Flowable` takes a list of `Flowables` and attempts to keep them in the same `Frame`. If the total height of the `Flowables` in the list `flowables` exceeds the current frame's available space then all the space is used and a frame break is forced.

9.8 TableOfContents()

A table of contents can be generated by using the `TableOfContents` flowable. The following steps are needed to add a table of contents to your document:

Create an instance of `TableOfContents`. Override the level styles (optional) and add the object to the story:

```
toc = TableOfContents()
PS = ParagraphStyle
toc.levelStyles = [
    PS(fontName='Times-Bold', fontSize=14, name='TOCHeading1',
            leftIndent=20, firstLineIndent=-20, spaceBefore=5, leading=16),
    PS(fontSize=12, name='TOCHeading2',
            leftIndent=40, firstLineIndent=-20, spaceBefore=0, leading=12),
    PS(fontSize=10, name='TOCHeading3',
            leftIndent=60, firstLineIndent=-20, spaceBefore=0, leading=12),
    PS(fontSize=10, name='TOCHeading4',
            leftIndent=100, firstLineIndent=-20, spaceBefore=0, leading=12),
]
story.append(toc)
```

Entries to the table of contents can be done either manually by calling the `addEntry` method on the `TableOfContents` object or automatically by sending a `'TOCEntry'` notification in the `afterFlowable` method of the `DocTemplate` you are using. The data to be passed to `notify` is a list of three or four items countaining a level number, the entry text, the page number and an optional destination key which the entry should point to. This list will usually be created in a document template's method like afterFlowable(), making notification calls using the notify() method with appropriate data like this:

```
def afterFlowable(self, flowable):
    """Detect Level 1 and 2 headings, build outline,
    and track chapter title."""
    if isinstance(flowable, Paragraph):
        txt = flowable.getPlainText()
        if style == 'Heading1':
            # ...
            self.notify('TOCEntry', (0, txt, self.page))
        elif style == 'Heading2':
            # ...
            key = 'h2-%s' % self.seq.nextf('heading2')
            self.canv.bookmarkPage(key)
            self.notify('TOCEntry', (1, txt, self.page, key))
        # ...
```

This way, whenever a paragraph of style `'Heading1'` or `'Heading2'` is added to the story, it will appear in the table of contents. `Heading2` entries will be clickable because a bookmarked key has been supplied.

Finally you need to use the `multiBuild` method of the DocTemplate because tables of contents need several passes to be generated:

```
doc.multiBuild(story)
```

Below is a simple but working example of a document with a table of contents:

```
from reportlab.lib.styles import ParagraphStyle as PS
from reportlab.platypus import PageBreak
from reportlab.platypus.paragraph import Paragraph
from reportlab.platypus.doctemplate import PageTemplate, BaseDocTemplate
from reportlab.platypus.tableofcontents import TableOfContents
from reportlab.platypus.frames import Frame
from reportlab.lib.units import cm

class MyDocTemplate(BaseDocTemplate):

    def __init__(self, filename, **kw):
        self.allowSplitting = 0
        BaseDocTemplate.__init__(self, filename, **kw)
        template = PageTemplate('normal', [Frame(2.5*cm, 2.5*cm, 15*cm, 25*cm, id='F1')])
        self.addPageTemplates(template)

    def afterFlowable(self, flowable):
        "Registers TOC entries."
        if flowable.__class__.__name__ == 'Paragraph':
            text = flowable.getPlainText()
            style = flowable.style.name
            if style == 'Heading1':
                self.notify('TOCEntry', (0, text, self.page))
            if style == 'Heading2':
                self.notify('TOCEntry', (1, text, self.page))

h1 = PS(name = 'Heading1',
       fontSize = 14,
       leading = 16)

h2 = PS(name = 'Heading2',
       fontSize = 12,
       leading = 14,
       leftIndent = delta)

# Build story.
story = []
toc = TableOfContents()
# For conciseness we use the same styles for headings and TOC entries
toc.levelStyles = [h1, h2]
story.append(toc)
story.append(PageBreak())
story.append(Paragraph('First heading', h1))
story.append(Paragraph('Text in first heading', PS('body')))
story.append(Paragraph('First sub heading', h2))
story.append(Paragraph('Text in first sub heading', PS('body')))
story.append(PageBreak())
story.append(Paragraph('Second sub heading', h2))
story.append(Paragraph('Text in second sub heading', PS('body')))
story.append(Paragraph('Last heading', h1))

doc = MyDocTemplate('mintoc.pdf')
doc.multiBuild(story)
```

9.9 `SimpleIndex()`

An index can be generated by using the `SimpleIndex` flowable. The following steps are needed to add an index to your document:

Use the index tag in paragraphs to index terms:

```
story = []

...

story.append('The third <index item="word" />word of this paragraph is indexed.')
```

Create an instance of `SimpleIndex` and add it to the story where you want it to appear:

```
index = SimpleIndex(dot=' . ', headers=headers)
story.append(index)
```

The parameters which you can pass into the SimpleIndex constructor are explained in the reportlab reference. Now, build the document by using the canvas maker returned by SimpleIndex.getCanvasMaker():

```
doc.build(story, canvasmaker=index.getCanvasMaker())
```

To build an index with multiple levels, pass a comma-separated list of items to the item attribute of an index tag:

```
<index item="terma,termb,termc" />
<index item="terma,termd" />
```

terma will respresent the top-most level and termc the most specific term. termd and termb will appear in the same level inside terma.

If you need to index a term containing a comma, you will need to escape it by doubling it. To avoid the ambiguity of three consecutive commas (an escaped comma followed by a list separator or a list separator followed by an escaped comma?) introduce a space in the right position. Spaces at the beginning or end of terms will be removed.

```
<index item="comma(,,), ,, ,...   " />
```

This indexes the terms "comma (,)", "," and "...".

9.10 `ListFlowable(),ListItem()`

Use these classes to make ordered and unordered lists. Lists can be nested.

`ListFlowable()` will create an ordered list, which can contain any flowable. The class has a number of parameters to change font, colour, size, style and position of list numbers, or of bullets in unordered lists. The type of numbering can also be set to use lower or upper case letters ('A,B,C' etc.) or Roman numerals (capitals or lowercase) using the bulletType property. To change the list to an unordered type, set bulletType='bullet'.

Items within a `ListFlowable()` list can be changed from their default appearance by wrapping them in a `ListItem()` class and setting its properties.

The following will create an ordered list, and set the third item to an unordered sublist.

```
from reportlab.platypus import ListFlowable, ListItem
from reportlab.lib.styles import getSampleStyleSheet
styles = getSampleStyleSheet()
style = styles["Normal"]
t = ListFlowable(
[
Paragraph("Item no.1", style),
ListItem(Paragraph("Item no. 2", style),bulletColor="green",value=7),
ListFlowable(
                [
                Paragraph("sublist item 1", style),
                ListItem(Paragraph('sublist item 2', style),bulletColor='red',value='square')
                ],
                bulletType='bullet',
                start='square',
                ),
Paragraph("Item no.4", style),
],
bulletType='i'
)
```

produces

i Item no.1

vii Item no. 2

viii \u25a0 Sublist item 1

\u25a0 Sublist item 2

ix Item no.4

Chapter 10 Writing your own `Flowable` Objects

Flowables are intended to be an open standard for creating reusable report content, and you can easily create your own objects. We hope that over time we will build up a library of contributions, giving reportlab users a rich selection of charts, graphics and other "report widgets" they can use in their own reports. This section shows you how to create your own flowables.
we should put the Figure class in the standard library, as it is a very useful base.

10.1 A very simple `Flowable`

Recall the `hand` function from the `pdfgen` section of this user guide which generated a drawing of a hand as a closed figure composed from Bezier curves.

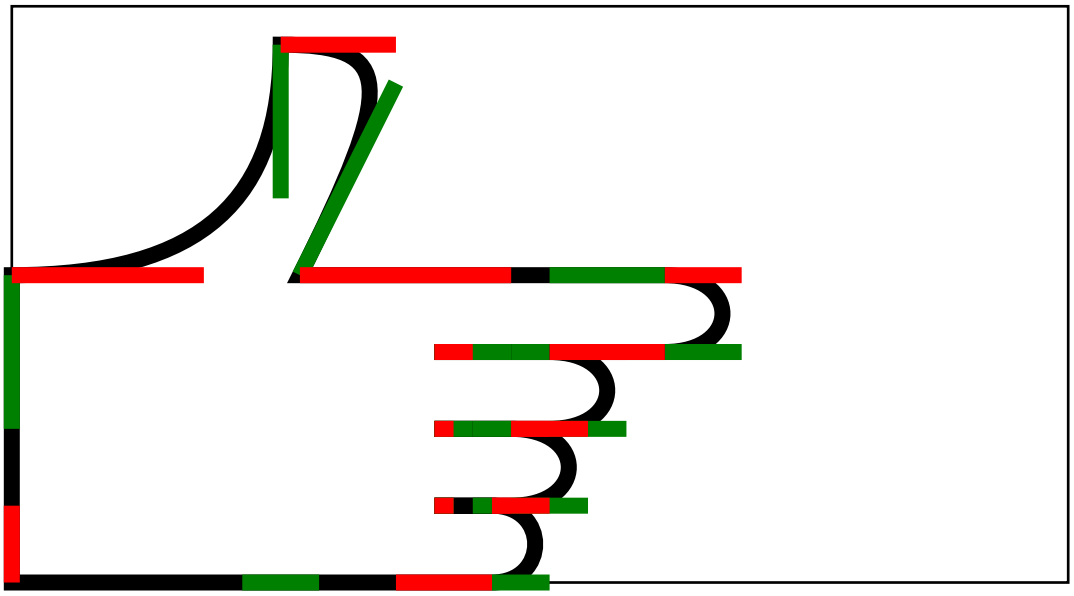


Figure 10-1: a hand

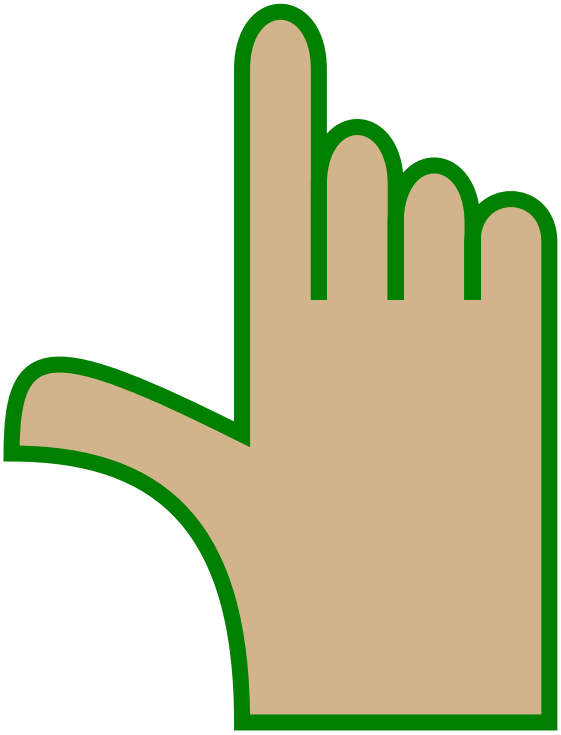
To embed this or any other drawing in a Platypus flowable we must define a subclass of `Flowable` with at least a `wrap` method and a `draw` method.

```
from reportlab.platypus.flowables import Flowable
from reportlab.lib.colors import tan, green
class HandAnnotation(Flowable):
    '''A hand flowable.'''
    def __init__(self, xoffset=0, size=None, fillcolor=tan, strokecolor=green):
        from reportlab.lib.units import inch
        if size is None: size=4*inch
        self.fillcolor, self.strokecolor = fillcolor, strokecolor
        self.xoffset = xoffset
        self.size = size
        # normal size is 4 inches
        self.scale = size/(4.0*inch)
    def wrap(self, *args):
        return (self.xoffset, self.size)
    def draw(self):
        canvas = self.canv
        canvas.setLineWidth(6)
        canvas.setFillColor(self.fillcolor)
        canvas.setStrokeColor(self.strokecolor)
        canvas.translate(self.xoffset+self.size,0)
        canvas.rotate(90)
        canvas.scale(self.scale, self.scale)
        hand(canvas, debug=0, fill=1)
```

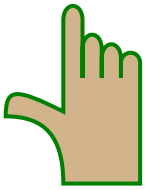
The `wrap` method must provide the size of the drawing -- it is used by the Platypus mainloop to decide whether this element fits in the space remaining on the current frame. The `draw` method performs the

drawing of the object after the Platypus mainloop has translated the `(0,0)` origin to an appropriate location in an appropriate frame.

Below are some example uses of the `HandAnnotation` flowable.



The default.



Just one inch high.

One inch high and shifted to the left with blue and cyan.

10.2 Modifying a Built in `Flowable`

To modify an existing flowable, you should create a derived class and override the methods you need to change to get the desired behaviour

As an example to create a rotated image you need to override the wrap and draw methods of the existing Image class

```
class RotatedImage(Image):
    def wrap(self,availWidth,availHeight):
        h, w = Image.wrap(self,availHeight,availWidth)
        return w, h
    def draw(self):
        self.canv.rotate(90)
```

```
            Image.draw(self)
        I = RotatedImage('../images/replogo.gif')
        I.hAlign = 'CENTER'
```

produces

Chapter 11 Graphics

11.1 Introduction

ReportLab Graphics is one of the sub-packages to the ReportLab library. It started off as a stand-alone set of programs, but is now a fully integrated part of the ReportLab toolkit that allows you to use its powerful charting and graphics features to improve your PDF forms and reports.

11.2 General Concepts

In this section we will present some of the more fundamental principles of the graphics library, which will show-up later in various places.

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 (renderPDF), Postscript (renderPS), and bitmap output (renderPM). The bitmap renderer uses Raph Levien's *libart* rasterizer and Fredrik Lundh's *Python Imaging Library* (PIL). The SVG renderer makes use of Python's standard library XML modules, so you don't need to install the XML-SIG's additional package named PyXML. 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.

The SVG renderer is special as it is still pretty experimental. The SVG code it generates is not really optimised in any way and maps only the features available in ReportLab Graphics (RLG) to SVG. This means there is no support for SVG animation, interactivity, scripting or more sophisticated clipping, masking or graduation shapes. So, be careful, and please report any bugs you find!

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*. For the SVG renderer this is actually no problem as it will take your drawings and flip things as needed, so your SVG output looks just as expected.

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...).

Getting Started

Let's create a simple drawing containing the string "Hello World" and some special characters, 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))
d.add(String(180,86, 'Special characters \
            \xc2\xa2\xc2\xa9\xc2\xae\xc2\xa3\xce\xb1\xce\xb2',
            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 11-1: '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.drawToFile(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.

To produce an SVG file containing the identical drawing, which may be imported into graphical editing tools such as Illustrator all we need to do is write code like this:

```
from reportlab.graphics import renderSVG
renderSVG.drawToFile(d, 'example1.svg')
```

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\andy\graphics\shapes.py", line 254, in __setattr__
    validateSetattr(self,attr,value)    #from reportlab.lib.attrmap
  File "C:\code\users\andy\lib\attrmap.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.

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.

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.

11.3 Charts

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

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 here)

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.

11.4 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)
```

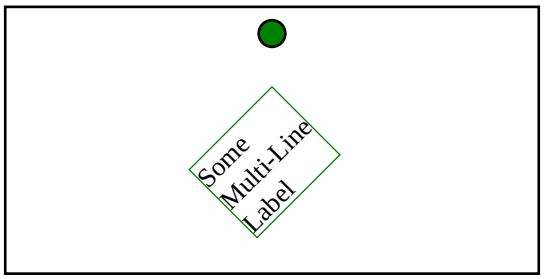


Figure 11-2: Label example

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 11-4 - 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 `tests`, run the script `test_charts_textlabels.py` and look at the PDF document it generates!

11.5 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:

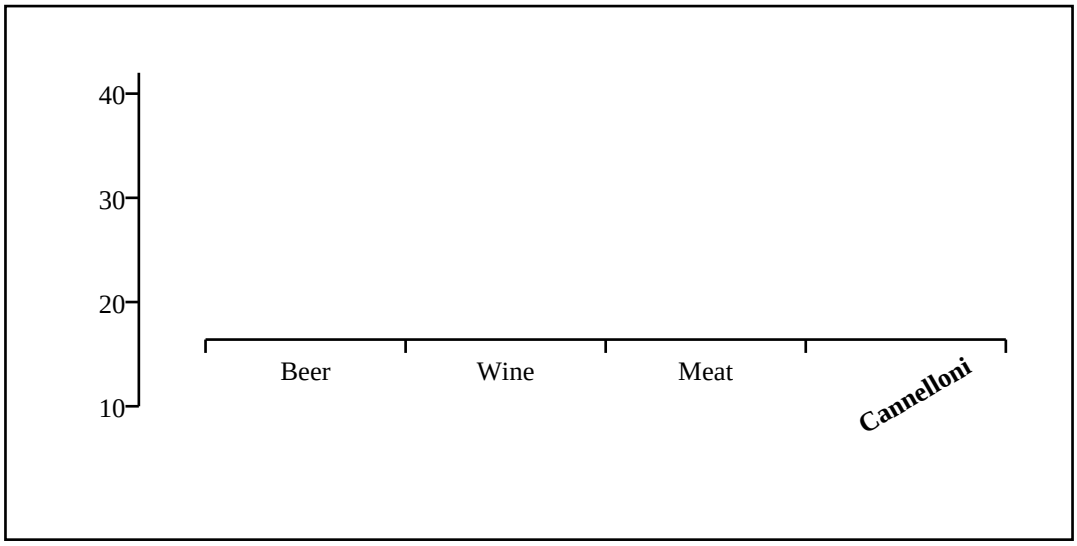


Figure 11-3: Two isolated axes

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. The `reversed` attribute (if 1) indicates that the categories should be reversed. 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 11-5 - 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 11-6 - 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)
```

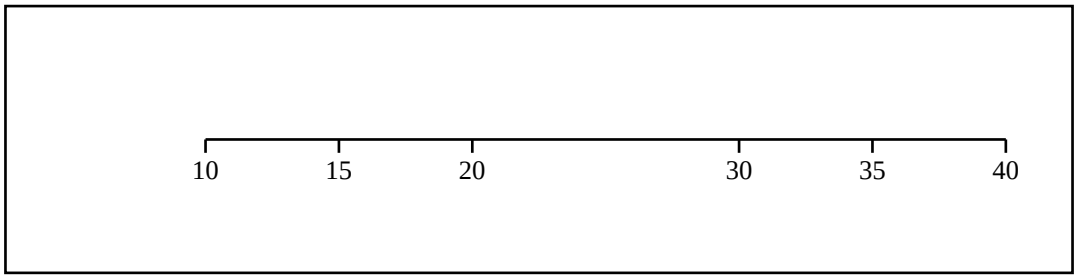


Figure 11-4: An axis with non-equidistant tick marks

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 11-7 - Axes joining properties

11.6 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:

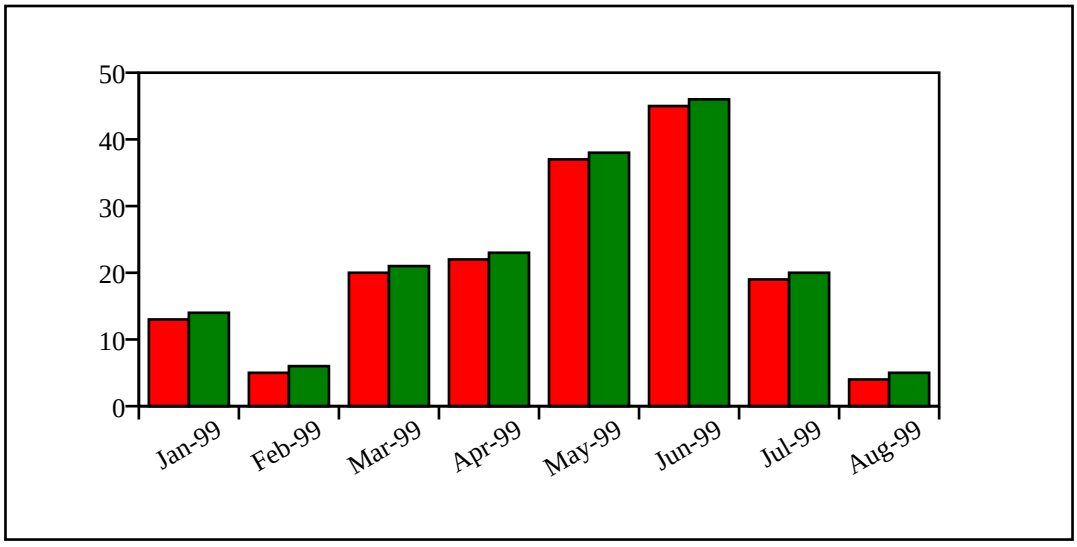


Figure 11-5: Simple bar chart with two data series

```
# 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)
`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. They are positioned automatically above the bar for positive values and below for negative ones.
`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 11-8 - 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.

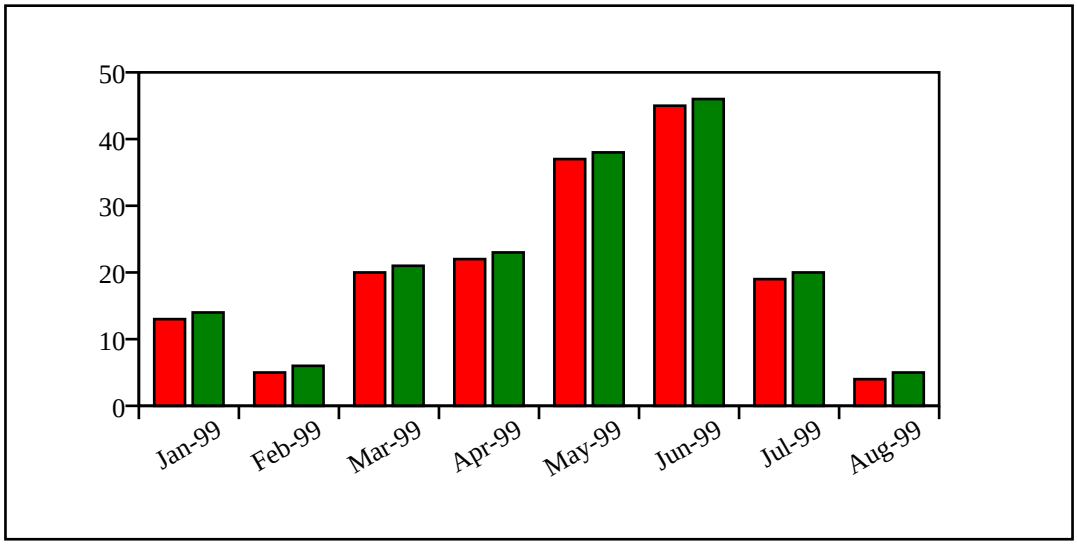


Figure 11-6: Like before, but with modified spacing

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.

Stacked bars are also supported for vertical bar graphs. You enable this layout for your chart by setting the `style` attribute to `'stacked'` on the `categoryAxis`.

```
bc.categoryAxis.style = 'stacked'
```

Here is an example of the previous chart values arranged in the stacked style.

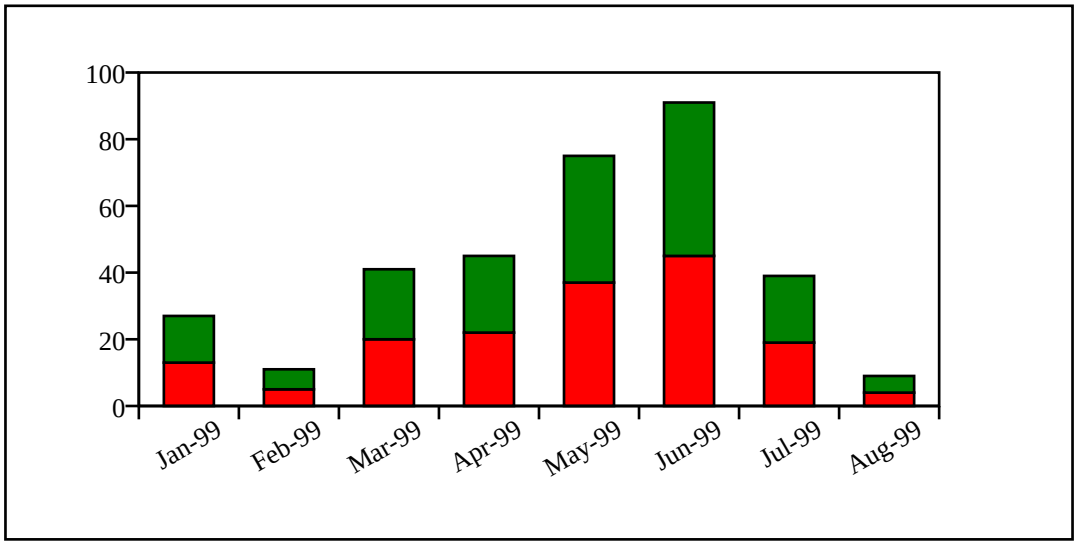


Figure 11-7: Stacking bars on top of each other.

11.7 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 = 'Jan Feb Mar Apr May Jun Jul Aug'.split(' ')
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)
```

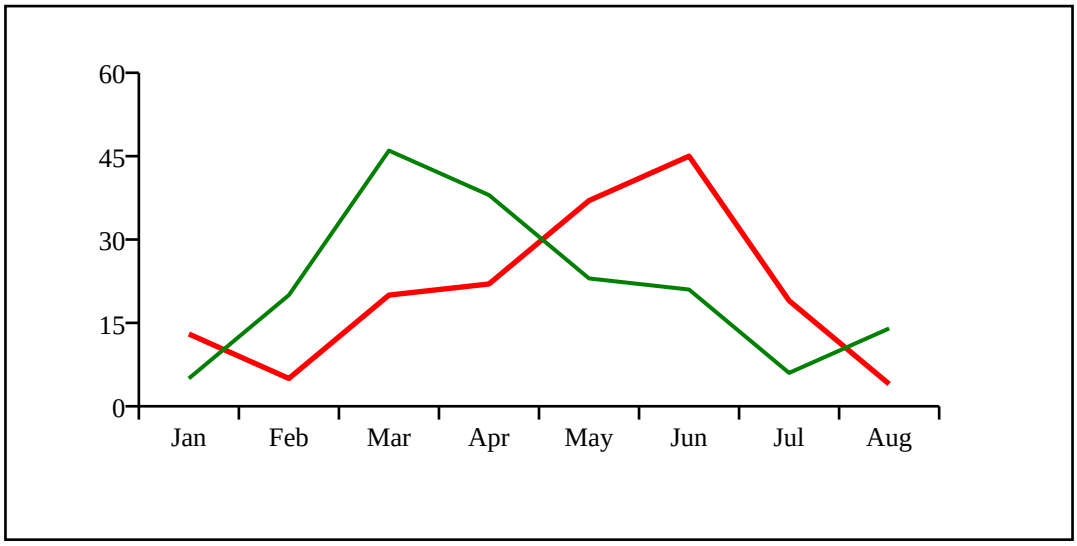


Figure 11-8: HorizontalLineChart sample

Property	Meaning
`data`	Data to be plotted, list of (lists of) numbers.
`x, y, width, height`	Bounding box of the line chart. Note that x and y do NOT specify the centre but the bottom left corner
`valueAxis`	The value axis, which may be formatted as described previously.
`categoryAxis`	The category axis, which may be formatted as described previously.
`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.
`lines.strokeColor`	Color of the line.
`lines.strokeWidth`	Width of the line.
`lineLabels`	A collection of labels used to format all line labels. Since this is a two-dimensional array, you may explicitly format the third label of the second line using this syntax: chart.lineLabels[(1,2)].fontSize = 12
`lineLabelFormat`	Defaults to None. As with the YValueAxis, if you supply a function or format string then labels will be drawn next to each line showing the numeric value. You can also set it to 'values' to display the values explicity defined in lineLabelArray.
`lineLabelArray`	Explicit array of line label values, must match size of data if present. These labels values will be displayed only if the property lineLabelFormat above is set to 'values'.

Table 11-9 - HorizontalLineChart properties

11.8 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.widgets.markers import makeMarker
```

```
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 = makeMarker('FilledCircle')
lp.lines[1].symbol = makeMarker('Circle')
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)
```

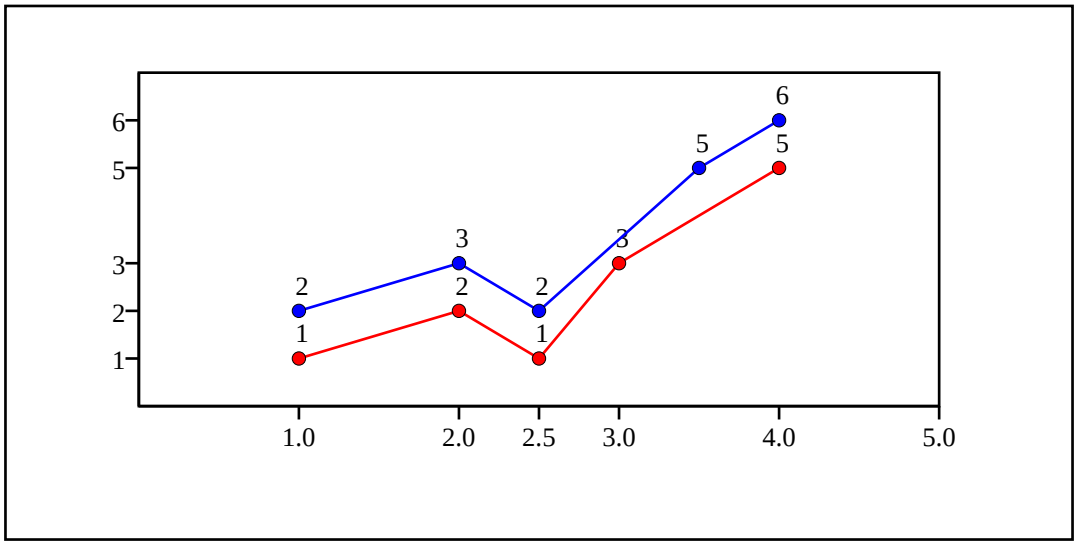


Figure 11-9: LinePlot sample

Property	Meaning
`data`	Data to be plotted, list of (lists of) numbers.
`x, y, width, height`	Bounding box of the line chart. Note that x and y do NOT specify the centre but the bottom left corner
`xValueAxis`	The vertical value axis, which may be formatted as described previously.
`yValueAxis`	The horizontal value axis, which may be formatted as described previously.
`strokeColor`	Defaults to None. This will draw a border around the plot rectangle, which may be useful in debugging. Axes will overwrite this.
`strokeWidth`	Defaults to None. Width of the border around the plot rectangle.
`fillColor`	Defaults to None. This will fill the plot rectangle with a solid color.
`lines.strokeColor`	Color of the line.

lines.strokeWidth	Width of the line.
lines.symbol	Marker used for each point. You can create a new marker using the function makeMarker(). For example to use a circle, the function call would be makeMarker('Circle')
lineLabels	A collection of labels used to format all line labels. Since this is a two-dimensional array, you may explicitly format the third label of the second line using this syntax: chart.lineLabels[(1,2)].fontSize = 12
lineLabelFormat	Defaults to None. As with the YValueAxis, if you supply a function or format string then labels will be drawn next to each line showing the numeric value. You can also set it to 'values' to display the values explicity defined in lineLabelArray.
lineLabelArray	Explicit array of line label values, must match size of data if present. These labels values will be displayed only if the property lineLabelFormat above is set to 'values'.

Table 11-10 - LinePlot properties

11.9 Pie Charts

As usual, we will start with an example:

```
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)
```

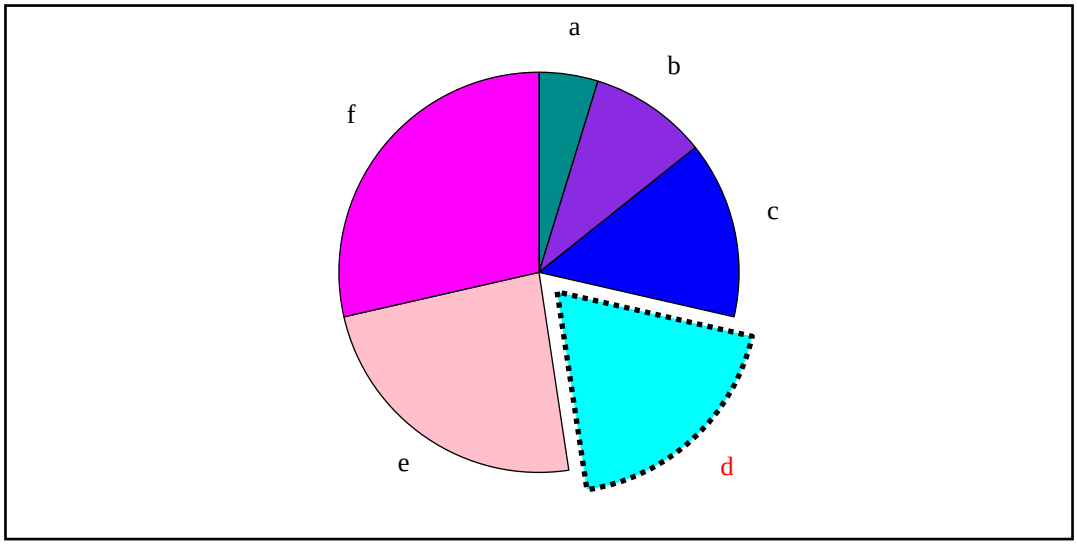


Figure 11-10: A bare bones pie chart

Properties are covered below. The pie has a 'wedges' collection and we document wedge properties in the same table.

Property	Meaning
`data`	A list or tuple of numbers
`x, y, width, height`	Bounding box of the pie. Note that x and y do NOT specify the centre but the bottom left corner, and that width and height do not have to be equal; pies may be elliptical and wedges will be drawn correctly.
`labels`	None, or a list of strings. Make it None if you don't want labels around the edge of the pie. Since it is impossible to know the size of slices, we generally discourage placing labels in or around pies; it is much better to put them in a legend alongside.
`startAngle`	Where is the start angle of the first pie slice? The default is '90' which is twelve o'clock.
`direction`	Which direction do slices progress in? The default is 'clockwise'.
`sideLabels`	This creates a chart with the labels in two columns, one on either side.
`sideLabelsOffset`	This is a fraction of the width of the pie that defines the horizontal distance between the pie and the columns of labels.
`simpleLabels`	Default is 1. Set to 0 to enable the use of customizable labels and of properties prefixed by label_ in the collection slices.
`wedges`	Collection of wedges. This lets you customise each wedge, or individual ones. See below
`wedges.strokeWidth`	Border width for wedge
`wedges.strokeColor`	Border color
`wedges.strokeDashArray`	Solid or dashed line configuration
`wedges.popout`	How far out should the slice(s) stick from the centre of the pie? Default is zero.
`wedges.fontName`	Name of the label font
`wedges.fontSize`	Size of the label font
`wedges.fontColor`	Color of the label text
`wedges.labelRadius`	This controls the anchor point for a text label. It is a fraction of the radius; 0.7 will place the text inside the pie, 1.2 will place it slightly outside. (note that if we add labels, we will keep this to specify their anchor point)

Table 11-11 - Pie properties

Customizing Labels

Each slide label can be customized individually by changing the properties prefixed by `label_` in the collection `wedges`. For example `pc.slices[2].label_angle = 10` changes the angle of the third label.

Before being able to use these customization properties, you need to disable simple labels with:
`pc.simplesLabels = 0`

Property	Meaning
label_dx	X Offset of the label
label_dy	Y Offset of the label
label_angle	Angle of the label, default (0) is horizontal, 90 is vertical, 180 is upside down
label_boxAnchor	Anchoring point of the label
label_boxStrokeColor	Border color for the label box
label_boxStrokeWidth	Border width for the label box
label_boxFillColor	Filling color of the label box
label_strokeColor	Border color for the label text
label_strokeWidth	Border width for the label text
label_text	Text of the label
label_width	Width of the label
label_maxWidth	Maximum width the label can grow to
label_height	Height of the label
label_textAnchor	Maximum height the label can grow to
label_visible	True if the label is to be drawn
label_topPadding	Padding at top of box
label_leftPadding	Padding at left of box
label_rightPadding	Padding at right of box
label_bottomPadding	Padding at bottom of box
label_simple_pointer	Set to 1 for simple pointers
label_pointer_strokeColor	Color of indicator line
label_pointer_strokeWidth	Width of indicator line

Table 11-12 - Pie.wedges label customization properties

Side Labels

If the sideLabels attribute is set to true, then the labels of the slices are placed in two columns, one on either side of the pie and the start angle of the pie will be set automatically. The anchor of the right hand column is set to 'start' and the anchor of the left hand column is set to 'end'. The distance from the edge of the pie from the edge of either column is decided by the sideLabelsOffset attribute, which is a fraction of the width of the pie. If xradius is changed, the pie can overlap the labels, and so we advise leaving xradius as None. There is an example below.

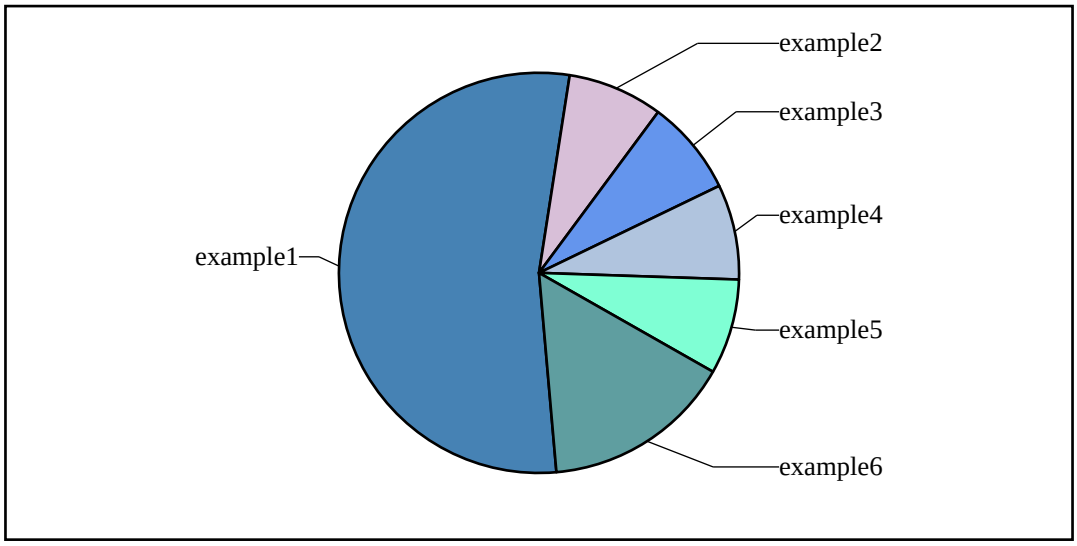


Figure 11-11: An example of a piechart with sideLabels =1

If you have sideLabels set to True, then some of the attributes become redundant, such as pointerLabelMode. Also sideLabelsOffset only changes the piechart if sideLabels is set to true.

Some issues

The pointers can cross if there are too many slices.

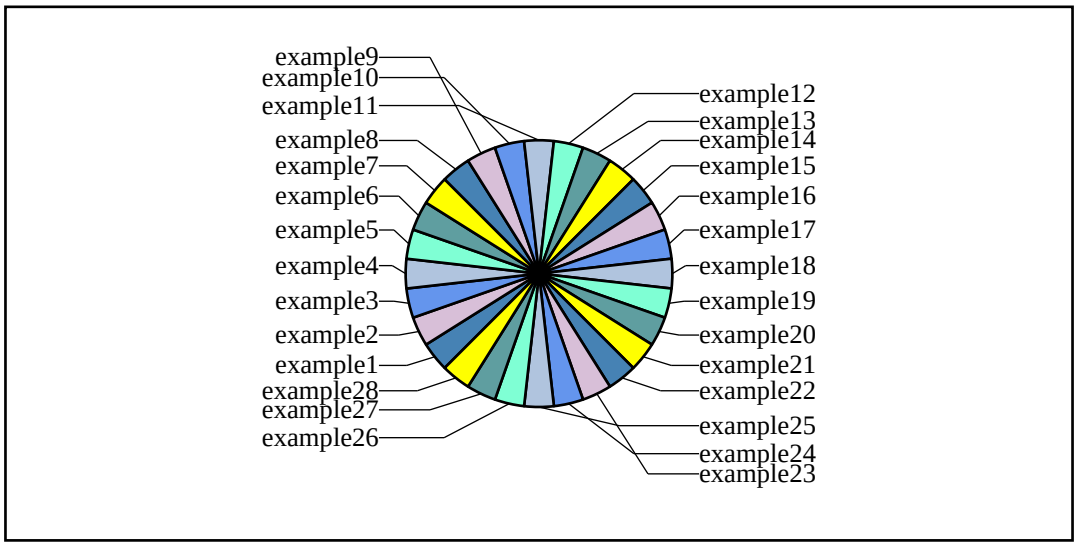


Figure 11-12: An example of pointers crossing

Also the labels can overlap despite checkLabelOverlap if they correspond to slices that are not adjacent.

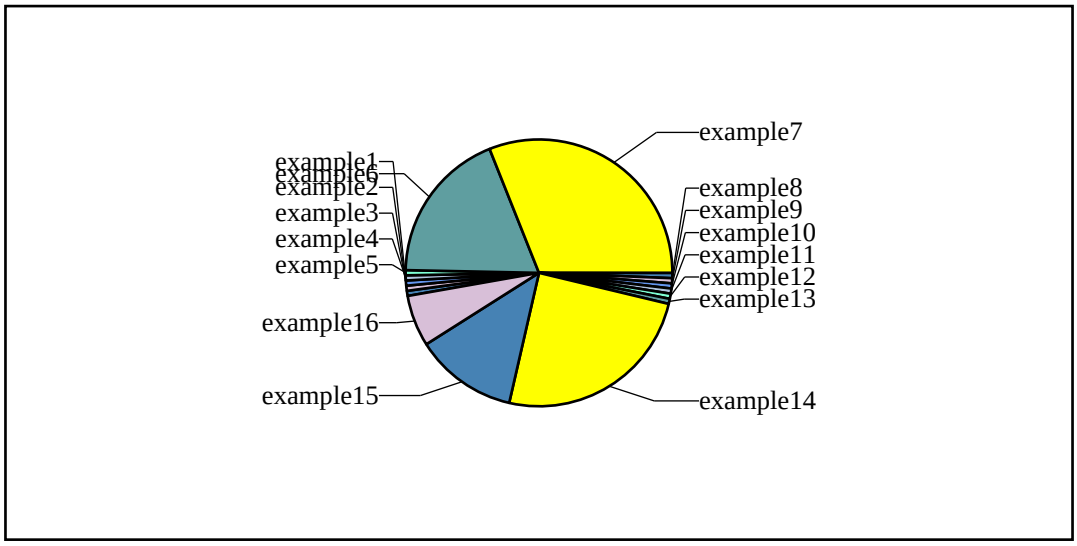


Figure 11-13: An example of labels overlapping

11.10 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?

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:

- ■■2022 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.
- ■■2022 Additional chart types - when the current design will have become more stable, we expect to add variants of bar charts to deal with percentile bars as well as the side-by-side variant seen here.

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'.

11.11 Shapes

This section 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.

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:

- ■■2022 Rect
- ■■2022 Circle
- ■■2022 Ellipse
- ■■2022 Wedge (a pie slice)
- ■■2022 Polygon
- ■■2022 Line

■■2022 PolyLine
■■2022 String
■■2022 Group
■■2022 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`).

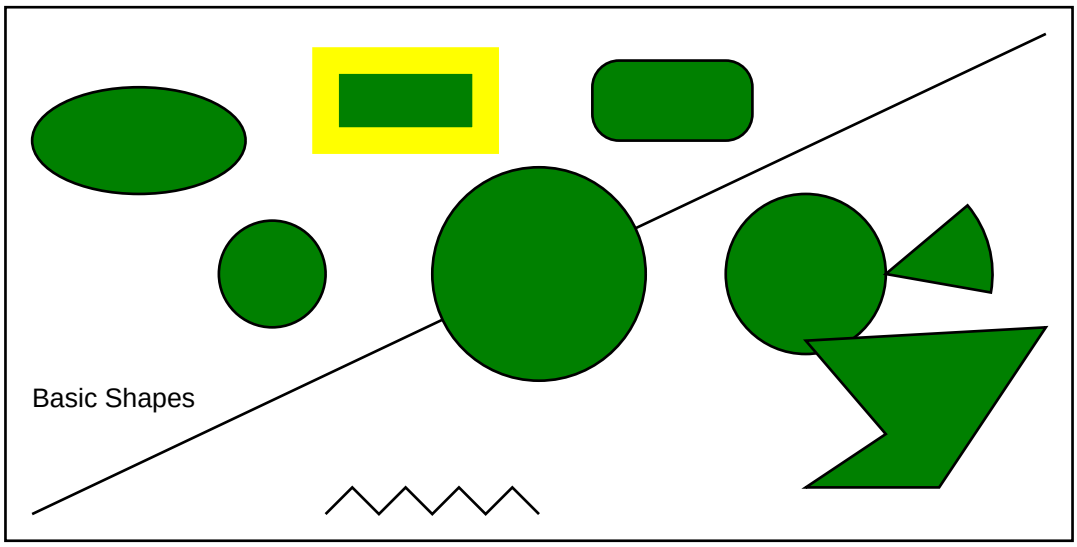


Figure 11-14: Basic shapes

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
>>>
```

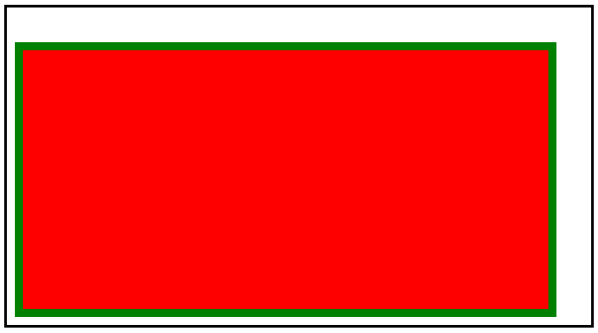


Figure 11-15: red rectangle with green border

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.

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)
```

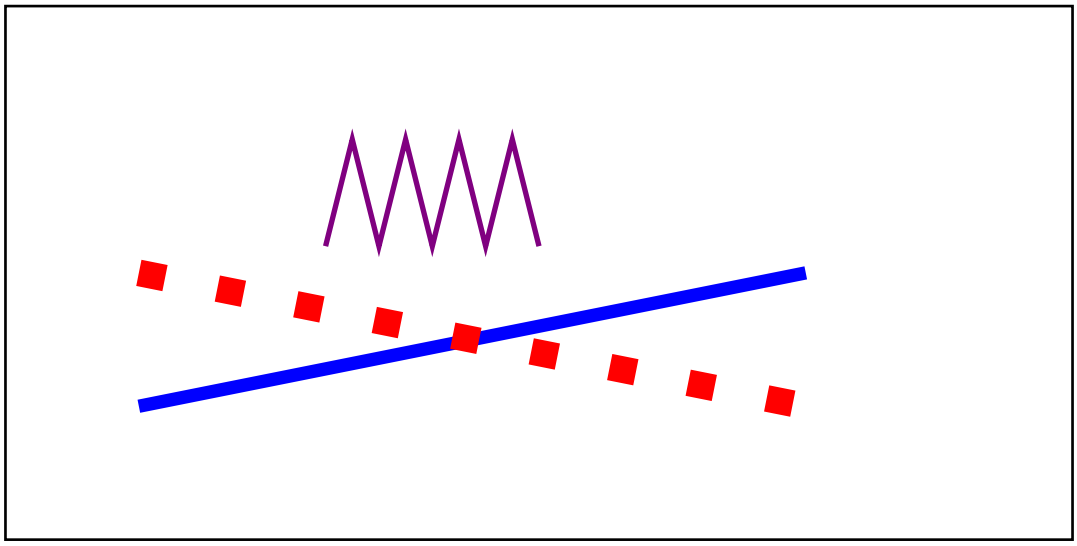


Figure 11-16: Line and PolyLine examples

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 'DarkGardenMK' 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='DarkGardenMK',
             fontSize=36))
```

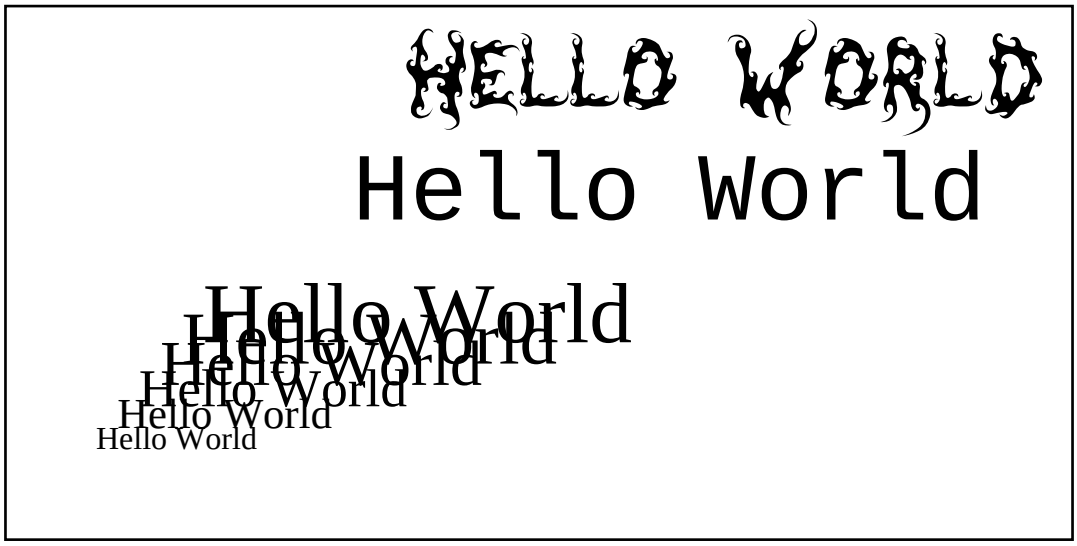


Figure 11-17: fancy font example

Paths

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

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)
```

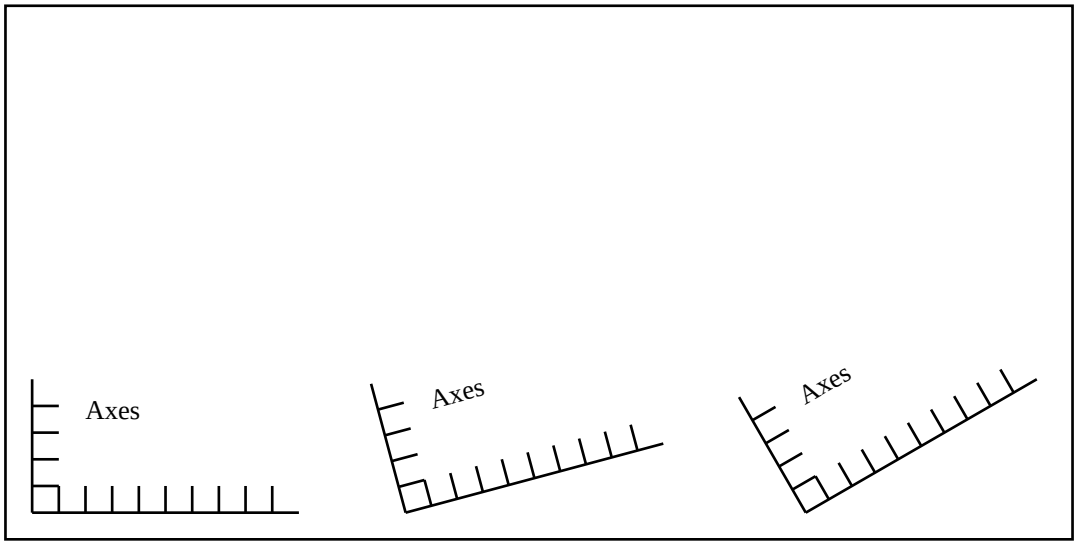


Figure 11-18: Groups examples

11.12 Widgets

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

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.

- ■■2022 A widget is a reusable shape
- ■■2022 it can be initialized with no arguments when its `draw()` method is called it creates a primitive Shape or a Group to represent itself
- ■■2022 It can have any parameters you want, and they can drive the way it is drawn
- ■■2022 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.

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')
```

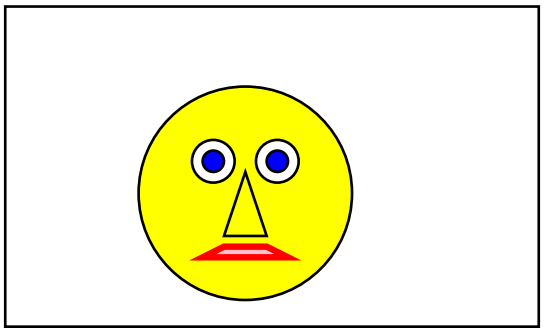


Figure 11-19: A sample widget

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 11-20: 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.NoEntry()
ds = signsandsymbols.DangerSign()
fd = signsandsymbols.FloppyDisk()
ns = signsandsymbols.NoSmoking()

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)
```

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.

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.

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()`.

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:

- ■■2022 the doc string for your widget class
- ■■2022 the code snippet from your *demo()* method, so people can see how to use it
- ■■2022 the drawing produced by the *demo()* method
- ■■2022 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!

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')
```

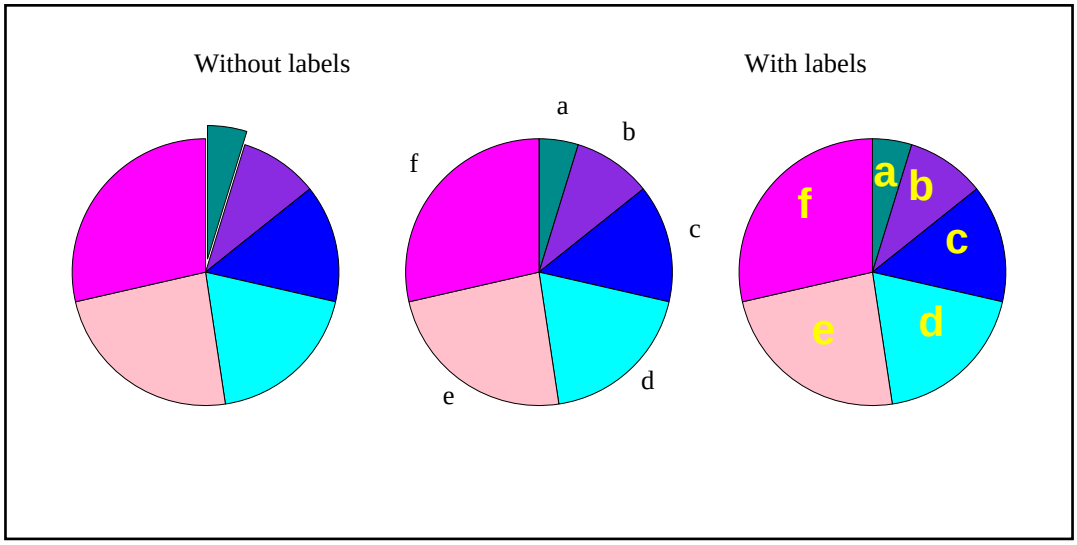


Figure 11-21: Some sample Pies

Appendix A ReportLab Demos

In the subdirectories of `reportlab/demos` there are a number of working examples showing almost all aspects of reportlab in use.

A.1 Odyssey

The three scripts odyssey.py, dodyssey.py and fodyssey.py all take the file odyssey.txt and produce PDF documents. The included odyssey.txt is short; a longer and more testing version can be found at ftp://ftp.reportlab.com/odyssey.full.zip.

```
Windows
cd reportlab\demos\odyssey
python odyssey.py
start odyssey.pdf

Linux
cd reportlab/demos/odyssey
python odyssey.py
acrord odyssey.pdf
```

Simple formatting is shown by the odyssey.py script. It runs quite fast, but all it does is gather the text and force it onto the canvas pages. It does no paragraph manipulation at all so you get to see the XML < & > tags.

The scripts fodyssey.py and dodyssey.py handle paragraph formatting so you get to see colour changes etc. Both scripts use the document template class and the dodyssey.py script shows the ability to do dual column layout and uses multiple page templates.

A.2 Standard Fonts and Colors

In `reportlab/demos/stdfonts` the script stdfonts.py can be used to illustrate ReportLab's standard fonts. Run the script using

```
cd reportlab\demos\stdfonts
python stdfonts.py
```

to produce two PDF documents, StandardFonts_MacRoman.pdf & StandardFonts_WinAnsi.pdf which show the two most common built in font encodings.

The colortest.py script in `reportlab/demos/colors` demonstrates the different ways in which reportlab can set up and use colors.

Try running the script and viewing the output document, colortest.pdf. This shows different color spaces and a large selection of the colors which are named in the `reportlab.lib.colors` module.

A.3 Py2pdf

Dinu Gherman contributed this useful script which uses reportlab to produce nicely colorized PDF documents from Python scripts including bookmarks for classes, methods and functions. To get a nice version of the main script try

```
cd reportlab/demos/py2pdf
python py2pdf.py py2pdf.py
acrord py2pdf.pdf
```

i.e. we used py2pdf to produce a nice version of py2pdf.py in the document with the same rootname and a .pdf extension.

The py2pdf.py script has many options which are beyond the scope of this simple introduction; consult the comments at the start of the script.

A.4 Gadflypaper

The Python script, gfe.py, in `reportlab/demos/gadflypaper` uses an inline style of document preparation. The script almost entirely produced by Aaron Watters produces a document describing Aaron's `gadfly` in memory database for Python. To generate the document use

```
cd reportlab\gadflypaper
python gfe.py
start gfe.pdf
```

everything in the PDF document was produced by the script which is why this is an inline style of document production. So, to produce a header followed by some text the script uses functions `header` and `p` which take some text and append to a global story list.

```
header("Conclusion")

p("""The revamped query engine design in Gadfly 2 supports
..........
and integration.""")
```

A.5 Pythonpoint

Andy Robinson has refined the pythonpoint.py script (in `reportlab\demos\pythonpoint`) until it is a really useful script. It takes an input file containing an XML markup and uses an xmllib style parser to map the tags into PDF slides. When run in its own directory pythonpoint.py takes as a default input the file pythonpoint.xml and produces pythonpoint.pdf which is documentation for Pythonpoint! You can also see it in action with an older paper

```
cd reportlab\demos\pythonpoint
python pythonpoint.py monterey.xml
start monterey.pdf
```

Not only is pythonpoint self documenting, but it also demonstrates reportlab and PDF. It uses many features of reportlab (document templates, tables etc). Exotic features of PDF such as fadeins and bookmarks are also shown to good effect. The use of an XML document can be contrasted with the *inline* style of the gadflypaper demo; the content is completely separate from the formatting