User Guide

ReportLab Version 1.02

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

103 Bayard Street
New Brunswick
New Jersey, 08904)
USA

Chapter 1 Introduction

1.1 About this document

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

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

In this chapter, we will cover the groundwork:

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

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

1.2 What is ReportLab?

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

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

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

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

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

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

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

1.3 What is Python?

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

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

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

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

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

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

1.4 Installation and Setup

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

Installation for experts

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

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

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

A note on available versions

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

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

Instructions for novices: Windows

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

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

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

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

Now you are ready to install reportlab itself.

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

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

Instructions for Python novices: Unix

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

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

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

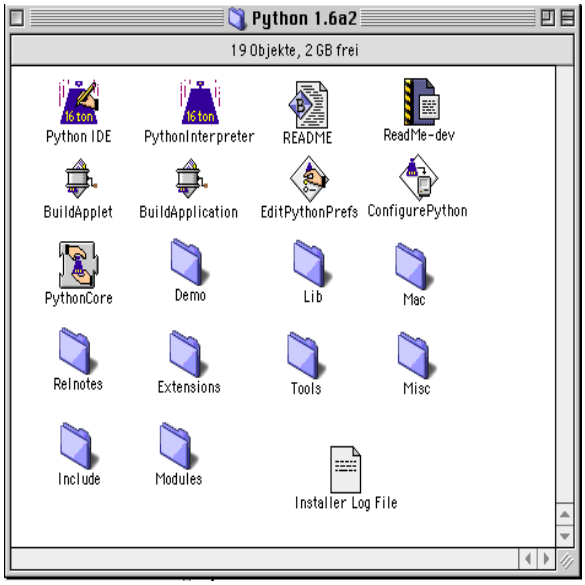
You should now be able to run python and execute the python statement

```
import reportlab
```

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

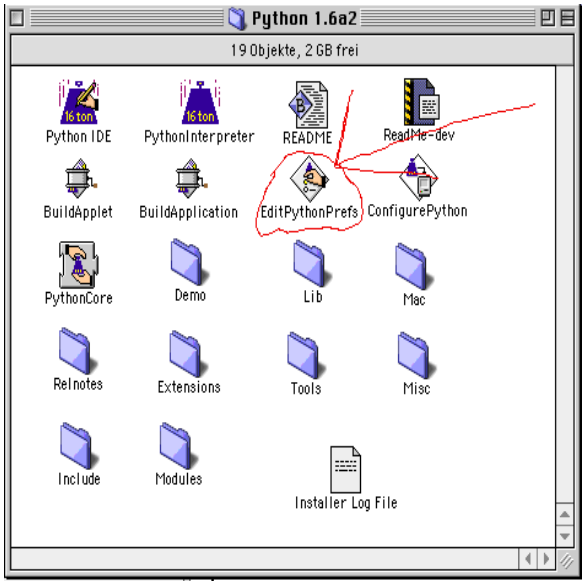
Instructions for Python novices: Mac

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

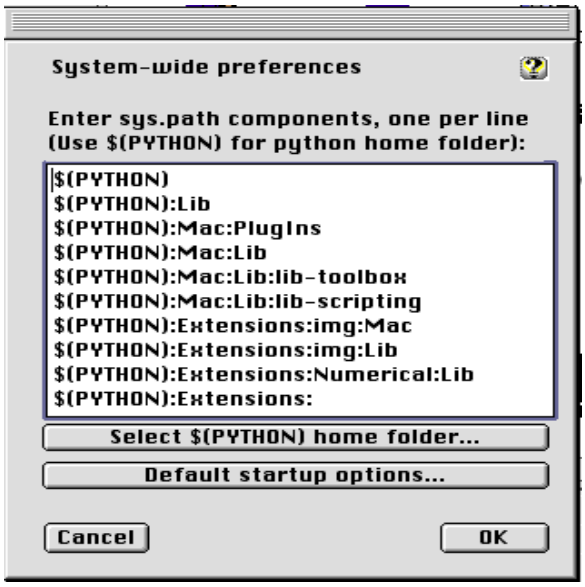


Now you can put Extensions in the Extensions-Folder; which is where you should unpack the **reportlab.zip** with your favorite unpack-utility. You¹ll get a subfolder named **reportlab**.

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



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



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

1.5 Getting Involved

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

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

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

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

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
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=pdfdoc.DEFAULT_ENCODING,
             verbosity=0):
```

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 determines which font encoding is used for the standard fonts; this should correspond to the encoding on your system. It has two values at present: `pdfdoc.WINANSI_ENCODING` or `pdfdoc.MACROMAN_ENCODING`. The `pdfdoc.DEFAULT_ENCODING` above points to the former, which is standard on Windows and many Unices (including Linux). If you are a Mac user and want to make a global change, modify the line at the top of *reportlab/pdfbase/pdfdoc.py* to switch it over.

We plan to add support for encodings on a per-font basis in future, so you can explicitly add in new fonts and say how the data is to be encoded. It is your responsibility to ensure that your string data is in an encoding matching that of the font. If conversions are needed, the Unicode library in Python 1.6 can be of great help.

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 octal escape sequence; 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.
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.

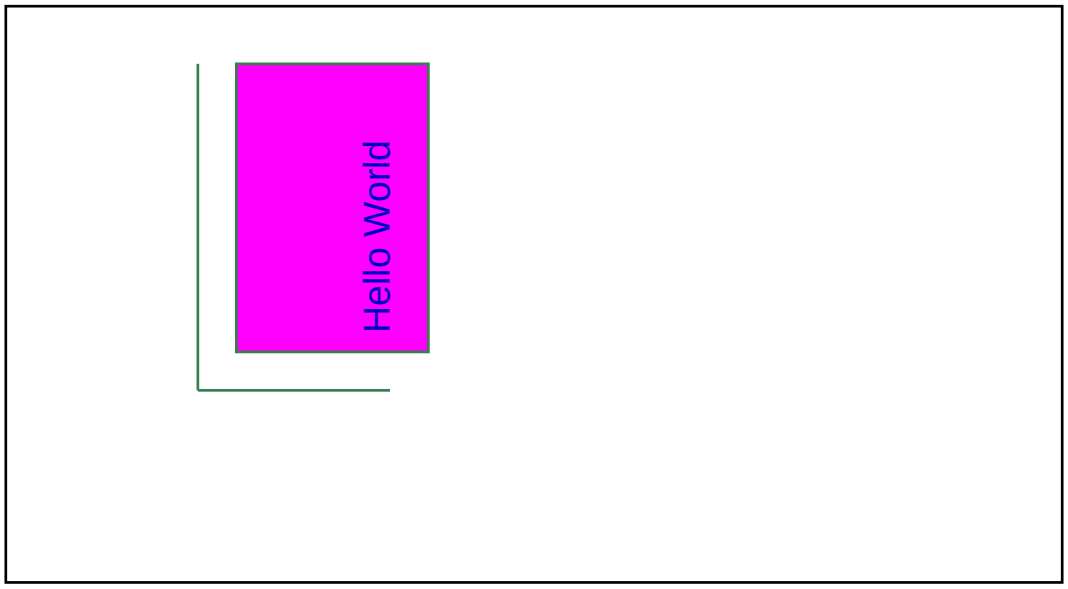


Figure 2-1: "Hello World" in pdfgen

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(x, y, width, height, 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

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

The `drawInlineImage` method places an image on the canvas.

You need the Python Imaging Library (PIL) to use images with the ReportLab package.

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.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.

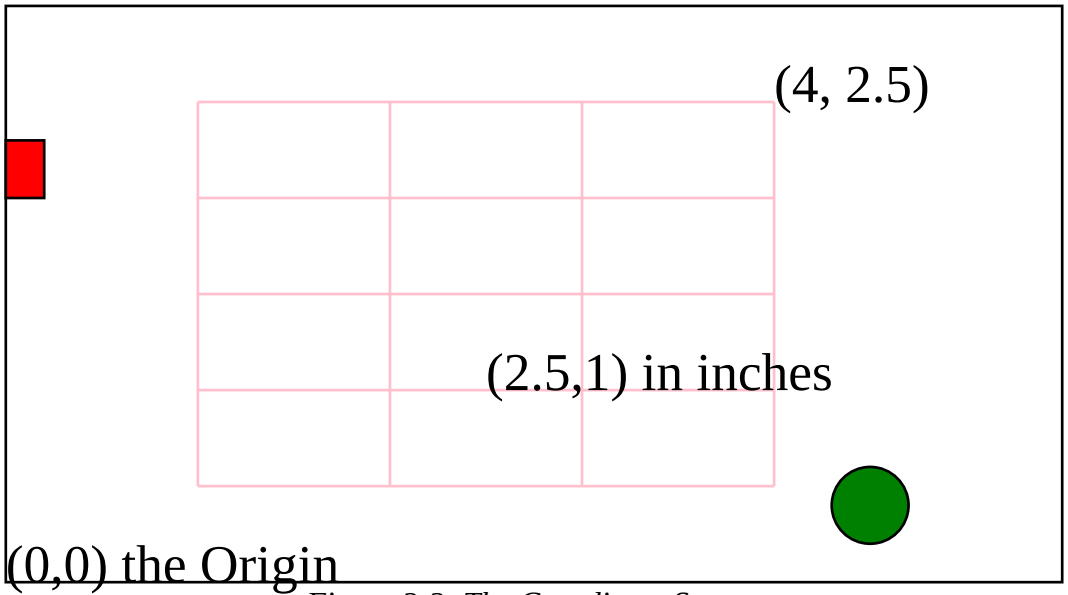


Figure 2-2: The Coordinate System

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.

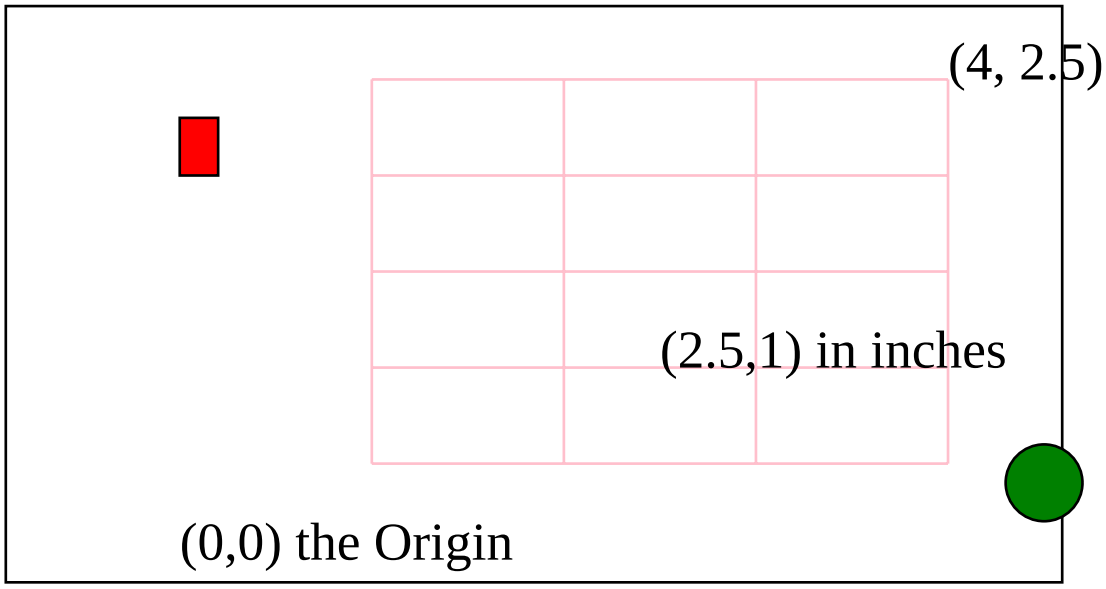


Figure 2-3: Moving the origin: the `translate` method

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.

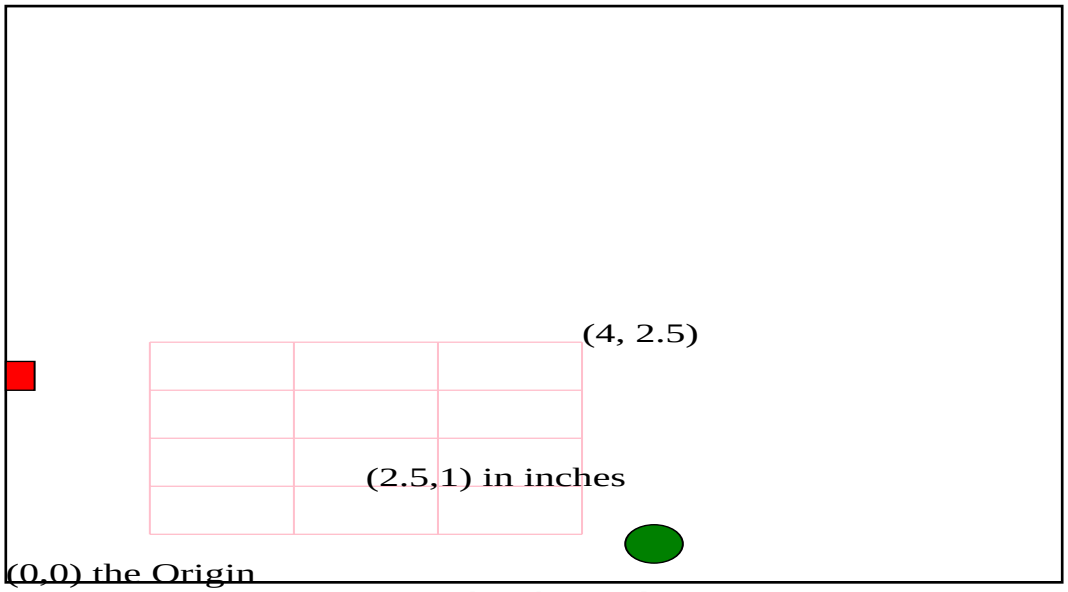


Figure 2-4: Scaling the coordinate system

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.

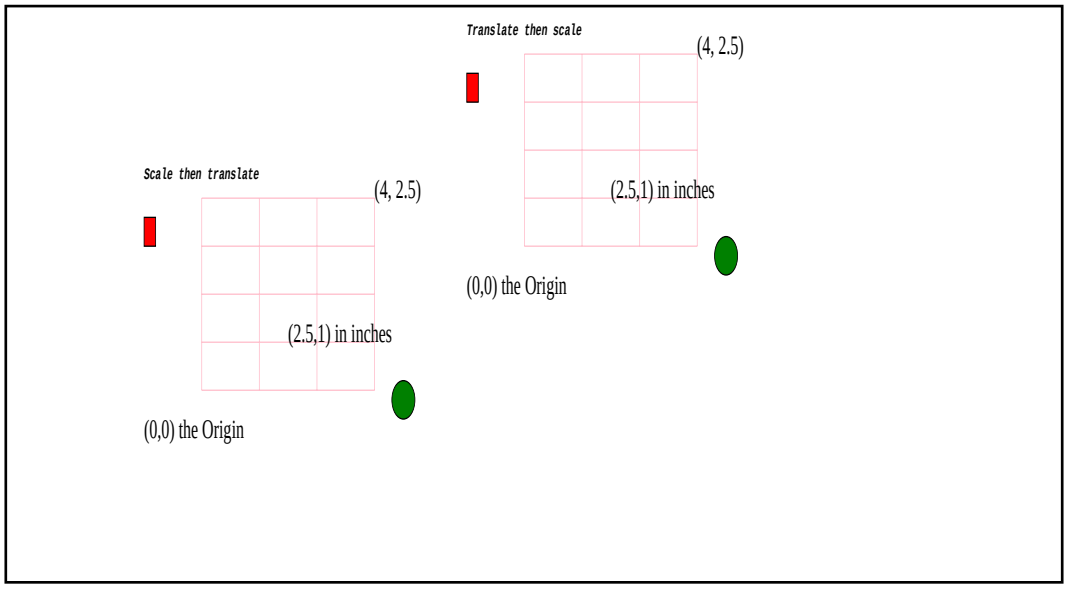


Figure 2-5: Scaling and Translating

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.

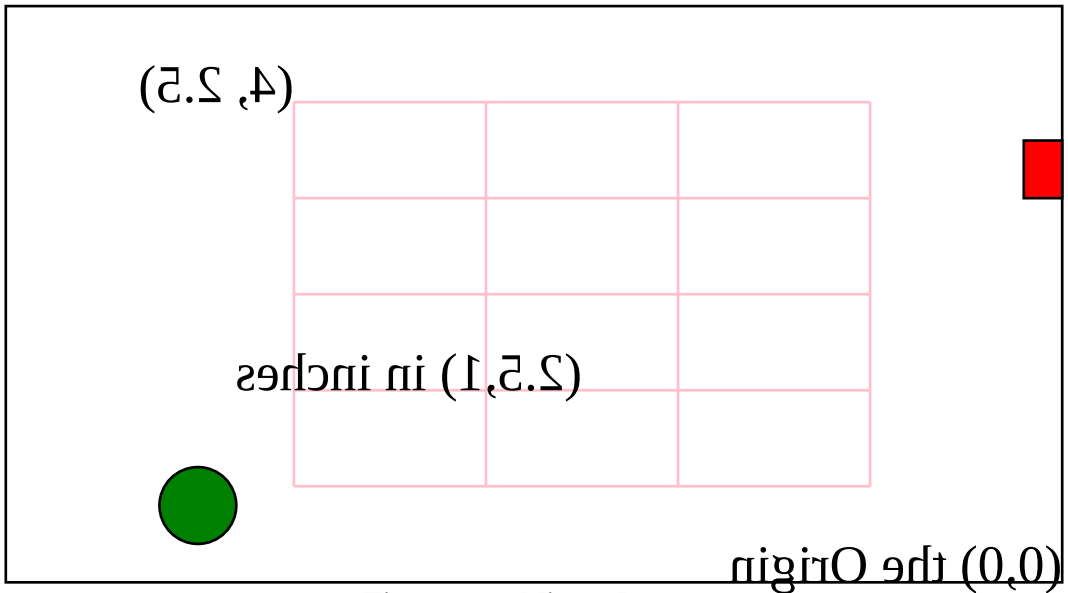


Figure 2-6: Mirror Images

Notice that the text strings are painted backwards.

2.8 Colors

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

```
def colors(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 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
    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
```

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

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't 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.

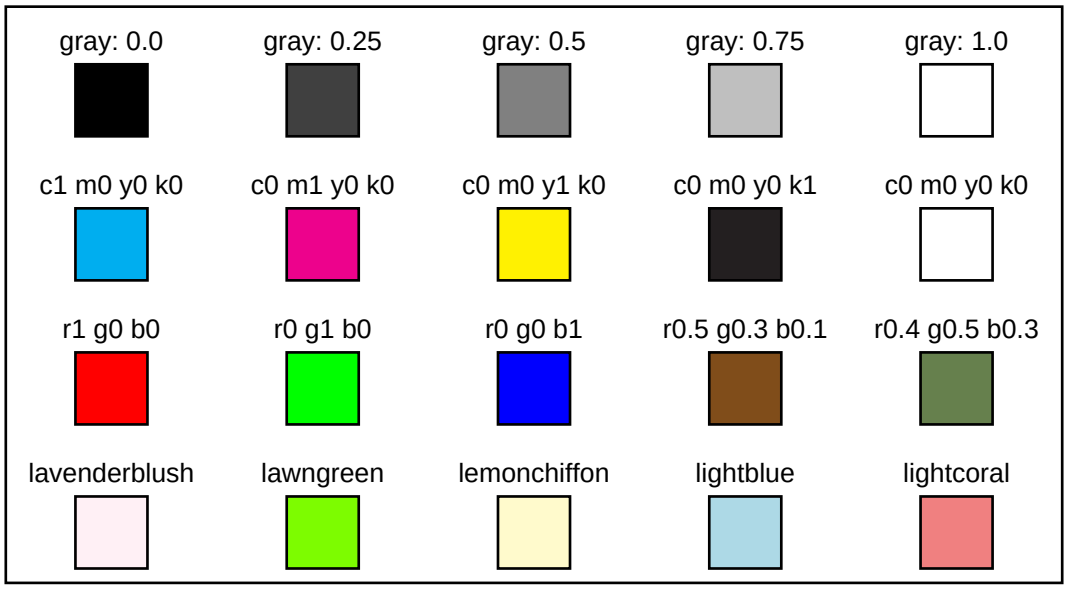


Figure 2-7: Color Models

2.9 Painting back to front

Objects may be painted over other objects to good effect in pdfgen. As in painting with oils the object painted last will show up on top. For example, the spumoni function below paints up a base of colors and then paints a white text over the base.

```
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")
```

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.

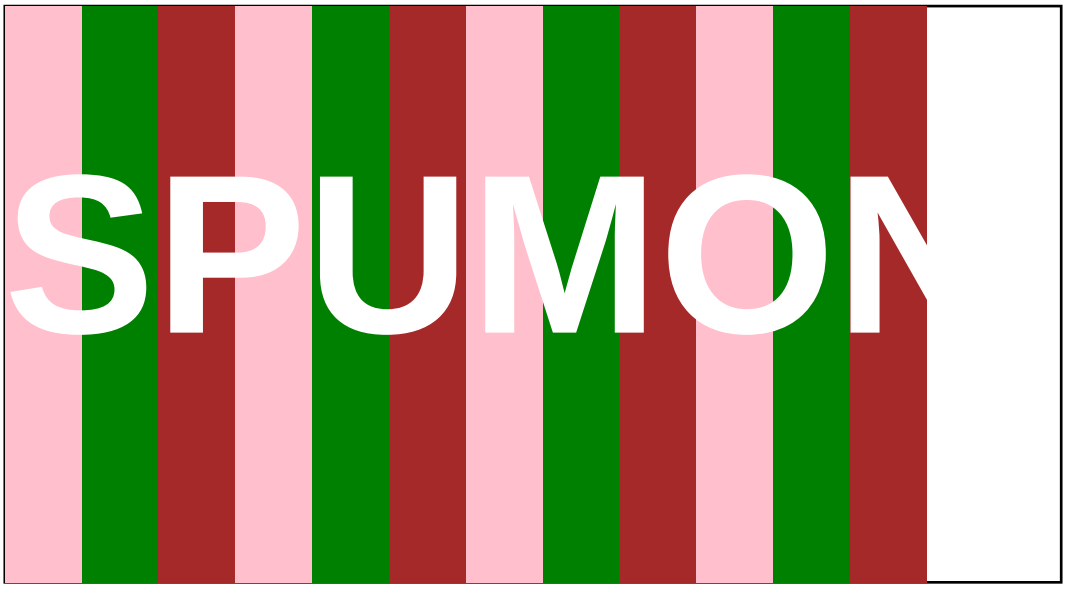


Figure 2-8: Painting over colors

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.

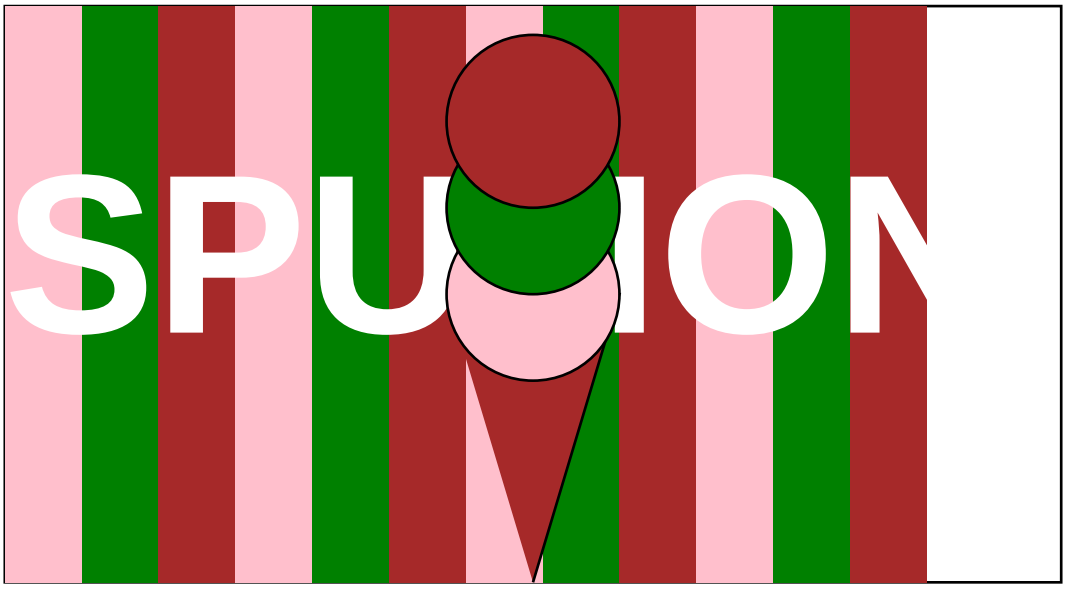


Figure 2-9: building up a drawing in layers

2.10 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.

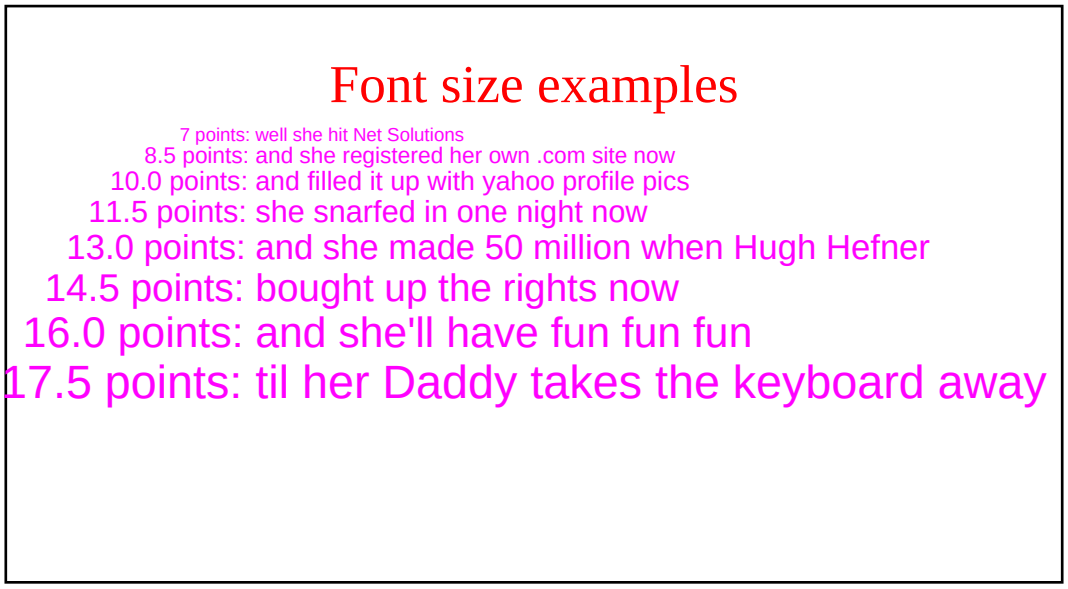


Figure 2-10: text in different fonts and sizes

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.

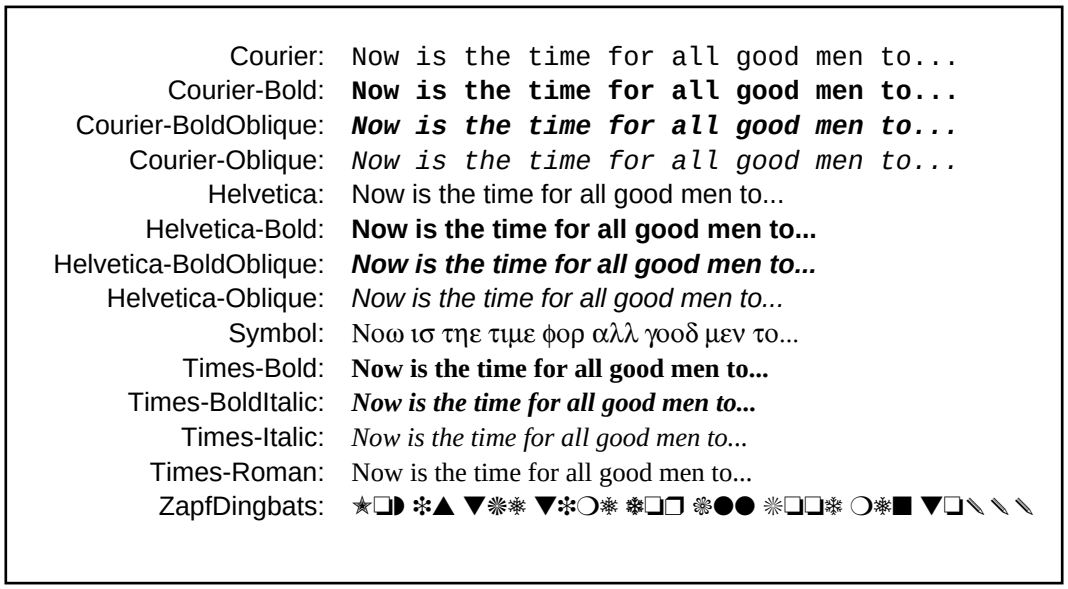


Figure 2-11: the 14 standard fonts

In the near future we will add the ability to embed other fonts within a PDF file. However, this will add a little to file size.

2.11 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 2-12: 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.

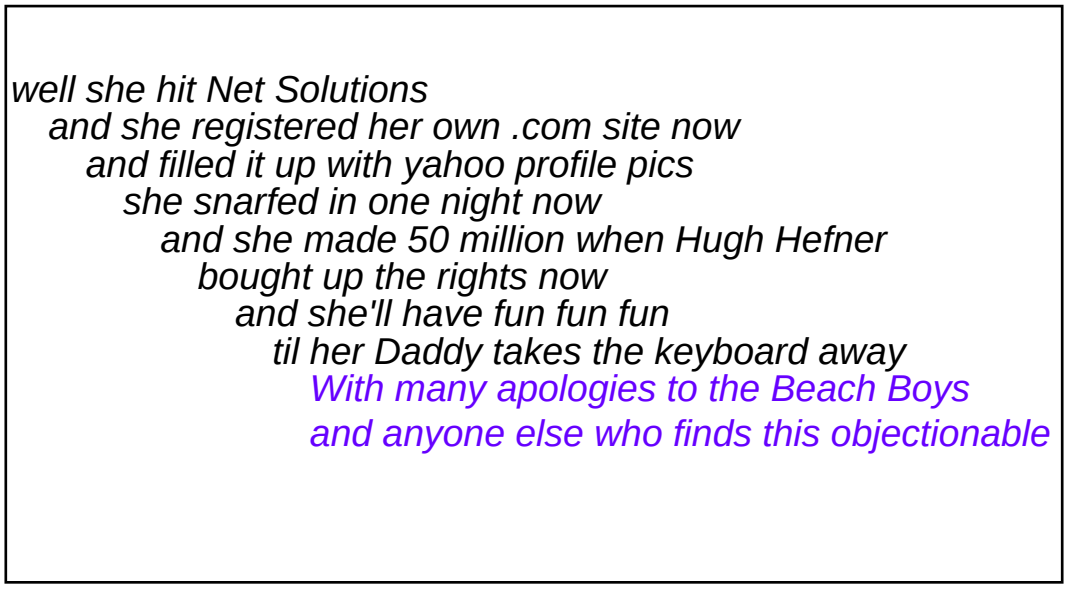


Figure 2-13: How the text cursor moves again

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.

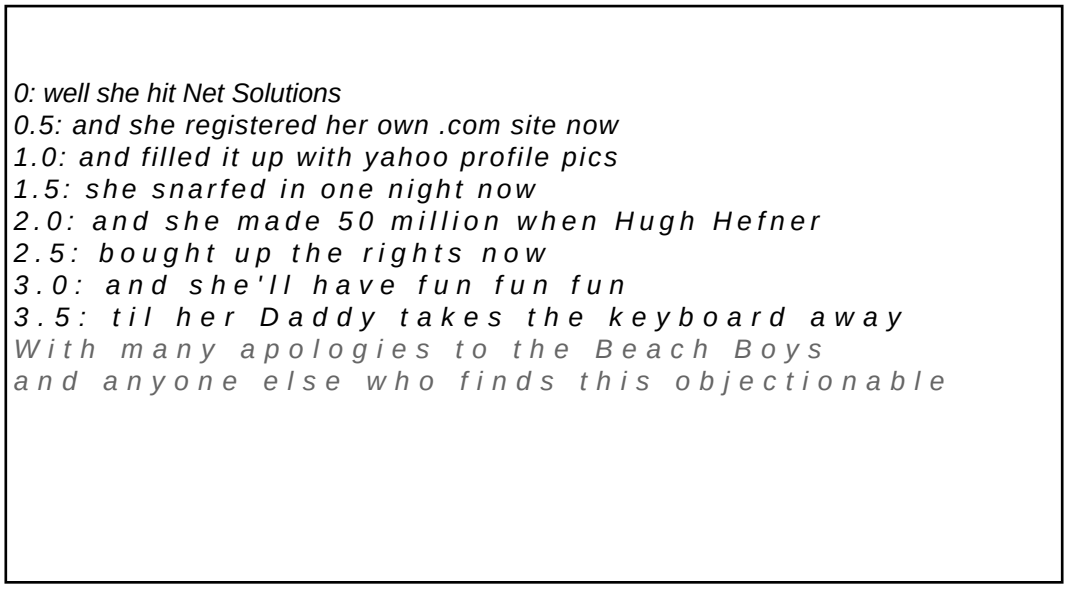


Figure 2-14: Adjusting inter-character spacing

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.

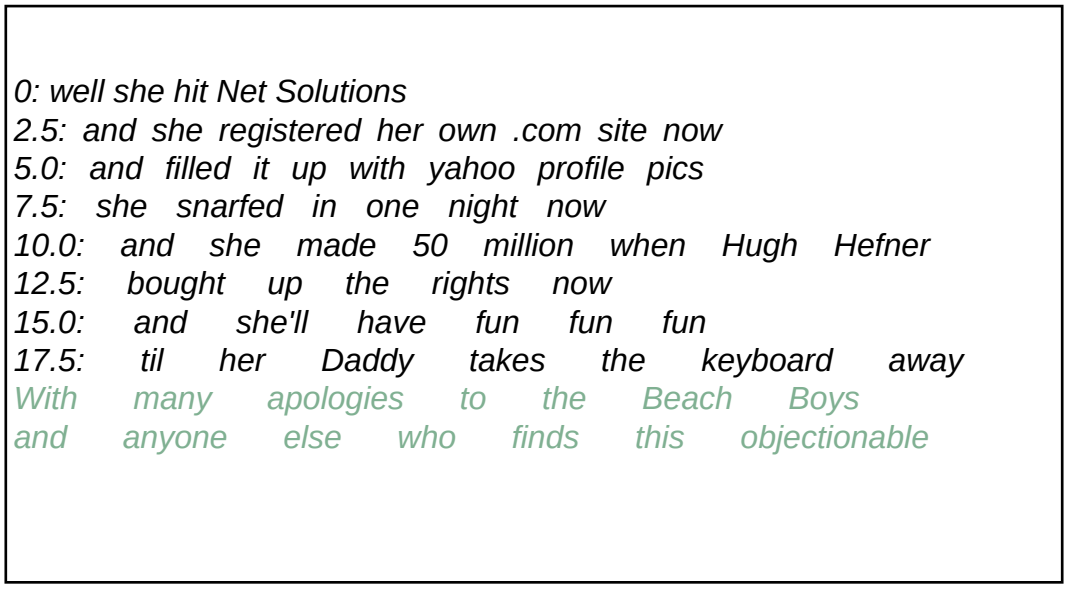


Figure 2-15: Adjusting word spacing

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.

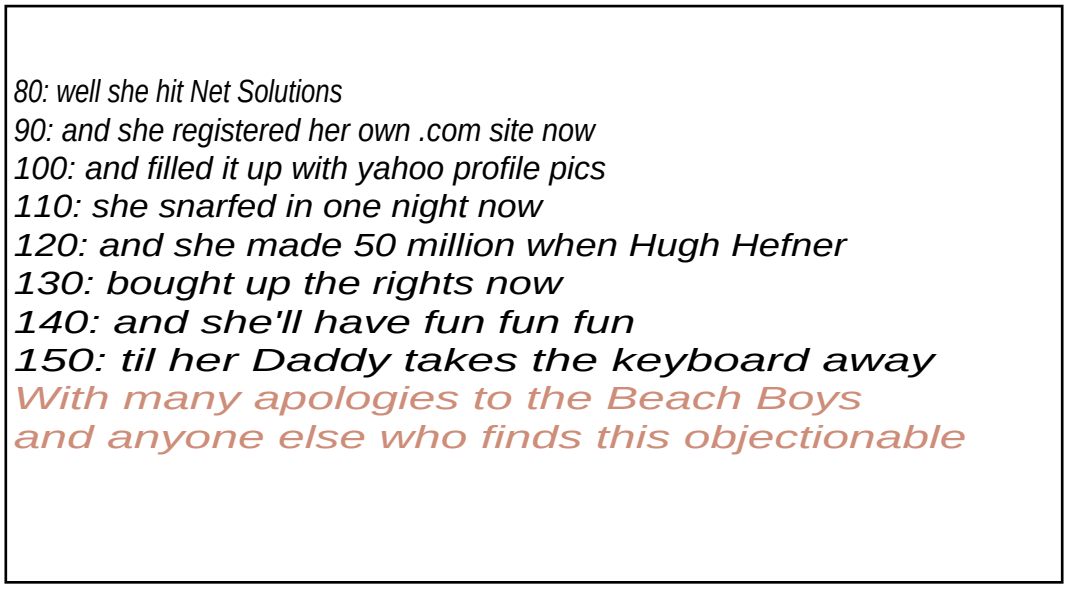


Figure 2-16: adjusting horizontal text scaling

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.

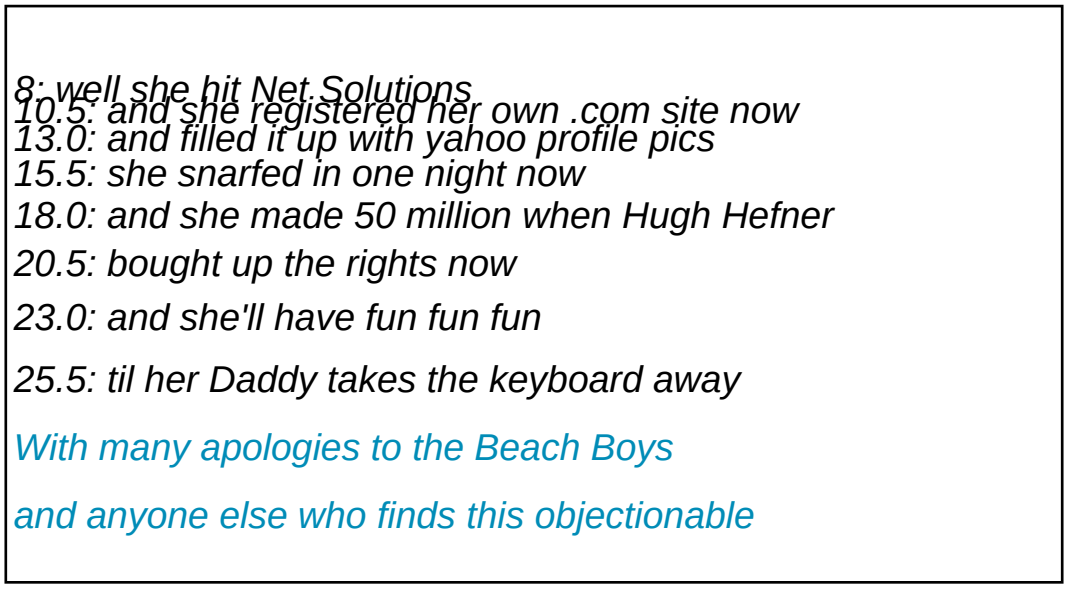


Figure 2-17: adjusting the leading

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.

2.12 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`.

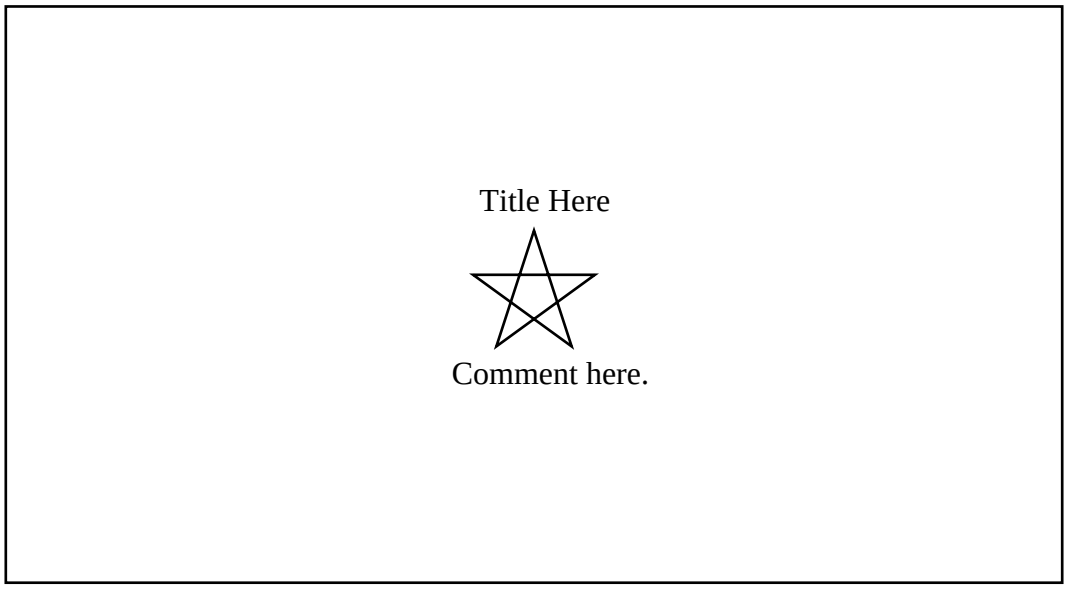


Figure 2-18: line style parameters

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.

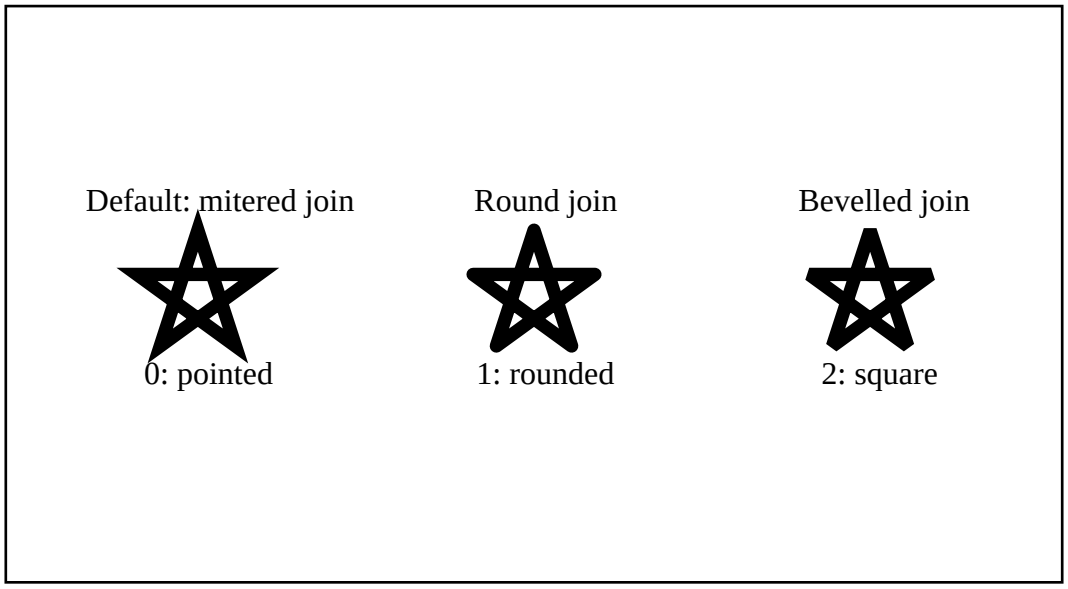


Figure 2-19: different line join styles

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.

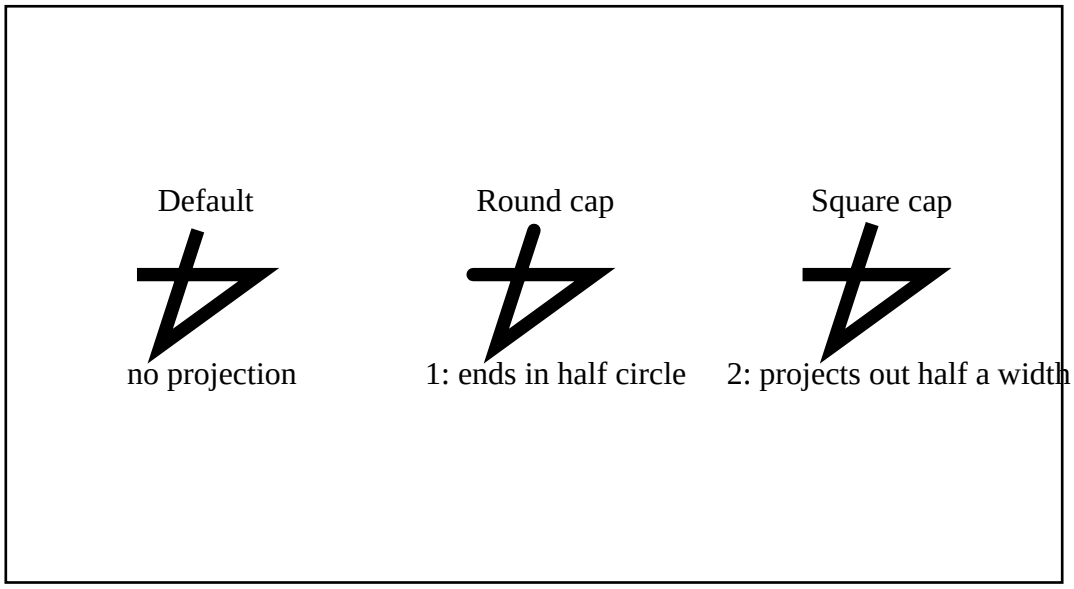


Figure 2-20: line cap settings

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.

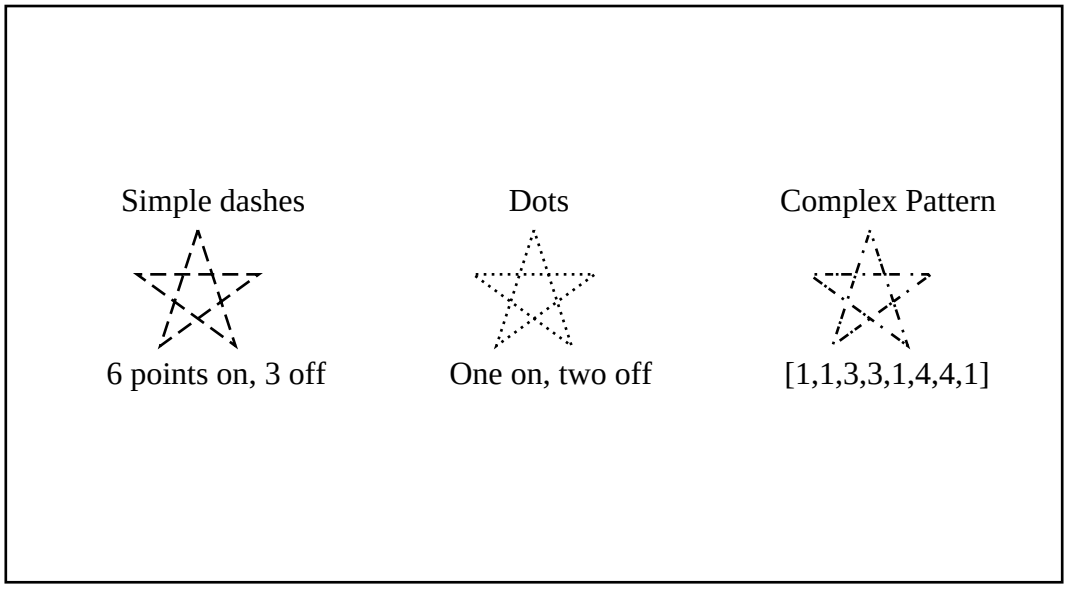


Figure 2-21: some dash patterns

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.

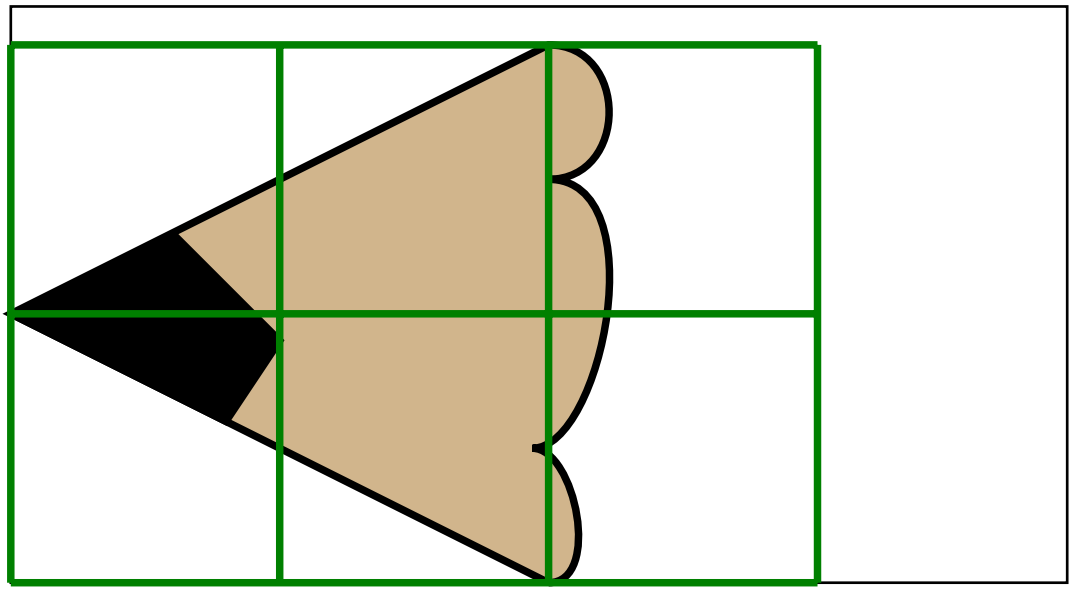


Figure 2-22: a pencil tip

2.13 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.

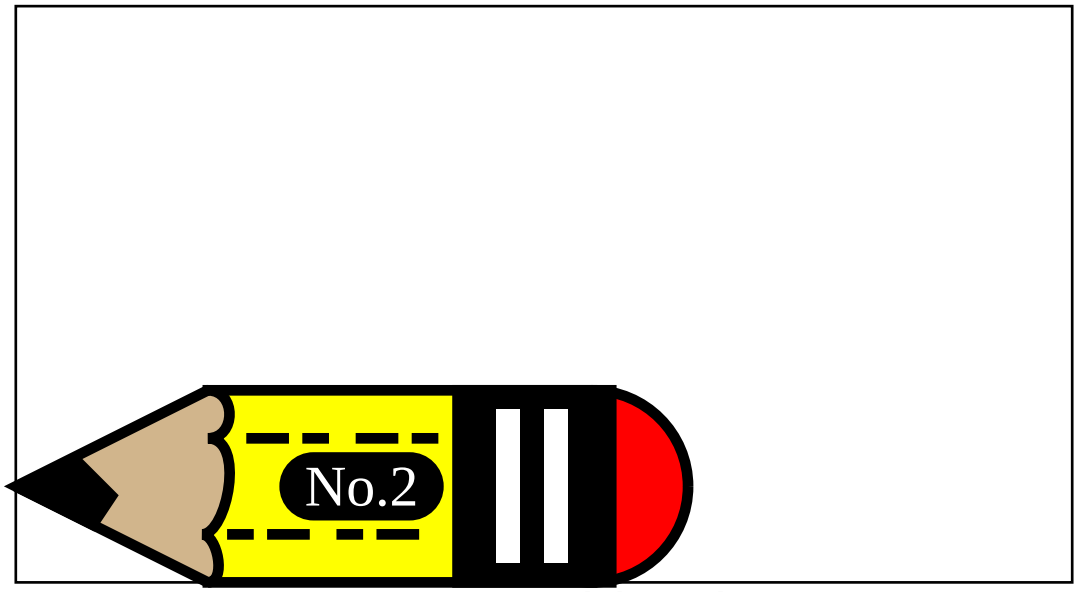


Figure 2-23: a whole pencil

2.14 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.

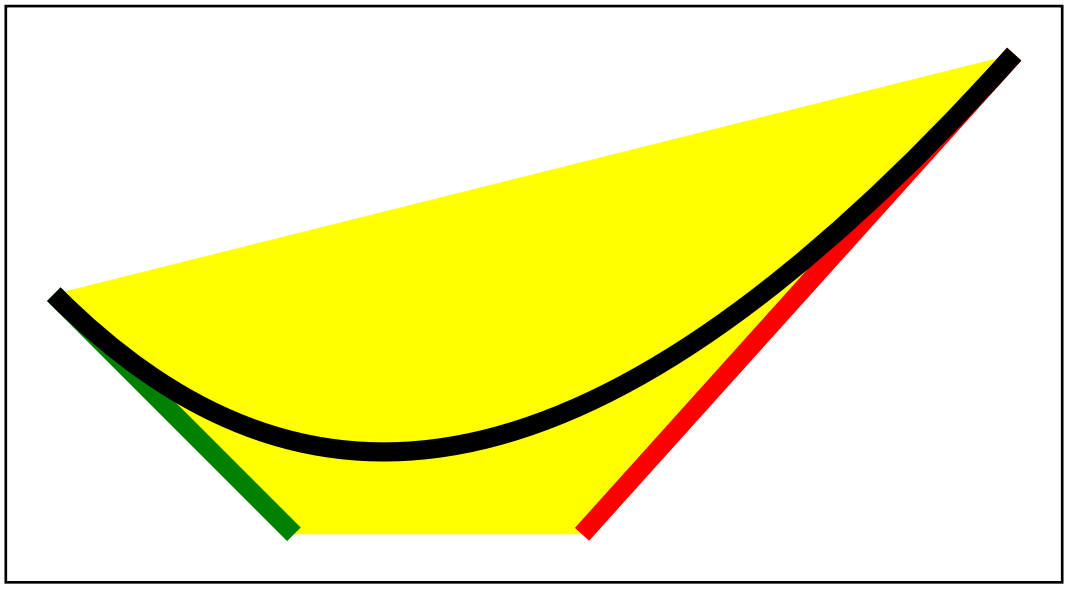


Figure 2-24: basic bezier curves

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.

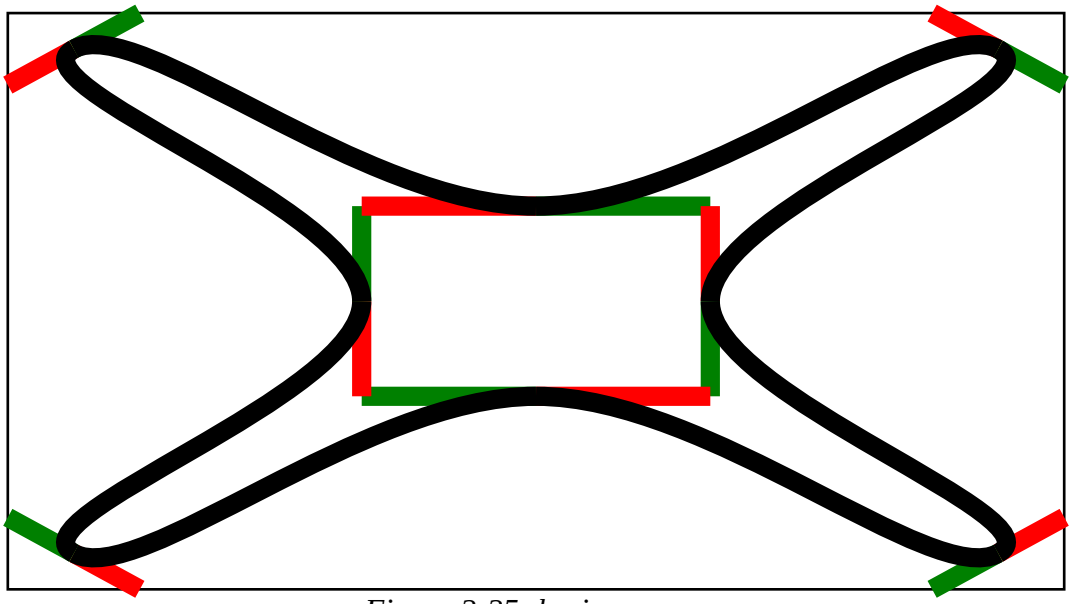


Figure 2-25: bezier curves

2.15 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.

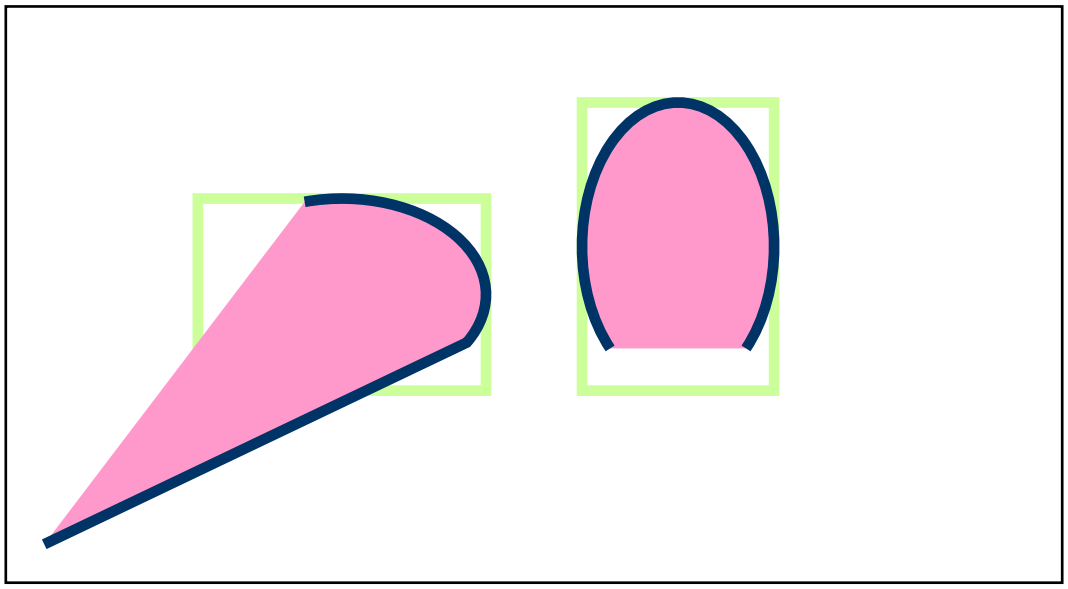


Figure 2-26: arcs in path objects

```
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(0,11*inch/2,inch/2), range(0,7*inch/2,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.

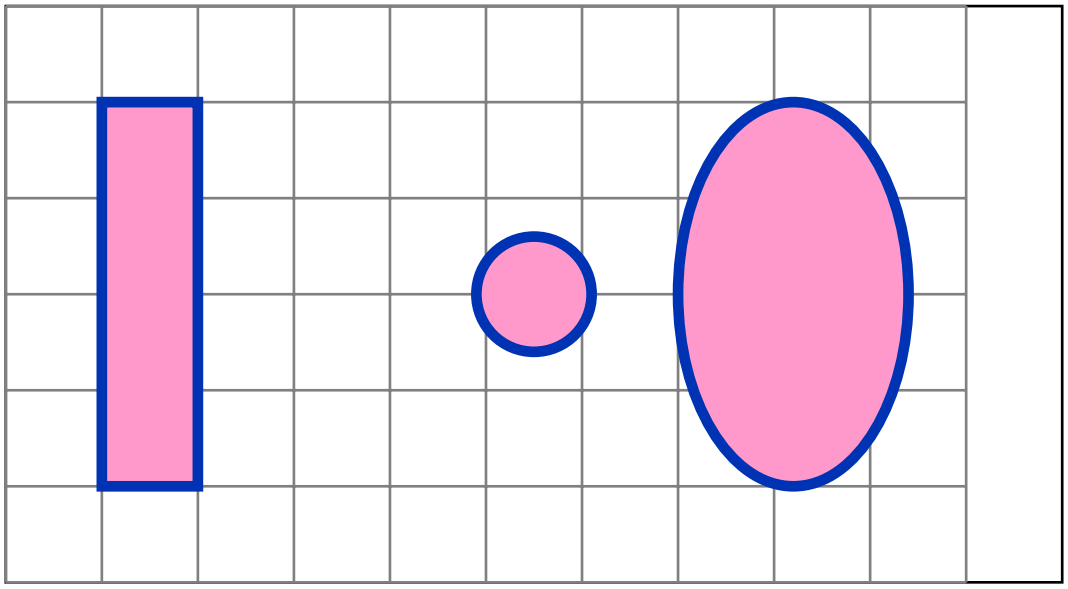


Figure 2-27: rectangles, circles, ellipses in path objects

```
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.

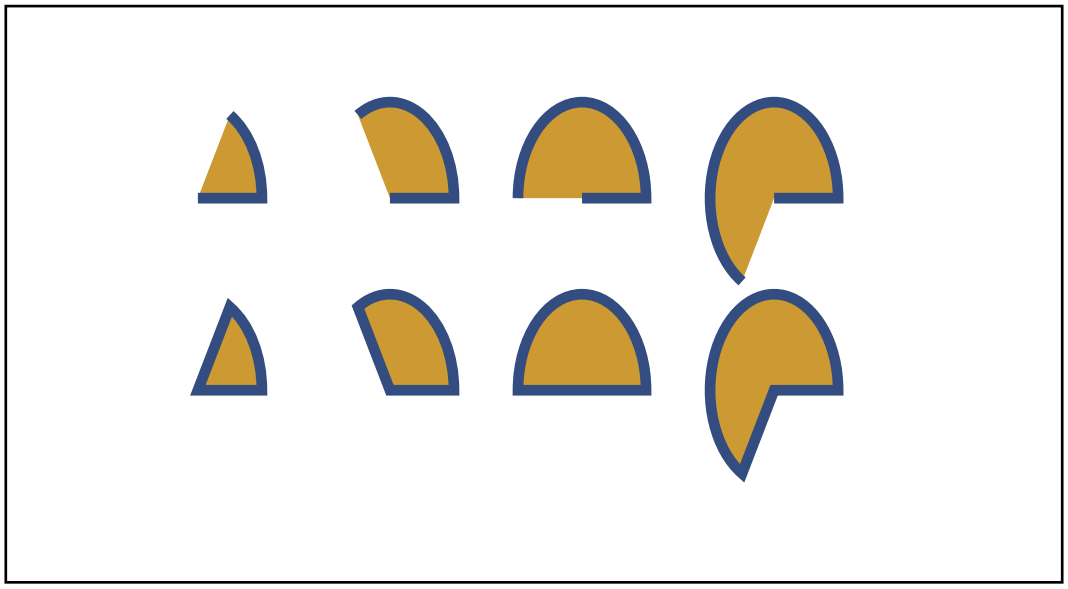


Figure 2-28: closing and not closing pathobject figures

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".

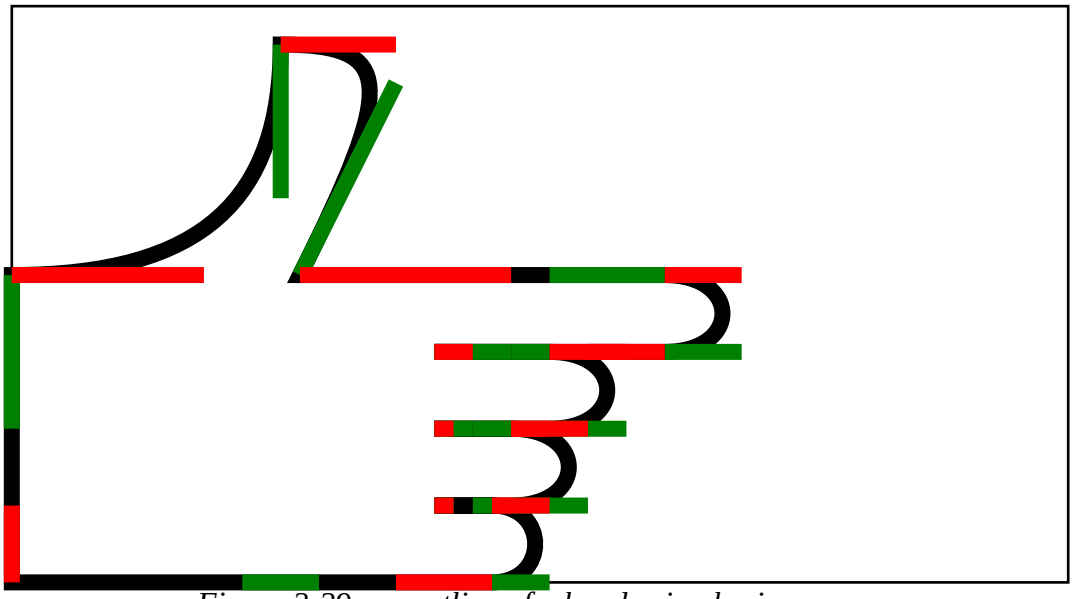


Figure 2-29: an outline of a hand using bezier curves

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.

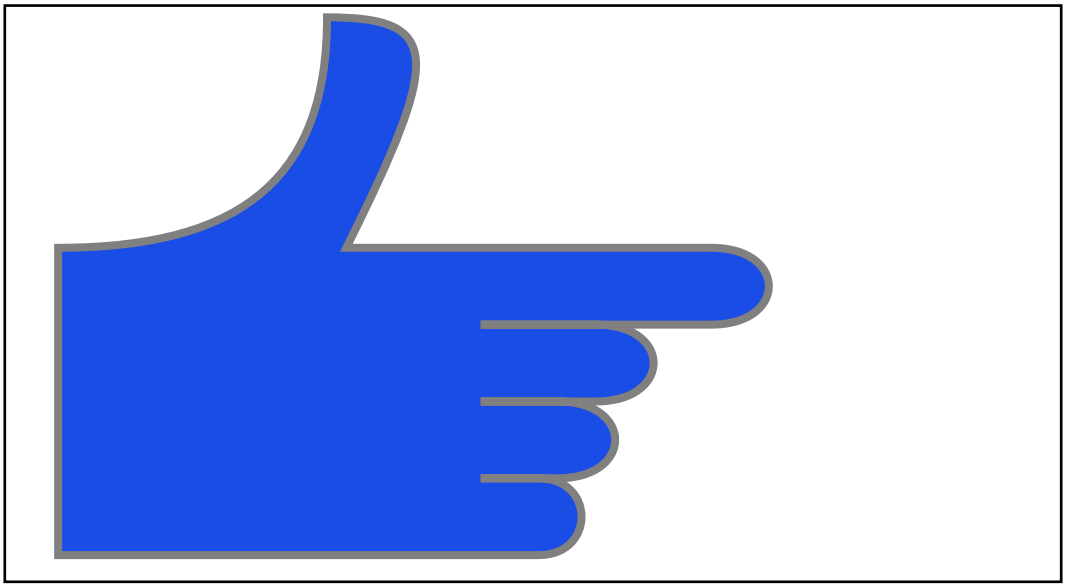


Figure 2-30: the finished hand, filled

Chapter 3 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.

3.1 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")
```

3.2 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. The bookmark methods define a destination that is the endpoint of a jump.

```
canvas.bookmarkPage(name)
canvas.bookmarkHorizontalAbsolute(name, yhorizontal)
```

The `bookmarkPage` method bookmarks the entire page. After jumping to an endpoint defined by `bookmarkPage` the PDF browser will display the whole page on the screen.

By contrast, `bookmarkHorizontalAbsolute` defines a destination associated with a horizontal position on a page. When the PDF browser jumps to a destination defined by `bookmarkHorizontalAbsolute` the screen will show a part of the page with the horizontal line at *`y=yhorizontal`* near the top, omitting parts of the rest of the page if appropriate.

Note: The horizontal position `yhorizontal` must be specified in terms of the *default user space*. In particular `bookmarkHorizontalAbsolute` ignores any modified geometric transform in effect in the canvas graphics state.

```
canvas.linkAbsolute(contents, destinationname, Rect=None, addtopage=1, name=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.bookmarkHorizontalAbsolute("Meaning_of_life", 5*inch)
```

defines horizontal location on the currently drawn page with the identifier `Meaning_of_life`. And the invocation (???)

```
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]')
```

3.3 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)
```

3.4 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.

3.5 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.

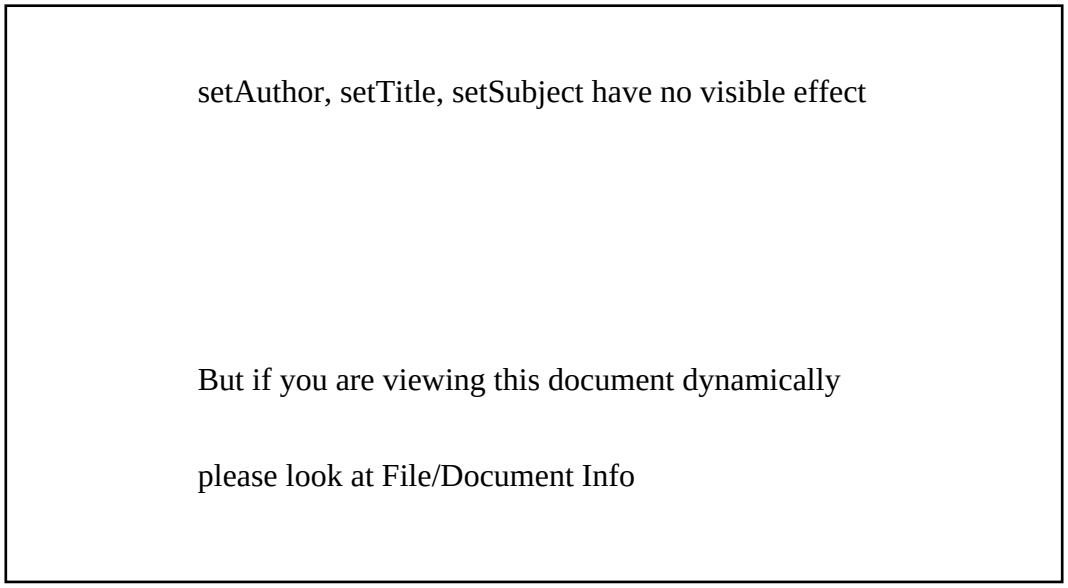


Figure 3-1: Setting document internal annotations

Chapter 4 PLATYPUS - Page Layout and Typography Using Scripts

4.1 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.

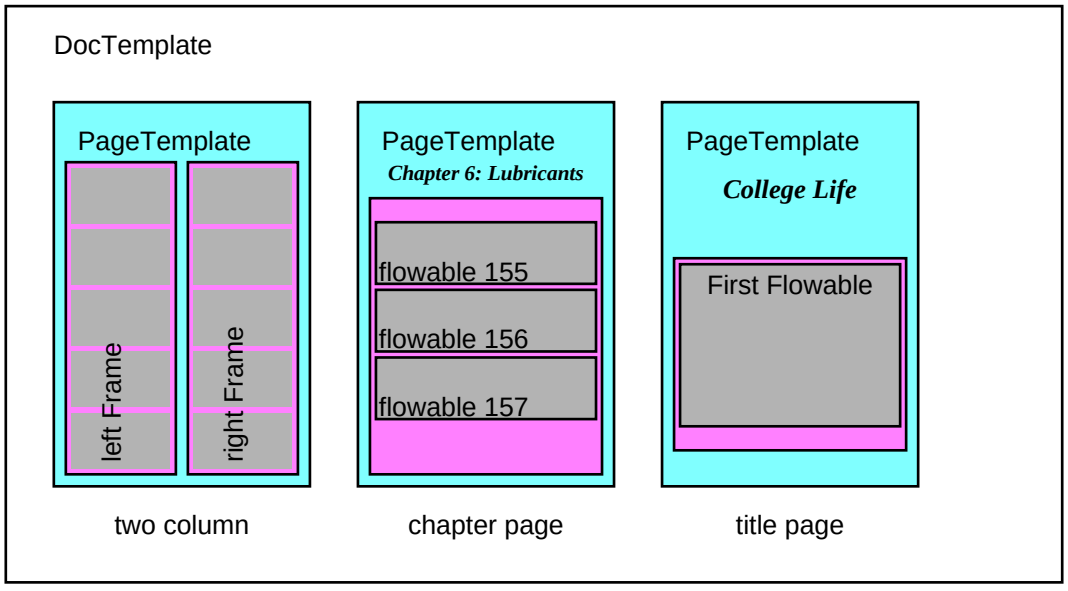


Figure 4-1: Illustration of DocTemplate structure

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 `Flowables` 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`.

4.2 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.lib.pagesizes import DEFAULT_PAGE_SIZE
from reportlab.lib.units import inch
PAGE_HEIGHT=DEFAULT_PAGE_SIZE[1]; PAGE_WIDTH=DEFAULT_PAGE_SIZE[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`.

4.3 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.

```
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.

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

4.4 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()
```

4.5 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=DEFAULT_PAGE_SIZE,
 pageTemplates=[],
 showBoundary=0,
 leftMargin=inch,
 rightMargin=inch,
 topMargin=inch,
 bottomMargin=inch,
 allowSplitting=1,
 title=None,
 author=None,
 _pageBreakQuick=1)
```

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 `StringIO` 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.

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 5 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
        }
```

5.1 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.

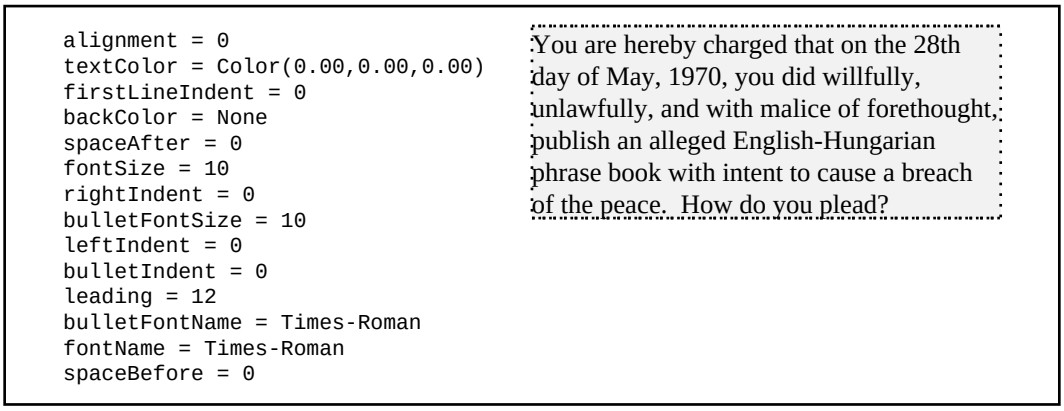


Figure 5-1: The default `ParagraphStyle`

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`.

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

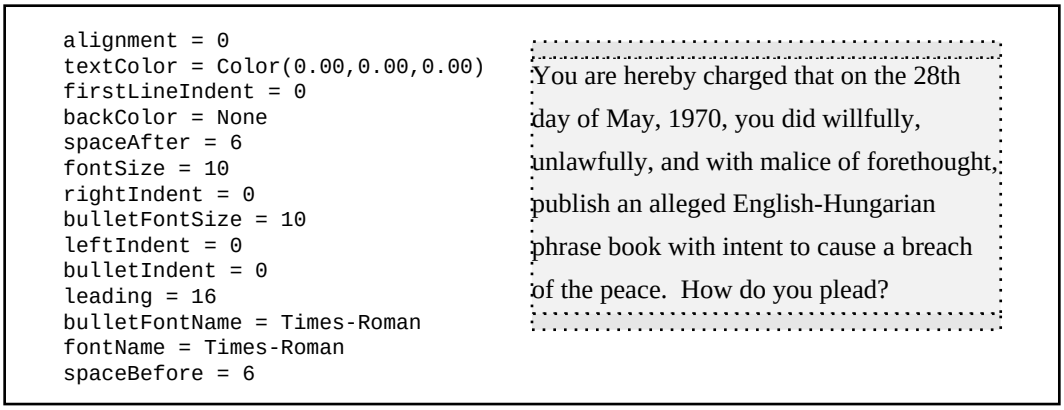


Figure 5-2: Space before and after and increased leading

The `firstLineIndent`, `leftIndent` and `rightIndent` attributes do exactly what you would expect. If you want a straight left edge, remember to set `firstLineIndent` equal to `leftIndent`.

```
alignment = 0
textColor = Color(0.00,0.00,0.00)
firstLineIndent = 48
backColor = None
spaceAfter = 0
fontSize = 10
rightIndent = 24
bulletFontSize = 10
leftIndent = 24
bulletIndent = 0
leading = 12
bulletFontName = Times-Roman
fontName = Times-Roman
spaceBefore = 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 5-3: one third inch indents at left and right, two thirds on first line

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

```
alignment = 0
textColor = Color(0.00,0.00,0.00)
firstLineIndent = 0
backColor = None
spaceAfter = 0
fontSize = 10
rightIndent = 0
bulletFontSize = 10
leftIndent = 36
bulletIndent = 0
leading = 12
bulletFontName = Times-Roman
fontName = Times-Roman
spaceBefore = 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 5-4: 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.

5.2 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	align, alignment
backColor	backcolor, bgcolor
bulletColor	bulletcolor, bcolor
bulletFontName	bfont, bulletfontname
bulletFontSize	bfontsize, bulletfontsize
bulletIndent	bindent, bulletindent
firstLineIndent	findent, firstlineindent

fontName	face, fontname, font
fontSize	size, fontsize
leading	leading
leftIndent	leftindent, lindent
rightIndent	rightindent, rindent
spaceAfter	spaceafter, spacea
spaceBefore	spacebefore, spaceb
textColor	fg, textcolor, color

Table 5-1 - 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.

5.3 Intra-paragraph markup

'Within each paragraph, we use a basic set of XML tags to provide markup. The most basic of these are bold (...) and italic (...). It is also legal to use an underline tag (<u>...</u> but it has no effect; PostScript fonts don't support underlining, and neither do we, yet.

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

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 5-5: Simple bold and italic 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.

```
<font face="times" color="red">
You are hereby charged</font>
that on the 28th day of May,
1970, you did willfully,
unlawfully, and <font
size=14>with malice of
forethought</font>, publish an
alleged English-Hungarian phrase
book with intent to cause a
breach of the peace.  How do you
plead?
```

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 5-6: The `font` tag

Superscripts and Subscripts

Superscripts and subscripts are supported with the <super> and <sub> tags, which work exactly as you might expect. In addition, most greek letters can be accessed by using the <greek></greek> tag, or with mathML entity names.

```
Equation (&alpha;):
<greek>e</greek>
<super><greek>ip</greek></super>
 = -1
```

Equation (α): ε $^{i\pi}$ = -1

Figure 5-7: Greek letters and superscripts

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:

```
<seq id="spam"/>, <seq
id="spam"/>, <seq id="spam"/>.
Reset<seqreset id="spam"/>.
<seq id="spam"/>, <seq
id="spam"/>, <seq id="spam"/>.
```

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

Figure 5-8: 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.

```
<seqdefault
id="spam"/>Continued... <seq/>,
<seq/>, <seq/>, <seq/>, <seq/>,
<seq/>, <seq/>.
```

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

Figure 5-9: 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 <seq
template="%(Chapter)s-%(FigureNo+)s"/>
- Multi-level templates
```

Figure 5-1 - Multi-level templates

Figure 5-10: 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!

5.4 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 <bullet>..</bullet> tag at its head. The 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. For genuine bullets, a good idea is to select the Symbol font in the style, and use a character such as \267):

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

Table 5-2 - <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.

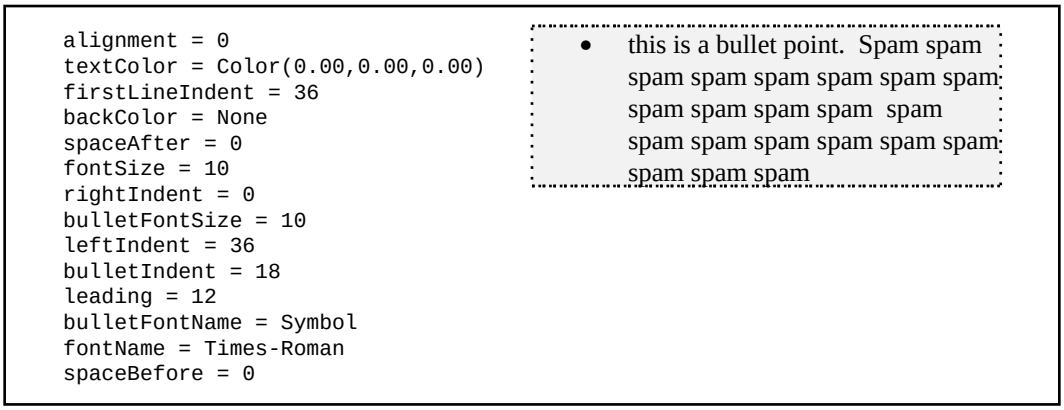


Figure 5-11: Basic use of bullet points

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 6 Tables and TableStyles

The `Table` class is derived from the `Flowable` class and is intended as a simple textual gridding mechanism. `Table` cells can hold anything which can be converted to a **Python** `string`.

`Tables` are created by passing the constructor a sequence of column widths, a 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.

6.1 `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 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.

The `repeatRows` and `repeatCols` arguments specify the number of leading rows and columns that should be repeated when the `Table` is asked to split itself.

`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.

6.2 `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.

6.3 `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()
```

6.4 `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).
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.
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
I.noImageCaching = 1
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

Chapter 7 Other Useful Flowables

7.1 Preformatted(text, style, bulletText = None, dedent=0)

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.

7.2 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['Normal']
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 & style entities as well for & < > and ".

7.3 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

```
Image("lj8100.jpg", width=2*inch, height=2*inch)
```

produces

7.4 `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.

7.5 `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.

7.6 `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`.

7.7 `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.

Chapter 8 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.

8.1 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.

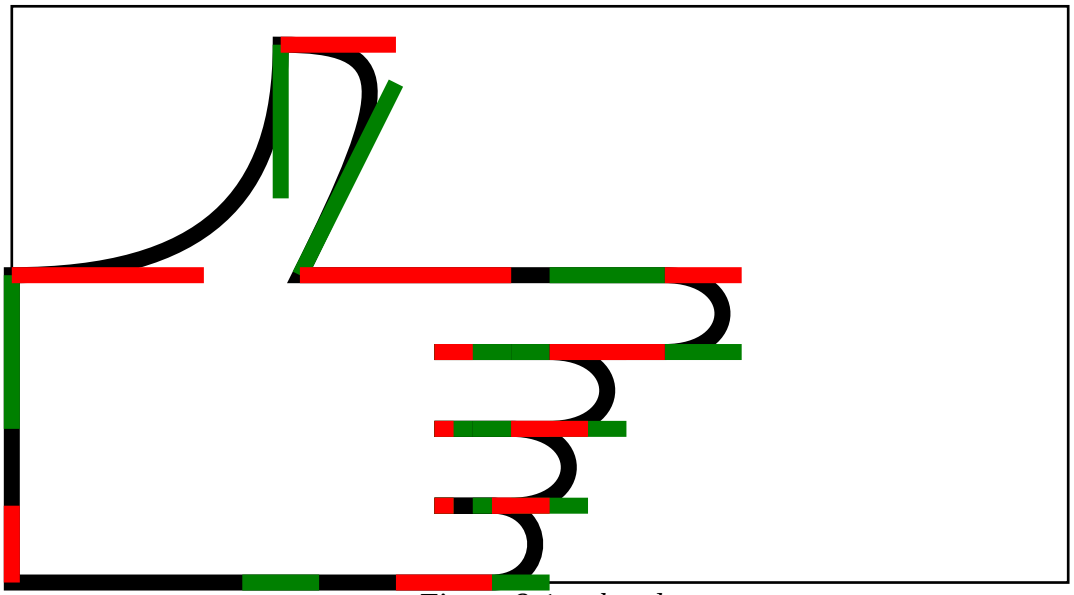


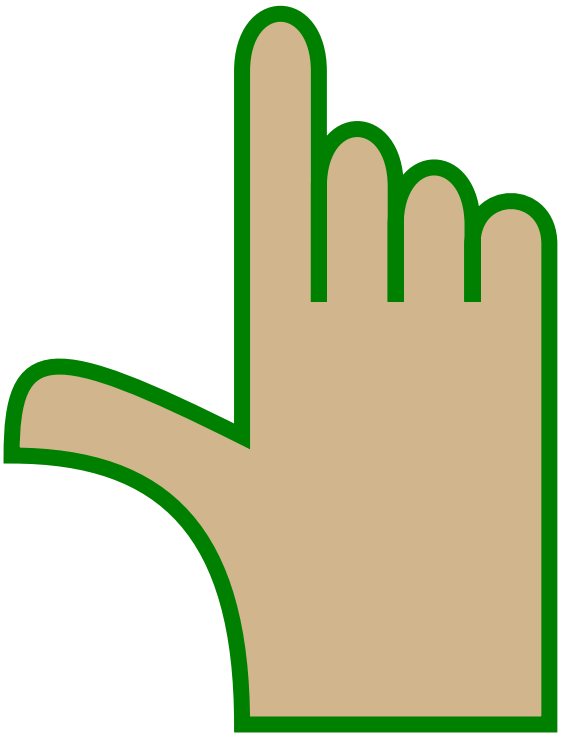
Figure 8-1: a hand

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.

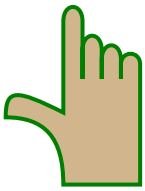
```
from reportlab.platypus.flowables import Flowable
from reportlab.lib.colors import tan, green
class HandAnnotation(Flowable):
    '''A hand flowable.'''
    def __init__(self, xoffset=0, size=None, fillcolor=tan, strokecolor=green):
        from reportlab.lib.units import inch
        if size is 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.

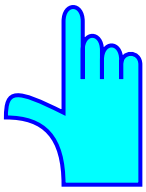
Below are some example uses of the `HandAnnotation` flowable.



The default.



Just one inch high.



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

8.2 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.noImageCaching = 1
```

produces

Chapter 9 Future Directions

We have a very long list of things we plan to do and what we do first will most likely be inspired by customer or user interest.

We plan to provide a large number of pre-designed Platypus example document types -- brochure, newsletter, business letter, thesis, memo, etcetera, to give our users a better boost towards the solutions they desire.

We plan to fully support adding fonts and internationalization, which are not well supported in the current release.

We plan to fully support some of the more obscure features of PDF such as general hyperlinks, which are not yet well supported.

We are also open for suggestions. Please let us know what you think is missing. You can also offer patches or contributions. Please look to `http://www.reportlab.com` for the latest mailing list and contact information.

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 (<gherman@europemail.com>) 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