# ReportLab API Reference

## Introduction

This is the API reference for the ReportLab library. All public classes, functions and methods are documented here.

Most of the reference text is built automatically from the documentation strings in each class, method and function. That's why it uses preformatted text and doesn't look very pretty.

Please note the following points:

(1) Items with one leading underscore are considered private to the modules they are defined in; they are not documented here and we make no commitment to their maintenance.

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

103 Bayard Street
New Brunswick
New Jersey, 08904)
USA

(2) Items ending in a digit (usually zero) are experimental; they are released to allow widespread testing, but are guaranteed to be broken in future (if only by dropping the zero). By all means play with these and give feedback, but do not use them in production scripts.

## Package Architecture

The reportlab package is broken into a number of subpackages. These are as follows:

**`reportlab.pdfgen`** - this is the programming interface to the PDF file format. The Canvas (and its co-workers, TextObject and PathObject) provide everything you need to create PDF output working at a low level - individual shapes and lines of text. Internally, it constructs blocks of *page marking operators* which match your drawing commands, and hand them over to the `pdfbase` package for drawing.

**`reportlab.pdfbase`** - this is not part of the public interface. It contains code to handle the 'outer structure' of PDF files, and utilities to handle text metrics and compressed streams.

**`reportlab.platypus`** - PLATYPUS stands for "Page Layout and Typography Using Scripts". It provides a higher level of abstraction dealing with paragraphs, frames on the page, and document templates. This is used for multi- page documents such as this reference.

**`reportlab.lib`** - this contains code of interest to application developers which cuts across both of our libraries, such as standard colors, units, and page sizes. It will also contain more drawable and flowable objects in future.

There is also a demos directory containing various demonstrations, and a docs directory. These can be accessed with package notation but should not be thought of as packages.

Each package is documented in turn.

# *reportlab.pdfgen* subpackage

This package contains three modules, canvas.py, textobject.py and pathobject.py, which define three classes of corresponding names. The only class users should construct directly is the Canvas, defined in reportlab.pdfgen.canvas; it provides methods to obtain PathObjects and TextObjects.

**Class Canvas:**

```
This class is the programmer's interface to the PDF file format.  Methods
are (or will be) provided here to do just about everything PDF can do.

The underlying model to the canvas concept is that of a graphics state machine
that at any given point in time has a current font, fill color (for figure
interiors), stroke color (for figure borders), line width and geometric transform, among
many other characteristics.

Canvas methods generally either draw something (like canvas.line) using the
current state of the canvas or change some component of the canvas
state (like canvas.setFont).  The current state can be saved and restored
using the saveState/restoreState methods.

Objects are "painted" in the order they are drawn so if, for example
two rectangles overlap the last draw will appear "on top".  PDF form
objects (supported here) are used to draw complex drawings only once,
for possible repeated use.

There are other features of canvas which are not visible when printed,
such as outlines and bookmarks which are used for navigating a document
in a viewer.

Here is a very silly example usage which generates a Hello World pdf document.

from reportlab.pdfgen import canvas
c = canvas.Canvas("hello.pdf")
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", 80)
# choose some colors
c.setStrokeColorRGB(0.2,0.5,0.3)
c.setFillColorRGB(1,0,1)
# draw a rectangle
c.rect(inch,inch,6*inch,9*inch, fill=1)
# make text go straight up
c.rotate(90)
# change color
c.setFillColorRGB(0,0,0.77)
# say hello (note after rotate the y coord needs to be negative!)
c.drawString(3*inch, -3*inch, "Hello World")
c.showPage()
c.save()
```

**def absolutePosition(self, x, y):**

```
return the absolute position of x,y in user space w.r.t. default user space
```

**def addFont(self, fontObj):**

```
add a new font for subsequent use.
```

**def addLiteral(self, s, escaped=1):**

```
introduce the literal text of PDF operations s into the current stream.
Only use this if you are an expert in the PDF file format.
```

**def addOutlineEntry(self, title, key, level=0, closed=None):**

```
Adds a new entry to the outline at given level.  If LEVEL not specified,
entry goes at the top level.  If level specified, it must be
no more than 1 greater than the outline level in the last call.

The key must be the (unique) name of a bookmark.
the title is the (non-unique) name to be displayed for the entry.

If closed is set then the entry should show no subsections by default
when displayed.
```

```
Example
```

```
   c.addOutlineEntry("first section", "section1")
   c.addOutlineEntry("introduction", "s1s1", 1, closed=1)
   c.addOutlineEntry("body", "s1s2", 1)
   c.addOutlineEntry("detail1", "s1s2s1", 2)
   c.addOutlineEntry("detail2", "s1s2s2", 2)
   c.addOutlineEntry("conclusion", "s1s3", 1)
   c.addOutlineEntry("further reading", "s1s3s1", 2)
   c.addOutlineEntry("second section", "section1")
   c.addOutlineEntry("introduction", "s2s1", 1)
   c.addOutlineEntry("body", "s2s2", 1, closed=1)
   c.addOutlineEntry("detail1", "s2s2s1", 2)
   c.addOutlineEntry("detail2", "s2s2s2", 2)
   c.addOutlineEntry("conclusion", "s2s3", 1)
   c.addOutlineEntry("further reading", "s2s3s1", 2)

generated outline looks like
    - first section
    |- introduction
    |- body
    |  |- detail1
    |  |- detail2
    |- conclusion
    |  |- further reading
    - second section
    |- introduction
    |+ body
    |- conclusion
    |  |- further reading

Note that the second "body" is closed.

Note that you can jump from level 5 to level 3 but not
from 3 to 5: instead you need to provide all intervening
levels going down (4 in this case).  Note that titles can
collide but keys cannot.
```

**def addPostScriptCommand(self, command, position=1):**

```
Embed literal Postscript in the document.

        With position=0, it goes at very beginning of page stream;
        with position=1, at current point; and
        with position=2, at very end of page stream.  What that does
        to the resulting Postscript depends on Adobe's header :-)

        Use with extreme caution, but sometimes needed for printer tray commands.
        Acrobat 4.0 will export Postscript to a printer or file containing
        the given commands.  Adobe Reader 6.0 no longer does as this feature is
        deprecated.  5.0, I don't know about (please let us know!). This was
        funded by Bob Marshall of Vector.co.uk and tested on a Lexmark 750.
        See test_pdfbase_postscript.py for 2 test cases - one will work on
        any Postscript device, the other uses a 'setpapertray' command which
        will error in Distiller but work on printers supporting it.
```

**def arc(self, x1,y1, x2,y2, startAng=0, extent=90):**

```
Draw a partial ellipse inscribed within the rectangle x1,y1,x2,y2,
starting at startAng degrees and covering extent degrees.   Angles
start with 0 to the right (+x) and increase counter-clockwise.
These should have x1<x2 and y1<y2.

Contributed to piddlePDF by Robert Kern, 28/7/99.
Trimmed down by AR to remove color stuff for pdfgen.canvas and
revert to positive coordinates.

The algorithm is an elliptical generalization of the formulae in
Jim Fitzsimmon's TeX tutorial <URL: http://www.tinaja.com/bezarc1.pdf>.
```

**def beginForm(self, name, lowerx=0, lowery=0, upperx=None, uppery=None):**

```
declare the current graphics stream to be a named form.
A graphics stream can either be a page or a form, not both.
Some operations (like bookmarking) are permitted for pages
but not forms.  The form will not automatically be shown in the
document but must be explicitly referenced using doForm in pages
that require the form.
```

**def beginPath(self):**

```
Returns a fresh path object.  Paths are used to draw
```

```
complex figures.  The object returned follows the protocol
for a pathobject.PDFPathObject instance
```

**def beginText(self, x=0, y=0):**

```
Returns a fresh text object.  Text objects are used
to add large amounts of text.  See textobject.PDFTextObject
```

**def bezier(self, x1, y1, x2, y2, x3, y3, x4, y4):**

```
Bezier curve with the four given control points
```

**def bookmarkHorizontal(self, key, relativeX, relativeY):**

```
w.r.t. the current transformation, bookmark this horizontal.
```

**def bookmarkHorizontalAbsolute(self, key, yhorizontal):**

```
Bind a bookmark (destination) to the current page at a horizontal position.
Note that the yhorizontal of the book mark is with respect to the default
user space (where the origin is at the lower left corner of the page)
and completely ignores any transform (translation, scale, skew, rotation,
etcetera) in effect for the current graphics state.  The programmer is
responsible for making sure the bookmark matches an appropriate item on
the page.
```

```
def bookmarkPage(self, key,
                 fitType="Fit",
                 left=None,
                 top=None,
                 bottom=None,
                 right=None,
                 zoom=None
                 ):
```

```
This creates a bookmark to the current page which can
be referred to with the given key elsewhere.

PDF offers very fine grained control over how Acrobat
reader is zoomed when people link to this. The default
is to keep the user's current zoom settings. the last
arguments may or may not be needed depending on the
choice of 'fitType'.

Fit types and the other arguments they use are:
/XYZ left top zoom - fine grained control.  null
  or zero for any of the parameters means 'leave
  as is', so "0,0,0" will keep the reader's settings.
  NB. Adobe Reader appears to prefer "null" to 0's.

/Fit - entire page fits in window

/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

(question: do we support /FitB, FitBH and /FitBV
which are hangovers from version 1.1 / Acrobat 3.0?)
```

**def circle(self, x_cen, y_cen, r, stroke=1, fill=0):**

```
draw a cirle centered at (x_cen,y_cen) with radius r (special case of ellipse)
```

**def clipPath(self, aPath, stroke=1, fill=0):**

```
clip as well as drawing
```

**def doForm(self, name):**

```
use a form XObj in current operation stream.

        The form should either have been defined previously using
        beginForm ... endForm, or may be defined later.  If it is not
        defined at save time, an exception will be raised. The form
        will be drawn within the context of the current graphics
        state.
```

**def drawAlignedString(self, x, y, text, pivotChar="."):**

```
Draws a string aligned on the first '.' (or other pivot character).

        The centre position of the pivot character will be used as x.
        So, you could draw a straight line down through all the decimals in a
        column of numbers, and anything without a decimal should be
        optically aligned with those that have.

        There is one special rule to help with accounting formatting.  Here's
        how normal numbers should be aligned on the 'dot'. Look at the
        LAST two:
           12,345,67
              987.15
               42
           -1,234.56
             (456.78)
             (456)
               27 inches
               13cm
        Since the last three do not contain a dot, a crude dot-finding
        rule would place them wrong. So we test for the special case
        where no pivot is found, digits are present, but the last character
        is not a digit.  We then work back from the end of the string
        This case is a tad slower but hopefully rare.
```

**def drawCentredString(self, x, y, text):**

```
Draws a string centred on the x coordinate.
```

**def drawImage(self, image, x, y, width=None, height=None, mask=None):**

```
Draws the image (ImageReader object or filename) as specified.

        "image" may be an image filename or a ImageReader object.  If width
        and height are not given, the "natural" width and height in pixels
        is used at a scale of 1 point to 1 pixel.

        The mask parameter 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 0-2, Green from 40-42 and
        Blue from 136-139  (on a scale of 0-255)

        The method returns the width and height of the underlying image since
        this is often useful for layout algorithms.

        Unlike drawInlineImage, this creates 'external images' which
        are only stored once in the PDF file but can be drawn many times.
        If you give it the same filename twice, even at different locations
        and sizes, it will reuse the first occurrence.  If you use ImageReader
        objects, it tests whether the image content has changed before deciding
        whether to reuse it.

        In general you should use drawImage in preference to drawInlineImage
        unless you have read the PDF Spec and understand the tradeoffs.
```

**def drawInlineImage(self, image, x,y, width=None,height=None):**

```
Draw an Image into the specified rectangle.  If width and
height are omitted, they are calculated from the image size.
Also allow file names as well as images.  The size in pixels
of the image is returned.
```

**def drawPath(self, aPath, stroke=1, fill=0):**

```
Draw the path object in the mode indicated
```

**def drawRightString(self, x, y, text):**

```
Draws a string right-aligned with the x coordinate
```

**def drawString(self, x, y, text):**

```
Draws a string in the current text styles.
```

**def drawText(self, aTextObject):**

```
Draws a text object
```

**def ellipse(self, x1, y1, x2, y2, stroke=1, fill=0):**

```
Draw an ellipse defined by an enclosing rectangle.

        Note that (x1,y1) and (x2,y2) are the corner points of
        the enclosing rectangle.

        Uses bezierArc, which conveniently handles 360 degrees.
        Special thanks to Robert Kern.
```

**def endForm(self):**

```
emit the current collection of graphics operations as a Form
as declared previously in beginForm.
```

**def getAvailableFonts(self):**

```
Returns the list of PostScript font names available.

        Standard set now, but may grow in future with font embedding.
```

**def getPageNumber(self):**

```
get the page number for the current page being generated.
```

**def getpdfdata(self):**

```
Returns the PDF data that would normally be written to a file.
If there is current data a ShowPage is executed automatically.
After this operation the canvas must not be used further.
```

**def grid(self, xlist, ylist):**

```
Lays out a grid in current line style.  Supply list of
x an y positions.
```

**def hasForm(self, name):**

```
Query whether form XObj really exists yet.
```

**def init_graphics_state(self):**

```
(no documentation string)
```

**def line(self, x1,y1, x2,y2):**

```
draw a line segment from (x1,y1) to (x2,y2) (with color, thickness and
other attributes determined by the current graphics state).
```

**def lines(self, linelist):**

```
Like line(), permits many lines to be drawn in one call.
for example for the figure
    |
  -- --
    |

  crosshairs = [(20,0,20,10), (20,30,20,40), (0,20,10,20), (30,20,40,20)]
  canvas.lines(crosshairs)
```

**def linkAbsolute(self, contents, destinationname, Rect=None, addtopage=1, name=None, **kw):**

```
rectangular link annotation positioned wrt the default user space.
The identified rectangle on the page becomes a "hot link" which
when clicked will send the viewer to the page and position identified
by the destination.

Rect identifies (lowerx, lowery, upperx, uppery) for lower left
and upperright points of the rectangle.  Translations and other transforms
are IGNORED (the rectangular position is given with respect
to the default user space.
destinationname should be the name of a bookmark (which may be defined later
but must be defined before the document is generated).

You may want to use the keyword argument Border='[0 0 0]' to
suppress the visible rectangle around the during viewing link.
```

**def linkRect(self, contents, destinationname, Rect=None, addtopage=1, name=None, **kw):**

```
rectangular link annotation w.r.t the current user transform.
if the transform is skewed/rotated the absolute rectangle will use the max/min x/y
```

**def linkURL(self, url, rect, relative=0, thickness=0, color=None, dashArray=None, kind="URI"):**

```
Create a rectangular URL 'hotspot' in the given rectangle.

        if relative=1, this is in the current coord system, otherwise
        in absolute page space.
        The remaining options affect the border appearance; the border is
        drawn by Acrobat, not us.  Set thickness to zero to hide it.
        Any border drawn this way is NOT part of the page stream and
        will not show when printed to a Postscript printer or distilled;
        it is safest to draw your own.
```

**def pageHasData(self):**

```
Info function - app can call it after showPage to see if it needs a save
```

**def pop_state_stack(self):**

```
(no documentation string)
```

**def push_state_stack(self):**

```
(no documentation string)
```

**def rect(self, x, y, width, height, stroke=1, fill=0):**

```
draws a rectangle with lower left corner at (x,y) and width and height as given.
```

**def resetTransforms(self):**

```
I want to draw something (eg, string underlines) w.r.t. the default user space.
Reset the matrix! This should be used usually as follows:
   canv.saveState()
   canv.resetTransforms()
   ...draw some stuff in default space coords...
   canv.restoreState() # go back!
```

**def restoreState(self):**

```
restore the graphics state to the matching saved state (see saveState).
```

**def rotate(self, theta):**

```
Canvas.rotate(theta)

        Rotate the canvas by the angle theta (in degrees).
```

**def roundRect(self, x, y, width, height, radius, stroke=1, fill=0):**

```
Draws a rectangle with rounded corners.  The corners are
approximately quadrants of a circle, with the given radius.
```

**def save(self):**

```
Saves and close the PDF document in the file.
If there is current data a ShowPage is executed automatically.
After this operation the canvas must not be used further.
```

**def saveState(self):**

```
Save the current graphics state to be restored later by restoreState.

        For example:
            canvas.setFont("Helvetica", 20)
            canvas.saveState()
            ...
            canvas.setFont("Courier", 9)
            ...
            canvas.restoreState()
            # if the save/restore pairs match then font is Helvetica 20 again.
```

**def scale(self, x, y):**

```
Scale the horizontal dimension by x and the vertical by y
(with respect to the current graphics state).
For example canvas.scale(2.0, 0.5) will make everything short and fat.
```

**def setAuthor(self, author):**

```
identify the author for invisible embedding inside the PDF document.
the author annotation will appear in the the text of the file but will
not automatically be seen when the document is viewed.
```

**def setDash(self, array=[], phase=0):**

```
Two notations.  pass two numbers, or an array and phase
```

**def setFont(self, psfontname, size, leading = None):**

```
Sets the font.  If leading not specified, defaults to 1.2 x
font size. Raises a readable exception if an illegal font
is supplied.  Font names are case-sensitive! Keeps track
of font name and size for metrics.
```

**def setFontSize(self, size=None, leading=None):**

```
Sets font size or leading without knowing the font face
```

**def setLineCap(self, mode):**

```
0=butt,1=round,2=square
```

**def setLineJoin(self, mode):**

```
0=mitre, 1=round, 2=bevel
```

**def setLineWidth(self, width):**

```
(no documentation string)
```

**def setMiterLimit(self, limit):**

```
(no documentation string)
```

**def setPageCallBack(self, func):**

```
func(pageNum) will be called on each page end.

       This is mainly a hook for progress monitoring.
        Call setPageCallback(None) to clear a callback.
```

**def setPageCompression(self, pageCompression=1):**

```
Possible values None, 1 or 0
If None the value from rl_config will be used.
If on, the page data will be compressed, leading to much
smaller files, but takes a little longer to create the files.
This applies to all subsequent pages, or until setPageCompression()
is next called.
```

**def setPageDuration(self, duration=None):**

```
Allows hands-off animation of presentations :-)

        If this is set to a number, in full screen mode, Acrobat Reader
        will advance to the next page after this many seconds. The
        duration of the transition itself (fade/flicker etc.) is controlled
        by the 'duration' argument to setPageTransition; this controls
        the time spent looking at the page.  This is effective for all
        subsequent pages.
```

**def setPageRotation(self, rot):**

```
Instruct display device that this page is to be rotated
```

**def setPageSize(self, size):**

```
accepts a 2-tuple in points for paper size for this
and subsequent pages
```

**def setPageTransition(self, effectname=None, duration=1,
                    direction=0,dimension='H',motion='I'):**

```
PDF allows page transition effects for use when giving
presentations.  There are six possible effects.  You can
just guive the effect name, or supply more advanced options
to refine the way it works.  There are three types of extra
argument permitted, and here are the allowed values:
    direction_arg = [0,90,180,270]
    dimension_arg = ['H', 'V']
    motion_arg = ['I','O'] (start at inside or outside)

This table says which ones take which arguments:

PageTransitionEffects = {
    'Split': [direction_arg, motion_arg],
    'Blinds': [dimension_arg],
    'Box': [motion_arg],
    'Wipe' : [direction_arg],
    'Dissolve' : [],
    'Glitter':[direction_arg]
    }
Have fun!
```

**def setSubject(self, subject):**

```
write a subject into the PDF file that won't automatically display
in the document itself.
```

**def setTitle(self, title):**

```
write a title into the PDF file that won't automatically display
in the document itself.
```

**def showOutline(self):**

```
Specify that Acrobat Reader should start with the outline tree visible.
showFullScreen() and showOutline() conflict; the one called last
wins.
```

**def showPage(self):**

```
Close the current page and possibly start on a new page.
```

**def skew(self, alpha, beta):**

```
(no documentation string)
```

**def stringWidth(self, text, fontName, fontSize, encoding=None):**

```
gets width of a string in the given font and size
```

**def textAnnotation(self, contents, Rect=None, addtopage=1, name=None, **kw):**

```
Experimental, but works.
```

**def transform(self, a,b,c,d,e,f):**

```
adjoin a mathematical transform to the current graphics state matrix.
Not recommended for beginners.
```

**def translate(self, dx, dy):**

```
move the origin from the current (0,0) point to the (dx,dy) point
(with respect to the current graphics state).
```

**def wedge(self, x1,y1, x2,y2, startAng, extent, stroke=1, fill=0):**

```
Like arc, but connects to the centre of the ellipse.
Most useful for pie charts and PacMan!
```

The method Canvas.beginPath allows users to construct a PDFPathObject, which is defined in reportlab/pdfgen/pathobject.py.

**Class PDFPathObject:**

```
Represents a graphic path.  There are certain 'modes' to PDF
drawing, and making a separate object to expose Path operations
ensures they are completed with no run-time overhead.  Ask
the Canvas for a PDFPath with getNewPathObject(); moveto/lineto/
curveto wherever you want; add whole shapes; and then add it back
into the canvas with one of the relevant operators.

Path objects are probably not long, so we pack onto one line
```

**def arc(self, x1,y1, x2,y2, startAng=0, extent=90):**

```
Contributed to piddlePDF by Robert Kern, 28/7/99.
Draw a partial ellipse inscribed within the rectangle x1,y1,x2,y2,
starting at startAng degrees and covering extent degrees.   Angles
start with 0 to the right (+x) and increase counter-clockwise.
These should have x1<x2 and y1<y2.

The algorithm is an elliptical generalization of the formulae in
Jim Fitzsimmon's TeX tutorial <URL: http://www.tinaja.com/bezarc1.pdf>.
```

**def arcTo(self, x1,y1, x2,y2, startAng=0, extent=90):**

```
Like arc, but draws a line from the current point to
the start if the start is not the current point.
```

**def circle(self, x_cen, y_cen, r):**

```
adds a circle to the path
```

**def close(self):**

```
draws a line back to where it started
```

**def curveTo(self, x1, y1, x2, y2, x3, y3):**

```
(no documentation string)
```

**def ellipse(self, x, y, width, height):**

```
adds an ellipse to the path
```

**def getCode(self):**

```
pack onto one line; used internally
```

**def lineTo(self, x, y):**

```
(no documentation string)
```

**def moveTo(self, x, y):**

```
(no documentation string)
```

**def rect(self, x, y, width, height):**

```
Adds a rectangle to the path
```

The method Canvas.beginText allows users to construct a PDFTextObject, which is defined in reportlab/pdfgen/textobject.py.

**Class PDFTextObject:**

PDF logically separates text and graphics drawing; text operations need to be bracketed between BT (Begin text) and ET operators. This class ensures text operations are properly encapusalted. Ask the canvas for a text object with beginText(x, y). Do not construct one directly. Do not use multiple text objects in parallel; PDF is not multi-threaded!

It keeps track of x and y coordinates relative to its origin.

**def getCode(self):**

pack onto one line; used internally

**def getCursor(self):**

Returns current text position relative to the last origin.

**def getStartOfLine(self):**

Returns a tuple giving the text position of the start of the current line.

**def getX(self):**

Returns current x position relative to the last origin.

**def getY(self):**

Returns current y position relative to the last origin.

**def moveCursor(self, dx, dy):**

Starts a new line at an offset dx,dy from the start of the current line. This does not move the cursor relative to the current position, and it changes the current offset of every future line drawn (i.e. if you next do a textLine() call, it will move the cursor to a position one line lower than the position specificied in this call.

**def setCharSpace(self, charSpace):**

Adjusts inter-character spacing

**def setFont(self, psfontname, size, leading = None):**

Sets the font. If leading not specified, defaults to 1.2 x font size. Raises a readable exception if an illegal font is supplied. Font names are case-sensitive! Keeps track of font anme and size for metrics.

**def setHorizScale(self, horizScale):**

Stretches text out horizontally

**def setLeading(self, leading):**

How far to move down at the end of a line.

**def setRise(self, rise):**

Move text baseline up or down to allow superscrip/subscripts

**def setTextOrigin(self, x, y):**

(no documentation string)

**def setTextRenderMode(self, mode):**

Set the text rendering mode.

0 = Fill text
1 = Stroke text
2 = Fill then stroke
3 = Invisible
4 = Fill text and add to clipping path
5 = Stroke text and add to clipping path
6 = Fill then stroke and add to clipping path
7 = Add to clipping path

**def setTextTransform(self, a, b, c, d, e, f):**

```
Like setTextOrigin, but does rotation, scaling etc.
```

**`def setWordSpace(self, wordSpace):`**

```
Adjust inter-word spacing.  This can be used
to flush-justify text - you get the width of the
words, and add some space between them.
```

**`def setXPos(self, dx):`**

```
Starts a new line dx away from the start of the
current line - NOT from the current point! So if
you call it in mid-sentence, watch out.
```

**`def textLine(self, text=''):`**

```
prints string at current point, text cursor moves down.
Can work with no argument to simply move the cursor down.
```

**`def textLines(self, stuff, trim=1):`**

```
prints multi-line or newlined strings, moving down.  One
comon use is to quote a multi-line block in your Python code;
since this may be indented, by default it trims whitespace
off each line and from the beginning; set trim=0 to preserve
whitespace.
```

**`def textOut(self, text):`**

```
prints string at current point, text cursor moves across.
```

# *reportlab.platypus* subpackage

The platypus package defines our high-level page layout API. The division into modules is far from final and has been based more on balancing the module lengths than on any particular programming interface. The __init__ module imports the key classes into the top level of the package.

## Overall Structure

Abstractly Platypus currently can be thought of has having four levels: documents, pages, frames and flowables (things which can fit into frames in some way). In practice there is a fifth level, the canvas, so that if you want you can do anything that pdfgen's canvas allows.

## Document Templates

### *BaseDocTemplate*

The basic document template class; it provides for initialisation and rendering of documents. A whole bunch of methods **`handle_XXX`** handle document rendering events. These event routines all contain some significant semantics so while these may be overridden that may require some detailed knowledge. Some other methods are completely virtual and are designed to be overridden.

### *BaseDocTemplate*

**`Class BaseDocTemplate:`**

```
First attempt at defining a document template class.

The basic idea is simple.
0)  The document has a list of data associated with it
    this data should derive from flowables. We'll have
    special classes like PageBreak, FrameBreak to do things
    like forcing a page end etc.

1)  The document has one or more page templates.

2)  Each page template has one or more frames.

3)  The document class provides base methods for handling the
    story events and some reasonable methods for getting the
    story flowables into the frames.

4)  The document instances can override the base handler routines.

Most of the methods for this class are not called directly by the user,
but in some advanced usages they may need to be overridden via subclassing.

EXCEPTION: doctemplate.build(...) must be called for most reasonable uses
since it builds a document using the page template.

Each document template builds exactly one document into a file specified
by the filename argument on initialization.

Possible keyword arguments for the initialization:

pageTemplates: A list of templates.  Must be nonempty.  Names
  assigned to the templates are used for referring to them so no two used
  templates should have the same name.  For example you might want one template
  for a title page, one for a section first page, one for a first page of
  a chapter and two more for the interior of a chapter on odd and even pages.
  If this argument is omitted then at least one pageTemplate should be provided
  using the addPageTemplates method before the document is built.
pageSize: a 2-tuple or a size constant from reportlab/lib/pagesizes.pu.
 Used by the SimpleDocTemplate subclass which does NOT accept a list of
 pageTemplates but makes one for you; ignored when using pageTemplates.

showBoundary: if set draw a box around the frame boundaries.
leftMargin:
rightMargin:
topMargin:
```

```
bottomMargin:  Margin sizes in points (default 1 inch)
  These margins may be overridden by the pageTemplates.  They are primarily of interest
  for the SimpleDocumentTemplate subclass.
allowSplitting:  If set flowables (eg, paragraphs) may be split across frames or pages
  (default: 1)
title: Internal title for document (does not automatically display on any page)
author: Internal author for document (does not automatically display on any page)
```

**def addPageTemplates(self,pageTemplates):**

```
add one or a sequence of pageTemplates
```

**def afterFlowable(self, flowable):**

```
called after a flowable has been rendered
```

**def afterInit(self):**

```
This is called after initialisation of the base class.
```

**def afterPage(self):**

```
This is called after page processing, and
immediately after the afterDrawPage method
of the current page template.
```

**def beforeDocument(self):**

```
This is called before any processing is
done on the document.
```

**def beforePage(self):**

```
This is called at the beginning of page
processing, and immediately before the
beforeDrawPage method of the current page
template.
```

**def build(self, flowables, filename=None, canvasmaker=canvas.Canvas):**

```
Build the document from a list of flowables.
If the filename argument is provided then that filename is used
rather than the one provided upon initialization.
If the canvasmaker argument is provided then it will be used
instead of the default.  For example a slideshow might use
an alternate canvas which places 6 slides on a page (by
doing translations, scalings and redefining the page break
operations).
```

**def clean_hanging(self):**

```
handle internal postponed actions
```

**def filterFlowables(self,flowables):**

```
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.
```

**def handle_breakBefore(self, flowables):**

```
preprocessing step to allow pageBreakBefore and frameBreakBefore attributes
```

**def handle_currentFrame(self,fx,resume=0):**

```
change to the frame with name or index fx
```

**def handle_documentBegin(self):**

```
implement actions at beginning of document
```

**def handle_flowable(self,flowables):**

```
try to handle one flowable from the front of list flowables.
```

**def handle_frameBegin(self,resume=0):**

```
What to do at the beginning of a frame
```

**def handle_frameEnd(self,resume=0):**

```
Handles the semantics of the end of a frame. This includes the selection of
the next frame or if this is the last frame then invoke pageEnd.
```

**def handle_keepWithNext(self, flowables):**

```
implements keepWithNext
```

**def handle_nextFrame(self,fx,resume=0):**

```
On endFrame change to the frame with name or index fx
```

**def handle_nextPageTemplate(self,pt):**

```
On endPage change to the page template with name or index pt
```

**def handle_pageBegin(self):**

```
Perform actions required at beginning of page.
shouldn't normally be called directly
```

**def handle_pageBreak(self,slow=None):**

```
some might choose not to end all the frames
```

**def handle_pageEnd(self):**

```
show the current page
check the next page template
hang a page begin
```

**def multiBuild(self, story,**
**filename=None,**
**canvasmaker=canvas.Canvas,**
**maxPasses = 10):**

```
Makes multiple passes until all indexing flowables
are happy.
```

**def notify(self, kind, stuff):**

```
"Forward to any listeners
```

**def pageRef(self, label):**

```
hook to register a page number
```

**def setPageCallBack(self, func):**

```
Simple progress monitor - func(pageNo) called on each new page
```

**def setProgressCallBack(self, func):**

```
Cleverer progress monitor - func(typ, value) called regularly
```

A simple document processor can be made using derived class, **`SimpleDocTemplate`**.

## *SimpleDocTemplate*

**Class SimpleDocTemplate:**

A special case document template that will handle many simple documents. See documentation for BaseDocTemplate. No pageTemplates are required for this special case. A page templates are inferred from the margin information and the onFirstPage, onLaterPages arguments to the build method.

A document which has all pages with the same look except for the first page may can be built using this special approach.

**def build(self,flowables,onFirstPage=_doNothing, onLaterPages=_doNothing, canvasmaker=canvas.Canvas):**

build the document using the flowables. Annotate the first page using the onFirstPage function and later pages using the onLaterPages function. The onXXX pages should follow the signature

```
def myOnFirstPage(canvas, document):
    # do annotations and modify the document
    ...
```

The functions can do things like draw logos, page numbers, footers, etcetera. They can use external variables to vary the look (for example providing page numbering or section names).

**def handle_pageBegin(self):**

override base method to add a change of page template after the firstpage.

## Flowables

**Class Paragraph:**

```
Paragraph(text, style, bulletText=None, caseSensitive=1)
text a string of stuff to go into the paragraph.
style is a style definition as in reportlab.lib.styles.
bulletText is an optional bullet defintion.
caseSensitive set this to 0 if you want the markup tags and their attributes to be case-insensitive.

This class is a flowable that can format a block of text
into a paragraph with a given style.

The paragraph Text can contain XML-like markup including the tags:
<b> ... </b> - bold
<i> ... </i> - italics
<u> ... </u> - underline
<strike> ... </strike> - strike through
<super> ... </super> - superscript
<sub> ... </sub> - subscript
<font name=fontfamily/fontname color=colorname size=float>
<onDraw name=callable label="a label">
<link>link text</link>
    attributes of links
        size/fontSize=num
        name/face/fontName=name
        fg/textColor/color=color
        backcolor/backColor/bgcolor=color
        dest/destination/target/href/link=target

The whole may be surrounded by <para> </para> tags

It will also be able to handle any MathML specified Greek characters.
```

**def beginText(self, x, y):**

```
(no documentation string)
```

**def breakLines(self, width):**

```
Returns a broken line structure. There are two cases

A) For the simple case of a single formatting input fragment the output is
    A fragment specifier with
        kind = 0
        fontName, fontSize, leading, textColor
        lines=  A list of lines
                Each line has two items.
                1) unused width in points
                2) word list

B) When there is more than one input formatting fragment the output is
    A fragment specifier with
        kind = 1
        lines=  A list of fragments each having fields
                    extraspace (needed for justified)
                    fontSize
                    words=word list
                        each word is itself a fragment with
                        various settings

This structure can be used to easily draw paragraphs with the various alignments.
You can supply either a single width or a list of widths; the latter will have its
last item repeated until necessary. A 2-element list is useful when there is a
different first line indent; a longer list could be created to facilitate custom wraps
around irregular objects.
```

**def draw(self):**

```
(no documentation string)
```

**def drawPara(self,debug=0):**

```
Draws a paragraph according to the given style.
Returns the final y position at the bottom. Not safe for
paragraphs without spaces e.g. Japanese; wrapping
algorithm will go infinite.
```

**def getPlainText(self,identify=None):**

```
Convenience function for templates which want access
to the raw text, without XML tags.
```

**def minWidth(self):**

```
Attempt to determine a minimum sensible width
```

**def split(self,availWidth, availHeight):**

```
(no documentation string)
```

**def wrap(self, availWidth, availHeight):**

```
(no documentation string)
```

**Class Flowable:**

```
Abstract base class for things to be drawn.  Key concepts:
1. It knows its size
2. It draws in its own coordinate system (this requires the
    base API to provide a translate() function.
```

**def drawOn(self, canvas, x, y, _sW=0):**

```
Tell it to draw itself on the canvas.  Do not override
```

**def getKeepWithNext(self):**

```
returns boolean determining whether the next flowable should stay with this one
```

**def getSpaceAfter(self):**

```
returns how much space should follow this item if another item follows on the same page.
```

**def getSpaceBefore(self):**

```
returns how much space should precede this item if another item precedess on the same page.
```

**def identity(self, maxLen=None):**

```
This method should attempt to return a string that can be used to identify
a particular flowable uniquely. The result can then be used for debugging
and or error printouts
```

**def isIndexing(self):**

```
Hook for IndexingFlowables - things which have cross references
```

**def minWidth(self):**

```
This should return the minimum required width
```

**def split(self, availWidth, availheight):**

```
This will be called by more sophisticated frames when
wrap fails. Stupid flowables should return []. Clever flowables
should split themselves and return a list of flowables
```

**def splitOn(self, canv, aW, aH):**

```
intended for use by packers allows setting the canvas on
during the actual split
```

**def wrap(self, 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.
```

**def wrapOn(self, canv, aW, aH):**

```
intended for use by packers allows setting the canvas on
during the actual wrap
```

**Class XBox:**

```
Example flowable - a box with an x through it and a caption.
This has a known size, so does not need to respond to wrap().
```

**def draw(self):**

```
(no documentation string)
```

**Class Preformatted:**

```
This is like the HTML <PRE> tag.
It attempts to display text exactly as you typed it in a fixed width "typewriter" font.
The line breaks are exactly where you put
them, and it will not be wrapped.
```

**def draw(self):**

(no documentation string)

**def split(self, availWidth, availHeight):**

(no documentation string)

**def wrap(self, availWidth, availHeight):**

(no documentation string)

**Class Image:**

```
an image (digital picture).  Formats supported by PIL/Java 1.4 (the Python/Java Imaging Library
are supported.  At the present time images as flowables are always centered horozontally
in the frame. We allow for two kinds of lazyness to allow for many images in a document
which could lead to file handle starvation.
lazy=1 don't open image until required.
lazy=2 open image when required then shut it.
```

**def draw(self):**

(no documentation string)

**def identity(self,maxLen=None):**

(no documentation string)

**def wrap(self, availWidth, availHeight):**

(no documentation string)

**Class Spacer:**

```
A spacer just takes up space and doesn't draw anything - it guarantees
a gap between objects.
```

**def draw(self):**

(no documentation string)

**Class PageBreak:**

```
Move on to the next page in the document.
This works by consuming all remaining space in the frame!
```

**Class CondPageBreak:**

Throw a page if not enough vertical space

**def wrap(self, availWidth, availHeight):**

(no documentation string)

**Class KeepTogether:**

(no documentation string)

**def identity(self, maxLen=None):**

(no documentation string)

**def split(self, aW, aH):**

(no documentation string)

**def wrap(self, aW, aH):**

(no documentation string)

**Class Macro:**

```
This is not actually drawn (i.e. it has zero height)
but is executed when it would fit in the frame.  Allows direct
access to the canvas through the object 'canvas'
```

**def draw(self):**

(no documentation string)

**def wrap(self, availWidth, availHeight):**

(no documentation string)

**Class XPreformatted:**

(no documentation string)

**def breakLines(self, width):**

```
Returns a broken line structure. There are two cases

A) For the simple case of a single formatting input fragment the output is
    A fragment specifier with
        kind = 0
        fontName, fontSize, leading, textColor
        lines=  A list of lines
                Each line has two items.
                1) unused width in points
                2) a list of words

B) When there is more than one input formatting fragment the out put is
    A fragment specifier with
        kind = 1
        lines=  A list of fragments each having fields
                    extraspace (needed for justified)
                    fontSize
                    words=word list
                        each word is itself a fragment with
                        various settings

This structure can be used to easily draw paragraphs with the various alignments.
You can supply either a single width or a list of widths; the latter will have its
last item repeated until necessary. A 2-element list is useful when there is a
different first line indent; a longer list could be created to facilitate custom wraps
around irregular objects.
```

**Class PythonPreformatted:**

Used for syntax-colored Python code, otherwise like XPreformatted.

**def escapeHtml(self, text):**

(no documentation string)

**def fontify(self, code):**

Return a fontified version of some Python code.

# *reportlab.lib* subpackage

This package contains a number of modules which either add utility to pdfgen and platypus, or which are of general use in graphics applications.

## *reportlab.lib.colors* module

**`def Blacker(c,f):`**

```
given a color combine with black as c*f+b*(1-f) 0<=f<=1
```

**`def HexColor(val):`**

```
This function converts a hex string, or an actual integer number,
into the corresponding color.  E.g., in "AABBCC" or 0xAABBCC,
AA is the red, BB is the green, and CC is the blue (00-FF).

HTML uses a hex string with a preceding hash; if this is present,
it is stripped off.  (AR, 3-3-2000)

For completeness I assume that #aabbcc or 0xaabbcc are hex numbers
otherwise a pure integer is converted as decimal rgb
```

**`def Whiter(c,f):`**

```
given a color combine with white as c*f w*(1-f) 0<=f<=1
```

**`def cmyk2rgb((c,m,y,k),density=1):`**

```
Convert from a CMYK color tuple to an RGB color tuple
```

**`def cmykDistance(col1, col2):`**

```
Returns a number between 0 and root(4) stating how similar
two colours are - distance in r,g,b, space.  Only used to find
names for things.
```

**`def color2bw(colorRGB):`**

```
Transform an RGB color to a black and white equivalent.
```

**`def colorDistance(col1, col2):`**

```
Returns a number between 0 and root(3) stating how similar
two colours are - distance in r,g,b, space.  Only used to find
names for things.
```

**`def describe(aColor,mode=0):`**

```
finds nearest colour match to aColor.
mode=0 print a string desription
mode=1 return a string description
mode=2 return (distance, colorName)
```

**`def getAllNamedColors():`**

```
(no documentation string)
```

**`def linearlyInterpolatedColor(c0, c1, x0, x1, x):`**

```
Linearly interpolates colors. Can handle RGB, CMYK and PCMYK
colors - give ValueError if colours aren't the same.
Doesn't currently handle 'Spot Color Interpolation'.
```

**`def rgb2cmyk(r,g,b):`**

```
one way to get cmyk from rgb
```

**`def setColors(**kw):`**

```
(no documentation string)
```

**`def toColor(arg,default=None):`**

```
try to map an arbitrary arg to a color instance
```

**`def toColorOrNone(arg,default=None):`**

```
as above but allows None as a legal value
```

**`Class CMYKColor:`**

```
This represents colors using the CMYK (cyan, magenta, yellow, black)
model commonly used in professional printing.  This is implemented
```

```
as a derived class so that renderers which only know about RGB "see it"
as an RGB color through its 'red','green' and 'blue' attributes, according
to an approximate function.

The RGB approximation is worked out when the object in constructed, so
the color attributes should not be changed afterwards.

Extra attributes may be attached to the class to support specific ink models,
and renderers may look for these.
```

**def cmyk(self):**

```
Returns a tuple of four color components - syntactic sugar
```

**Class Color:**

```
This class is used to represent color.  Components red, green, blue
are in the range 0 (dark) to 1 (full intensity).
```

**def bitmap_rgb(self):**

```
(no documentation string)
```

**def hexval(self):**

```
(no documentation string)
```

**def rgb(self):**

```
Returns a three-tuple of components
```

**Class PCMYKColor:**

```
100 based CMYKColor with density and a spotName; just like Rimas uses
```

## *reportlab.lib.corp* module

```
This module includes some reusable routines for ReportLab's 'Corporate Image' - the
logo, standard page backdrops and so on - you are advised to do the same for your own
company!
```

**def test():**

```
This function produces a pdf with examples.
```

**Class RL_BusinessCard:**

```
Widget that creates a single business card.
Uses RL_CorpLogo for the logo.

For a black border around your card, set self.border to 1.
To change the details on the card, over-ride the following properties:
self.name, self.position, self.telephone, self.mobile, self.fax, self.email, self.web
The office locations are set in self.rh_blurb_top ("London office" etc), and
self.rh_blurb_bottom ("New York office" etc).
```

**def demo(self):**

```
(no documentation string)
```

**def draw(self):**

```
(no documentation string)
```

**Class RL_CorpLogo:**

```
Dinu's fat letter logo as hacked into decent paths by Robin
```

**def demo(self):**

```
(no documentation string)
```

**def draw(self):**

```
(no documentation string)
```

**Class RL_CorpLogoReversed:**

```
(no documentation string)
```

**Class RL_CorpLogoThin:**

```
The ReportLab Logo.

    New version created by John Precedo on 7-8 August 2001.
    Based on bitmapped imaged from E-Id.
```

Improved by Robin Becker.

**def demo(self):**

(no documentation string)

**def draw(self):**

(no documentation string)

**Class ReportLabLogo:**

vector reportlab logo centered in a 250x by 150y rectangle

**def draw(self, canvas):**

(no documentation string)

## *reportlab.lib.enums* module

holder for all reportlab's enumerated types

## *reportlab.lib.fonts* module

**def addMapping(face, bold, italic, psname):**

allow a custom font to be put in the mapping

**def ps2tt(psfn):**

ps fontname to family name, bold, italic

**def tt2ps(fn,b,i):**

family name + bold & italic to ps font name

## *reportlab.lib.pagesizes* module

This module defines a few common page sizes in points (1/72 inch). To be expanded to include things like label sizes, envelope windows etc.

**def landscape(pagesize):**

Use this to get page orientation right

**def portrait(pagesize):**

Use this to get page orientation right

## *reportlab.lib.sequencer* module

**def getSequencer():**

(no documentation string)

**def setSequencer(seq):**

(no documentation string)

**def test():**

(no documentation string)

**Class Sequencer:**

```
Something to make it easy to number paragraphs, sections,
images and anything else.  The features include registering
new string formats for sequences, and 'chains' whereby
some counters are reset when their parents.
It keeps track of a number of
'counters', which are created on request:
Usage:
    >>> seq = layout.Sequencer()
    >>> seq.next('Bullets')
    1
    >>> seq.next('Bullets')
    2
    >>> seq.next('Bullets')
    3
    >>> seq.reset('Bullets')
    >>> seq.next('Bullets')
```

```
1
>>> seq.next('Figures')
1
>>>
```

**def chain(self, parent, child):**

(no documentation string)

**def dump(self):**

Write current state to stdout for diagnostics

**def format(self, template):**

The crowning jewels - formats multi-level lists.

**def next(self, counter=None):**

Retrieves the numeric value for the given counter, then increments it by one.  New counters start at one.

**def nextf(self, counter=None):**

Retrieves the numeric value for the given counter, then increments it by one.  New counters start at one.

**def registerFormat(self, format, func):**

Registers a new formatting function.  The funtion must take a number as argument and return a string; fmt is a short menmonic string used to access it.

**def reset(self, counter=None, base=0):**

(no documentation string)

**def setDefaultCounter(self, default=None):**

Changes the key used for the default

**def setFormat(self, counter, format):**

Specifies that the given counter should use the given format henceforth.

**def thisf(self, counter=None):**

(no documentation string)

# Appendix A - CVS Revision History

```
$Log: reference.yml,v $
Revision 1.1  2001/10/05 12:33:33  rgbecker
Moved from original project docs, history lost

Revision 1.13  2001/08/30 10:32:38  dinu_gherman
Added missing flowables.

Revision 1.12  2001/07/11 09:21:27  rgbecker
Typo fix from Jerome Alet

Revision 1.11  2000/07/10 23:56:09  andy_robinson
Paragraphs chapter pretty much complete.  Fancy cover.

Revision 1.10  2000/07/03 15:39:51  rgbecker
Documentation fixes

Revision 1.9  2000/06/28 14:52:43  rgbecker
Documentation changes

Revision 1.8  2000/06/19 23:52:31  andy_robinson
rltemplate now simple, based on UserDocTemplate

Revision 1.7  2000/06/17 07:46:45  andy_robinson
Small text changes

Revision 1.6  2000/06/14 21:22:52  andy_robinson
Added docs for library

Revision 1.5  2000/06/12 11:26:34  andy_robinson
Numbered list added

Revision 1.4  2000/06/12 11:13:09  andy_robinson
Added sequencer tags to paragraph parser

Revision 1.3  2000/06/09 01:44:24  aaron_watters
added automatic generation for pathobject and textobject modules.

Revision 1.2  2000/06/07 13:39:22  andy_robinson
Added some text to the first page of reference, and a build batch file

Revision 1.1.1.1  2000/06/05 16:39:04  andy_robinson
initial import
```