



ReportLab PDF Library

User Guide

ReportLab Version 2.0

Chapter 1 Introduction

1.1 About this document

This document is an introduction to the ReportLab PDF library. Some previous programming experience is presumed and familiarity with the Python Programming language is recommended. If you are new to Python, we tell you in the next section where to go for orientation.

This manual does not cover 100% of the features, but should explain all the main concepts and help you get started, and point you at other learning resources. After working your way through this, you should be ready to begin writing programs to produce sophisticated reports.

In this chapter, we will cover the groundwork:

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

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

1.2 What is the ReportLab PDF Library?

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

PDF is the global standard for electronic documents. It supports high-quality printing yet is totally portable across platforms, thanks to the freely available Acrobat Reader. Any application which previously generated hard copy reports or driving a printer can benefit from making PDF documents instead; these can be archived, emailed, placed on the web, or printed out the old-fashioned way. However, the PDF file format is a complex indexed binary format which is impossible to type directly. The PDF format specification is more than 600 pages long and PDF files must provide precise byte offsets -- a single extra character placed anywhere in a valid PDF document can render it invalid. This makes it harder to generate than HTML.

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

The ReportLab library directly creates PDF based on your graphics commands. There are no intervening steps. Your applications can generate reports extremely fast - sometimes orders of magnitude faster than traditional report-writing tools. This approach is shared by several other libraries - PDFlib for C, iText for Java, iTextSharp for .NET and others. However, The ReportLab library differs in that it can work at much higher levels, with a full featured engine for laying out documents complete with tables and charts.

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

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

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

1.3 What is Python?

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

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

Python is as old as Java and has been growing steadily in popularity for 13 years; since our library first came out it has entered the mainstream. Many ReportLab library users are already Python devotees, but if you are not, we feel that the language is an excellent choice for document-generation apps because of its expressiveness and ability to get data from anywhere.

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

1.4 Acknowledgements

Many people have contributed to ReportLab. We would like to thank in particular (in approximately chronological order) Chris Lee, Magnus Lie Hetland, Robert Kern, Jeff Bauer (who contributed `normalDate.py`); Jerome Alet (numerous patches and the `rlzope` demo), Andre Reitz, Max M, Albertas Agejevas, T Blatter, Ron Peleg, Gary Poster, Steve Halasz, Andrew Mercer, Paul McNett, Chad Miller, Tim Roberts, Jorge Godoy and Benn B.

Special thanks go to Just van Rossum for his valuable assistance with font technicalities and the LettErrorRobot-Chrome type 1 font.

Marius Gedminas deserves a big hand for contributing the work on TrueType fonts and we are glad to include these in the toolkit. Finally we thank Bigelow & Holmes Inc (design@bigelowandholmes.com) for Luxi Serif Regular and Ray Larabie (<http://www.larabiefonts.com>) for the Rina TrueType font.

1.5 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 2.3 or later (2.4 recommended). ReportLab 2.x uses Python 2.3 features and will use 2.4 going forwards. We also maintain a 1.x branch which works back to Python 2.1.
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. (You can also use `python setup.py install` if you wish)
5. Unpack the `rl_addons` package and build the C extensions with `distutils`; or grab the corresponding `.pyd` files from our download page.
6. `cd` to `reportlab/test` and execute `runAll.py`. This will create many PDF files.
7. You may also want to download and run the `rl_check.py` on our site, which health-checks an installation and reports on any missing options.

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 or at `http://www.reportlab.com/ftp/`. Each successive version is stored in both zip and tgz format, but the contents are identical apart from line endings. Versions are numbered: `ReportLab_1_00.zip`, `ReportLab_1_01.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. Finally, daily snapshots off the trunk are available as `current.zip` (or `current.tgz`).

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/`. Reportlab 2.x works with Python 2.3 upwards but we strongly recommend to use the latest stable version of Python (2.4.3 at the time of writing). Follow the links to 'Download' and get the latest official version. This will install itself into `C:\Python24`. After installing, you should be able to run the 'Python (command line)' option from the Start Menu.
2. If on Windows, we strongly recommend installing the Python Windows Extensions, which let you use access all the Windows data sources, and provide a very nice IDE. This can be found at `http://sourceforge.net/projects/pywin32/`. Once this is installed, you can start Pythonwin from the Start Menu and get a GUI application.
3. The next step is optional and only necessary if you want to include images in your reports; it can also be carried out later. However we always recommend a full installation if time permits.
4. Install the Python Imaging Library (PIL) from `http://www.pythonware.com/products/pil/`.
5. Now you are ready to install reportlab itself. 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.
6. Download the zip file of precompiled DLLs for your Python version from the bottom of the `http://www.reportlab.org/downloads.html` downloads page, and unzip them into `C:\Python24\lib\site-packages` (or its equivalent for other Python versions).
7. Open up a MS-DOS command prompt and CD to `..\reportlab\test`. Enter `"runAll.py"`. You should see lots of dots and no error messages. This will also create many PDF files and generate the manuals in `reportlab/docs` (including this one).
8. Finally, we recommend you download and run the script `rl_check.py` from `http://www.reportlab.org/ftp/`. This will health-check all the above steps and warn you if anything is missing or mismatched.

Instructions for Python novices: Unix

1. On a large number of Unix and Linux distributions, Python is already installed, or is available as a standard package you can install with the relevant package manager.
2. 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/2.4.3/Python-2.4.3.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).
3. If you are building Python yourself, unpack the sources into a temporary directory using a tar command e.g. `tar xzvf Python-2.4.3.tgz`; this will create a subdirectory called `Python-2.4.3` (or whatever). cd into this directory. Then read the file `README`! It contains the latest information on how to install Python.
4. 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.

5. 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`.
6. 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.
7. 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 necessary 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
```

8. If you want to use images you should certainly consider getting & installing the Python Imaging Library - follow the directions from <http://www.python.org/sigs/image-sig/index.html> or get it directly from <http://www.pythonware.com/products/pil/>.

Instructions for Python novices: Mac

This is much, much easier with Mac OS X since Python (usually 2.3) is installed on your system as standard. Just follow the instructions for installing the ReportLab archive above.

Instructions for Jython (Java implementation of Python) users

A port to Java was done in 2004. This involved some changes to the framework and creating Java equivalents of the C extensions. At the end of this work the entire output of the test suite produced byte-for-byte identical output. However, we have not been testing against Jython since, because (a) as far as we know no one used it, and (b) Jython has not kept up with Python features which we need to use. We suggest you use ReportLab v1.19 or v1.20 which were Python-2.1 compatible. We'd welcome test reports and/or a volunteer to refresh things now that Jython is progressing.

The Jython version was tested under Sun's J2SDK 1.3.1. It is known that under J2SDK 1.4.0_01 `test_pdfbase_ttfonts.py` fails horribly with an `outOfMemory` exception, probably caused by a JVM bug.

1. Before installing Jython, make sure you have a supported version of Java Virtual Machine installed. For the list of supported JVM's see <http://www.jython.org/platform.html>
2. To install Jython, download the setup package from www.jython.org and follow installation instructions.
3. To set ReportLab toolkit under Jython `PATH`, edit `JYTHON_HOME/registry` file and include line that tells Jython where to look for packages. To include ReportLab toolkit under Jython `PATH`, directory that contains Reportlab should be included:
`python.path=REPORTLAB_HOME_PARENT_DIR` For example, if your Reportlab toolkit is installed under `C:\code\reportlab` the path line should be:
`python.path=C:\\code` (note two backslashes!)

Instructions for IronPython (Python for .NET) users

We haven't tackled this yet officially, but IronPython can apparently run much of our code. We do need to go through the same exercises we did for Jython - finding the .NET equivalents of `_rl_accel`, `pyRXP`, `_renderPM` and `PIL` - to get 100% managed code. Hopefully this will happen soon and we'd be delighted to

work with anyone on this.

1.6 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. To Subscribe visit <http://two.pairlist.net/mailman/listinfo/reportlab-users>. From there you can also browse through the group's archives and contributions. The mailing list is the place to report bugs and get support.

1.7 Site Configuration

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

- `verbose`: set to integer values to control diagnostic output.
- `shapeChecking`: set this to zero to turn off a lot of error checking in the graphics modules
- `defaultEncoding`: set this to `WinAnsiEncoding` or `MacRomanEncoding`.
- `defaultPageSize`: set this to one of the values defined in `reportlab/lib/pagesizes.py`; as delivered it is set to `pagesizes.A4`; other values are `pagesizes.letter` etc.
- `defaultImageCaching`: set to zero to inhibit the creation of .a85 files on your hard-drive. The default is to create these preprocessed PDF compatible image files for faster loading
- `T1SearchPath`: this is a python list of strings representing directories that may be queried for information on Type 1 fonts
- `TTFSearchPath`: this is a python list of strings representing directories that may be queried for information on TrueType fonts
- `CMapSearchPath`: this is a python list of strings representing directories that may be queried for information on font code maps.
- `showBoundary`: set to non-zero to get boundary lines drawn.
- `ZLIB_WARNINGS`: set to non-zero to get warnings if the Python compression extension is not found.
- `pageComression`: set to non-zero to try and get compressed PDF.
- `allowtableBoundsErrors`: set to 0 to force an error on very large Platypus table elements
- `emptyTableAction`: Controls behaviour for empty tables, can be 'error' (default), 'indicate' or 'ignore'.

1.8 Learning More About Python

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

- **Introductory Material on Python.** A list of tutorials on the Python.org web site.
<http://www.python.org/doc/Intros.html>

- **Python Tutorial.** The official Python Tutorial by Guido van Rossum (edited by Fred L. Drake, Jr.) <http://www.python.org/doc/tut/>
- **Learning to Program.** A tutorial on programming by Alan Gauld. Has a heavy emphasis on Python, but also uses other languages.
<http://www.freenetpages.co.uk/hp/alan.gauld/>
- **How to think like a computer scientist** (Python version).
<http://www.ibiblio.org/obp/thinkCSPy/>
- **Instant Python.** A 6-page minimal crash course by Magnus Lie Hetland.
<http://www.hetland.org/python/instant-python.php>
- **Dive Into Python.** A free Python tutorial for experienced programmers.
<http://diveintopython.org/>

1.9 What's New in ReportLab 2.0

Many new features have been added, foremost amongst which is the support for unicode. This page documents what has changed since version 1.20.

Adding full unicode support meant that we had to break backwards-compatibility, so old code written for ReportLab 1 will sometimes need changes before it will run correctly with ReportLab 2. Now that we have made the clean break to introduce this important new feature, we intend to keep the API backwards-compatible throughout the 2.* series.

Goals for the 2.x series

The main rationale for 2.0 was an incompatible change at the character level: to properly support Unicode input. Now that it's out we will maintain compatibility with 2.0. There are no pressing feature wishlists and new features will be driven, as always, by contributions and the demands of projects.

Our 1.x code base is still Python 2.1 compatible. The new version lets us move forwards with a baseline of Python 2.4 (2.3 will work too, for the moment, but we don't promise that going forwards) so we can use newer language features freely in our development.

One area where we do want to make progress from release to release is with documentation and installability. We'll be looking into better support for distutils, setuptools, eggs and so on; and into better examples and tools to help people learn what's in the (substantial) code base.

Bigger ideas and more substantial rewrites are deferred to Version 3.0, with no particular target dates.

Contributions

Thanks to everybody who has contributed to the open-source toolkit in the run-up to the 2.0 release, whether by reporting bugs, sending patches, or contributing to the reportlab-users mailing list. Thanks especially to the following people, who contributed code that has gone into 2.0: Andre Reitz, Max M, Albertas Agejevas, T Blatter, Ron Peleg, Gary Poster, Steve Halasz, Andrew Mercer, Paul McNett, Chad Miller.

If we missed you, please let us know!

Unicode support

This is the Big One, and the reason some apps may break. You must now pass in text either in UTF-8 or as unicode string objects. The library will handle everything to do with output encoding. There is more information on this below. Since this is the biggest change, we'll start by reviewing how it worked in the past.

In ReportLab 1.x, any string input you passed to our APIs was supposed to be in the same encoding as the font you selected for output. If using the default fonts in Acrobat Reader (Helvetica/Times/Courier), you would have implicitly used WinAnsi encoding, which is almost exactly the same as Latin-1. However, if using TrueType fonts, you would have been using UTF-8.

For Asian fonts, you had a wide choice of encodings but had to specify which one (e.g Shift-JIS or EUC for Japanese). This state of affairs meant that you had to make sure that every piece of text input was in the same encoding as the font used to display it.

Input text encoding is UTF-8 or Python Unicode strings

Any text you pass to a canvas API (drawString etc.), Paragraph or other flowable constructor, into a table cell, or as an attribute of a graphic (e.g. chart.title.text), is supposed to be unicode. If you use a traditional Python string, it is assumed to be UTF-8. If you pass a Unicode object, we know it's unicode.

Font encodings

Fonts still work in different ways, and the built-in ones will still use WinAnsi or MacRoman internally while TrueType will use UTF-8. However, the library hides this from you; it converts as it writes out the PDF file. As before, it's still your job to make sure the font you use has the characters you need, or you may get either a traceback or a visible error character.

Asian CID fonts

You no longer need to specify the encoding for the built-in Asian fonts, just the face name. ReportLab knows about the standard fonts in Adobe's Asian Language Packs

Asian Truetype fonts

The standard Truetype fonts differ slightly for Asian languages (e.g. msmincho.ttc). These can now be read and used, albeit somewhat inefficiently.

Asian word wrapping

Previously we could display strings in Asian languages, but could not properly wrap paragraphs as there are no gaps between the words. We now have a basic word wrapping algorithm.

unichar tag

A convenience tag, <unichar/> has also been added. You can now do ü or <unichar name='LATIN SMALL LETTER U WITH DIAERESIS'/> and get a lowercase u umlaut. Names should be those in the Unicode Character Database.

Accents, greeks and symbols

The correct way to refer to all non-ASCII characters is to use their unicode representation. This can be literal Unicode or UTF-8. Special symbols and Greek letters (collectively, "greeks") inserted in paragraphs using the greek tag (e.g. <greek>lambda</greek>) or using the entity references (e.g. λ) are now processed in a different way than in version 1.

Previously, these were always rendered using the Zapf Dingbats font. Now they are always output in the font you specified, unless that font does not support that character. If the font does not support the character, and the font you specified was an Adobe Type 1 font, Zapf Dingbats is used as a fallback. However, at present there is no fallback in the case of TTF fonts. Note that this means that documents that contain greeks and specify a TTF font may need changing to explicitly specify the font to use for the greek character, or you will see a black square in place of that character when you view your PDF output in Acrobat Reader.

Other New Features

PDF

Improved low-level annotation support for PDF "free text annotations"

FreeTextAnnotation allows showing and hiding of an arbitrary PDF "form" (reusable chunk of PDF content) depending on whether the document is printed or viewed on-screen, or depending on whether the mouse is hovered over the content, etc.

TTC font collection files are now readable"

ReportLab now supports using TTF fonts packaged in .TTC files

East Asian font support (CID and TTF)

You no longer need to specify the encoding for the built-in Asian fonts, just the face name. ReportLab knows about the standard fonts in Adobe's Asian Language Packs.

Native support for JPEG CMYK images

ReportLab now takes advantage of PDF's native JPEG CMYK image support, so that JPEG CMYK images are no longer (lossily) converted to RGB format before including them in PDF.

Platypus

Link support in paragraphs

Platypus paragraphs can now contain link elements, which support both internal links to the same PDF document, links to other local PDF documents, and URL links to pages on the web. Some examples:

Web links:

```
<link href="http://www.reportlab.com/">ReportLab<link>
```

Internal link to current PDF document:

```
<link href="summary">ReportLab<link>
```

External link to a PDF document on the local filesystem:

```
<link href="pdf:C:/john/report.pdf">ReportLab<link>
```

Improved wrapping support

Support for wrapping arbitrary sequence of flowables around an image, using `reportlab.platypus.flowables.ImageAndFlowables` (similar to `ParagraphAndImage`).

KeepInFrame

Sometimes the length of a piece of text you'd like to include in a fixed piece of page "real estate" is not guaranteed to be constrained to a fixed maximum length. In these cases, `KeepInFrame` allows you to specify an appropriate action to take when the text is too long for the space allocated for it. In particular, it can shrink the text to fit, mask (truncate) overflowing text, allow the text to overflow into the rest of the document, or raise an error.

Improved convenience features for inserting unicode symbols and other characters

lets you conveniently insert unicode characters using the standard long name or code point. Characters inserted with the `<greek>` tags (e.g. `<greek>λαμβδα`) or corresponding entity references (e.g. `<greek>λ`) support arbitrary fonts (rather than only Zapf Dingbats).

Improvements to Legending

Instead of manual placement, there is now an attachment point (N, S, E, W, etc.), so that the legend is always automatically positioned correctly relative to the chart. Swatches (the small sample squares of colour / pattern fill sometimes displayed in the legend) can now be automatically created from the graph data. Legends can now have automatically-computed totals (useful for financial applications).

More and better ways to place piechart labels

New smart algorithms for automatic pie chart label positioning have been added. You can now produce nice-looking labels without manual positioning even for awkward cases in big runs of charts.

Adjustable piechart slice ordering

For example, pie charts with lots of small slices can be configured to alternate thin and thick slices to help the label placement algorithm work better.

Improved spiderplots

Noteworthy bug fixes

Fixes to TTF splitting (patch from Albertas Agejevas)

This affected some documents using font subsetting

Tables with spans improved splitting

Splitting of tables across pages did not work correctly when the table had row/column spans

Fix runtime error affecting `keepWithNext`

Chapter 2 Graphics and Text with pdfgen

2.1 Basic Concepts

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

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

A simple example program that uses a canvas follows.

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

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

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

2.2 More about the Canvas

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

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

```
def __init__(self,filename,
             pagesize=(595.27,841.89),
             bottomup = 1,
             pageCompression=0,
             encoding=rl_config.defaultEncoding,
             verbosity=0):
```

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 [GUT's]) place the origin at the top left. The `bottomup` argument is deprecated and may be dropped in future
Need to see if it really works for all tasks, and if not then get rid of it

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

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

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

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

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.setFillColorsRGB(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.setFillColorsRGB(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.

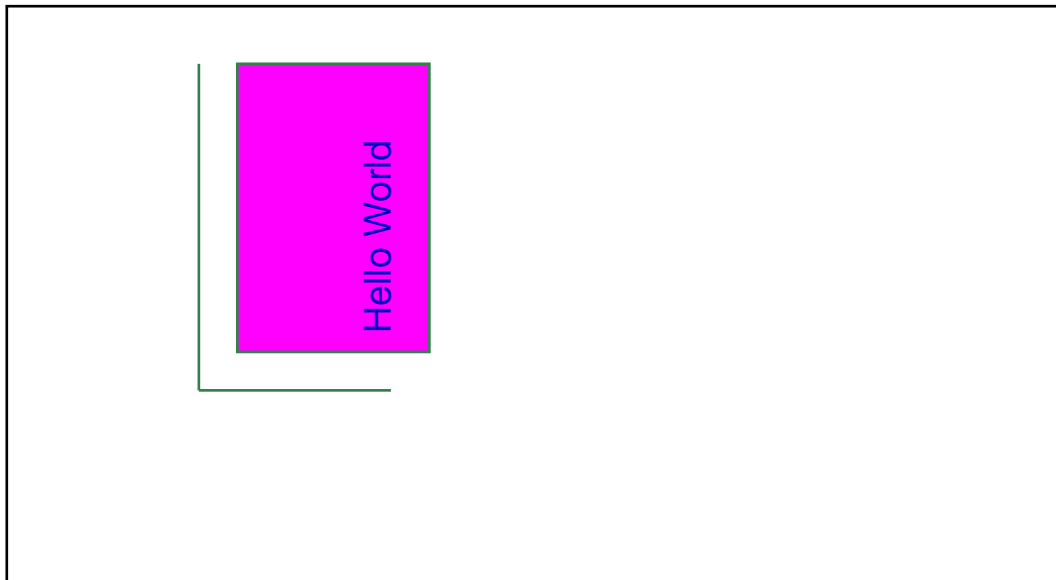


Figure A-1: "Hello World" in pdfgen

About the demos in this document

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

2.4 The tools: the "draw" operations

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

Line methods

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

```
canvas.lines(linelist)
```

The line methods draw straight line segments on the canvas.

Shape methods

```
canvas.grid(xlist, ylist)

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

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

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

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

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

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

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

The shape methods draw common complex shapes on the canvas.

String drawing methods

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

canvas.drawRightString(x, y, text)

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

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

The text object methods

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

canvas.drawText(textobject)
```

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

The path object methods

```
path = canvas.beginPath()

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

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

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

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

Image methods



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

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

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

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

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

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

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

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

Ending a page

```
canvas.showPage()
```

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



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

2.5 The toolbox: the "state change" operations

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

Changing Colors

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

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

```
canvas.setFillColorRGB(r, g, b)
```

```
canvas.setStrokeColorRGB(r, g, b)
```

```
canvas.setFillColor(acolor)

canvas.setStrokeColor(acolor)

canvas.setFillGray(gray)

canvas.setStrokeGray(gray)
```

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

Changing Fonts

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

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

Changing Graphical Line Styles

```
canvas.setLineWidth(width)

canvas.setLineCap(mode)

canvas.setLineJoin(mode)

canvas.setMiterLimit(limit)

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

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

Changing Geometry

```
canvas.setPageSize(pair)

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

canvas.translate(dx, dy)

canvas.scale(x, y)

canvas.rotate(theta)

canvas.skew(alpha, beta)
```

All PDF drawings fit into a specified page size. Elements drawn outside of the specified page size are not visible. Furthermore all drawn elements are passed through an affine transformation which may adjust their location and/or distort their appearance. 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 invocation 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.

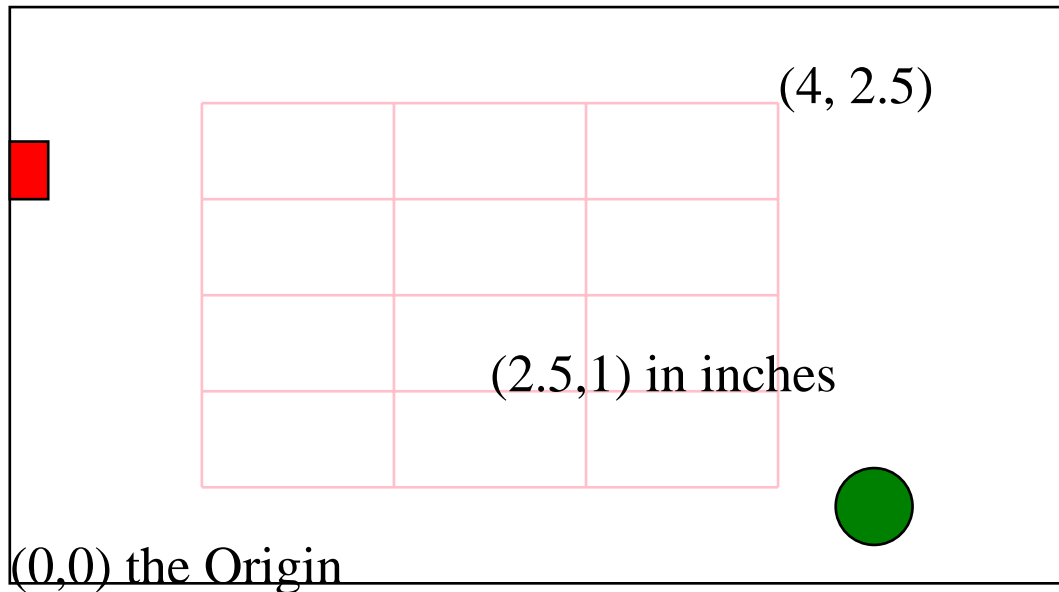


Figure A-2: The Coordinate System

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.

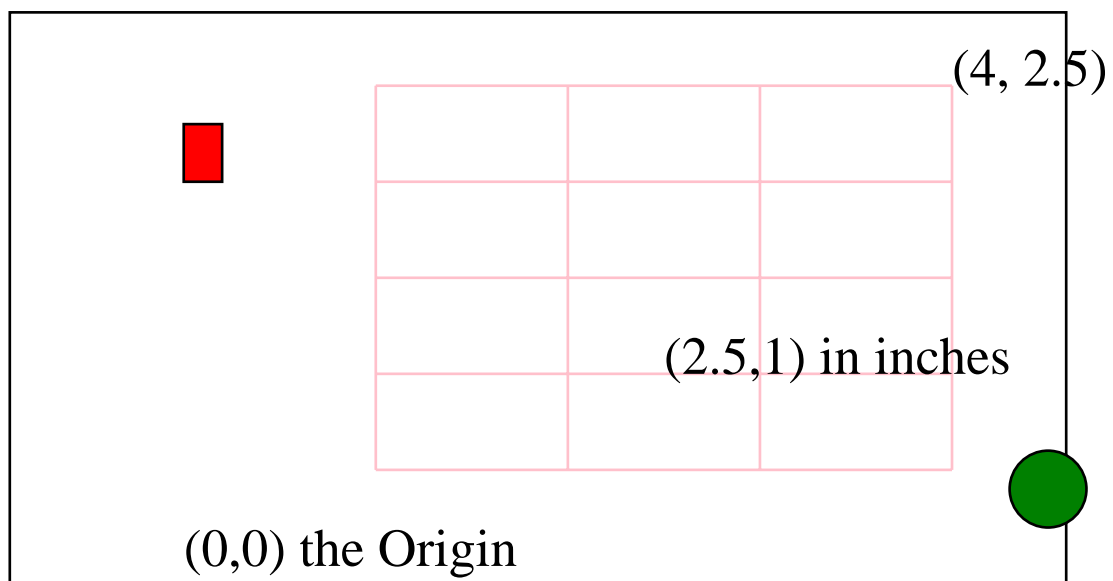


Figure A-3: Moving the origin: the `translate` method



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.

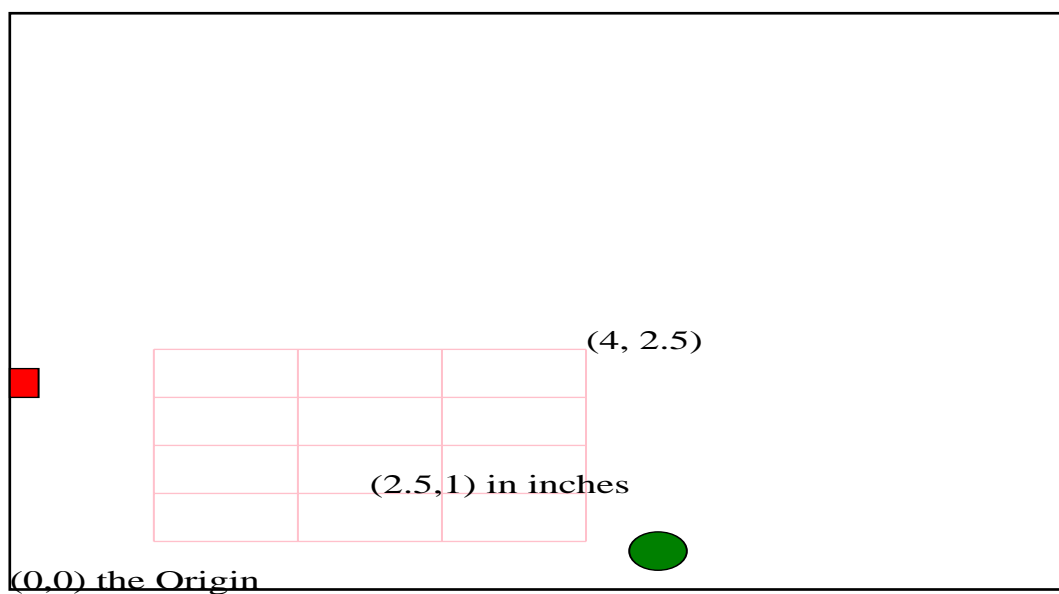


Figure A-4: Scaling the coordinate system



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.

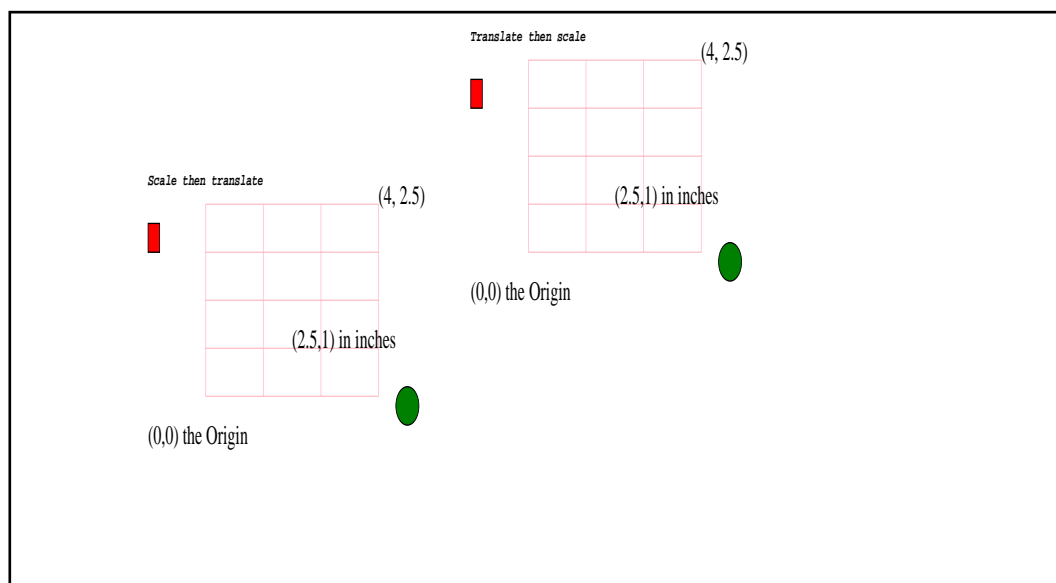


Figure A-5: Scaling and Translating



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()` and `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.

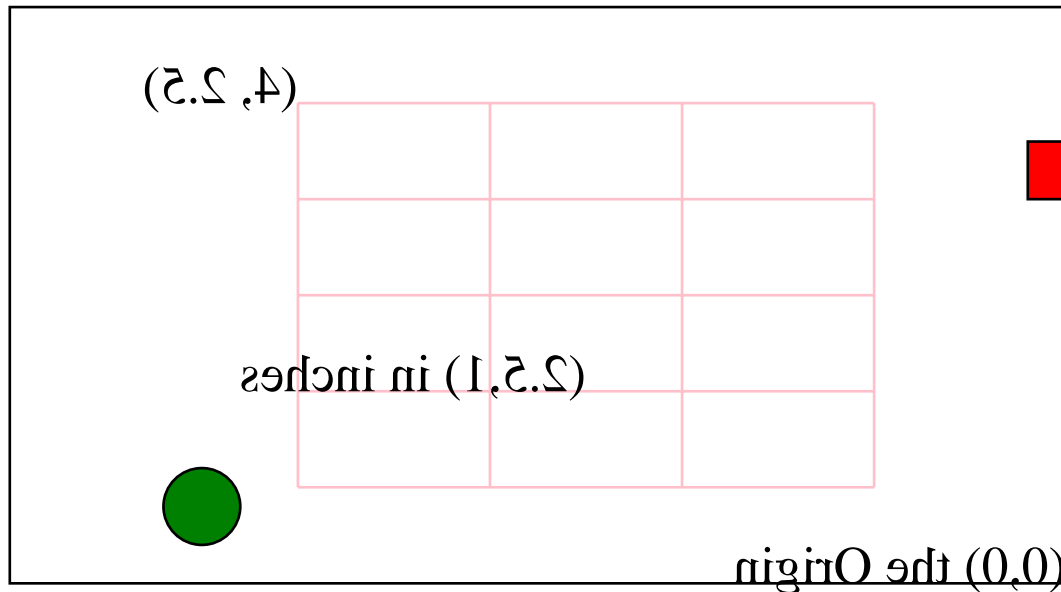


Figure A-6: Mirror Images

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.

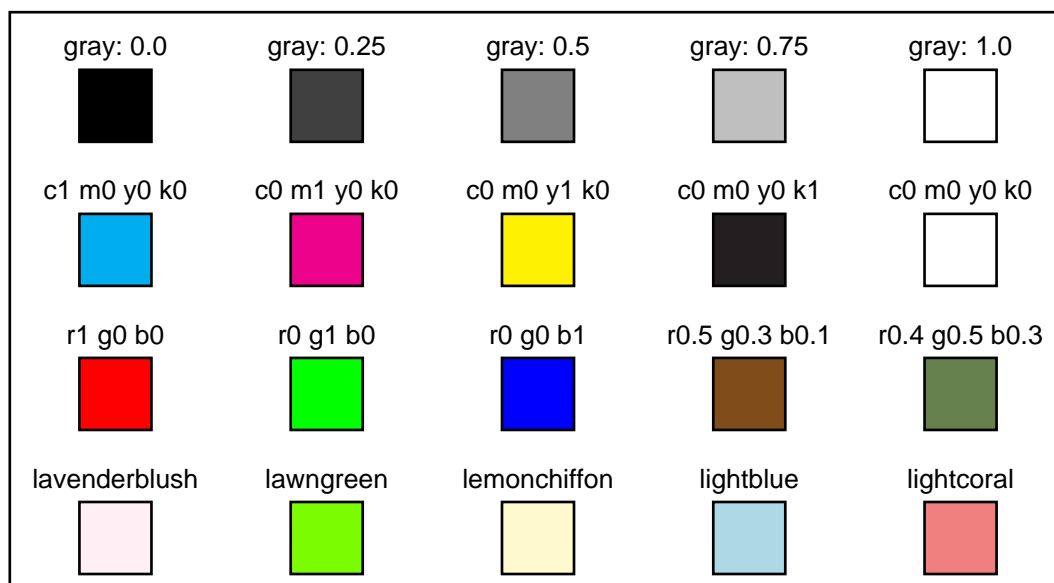


Figure A-7: Color Models

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.

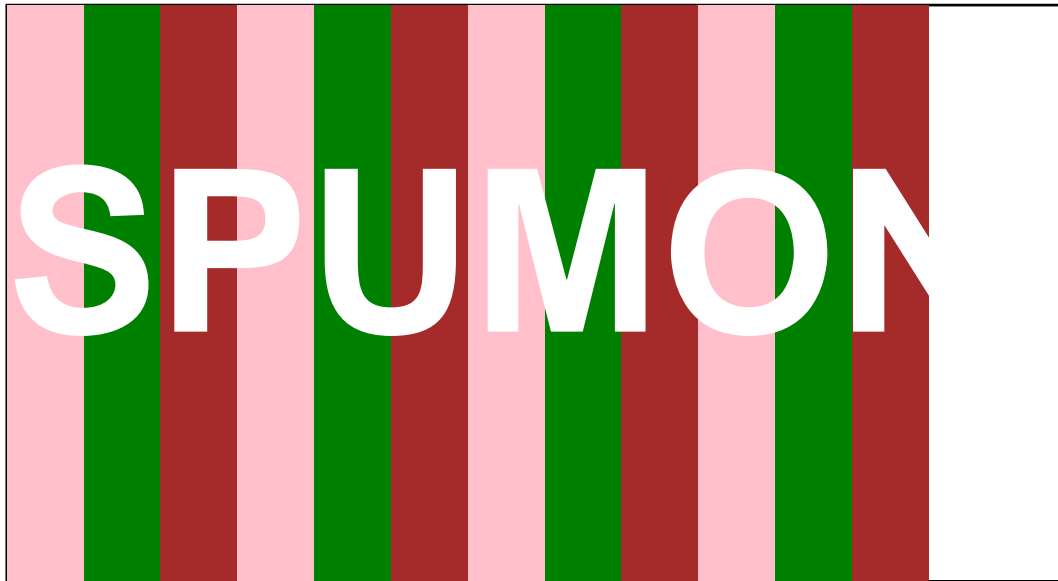


Figure A-8: Painting over colors

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 each other as well.

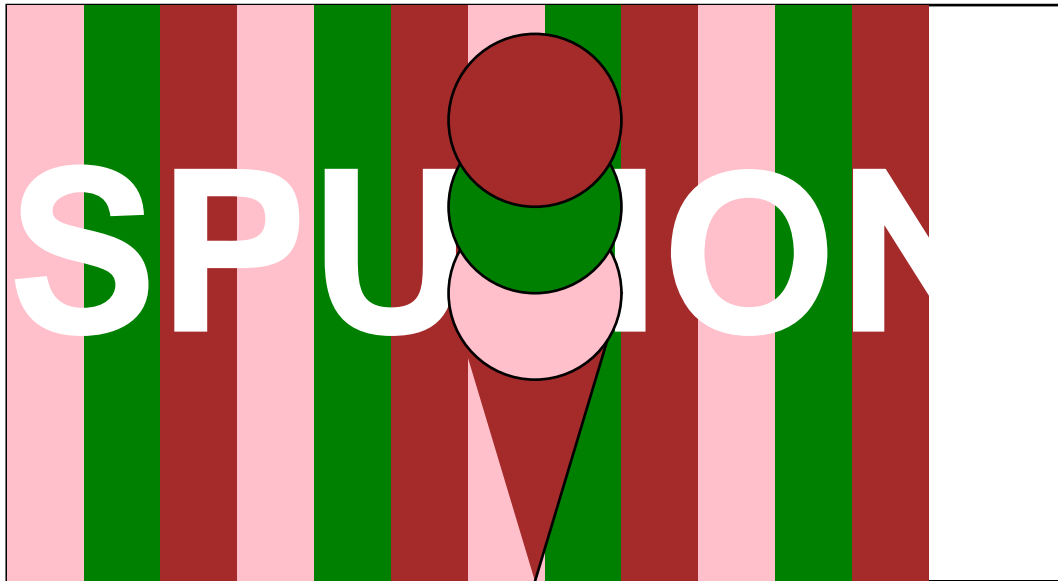


Figure A-9: building up a drawing in layers

2.10 Standard fonts and text objects

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

```
def textsize(canvas):
    from reportlab.lib.units import inch
    from reportlab.lib.colors import magenta, red
    canvas.setFont("Times-Roman", 20)
    canvas.setFillColor(red)
    canvas.drawCentredString(2.75*inch, 2.5*inch, "Font size examples")
    canvas.setFillColor(magenta)
    size = 7
    y = 2.3*inch
    x = 1.3*inch
    for line in lyrics:
        canvas.setFont("Helvetica", size)
        canvas.drawRightString(x,y,"%s points: " % size)
        canvas.drawString(x,y, line)
        y = y-size*1.2
        size = size+1.5
```

The `textsize` function generates the following page.

Font size examples

7 points: well she hit Net Solutions
8.5 points: and she registered her own .com site now
10.0 points: and filled it up with yahoo profile pics
11.5 points: she snarfed in one night now
13.0 points: and she made 50 million when Hugh Hefner
14.5 points: bought up the rights now
16.0 points: and she'll have fun fun fun
17.5 points: til her Daddy takes the keyboard away

Figure A-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.

Courier:	Now is the time for all good men to...
Courier-Bold:	Now is the time for all good men to...
Courier-BoldOblique:	<i>Now is the time for all good men to...</i>
Courier-Oblique:	<i>Now is the time for all good men to...</i>
Helvetica:	Now is the time for all good men to...
Helvetica-Bold:	Now is the time for all good men to...
Helvetica-BoldOblique:	<i>Now is the time for all good men to...</i>
Helvetica-Oblique:	<i>Now is the time for all good men to...</i>
Symbol:	■■■■ ■■ ■■■ ■■■■ ■■■ ■■■ ■■■■ ■■■ ■■■...
Times-Bold:	Now is the time for all good men to...
Times-BoldItalic:	<i>Now is the time for all good men to...</i>
Times-Italic:	<i>Now is the time for all good men to...</i>
Times-Roman:	Now is the time for all good men to...
ZapfDingbats:	■■■■ ■■ ■■■ ■■■■ ■■■ ■■■ ■■■■ ■■■ ■■■■■■

Figure A-11: the 14 standard fonts

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

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.

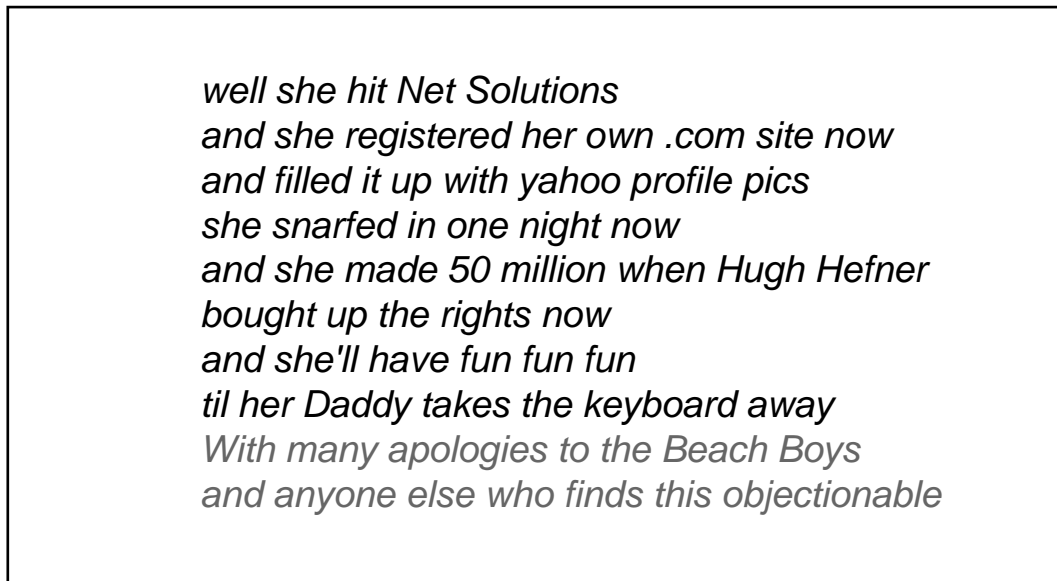


Figure A-12: How the text cursor moves

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.beginPath()
    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.setFillColorsRGB(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.

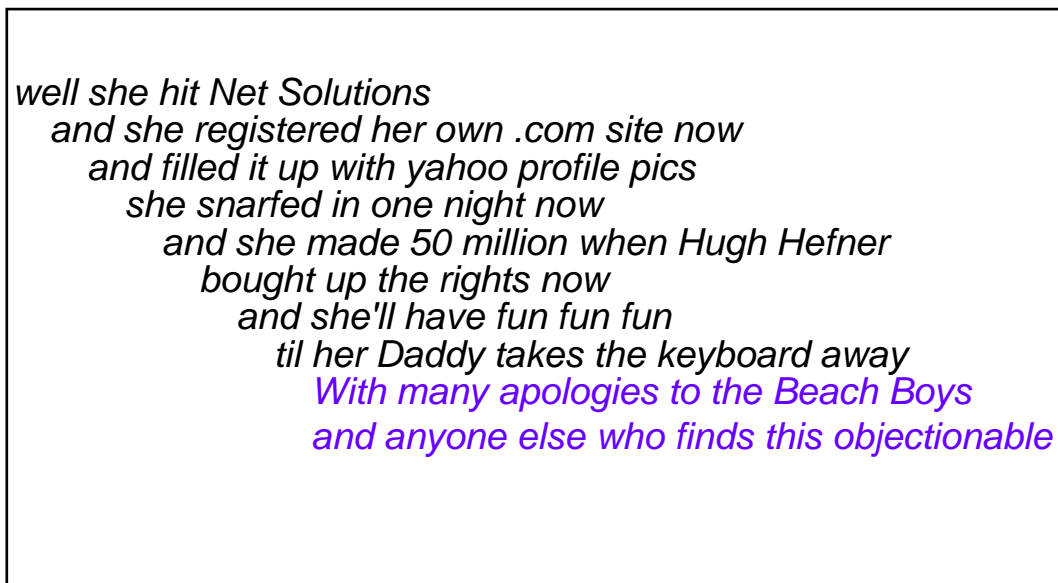


Figure A-13: How the text cursor moves again

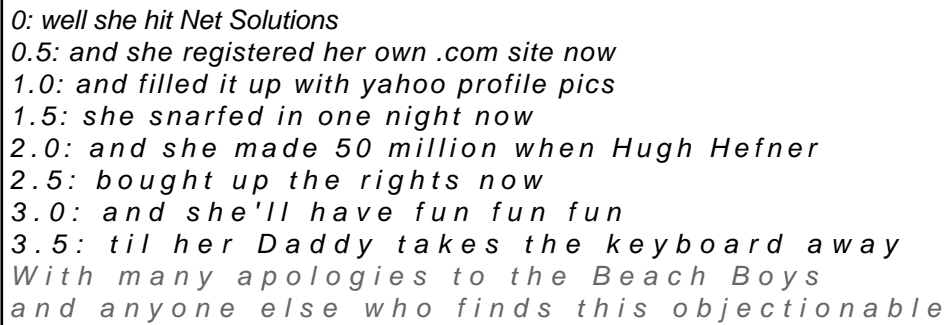
Character Spacing

```
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.



0: well she hit Net Solutions
0.5: and she registered her own .com site now
1.0: and filled it up with yahoo profile pics
1.5: she snarfed in one night now
2.0: and she made 50 million when Hugh Hefner
2.5: bought up the rights now
3.0: and she'll have fun fun fun
3.5: til her Daddy takes the keyboard away
With many apologies to the Beach Boys
and anyone else who finds this objectionable

Figure A-14: Adjusting inter-character spacing

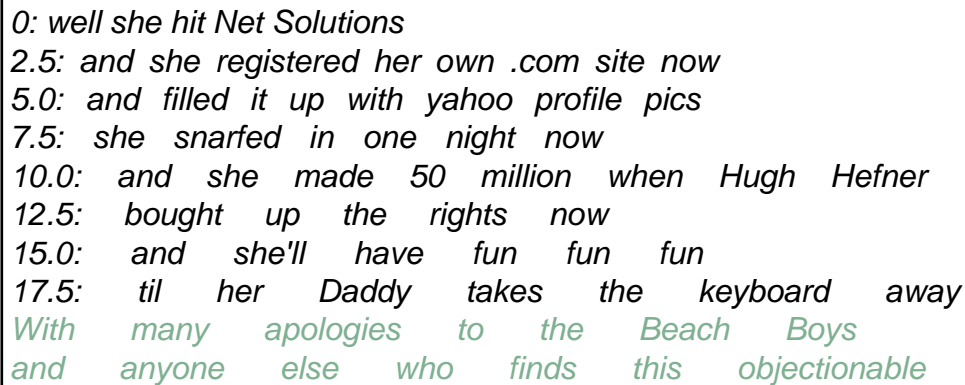
Word Spacing

```
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.setFillColorsCMYK(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.



0: well she hit Net Solutions
 2.5: and she registered her own .com site now
 5.0: and filled it up with yahoo profile pics
 7.5: she snarfed in one night now
 10.0: and she made 50 million when Hugh Hefner
 12.5: bought up the rights now
 15.0: and she'll have fun fun fun
 17.5: til her Daddy takes the keyboard away
 With many apologies to the Beach Boys
 and anyone else who finds this objectionable

Figure A-15: Adjusting word spacing

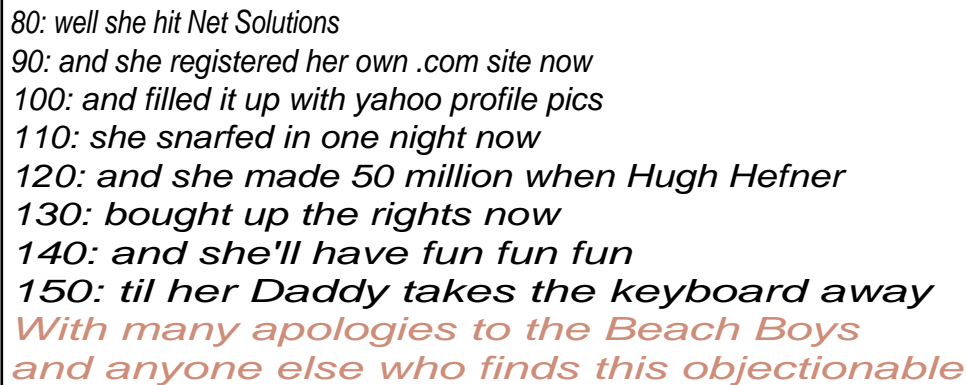
Horizontal Scaling

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

Lines of text can be stretched or shrunk 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.setFillColorsCMYK(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.



80: well she hit Net Solutions
 90: and she registered her own .com site now
 100: and filled it up with yahoo profile pics
 110: she snarfed in one night now
 120: and she made 50 million when Hugh Hefner
 130: bought up the rights now
 140: and she'll have fun fun fun
 150: til her Daddy takes the keyboard away
 With many apologies to the Beach Boys
 and anyone else who finds this objectionable

Figure A-16: adjusting horizontal text scaling

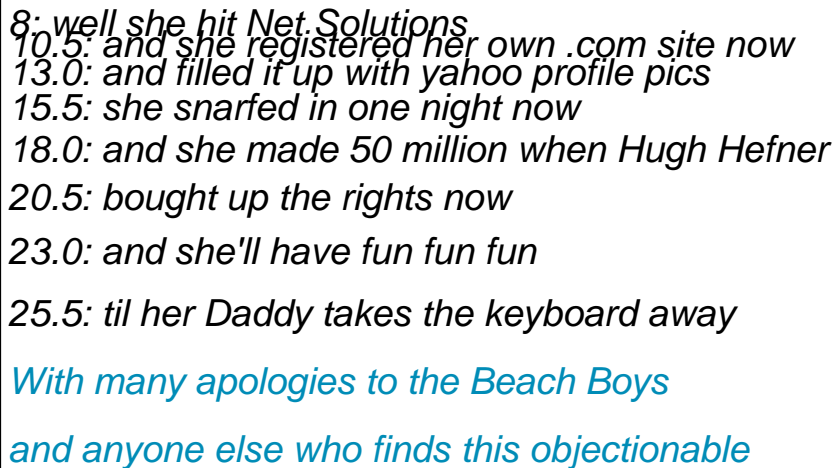
Interline spacing (Leading)

```
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.beginPath()
    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.setFillColorsCMYK(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 may write over the bottom parts of characters in the previous line.



8: well she hit Net Solutions
 10.5: and she registered her own .com site now
 13.0: and filled it up with yahoo profile pics
 15.5: she snarfed in one night now
 18.0: and she made 50 million when Hugh Hefner
 20.5: bought up the rights now
 23.0: and she'll have fun fun fun
 25.5: til her Daddy takes the keyboard away
 With many apologies to the Beach Boys
 and anyone else who finds this objectionable

Figure A-17: adjusting the leading

Other text object methods

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

The `setTextRenderMode` method allows text to be used as a foreground 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 A-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.

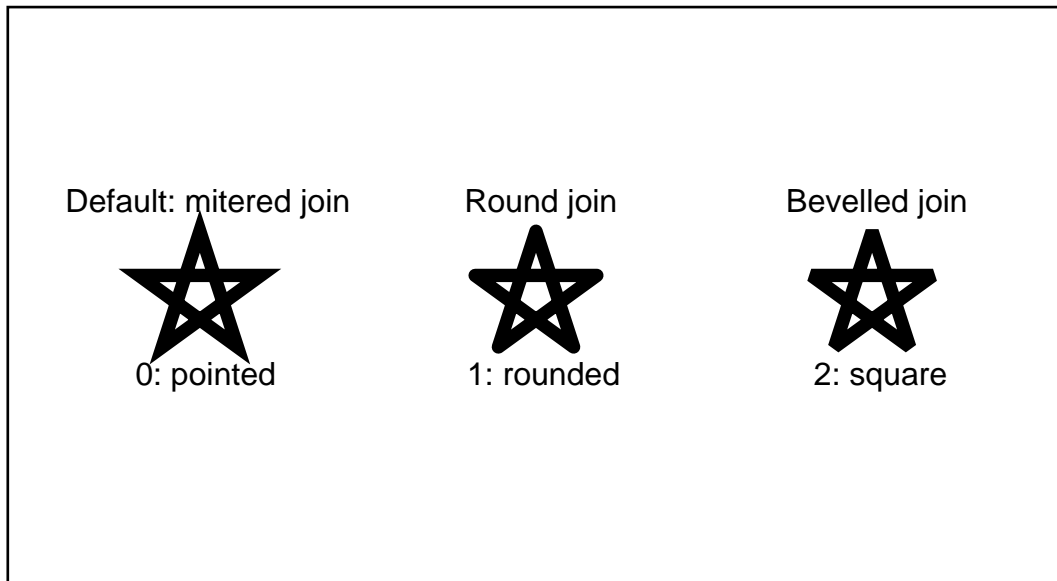


Figure A-19: different line join styles

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.

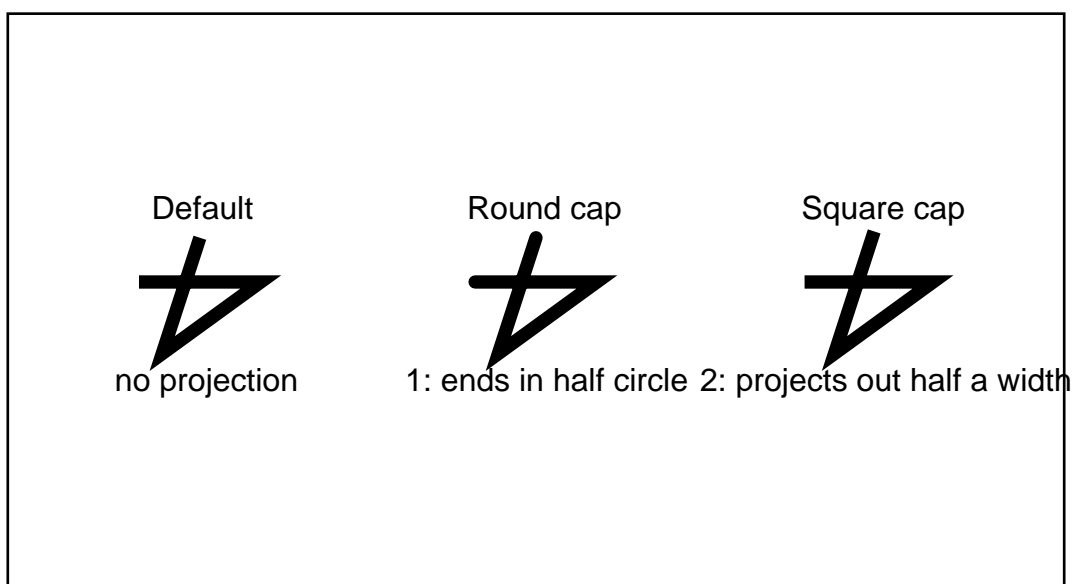


Figure A-20: line cap settings

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.

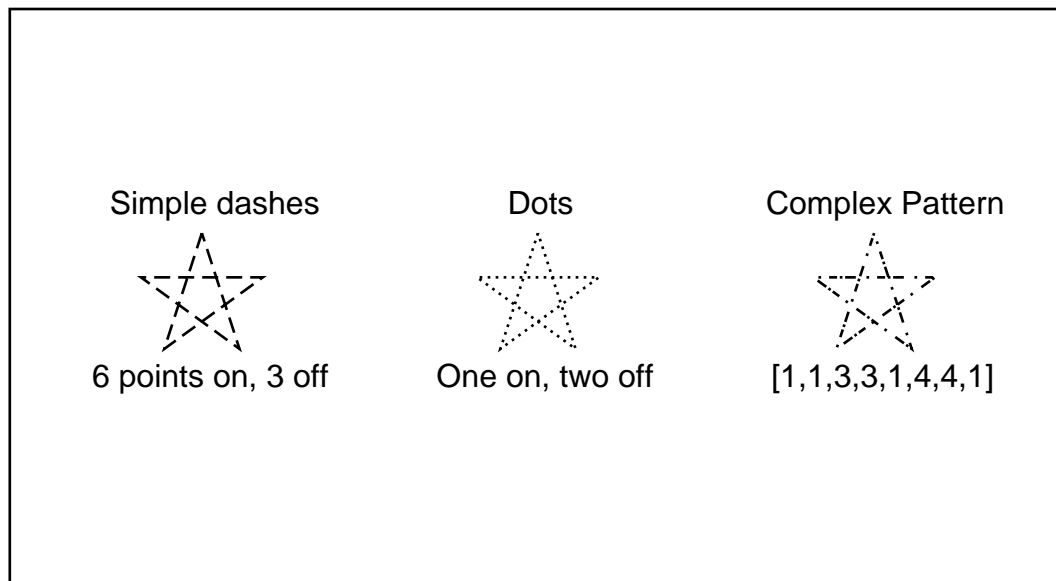


Figure A-21: some dash patterns

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.

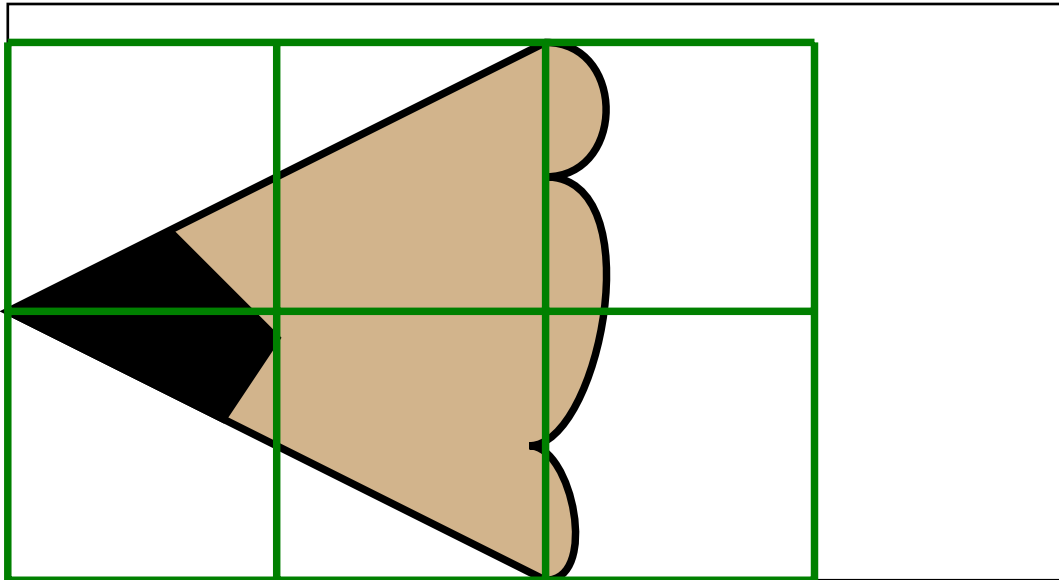


Figure A-22: a pencil tip

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 eraser
    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.

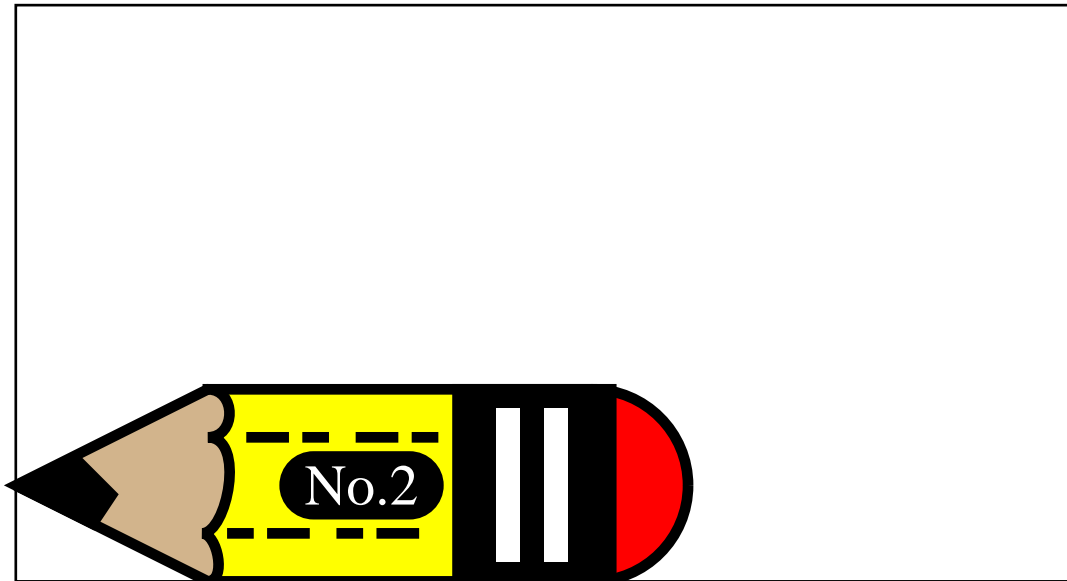


Figure A-23: a whole pencil

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 (x_1, y_1) , (x_2, y_2) , (x_3, y_3) , (x_4, y_4) . The curve starts at (x_1, y_1) and ends at (x_4, y_4) and the line segment from (x_1, y_1) to (x_2, y_2) and the line segment from (x_3, y_3) to (x_4, y_4) both form tangents to the curve. Furthermore the curve is entirely contained in the convex figure with vertices at the control points.

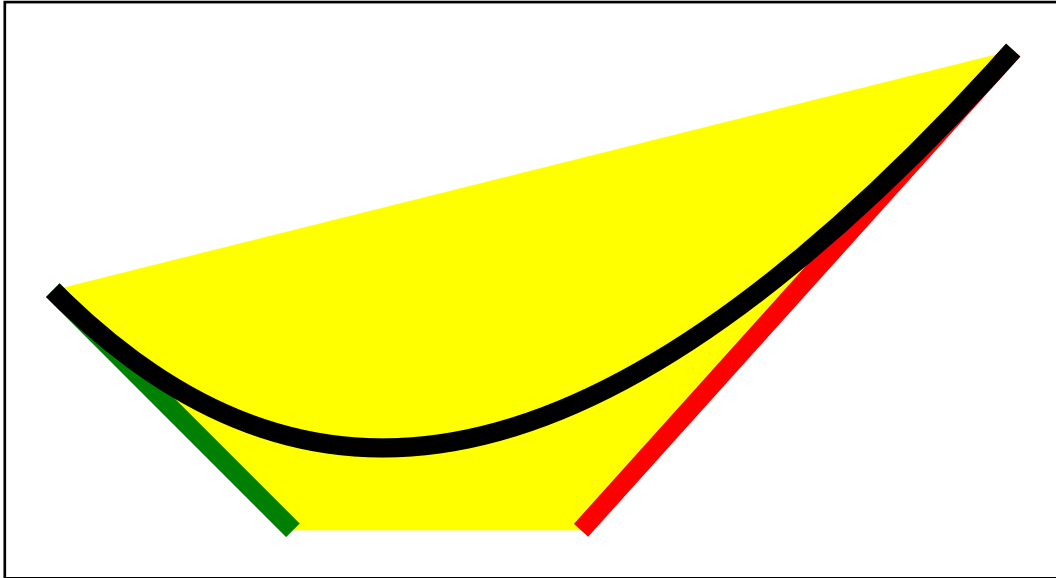


Figure A-24: basic bezier curves

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.

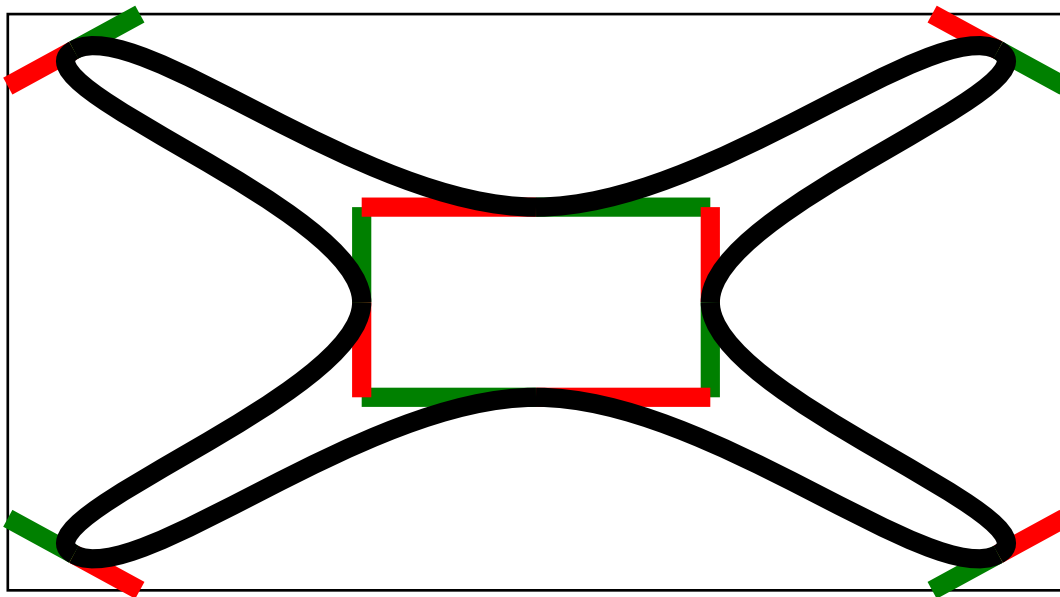


Figure A-25: bezier curves

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.lineTo(1*inch, 1*inch, 2.5*inch, 2*inch, startAng=-30, extent=135)
p.arc(3*inch, 1*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.

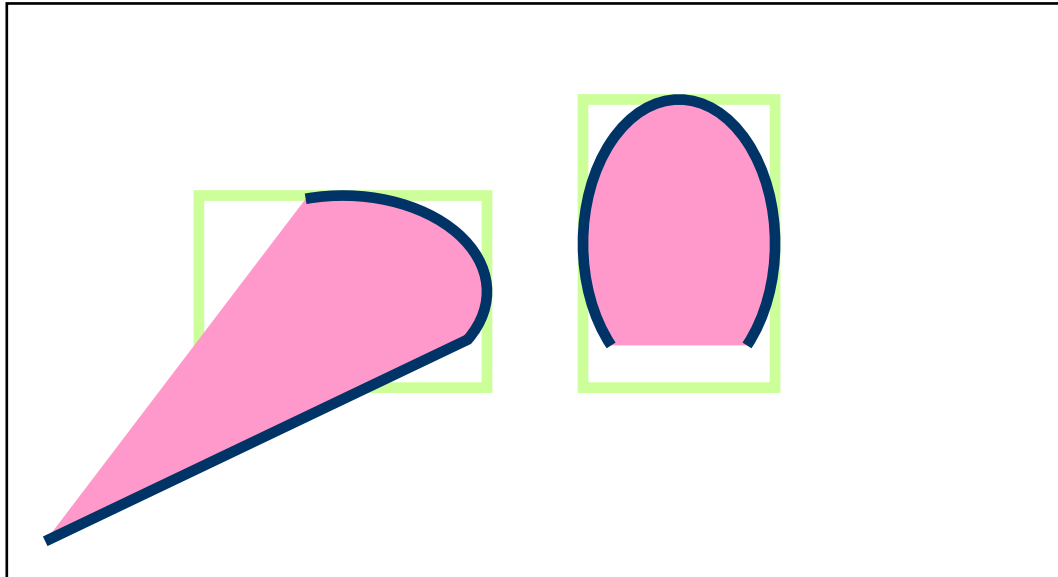


Figure A-26: arcs in path objects

```
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 rectangle 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.

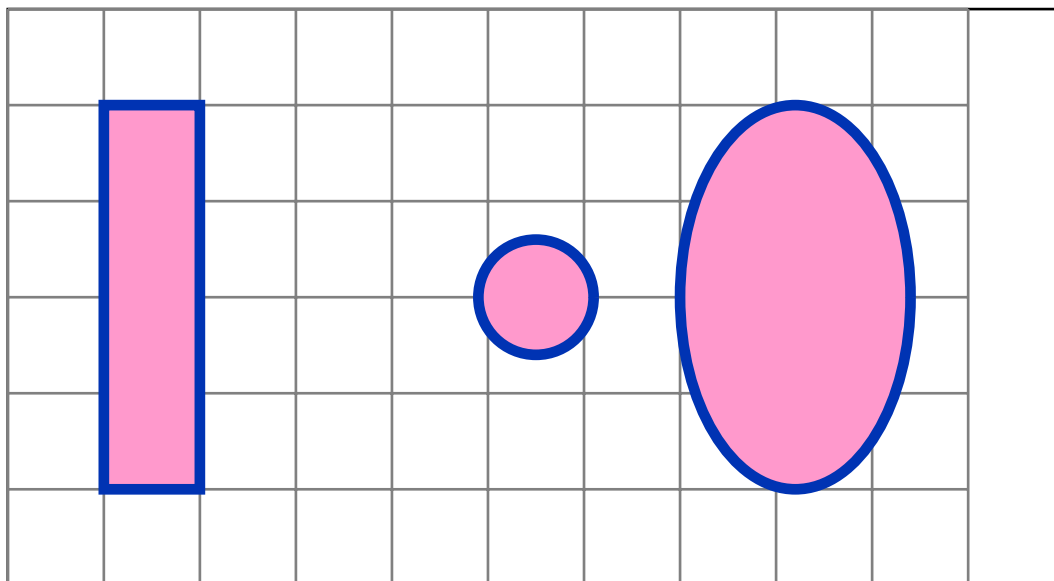


Figure A-27: rectangles, circles, ellipses in path objects

```
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.setFillColorsRGB(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.

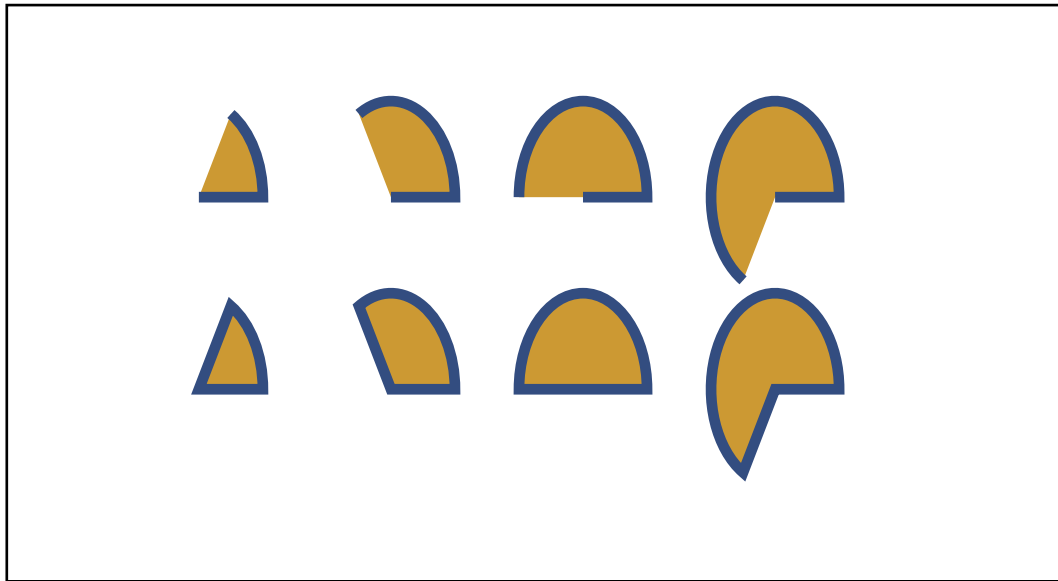


Figure A-28: closing and not closing pathobject figures

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".

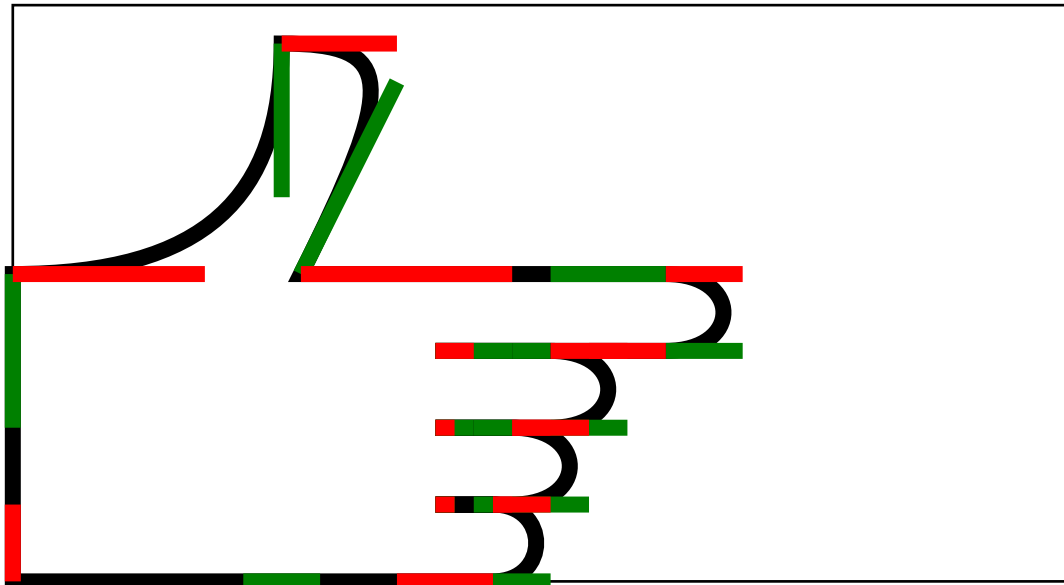


Figure A-29: an outline of a hand using bezier curves

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.setFillColorsRGB(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.

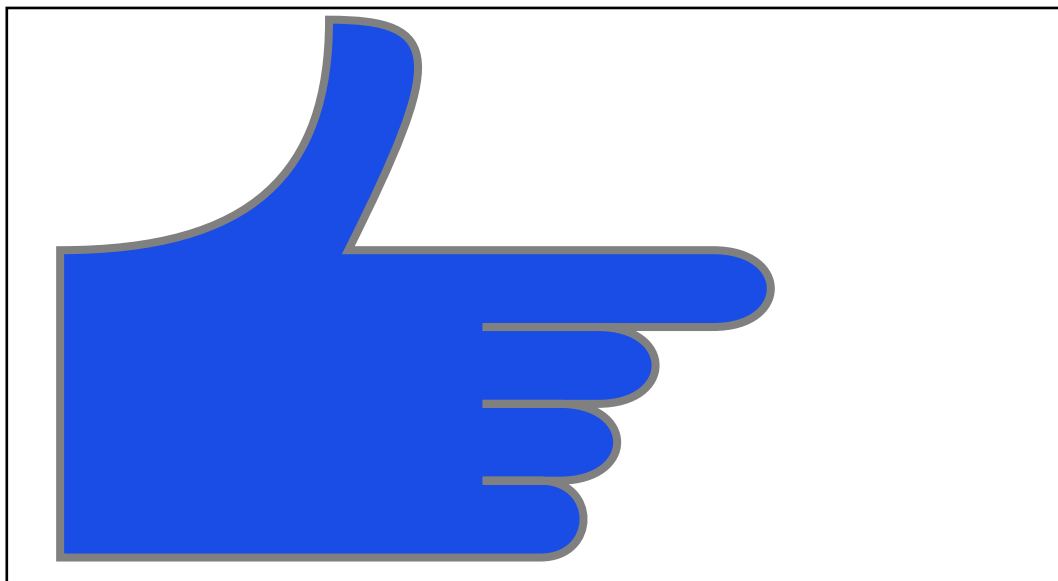


Figure A-30: the finished hand, filled

2.16 Further Reading: The ReportLab Graphics Library

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

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

A thorough description of its philosophy and features is beyond the scope of this general user guide and the reader is recommended to continue with the "*ReportLab Graphics Guide*". There she will find information about the existing components and how to create customized ones.

Also, the graphics guide contains a presentation of an emerging charting package and its components (labels, axes, legends and different types of charts like bar, line and pie charts) that builds directly on the graphics library.

Chapter 3 Fonts and encodings

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

3.1 Unicode and UTF8 are the default input encodings

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

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

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

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

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

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

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

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

3.2 Changing the built-in fonts output encoding

There are still a number of places in the code, including the `rl_config.defaultEncoding` parameter, and arguments passed to various Font constructors. These generally relate to the OUTPUT encoding used when we write data in the font file. This affects which characters are actually available in the font if you are using Type 1 fonts, since only 256 glyphs can be available at one time. Unless you have a very specific need for MacRoman or MacExpert encoding characters, we advise you to ignore this. By default the standard fonts (Helvetica, Courier, Times Roman) will offer the glyphs available in Latin-1. If you try to print a non-Latin-1 character using the built-in Helvetica, you'll see a rectangle or blob.

3.3 Using non-standard Type 1 fonts

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

You can use the mechanism described below to include arbitrary fonts in your documents. Just van Rossum has kindly donated a Type 1 font named *LettErrorRobot-Chrome* which we may use for testing and/or documenting purposes (and which you may use as well). It comes bundled with the ReportLab distribution in the directory `reportlab/fonts`.

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

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

```
import os
import reportlab
folder = os.path.dirname(reportlab.__file__) + os.sep + 'fonts'
afmFile = os.path.join(folder, 'LeERC____.AFM')
pfbFile = os.path.join(folder, 'LeERC____.PFB')

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

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

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

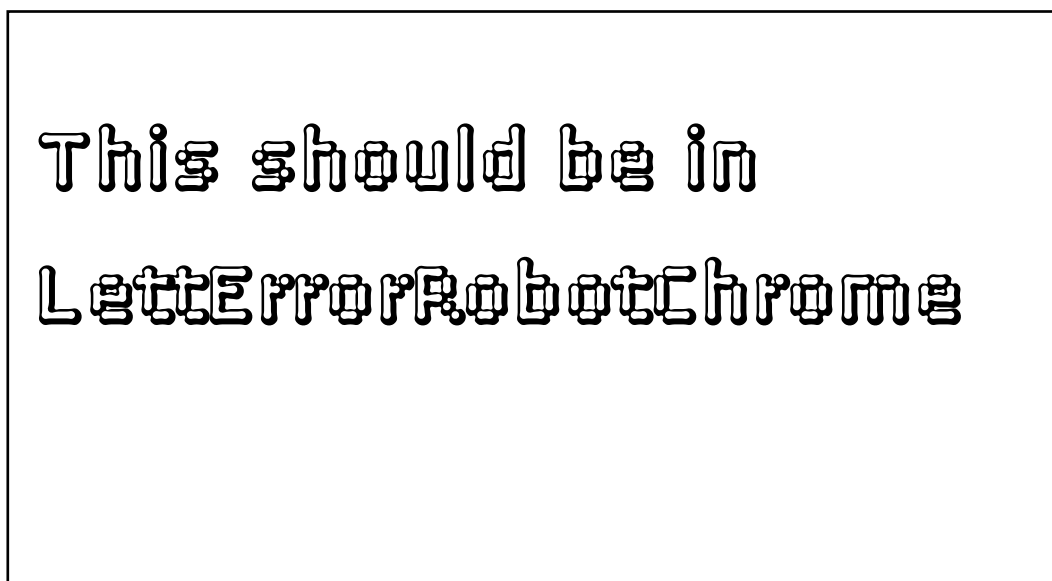


Figure A-31: Using a very non-standard font

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

```
class FontNameNotFoundError(Exception):
    pass

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

    f = open(path)

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

    return fontName
```

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

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

Warnings about missing glyphs

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

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

3.4 Standard Single-Byte Font Encodings

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

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

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00																																
20		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	•
80	€	•	,	f	„	...	†	‡	^	%	Š	‹	œ	•	Ž	•	•	‘	’	“	”	•	—	—	~	™	š	›	œ	•	ž	ÿ
A0		ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figure A-32: WinAnsi Encoding

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

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00																																
20		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{		}	~	■
80	Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ã	ä	å	ç	é	è	ê	ë	í	ì	ï	ñ	ó	ò	ô	õ	ö	ù	û	ü		
A0	†	°	¢	£	§	•	¶	ß	®	©	™	´	¨	≠	Æ	Ø	∞	±	≤	≥	¥	µ	∂	Σ	Π	π	∫	α	ο	Ω	æ	ø
C0	¿	ı	¬	√	ƒ	≈	Δ	«	»	…	■	À	Ã	Ö	œ	—	“	”	‘	’	÷	◊	ÿ	/	¤	‹	›	fi	fl			
E0	‡	•	,	„	%	Â	Ê	Á	Ë	È	Í	Î	Ì	Ó	Ô	■	Ò	Ù	Ú	Û	ı	ˆ	˜	˘	˙	˚	˛	˜	˜	˜	˜	˜

Figure A-33: MacRoman Encoding

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

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00																																
20		✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	
40	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	
60	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	✂	
80	()	()	()	<	>	()	()	()	()	()	()	()	()	()	()	()	(
A0	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪	⑫	⑬	⑭	⑮	
C0	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪	⑫	⑬	⑭	⑮	⑯	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	
E0	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	➔	

Figure A-34: ZapfDingbats and its one and only encoding

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00																																
20		!	∇	#	∑	%	&	∑	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	≡	A	B	X	Δ	E	Φ	Γ	H	I	∅	K	Λ	M	N	O	Π	Θ	P	Σ	T	Y	ç	Ω	Ξ	Ψ	Z		∴		±	
60	␣	α	β	γ	δ	ε	φ	γ	η	ι	φ	κ	λ	μ	ν	ο	π	θ	ρ	σ	τ	υ	ω	ω	ξ	ψ	ζ	{		}	~	■
80	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
A0	€	Υ	′	≤	/	∞	f	♣	♦	♥	♠	↔	←	↑	→	↓	°	±	″	≥	×	∞	∂	•	÷	≠	≡	≈	...		—	↓
C0	ℵ	ℶ	ℷ	ℸ	⊗	⊕	∅	∩	∪	⊃	⊂	⊄	⊆	⊆	∈	∉	∠	∇	®	©	™	Π	√	·	¬	^	∨	↔	⇐	↑	⇒	↓
E0	⋄	⋀	®	©	™	Σ	(■	〉															

Figure A-35: Symbol and its one and only encoding

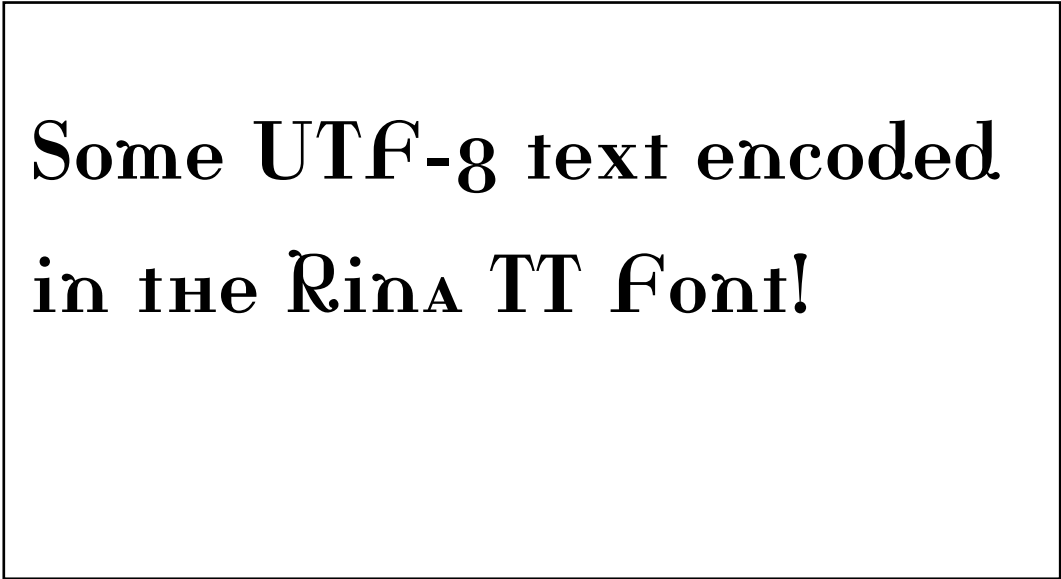
3.5 TrueType Font Support

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

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

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

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



Some UTF-8 text encoded
in the Rina TT Font!

Figure A-36: Using a the Rina TrueType Font

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

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

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

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

```
from reportlab.lib.fonts import addMapping
addMapping('Rina', 0, 0, 'Rina')      #normal
addMapping('Rina', 0, 1, 'Rina')      #italic
addMapping('Rina', 1, 0, 'Rina')      #bold
addMapping('Rina', 1, 1, 'Rina')      #italic and bold
```

We only have a Rina regular font, no bold or italic, so we must map all to the same internal fontname. `` and `<i>` tags may now be used safely, but have no effect. After registering and mapping the Rina font as above we can use paragraph text like



Figure A-37: Using TTF fonts in paragraphs

3.6 Asian Font Support

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

Asian Language Packs

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

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

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

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

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

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0		亜	啞	娃	阿	哀	愛	挨	始	逢	葵	茜	穉	愚	握	渥	旭	葦	芦	鰻
20		梓	圧	幹	扱	宛	姐	虻	飴	絢	綾	鮎	或	粟	裕	安	庵	按	暗	案
40		鞍	杏	以	伊	位	依	偉	圉	夷	委	威	尉	惟	意	慰	易	椅	為	畏
60		移	維	緯	胃	萎	衣	謂	違	遺	医	井	亥	域	育	郁	磯	一	壹	溢
80		稲	茨	芋	鰯	允	印	咽	員	因	姻	引	飲	淫	胤	蔭				

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

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

# the two unicode characters below are "Tokyo"
msg = u'\u6771\u4eac : Unicode font, unicode input'
canvas.drawString(100, 675, msg)
```

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

```
reportlab/test/test_multibyte_jpn.py
reportlab/test/test_multibyte_kor.py
```

```
reportlab/test/test_multibyte_chs.py
reportlab/test/test_multibyte_cht.py
```

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

TrueType fonts with Asian characters

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

To Do

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

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

3.7 RenderPM tests

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

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

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

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

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

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

Chapter 4 Exposing PDF Special Capabilities

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

4.1 Forms

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

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

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

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

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

4.2 Links and Destinations

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

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

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

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

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

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

Table 4-1 - Required attributes for different fit types

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

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

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

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

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

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

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

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



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

Defining internal links

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

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

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

For example the code

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

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

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

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

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

4.3 Outline Trees

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

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

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

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

4.4 Page Transition Effects

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

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

4.5 Internal File Annotations

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

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

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

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

setAuthor, setTitle, setSubject have no visible effect

But if you are viewing this document dynamically

please look at File/Document Info

Figure A-38: Setting document internal annotations

Chapter 5 PLATYPUS - Page Layout and Typography Using Scripts

5.1 Design Goals

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

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

The overall design of Platypus can be thought of as 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.

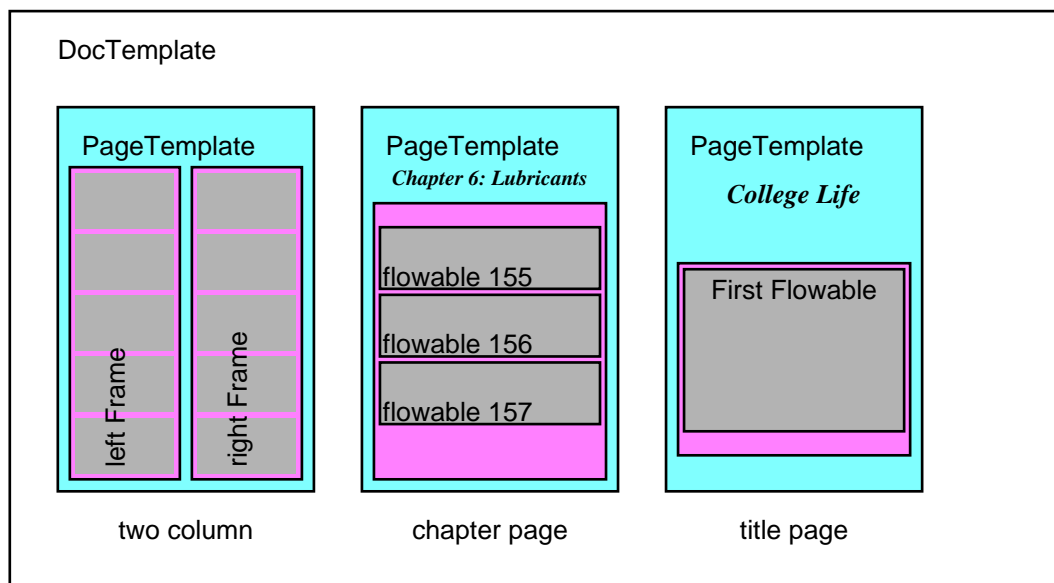


Figure A-39: Illustration of DocTemplate structure

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`.

5.2 Getting started

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

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

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

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

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

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

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

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

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

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

5.3 Flowables

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

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

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

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

Flowable User Methods

```
Flowable.draw()
```

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

```
Flowable.drawOn(canvas,x,y)
```

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

```
Flowable.wrap(availWidth, availHeight)
```

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

```
Flowable.split(self, availWidth, availheight):
```

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

5.4 Guidelines for flowable positioning

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

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

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

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

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

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

5.5 Frames

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

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

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

Frame User Methods

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

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

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

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

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

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

Using Frames

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

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

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

5.6 Documents and Templates

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

The `BaseDocTemplate` class

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

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 override 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 derived class version. We make this easy by providing a base class synonym for each of the handler methods with the same name prefixed by an underscore '_'.

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

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

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

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

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

```
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.lib.pagesizes import letter
from reportlab.platypus import Paragraph, SimpleDocTemplate
styles = getSampleStyleSheet()
styleN = styles['Normal']
styleH = styles['Heading1']
story = []

#add some flowables
story.append(Paragraph("This is a Heading",styleH))
```

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

PageTemplates

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

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

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

Chapter 6 Paragraphs

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

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

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

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

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

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

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

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

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

class ParagraphStyle

```
class ParagraphStyle(PropertySet):
    defaults = {
        'fontName': 'Times-Roman',
        'fontSize': 10,
        'leading': 12,
        'leftIndent': 0,
        'rightIndent': 0,
        'firstLineIndent': 0,
        'alignment': TA_LEFT,
        'spaceBefore': 0,
        'spaceAfter': 0,
        'bulletFontName': 'Times-Roman',
        'bulletFontSize': 10,
        'bulletIndent': 0,
        'textColor': black
    }
```

6.1 Using Paragraph Styles

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

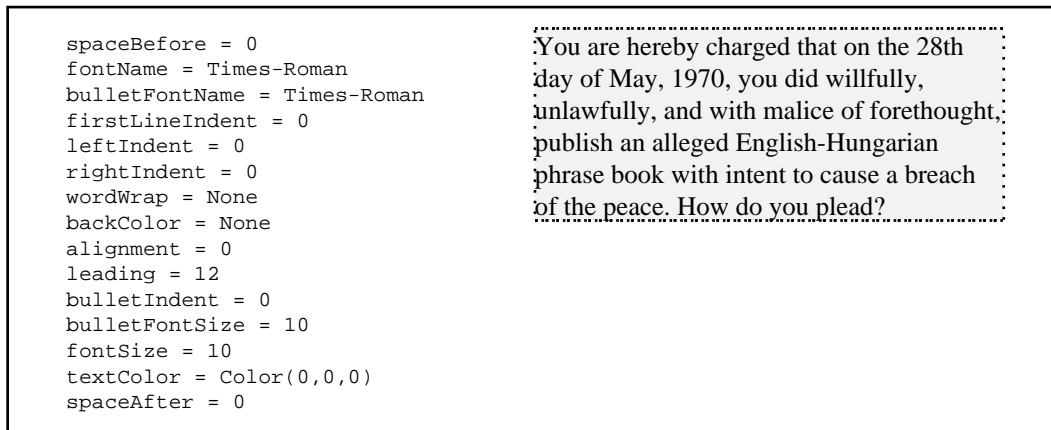


Figure A-40: The default ParagraphStyle

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`:

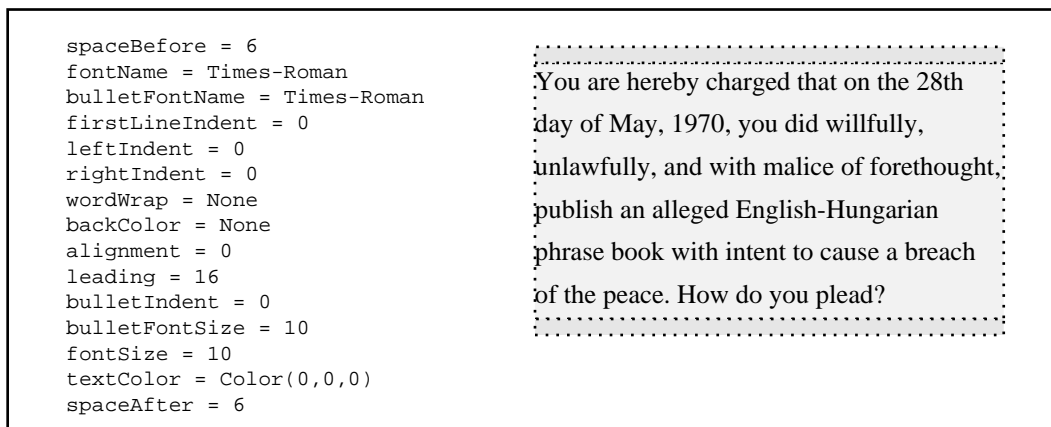


Figure A-41: Space before and after and increased leading

The `leftIndent` and `rightIndent` attributes do exactly what you would expect; `firstLineIndent` is added to the `leftIndent` of the first line. If you want a straight left edge, remember to set `firstLineIndent` equal to 0.

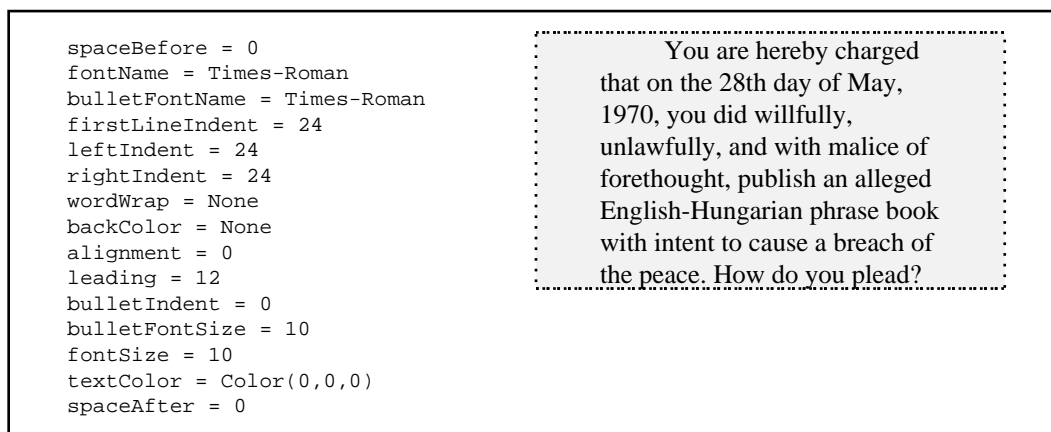


Figure A-42: one third inch indents at left and right, two thirds on first line

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

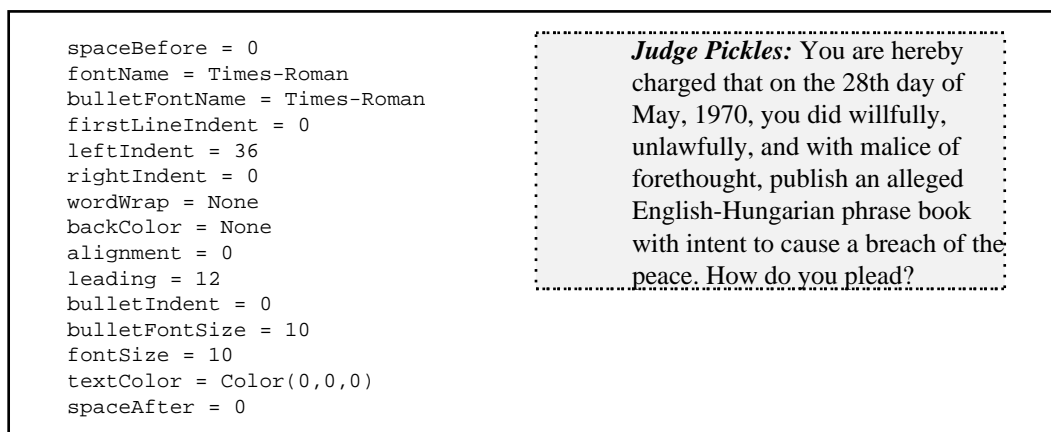


Figure A-43: Definition Lists

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.

6.2 Paragraph XML Markup Tags

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

The outermost < para > tag

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

Attribute	Synonyms
<code>alignment</code>	<code>alignment</code> , <code>align</code>
<code>backColor</code>	<code>backColor</code> , <code>bgcolor</code> , <code>backcolor</code> , <code>bg</code>
<code>bulletColor</code>	<code>bulletColor</code> , <code>bcolor</code> , <code>bulletcolor</code>
<code>bulletFontName</code>	<code>bulletFontName</code> , <code>bulletfontname</code> , <code>bfont</code>
<code>bulletFontSize</code>	<code>bulletfontsize</code> , <code>bulletFontSize</code> , <code>bfontsize</code>
<code>bulletIndent</code>	<code>bindent</code> , <code>bulletindent</code> , <code>bulletIndent</code>
<code>firstLineIndent</code>	<code>findent</code> , <code>firstlineindent</code> , <code>firstLineIndent</code>

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

Table 6-2 - Synonyms for style attributes

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

6.3 Intra-paragraph markup

Within each paragraph, we use a basic set of XML tags to provide markup. The most basic of these are bold (`...`) and italic (`<i>...</i>`). 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.

<pre>You are hereby charged 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>?</pre>	<p>You are hereby charged 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></p>
---	---

Figure A-44: Simple bold and italic tags

The `` tag

The `` 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.

<pre> 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?</pre>	<p>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?</p>
--	--

Figure A-45: 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 (α): <code><greek>e</greek></code> <code><super><greek>ip</greek></super></code> <code>= -1</code>	Equation (α): $\varepsilon^{ip} = -1$
---	--

Figure A-46: 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:

<code><seq id="spam"/>, <seq</code> <code>id="spam"/>, <seq id="spam"/>.</code> <code>Reset<seqreset id="spam"/>. <seq</code> <code>id="spam"/>, <seq id="spam"/>.</code> <code><seq id="spam"/>.</code>	1, 2, 3. Reset. 1, 2, 3.
--	--------------------------

Figure A-47: 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.

<code><seqdefault</code> <code>id="spam"/>Continued... <seq/>.</code> <code><seq/>, <seq/>, <seq/>, <seq/>.</code> <code><seq/>, <seq/>.</code>	Continued... 4, 5, 6, 7, 8, 9, 10.
--	------------------------------------

Figure A-48: 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:

<code>Figure <seq</code> <code>template="%(Chapter)s-%(FigureNo+)s"/></code> <code>- Multi-level templates</code>	Figure 6-1 - Multi-level templates
---	------------------------------------

Figure A-49: Multi-level templates

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

6.4 Bullets and Paragraph Numbering

In addition to the three indent properties, some other parameters are needed to correctly handle bulleted and numbered lists. We discuss this here because you have now seen how to handle numbering. A paragraph may have an optional `bulletText` argument passed to its constructor; alternatively, bullet text may be placed

in a `<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 Times-Roman font in the style, and use a character such as `\xe2\x80\xa2`:

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

Table 6-3 - <bullet> attributes & synonyms

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

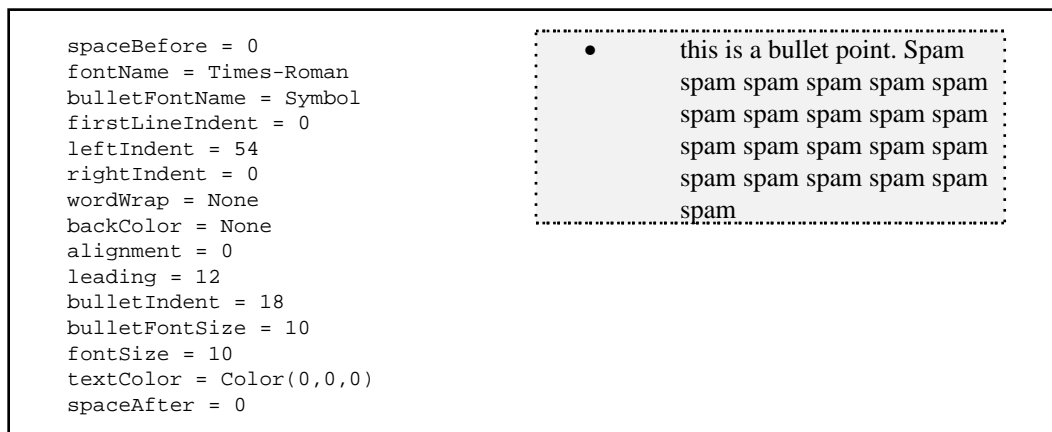


Figure A-50: Basic use of bullet points

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

Chapter 7 Tables and TableStyles

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

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

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

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

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

7.1 Table User Methods

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

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

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

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

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

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

The `splitByRow` argument is only needed for tables both too tall and too wide to fit in the current context. In this case you must decide whether to 'tile' down and across, or across and then down. This parameter is a

Boolean indicating that the Table should split itself by row before attempting to split itself by column when too little space is available in the current drawing area and the caller wants the Table to split.

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.

7.2 TableStyle

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

7.3 TableStyle User Methods

TableStyle(commandSequence)

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

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

TableStyle.add(commandSequence)

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

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

TableStyle.getCommands()

This method returns the sequence of commands of the instance.

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

7.4 TableStyle Commands

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

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

TableStyle Cell Formatting Commands

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

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

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

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

produces

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

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

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

produces

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

TableStyle Line Commands

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

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

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

produces

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

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

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

produces

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

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

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

20	21	22	23	24
30	31	32	33	34

When unsplit and split at the first or second row.

Complex Cell Values



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

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

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

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

produces

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

20	21	22	23	24
30	31	32	33	34

TableStyle Span Commands

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

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

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

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

produces

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

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

Special TableStyle Indexes

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

Chapter 8 Other Useful Flowables

8.1 `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.

8.2 `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; style entities as well for &amp; < > 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 `"`.

8.3 `Image(filename, width=None, height=None)`

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

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

will display as



whereas

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

produces



8.4 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.

8.5 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.

8.6 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`.

8.7 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 9 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.

9.1 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.

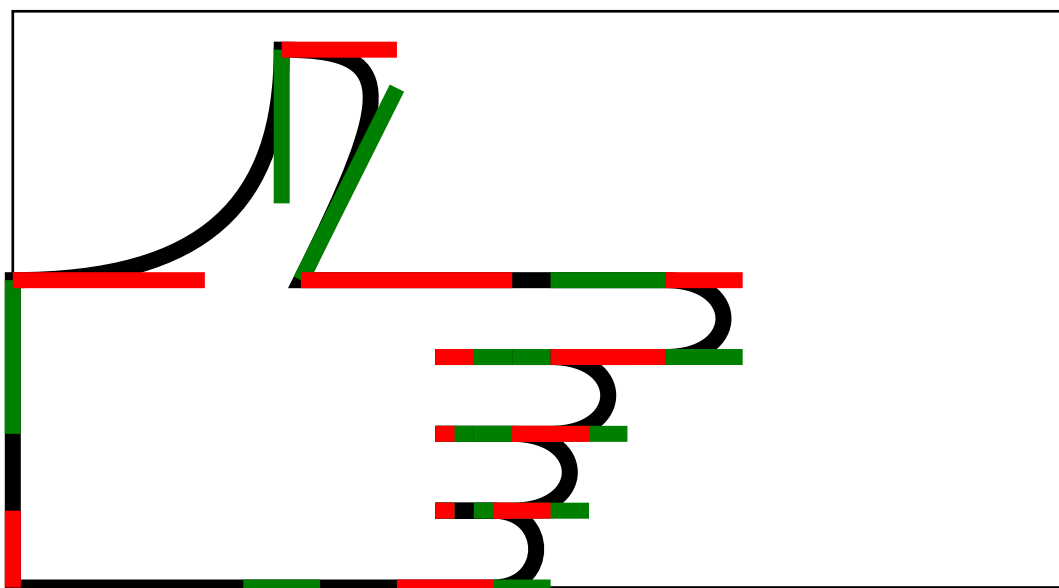


Figure A-51: a hand

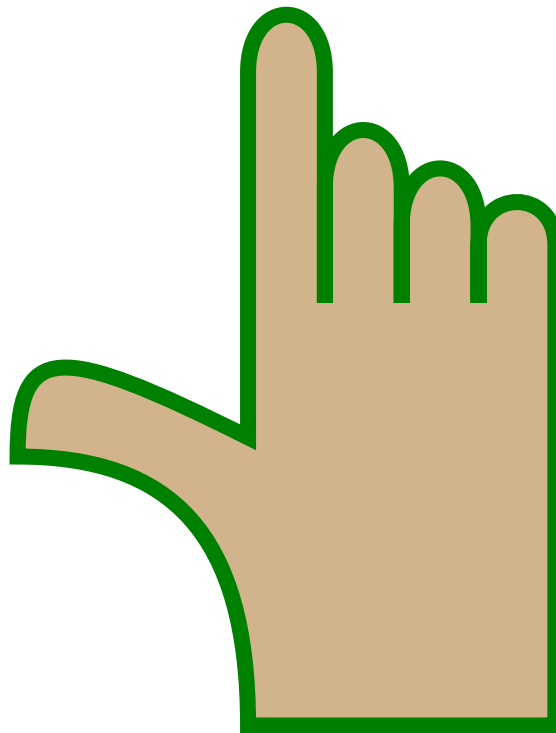
To embed this or any other drawing in a Platypus flowable we must define a subclass of Flowable with at least a wrap method and a draw method.

```
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)
```

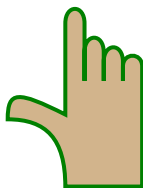
The wrap method must provide the size of the drawing -- it is used by the Platypus mainloop to decide whether this element fits in the space remaining on the current frame. The draw method performs the drawing of the object after the Platypus mainloop has translated the (0 , 0) origin to an appropriate location

in an appropriate frame.

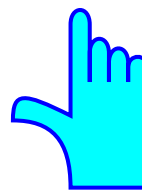
Below are some example uses of the `HandAnnotation` flowable.



The default.



Just one inch high.



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

9.2 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/replologo.gif')  
I.hAlign = 'CENTER'
```

produces



Chapter 10 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