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.

(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:

Media House
3 Palmerston Road
Wimbledon
London SW19 1PG, UK

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

reportlab.pdfgen.canvas module

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.

    Example::

       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 addPageLabel(self, pageNum, style=None, start=None, prefix=None):

```
    add a PDFPageLabel for pageNum
```

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, **kw):

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

def bookmarkHorizontalAbsolute(self, key, top, left=0, fit='XYZ', **kw):

    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,
                 fit="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 delViewerPreference(self,pref):

    you'll get an error here if none have been set
```

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,mode=None):

Draws a string centred on the x coordinate.

We're British, dammit, and proud of our spelling!

def drawImage(self, image, x, y, width=None, height=None, mask=None, preserveAspectRatio=False, anchor='c'):

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

"image" may be an image filename or an ImageReader object.

x and y define the lower left corner of the image you wish to draw (or of its bounding box, if using preserveAspectRation below).

If width and height are not given, the width and height of the image in pixels is used at a scale of 1 point to 1 pixel.

If width and height are given, the image will be stretched to fill the given rectangle bounded by (x, y, x+width, y-height).

If you supply negative widths and/or heights, it inverts them and adjusts x and y accordingly.

The method returns the width and height of the underlying image, since this is often useful for layout algorithms and saves you work if you have not specified them yourself.

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

New post version 2.0: drawImage can center an image in a box you provide, while preserving its aspect ratio. For example, you might have a fixed square box in your design, and a collection of photos which might be landscape or portrait that you want to appear within the box. If preserveAspectRatio is true, your image will appear within the box specified.

```
            If preserveAspectRatio is True, the anchor property can be used to
            specify how images should fit into the given box.  It should
            be set to one of the following values, taken from the points of
            the compass (plus 'c' for 'centre'):

                    nw   n   ne
                    w    c    e
                    sw   s   se

            The default value is 'c' for 'centre'.  Thus, if you want your
            bitmaps to always be centred and appear at the top of the given box,
            set anchor='n'.      There are good examples of this in the output
            of test_pdfgen_general.py

            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, resulting in a saving
            in file size and generation time.  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, preserveAspectRatio=False,anchor='c'):

```
    See drawImage, which should normally be used instead...

    drawInlineImage behaves like drawImage, but stores the image content
    within the graphics stream for the page.  This means that the mask
    parameter for transparency is not available.  It also means that there
    is no saving in file size or time if the same image is reused.

    In theory it allows images to be displayed slightly faster; however,
    we doubt if the difference is noticeable to any human user these days.
    Only use this if you have studied the PDF specification and know the
    implications.
```

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

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

def drawRightString(self, x, y, text, mode=None):

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

def drawString(self, x, y, text, mode=None):

```
    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 freeTextAnnotation(self, contents, DA, Rect=None, addtopage=1, name=None, relative=0, **kw):

```
    DA is the default appearance string???
```

def getAvailableFonts(self):

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

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

def getCurrentPageContent(self):

```
    Return uncompressed contents of current page buffer.
```

```
            This is useful in creating test cases and assertions of what
            got drawn, without necessarily saving pages to disk
```

def getPageNumber(self):

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

def getViewerPreference(self,pref):

```
    you'll get an error here if none have been set
```

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 inkAnnotation(self, contents, InkList=None, Rect=None, addtopage=1, name=None, relative=0, **kw):

```
    (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,
        thickness=0, color=None, dashArray=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, relative=1,
        thickness=0, color=None, dashArray=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", **kw):

```
    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, but is visible
    in document properties etc etc.
```

def setCreator(self, creator):

```
    write a creator into the PDF file that won't automatically display
    in the document itself. This should be used to name the original app
    which is passing data into ReportLab, if you wish to name it.
```

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

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

def setDateFormatter(self, dateFormatter):

accepts a func(yyyy,mm,dd,hh,m,s) used to create embedded formatted date

def setEncrypt(self, encrypt):

Set the encryption used for the pdf generated by this canvas.
If encrypt is a string object, it is used as the user password for the pdf.
If encrypt is an instance of reportlab.lib.pdfencrypt.StandardEncryption, this object is used to encrypt the pdf. This allows more finegrained control over the encryption settings.

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 setKeywords(self, keywords):

write a list of keywords into the PDF file which shows in document properties.
Either submit a single string or a list/tuple

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 setViewerPreference(self,pref,value):

```
    set one of the allowed enbtries in the documents viewer preferences
```

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=None, fontSize=None):

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

def textAnnotation(self, contents, Rect=None, addtopage=1, name=None, relative=0, **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!
```

reportlab.pdfgen.pathobject module

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
```

reportlab.pdfgen.textobject module

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.pdfgen.pdfgeom module

This module includes any mathematical methods needed for PIDDLE. It should have no dependencies beyond the Python library. So far, just Robert Kern's bezierArc.

def bezierArc(x1,y1, x2,y2, startAng=0, extent=90):

bezierArc(x1,y1, x2,y2, startAng=0, extent=90) --> List of Bezier curve control points.

(x1, y1) and (x2, y2) are the corners of the enclosing rectangle. The coordinate system has coordinates that increase to the right and down. Angles, measured in degress, start with 0 to the right (the positive X axis) and increase counter-clockwise. The arc extends from startAng to startAng+extent. I.e. startAng=0 and extent=180 yields an openside-down semi-circle.

The resulting coordinates are of the form (x1,y1, x2,y2, x3,y3, x4,y4) such that the curve goes from (x1, y1) to (x4, y4) with (x2, y2) and (x3, y3) as their respective Bezier control points.

reportlab.pdfgen.pdfimages module

Image functionality sliced out of canvas.py for generalization

Class PDFImage:

Wrapper around different "image sources". You can make images from a PIL Image object, a filename (in which case it uses PIL), an image we previously cached (optimisation, hardly used these days) or a JPEG (which PDF supports natively).

def PIL_imagedata(self):

(no documentation string)

def cache_imagedata(self):

(no documentation string)

def drawInlineImage(self, canvas, preserveAspectRatio=False,anchor='sw'):

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. This allows a caching mechanism

def format(self, document):

Allow it to be used within pdfdoc framework. This only defines how it is stored, not how it is drawn later.

def getImageData(self,preserveAspectRatio=False):

Gets data, height, width - whatever type of image

def jpg_imagedata(self):

(no documentation string)

def non_jpg_imagedata(self,image):

(no documentation string)

reportlab.pdfgen.pycanvas module

A contributed Canvas class which can also output Python source code to "replay" operations later pycanvas.Canvas class works exactly like canvas.Canvas, but you can call str() on pycanvas.Canvas instances. Doing so will return the Python source code equivalent to your own program, which would, when run, produce the same PDF document as your original program. Generated Python source code defines a doIt() function which accepts a filename or file-like object as its first parameter, and an optional boolean parameter named "regenerate". The doIt() function will generate a PDF document and save it in the file you specified in this argument. If the regenerate parameter is set then it will also return an automatically generated equivalent Python source code as a string of text, which you can run again to produce the very same PDF document and the Python source code, which you can run again... ad nauseam ! If the regenerate parameter is unset or not used at all (it then defaults to being unset) then None is returned and the doIt() function is much much faster, it is also much faster than the original non-serialized program. the tests/test_pdfgen_pycanvas.py program is the test suite for pycanvas, you can do the following to run it : First set verbose=1 in

reportlab/rl_config.py then from the command interpreter : $ cd tests $ python test_pdfgen_pycanvas.py >n1.py this will produce both n1.py and test_pdfgen_pycanvas.pdf then : $ python n1.py n1.pdf >n2.py $ python n2.py n2.pdf >n3.py $ ... n1.py, n2.py, n3.py and so on will be identical files. they eventually may end being a bit different because of rounding problems, mostly in the comments, but this doesn't matter since the values really are the same (e.g. 0 instead of 0.0, or .53 instead of 0.53) n1.pdf, n2.pdf, n3.pdf and so on will be PDF files similar to test_pdfgen_pycanvas.pdf. Alternatively you can import n1.py (or n3.py, or n16384.py if you prefer) in your own program, and then call its doIt function : import n1 pythonsource = n1.doIt("myfile.pdf", regenerate=1) Or if you don't need the python source code and want a faster result : import n1 n1.doIt("myfile.pdf") When the generated source code is run directly as an independant program, then the equivalent python source code is printed to stdout, e.g. : python n1.py will print the python source code equivalent to n1.py Why would you want to use such a beast ? - To linearize (serialize?) a program : optimizing some complex parts for example. - To debug : reading the generated Python source code may help you or the ReportLab team to diagnose problems. The generated code is now clearly commented and shows nesting levels, page numbers, and so on. You can use the generated script when asking for support : we can see the results you obtain without needing your datas or complete application. - To create standalone scripts : say your program uses a high level environment to generate its output (databases, RML, etc...), using this class would give you an equivalent program but with complete independance from the high level environment (e.g. if you don't have Oracle). - To contribute some nice looking PDF documents to the ReportLab website without having to send a complete application you don't want to distribute. - ... Insert your own ideas here ... - For fun because you can do it !

def buildargs(*args, **kwargs) :

Constructs a printable list of arguments suitable for use in source function calls.

Class Canvas:

Our fake Canvas class, which will intercept each and every method or attribute access.

Class PDFAction:

Base class to fake method calls or attributes on PDF objects (Canvas, PDFPathObject, PDFTextObject).

Class PDFObject:

Base class for PDF objects like PDFPathObject and PDFTextObject.

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.

    1)  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.

    2)  The document has one or more page templates.

    3)  Each page template has one or more frames.

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

    5)  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 docAssign(self,var,expr,lifetime):

(no documentation string)

def docEval(self,expr):

(no documentation string)

def docExec(self,stmt,lifetime):

(no documentation string)

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,
               maxPasses = 10,
               **buildKwds
               ):
```

Makes multiple passes until all indexing flowables
are happy.

Returns number of passes

```
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"/>
    <index [name="callablecanvasattribute"] 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
    <a>anchor text</a>
    attributes of anchors
    fontSize=num
    fontName=name
    fg/textColor/color=color
    backcolor/backColor/bgcolor=color
    href=href
    <a name="anchorpoint"/>
    <unichar name="unicode character name"/>
    <unichar value="unicode code point"/>
    <img src="path" width="1in" height="1in" valign="bottom"/>
            width="w%" --> fontSize*w/100   idea from Roberto Alsina
            height="h%" --> linewidth*h/100 <ralsina@netmanagers.com.ar>

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

    The <b> and <i> tags will work for the built-in fonts (Helvetica
    /Times / Courier).  For other fonts you need to register a family
    of 4 fonts using reportlab.pdfbase.pdfmetrics.registerFont; then
    use the addMapping function to tell the library that these 4 fonts
    form a family e.g.
    from reportlab.lib.fonts import addMapping
    addMapping('Vera', 0, 0, 'Vera')    #normal
    addMapping('Vera', 0, 1, 'Vera-Italic')    #italic
    addMapping('Vera', 1, 0, 'Vera-Bold')    #bold
    addMapping('Vera', 1, 1, 'Vera-BoldItalic')    #italic and bold

    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 breakLinesCJK(self, width):

Initially, the dumbest possible wrapping algorithm. Cannot handle font variations.

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.
        If they decide that nothing useful can be fitted in the
        available space (e.g. if you have a table and not enough
        space for the first row), also return []
```

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 minWidth(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 NullDraw:

```
    (no documentation string)
```

def draw(self):

(no documentation string)

Class Spacer:

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

Class UseUpSpace:

(no documentation string)

def wrap(self, availWidth, availHeight):

(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 SlowPageBreak:

(no documentation string)

Class CondPageBreak:

use up a frame if not enough vertical space effectively CondFrameBreak

def identity(self,maxLen=None):

(no documentation string)

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 CallerMacro:

like Macro, but with callable command(s)
drawCallable(self)
wrapCallable(self,aW,aH)

def draw(self):

(no documentation string)

def wrap(self, aW, aH):

(no documentation string)

Class ParagraphAndImage:

combine a Paragraph and an Image

def draw(self):

(no documentation string)

def getSpaceAfter(self):

(no documentation string)

def getSpaceBefore(self):

(no documentation string)

def split(self,availWidth, availHeight):

(no documentation string)

def wrap(self,availWidth,availHeight):

(no documentation string)

Class KeepInFrame:

(no documentation string)

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

(no documentation string)

def identity(self, maxLen=None):

(no documentation string)

def wrap(self,availWidth,availHeight):

(no documentation string)

Class ImageAndFlowables:

combine a list of flowables and an Image

def deepcopy(self):

(no documentation string)

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

(no documentation string)

def getSpaceAfter(self):

(no documentation string)

def getSpaceBefore(self):

(no documentation string)

def split(self,availWidth, availHeight):

(no documentation string)

def wrap(self,availWidth,availHeight):

(no documentation string)

Class AnchorFlowable:

create a bookmark in the pdf

def draw(self):

(no documentation string)

def wrap(self,aW,aH):

(no documentation string)

Class FrameSplitter:

When encountered this flowable should either switch directly to nextTemplate
if remaining space in the current frame is less than gap+required or it should
temporarily modify the current template to have the frames from nextTemplate
that are listed in nextFrames and switch to the first of those frames.

def wrap(self,aW,aH):

(no documentation string)

Class TableOfContents:

This creates a formatted table of contents.

It presumes a correct block of data is passed in.
The data block contains a list of (level, text, pageNumber)
triplets. You can supply a paragraph style for each level
(starting at zero).
Set dotsMinLevel to determine from which level on a line of
dots should be drawn between the text and the page number.
If dotsMinLevel is set to a negative value, no dotted lines are drawn.

def addEntries(self, listOfEntries):

Bulk creation of entries in the table of contents.

If you knew the titles but not the page numbers, you could

supply them to get sensible output on the first run.

def addEntry(self, level, text, pageNum, key=None):

Adds one entry to the table of contents.

This allows incremental buildup by a doctemplate.
Requires that enough styles are defined.

def beforeBuild(self):

(no documentation string)

def clearEntries(self):

(no documentation string)

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

Don't do this at home! The standard calls for implementing
draw(); we are hooking this in order to delegate ALL the drawing
work to the embedded table object.

def getLevelStyle(self, n):

Returns the style for level n, generating and caching styles on demand if not present.

def isIndexing(self):

(no documentation string)

def isSatisfied(self):

(no documentation string)

def notify(self, kind, stuff):

The notification hook called to register all kinds of events.

Here we are interested in 'TOCEntry' events only.

def split(self, availWidth, availHeight):

At this stage we do not care about splitting the entries,
we will just return a list of platypus tables. Presumably the
calling app has a pointer to the original TableOfContents object;
Platypus just sees tables.

def wrap(self, availWidth, availHeight):

All table properties should be known by now.

Class SimpleIndex:

Creates multi level indexes.
The styling can be cutomized and alphabetic headers turned on and off.

def addEntry(self, text, pageNum, key=None):

Allows incremental buildup

def beforeBuild(self):

(no documentation string)

def clearEntries(self):

(no documentation string)

def draw(self):

(no documentation string)

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

Don't do this at home! The standard calls for implementing
draw(); we are hooking this in order to delegate ALL the drawing
work to the embedded table object.

def getCanvasMaker(self, canvasmaker=canvas.Canvas):

(no documentation string)

def getFormatFunc(self,format):

(no documentation string)

def getLevelStyle(self, n):

Returns the style for level n, generating and caching styles on demand if not present.

def isIndexing(self):

(no documentation string)

def isSatisfied(self):

(no documentation string)

def notify(self, kind, stuff):

The notification hook called to register all kinds of events.

Here we are interested in 'IndexEntry' events only.

def setup(self, style=None, dot=None, tableStyle=None, headers=True, name=None, format='123', offset=0):

This method makes it possible to change styling and other parameters on an existing object.

style is the paragraph style to use for index entries.
dot can either be None or a string. If it's None, entries are immediatly followed by their
corresponding page numbers. If it's a string, page numbers are aligned on the right side
of the document and the gap filled with a repeating sequence of the string.
tableStyle is the style used by the table which the index uses to draw itself. Use this to
change properties like spacing between elements.
headers is a boolean. If it is True, alphabetic headers are displayed in the Index when the first
letter changes. If False, we just output some extra space before the next item
name makes it possible to use several indexes in one document. If you want this use this
parameter to give each index a unique name. You can then index a term by refering to the
name of the index which it should appear in:

<index item="term" name="myindex" />

format can be 'I', 'i', '123', 'ABC', 'abc'

def split(self, availWidth, availHeight):

At this stage we do not care about splitting the entries,
we will just return a list of platypus tables. Presumably the
calling app has a pointer to the original TableOfContents object;
Platypus just sees tables.

def wrap(self, availWidth, availHeight):

All table properties should be known by now.

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

```
    Defines standard colour-handling classes and colour names. We define standard classes
    to hold colours in two models: RGB and CMYK. These can be constructed from several
    popular formats. We also include - pre-built colour objects for the HTML standard
    colours - pre-built colours used in ReportLab's branding - various conversion and
    construction functions
```

def Blacker(c,f):

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

def HexColor(val, htmlOnly=False, alpha=False):

```
    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).

    An alpha value can also be given in the form #AABBCCDD or 0xAABBCCDD where
    DD is the alpha value.

    For completeness I assume that #aabbcc or 0xaabbcc are hex numbers
    otherwise a pure integer is converted as decimal rgb.  If htmlOnly is true,
    only the #aabbcc form is allowed.

    >>> HexColor('#ffffff')
    Color(1,1,1,1)
    >>> HexColor('#FFFFFF')
    Color(1,1,1,1)
    >>> HexColor('0xffffff')
    Color(1,1,1,1)
    >>> HexColor('16777215')
    Color(1,1,1,1)

    An '0x' or '#' prefix is required for hex (as opposed to decimal):

    >>> HexColor('ffffff')
    Traceback (most recent call last):
    ValueError: invalid literal for int() with base 10: 'ffffff'

    >>> HexColor('#FFFFFF', htmlOnly=True)
    Color(1,1,1,1)
    >>> HexColor('0xffffff', htmlOnly=True)
    Traceback (most recent call last):
    ValueError: not a hex string
    >>> HexColor('16777215', htmlOnly=True)
    Traceback (most recent call last):
    ValueError: not a hex string
```

def Whiter(c,f):

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

def cmyk2rgb(cmyk,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 fade(aSpotColor, percentages):

Waters down spot colors and returns a list of new ones

e.g fade(myColor, [100,80,60,40,20]) returns a list of five colors

def getAllNamedColors():

(no documentation string)

def hsl2rgb(h, s, l):

(no documentation string)

def hue2rgb(m1, m2, h):

(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 obj_R_G_B(c):

attempt to convert an object to (red,green,blue)

def parseColorClassFromString(arg):

(no documentation string)

def rgb2cmyk(r,g,b):

one way to get cmyk from rgb

def setColors(kw):**

(no documentation string)

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

def cmyka(self):

Returns a tuple of five color components - syntactic sugar

def fader(self, n, reverse=False):

return n colors based on density fade
NB note this dosen't reach density zero

Class CMYKColorSep:

special case color for making separating pdfs

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 bitmap_rgba(self):

(no documentation string)

def clone(self,kwds):**

copy then change values in kwds

def hexval(self):

(no documentation string)

def hexvala(self):

(no documentation string)

def rgb(self):

Returns a three-tuple of components

def rgba(self):

Returns a four-tuple of components

Class PCMYKColor:

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

Class PCMYKColorSep:

special case color for making separating pdfs

reportlab.lib.corp module

Generate ReportLab logo in a variety of sizes and formats. 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

Container for constants. Hardly used!

reportlab.lib.fonts module

Utilities to associate bold and italic versions of fonts into families Bold, italic and plain fonts are usually implemented in separate disk files; but non-trivial apps want **this** to do the right thing. We therefore need to keep 'mappings' between the font family name and the right group of up to 4 implementation fonts to use. Most font-handling code lives in pdfbase, and this probably should too.

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

A Sequencer class counts things. It aids numbering and formatting lists.

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)

reportlab.lib.abag module

Data structure to hold a collection of attributes, used by styles.

Class ABag:

'Attribute Bag' - a trivial BAG class for holding attributes.

This predates modern Python. Doing this again, we'd use a subclass
of dict.

You may initialize with keyword arguments.
a = ABag(k0=v0,....,kx=vx,....) ==> getattr(a,'kx')==vx

c = a.clone(ak0=av0,.....) copy with optional additional attributes.

def clone(self,attr):**

(no documentation string)

reportlab.lib.attrmap module

Framework for objects whose assignments are checked. Used by graphics. We developed
reportlab/graphics prior to Python 2 and metaclasses. For the graphics, we wanted to be
able to declare the attributes of a class, check them on assignment, and convert from
string arguments. Examples of attrmap-based objects can be found in
reportlab/graphics/shapes. It lets us defined structures like the one below, which are
seen more modern form in Django models and other frameworks. We'll probably replace
this one day soon, hopefully with no impact on client code. class Rect(SolidShape):
"""Rectangle, possibly with rounded corners.""" _attrMap = AttrMap(BASE=SolidShape, x =

```
    AttrMapValue(isNumber), y = AttrMapValue(isNumber), width = AttrMapValue(isNumber),
    height = AttrMapValue(isNumber), rx = AttrMapValue(isNumber), ry =
    AttrMapValue(isNumber), )
```

def addProxyAttribute(src,name,validate=None,desc=None,initial=None,dst=None):

```
    Add a proxy attribute 'name' to src with targets dst
```

def hook__setattr__(obj):

```
    (no documentation string)
```

def validateSetattr(obj,name,value):

```
    validate setattr(obj,name,value)
```

Class AttrMap:

```
    (no documentation string)
```

def clone(self,UNWANTED=[],kw):**

```
        (no documentation string)
```

def remove(self,unwanted):

```
        (no documentation string)
```

def update(self,kw):

```
        (no documentation string)
```

Class AttrMapValue:

```
    Simple multi-value holder for attribute maps
```

Class CallableValue:

```
    a class to allow callable initial values
```

reportlab.lib.boxstuff module

```
    Utility functions to position and resize boxes within boxes
```

def aspectRatioFix(preserve,anchor,x,y,width,height,imWidth,imHeight):

```
    This function helps position an image within a box.

        It first normalizes for two cases:
        - if the width is None, it assumes imWidth
        - ditto for height
        - if width or height is negative, it adjusts x or y and makes them positive

        Given
        (a) the enclosing box (defined by x,y,width,height where x,y is the         lower left corner) which
        (b) the image size (imWidth, imHeight), and
        (c) the 'anchor point' as a point of the compass - n,s,e,w,ne,se etc         and c for centre,

        this should return the position at which the image should be drawn,
        as well as a scale factor indicating what scaling has happened.

        It returns the parameters which would be used to draw the image
        without any adjustments:

            x,y, width, height, scale

        used in canvas.drawImage and drawInlineImage
```

reportlab.lib.codecharts module

```
    Routines to print code page (character set) drawings. Predates unicode. To be sure we
    can accurately represent characters in various encodings and fonts, we need some
    routines to display all those characters. These are defined herein. The idea is to
    include flowable, drawable and graphic objects for single and multi-byte fonts.
```

def hBoxText(msg, canvas, x, y, fontName):

```
    Helper for stringwidth tests on Asian fonts.

        Registers font if needed.  Then draws the string,
        and a box around it derived from the stringWidth function
```

def test():

(no documentation string)

Class Big5CodeChart:

Formats one 'row' of the 94x160 space used in Big 5

These deliberately resemble the code charts in Ken Lunde's "Understanding
CJKV Information Processing", to enable manual checking.

def draw(self):

(no documentation string)

def makeRow(self, row):

Works out the character values for this Big5 row.
Rows start at 0xA1

Class CodeChartBase:

Basic bits of drawing furniture used by
single and multi-byte versions: ability to put letters
into boxes.

def calcLayout(self):

Work out x and y positions for drawing

def drawChars(self, charList):

Fills boxes in order. None means skip a box.
Empty boxes at end get filled with gray

def drawLabels(self, topLeft = ''):

Writes little labels in the top row and first column

def formatByte(self, byt):

(no documentation string)

Class CodeWidget:

Block showing all the characters

def draw(self):

(no documentation string)

Class KutenRowCodeChart:

Formats one 'row' of the 94x94 space used in many Asian encodings.aliases

These deliberately resemble the code charts in Ken Lunde's "Understanding
CJKV Information Processing", to enable manual checking. Due to the large
numbers of characters, we don't try to make one graphic with 10,000 characters,
but rather output a sequence of these.

def draw(self):

(no documentation string)

def makeRow(self, row):

Works out the character values for this kuten row

Class SingleByteEncodingChart:

(no documentation string)

def draw(self):

(no documentation string)

reportlab.lib.extformat module

Apparently not used anywhere, purpose unknown!

def dictformat(_format, L={}, G={}):

(no documentation string)

def magicformat(format):

Evaluate and substitute the appropriate parts of the string.

reportlab.lib.fontfinder module

This provides some general-purpose tools for finding fonts. The FontFinder object can search for font files. It aims to build a catalogue of fonts which our framework can work with. It may be useful if you are building GUIs or design-time interfaces and want to present users with a choice of fonts. There are 3 steps to using it 1. create FontFinder and set options and directories 2. search 3. query >>> import fontfinder >>> ff = fontfinder.FontFinder() >>> ff.addDirectories([dir1, dir2, dir3]) >>> ff.search() >>> ff.getFamilyNames() #or whichever queries you want... Because the disk search takes some time to find and parse hundreds of fonts, it can use a cache to store a file with all fonts found. The cache file name For each font found, it creates a structure with - the short font name - the long font name - the principal file (.pfb for type 1 fonts), and the metrics file if appropriate - the time modified (unix time stamp) - a type code ('ttf') - the family name - bold and italic attributes One common use is to display families in a dialog for end users; then select regular, bold and italic variants of the font. To get the initial list, use getFamilyNames; these will be in alpha order. >>> ff.getFamilyNames() ['Bitstream Vera Sans', 'Century Schoolbook L', 'Dingbats', 'LettErrorRobot', 'MS Gothic', 'MS Mincho', 'Nimbus Mono L', 'Nimbus Roman No9 L', 'Nimbus Sans L', 'Vera', 'Standard Symbols L', 'URW Bookman L', 'URW Chancery L', 'URW Gothic L', 'URW Palladio L'] One can then obtain a specific font as follows >>> f = ff.getFont('Bitstream Vera Sans', bold=False, italic=True) >>> f.fullName 'Bitstream Vera Sans' >>> f.fileName 'C:\code\reportlab\fonts\Vera.ttf' >>> It can also produce an XML report of fonts found by family, for the benefit of non-Python applications. Future plans might include using this to auto-register fonts; and making it update itself smartly on repeated instantiation.

def test():

(no documentation string)

Class FontDescriptor:

This is a short descriptive record about a font.

typeCode should be a file extension e.g. ['ttf','ttc','otf','pfb','pfa']

def getTag(self):

Return an XML tag representation

Class FontFinder:

(no documentation string)

def addDirectories(self, dirNames):

(no documentation string)

def addDirectory(self, dirName):

(no documentation string)

def getFamilyNames(self):

Returns a list of the distinct font families found

def getFamilyXmlReport(self):

Reports on all families found as XML.

def getFont(self, familyName, bold=False, italic=False):

Try to find a font matching the spec

def getFontsInFamily(self, familyName):

Return list of all font objects with this family name

def getFontsWithAttributes(self, **kwds):

This is a general lightweight search.

def load(self, fileName):

(no documentation string)

def save(self, fileName):

(no documentation string)

def search(self):

(no documentation string)

reportlab.lib.formatters module

These help format numbers and dates in a user friendly way. Used by the graphics framework.

Class DecimalFormatter:

lets you specify how to build a decimal.

A future NumberFormatter class will take Microsoft-style patterns instead - "$#,##0.00" is WAY easier than this.

def format(self, num):

(no documentation string)

Class Formatter:

Base formatter - simply applies python format strings

def format(self, obj):

(no documentation string)

reportlab.lib.geomutils module

Utility functions for geometrical operations.

def normalizeTRBL(p):

Useful for interpreting short descriptions of paddings, borders, margin, etc.
Expects a single value or a tuple of length 2 to 4.
Returns a tuple representing (clockwise) the value(s) applied to the 4 sides of a rectangle:
If a single value is given, that value is applied to all four sides.
If two or three values are given, the missing values are taken from the opposite side(s).
If four values are given they are returned unchanged.

```
>>> normalizeTRBL(1)
(1, 1, 1, 1)
>>> normalizeTRBL((1, 1.2))
(1, 1.2, 1, 1.2)
>>> normalizeTRBL((1, 1.2, 0))
(1, 1.2, 0, 1.2)
>>> normalizeTRBL((1, 1.2, 0, 8))
(1, 1.2, 0, 8)
```

reportlab.lib.logger module

Logging and warning framework, predating Python's logging package

Class Logger:

An extended file type thing initially equivalent to sys.stderr
You can add/remove file type things; it has a write method

def add(self,fp):

add the file/string fp to the destinations

def remove(self,fp):

remove the file/string fp from the destinations

def write(self,text):

write text to all the destinations

Class WarnOnce:

(no documentation string)

def once(self,warning):

(no documentation string)

reportlab.lib.normalDate module

Jeff Bauer's lightweight date class, extended by us. Predates Python's datetime module.

def FND(d):

convert to ND if required

```
def bigBang():
    return lower boundary as a NormalDate
def bigCrunch():
    return upper boundary as a NormalDate
def dayOfWeek(y, m, d):
    return integer representing day of week, Mon=0, Tue=1, etc.
def firstDayOfYear(year):
    number of days to the first of the year, relative to Jan 1, 1900
def getStdDayNames():
    (no documentation string)
def getStdMonthNames():
    (no documentation string)
def getStdShortDayNames():
    (no documentation string)
def getStdShortMonthNames():
    (no documentation string)
def isLeapYear(year):
    determine if specified year is leap year, returns Python boolean
Class BusinessDate:
    Specialised NormalDate
    def add(self, days):
        add days to date; use negative integers to subtract
    def asNormalDate(self):
        (no documentation string)
    def daysBetweenDates(self, normalDate):
        (no documentation string)
    def normalize(self, i):
        (no documentation string)
    def scalar(self):
        (no documentation string)
    def setNormalDate(self, normalDate):
        (no documentation string)
Class ND:
    NormalDate is a specialized class to handle dates without
    all the excess baggage (time zones, daylight savings, leap
    seconds, etc.) of other date structures.  The minimalist
    strategy greatly simplifies its implementation and use.

    Internally, NormalDate is stored as an integer with values
    in a discontinuous range of -99990101 to 99991231.  The
    integer value is used principally for storage and to simplify
    the user interface.  Internal calculations are performed by
    a scalar based on Jan 1, 1900.

    Valid NormalDate ranges include (-9999,1,1) B.C.E. through
    (9999,12,31) C.E./A.D.


    1.0
        No changes, except the version number.  After 3 years of use by
        various parties I think we can consider it stable.

    0.8
        Added Prof. Stephen Walton's suggestion for a range method
         - module author resisted the temptation to use lambda <0.5 wink>
```

```
0.7
    Added Dan Winkler's suggestions for __add__, __sub__ methods

0.6
    Modifications suggested by Kevin Digweed to fix:
     - dayOfWeek, dayOfWeekAbbrev, clone methods
     - Permit NormalDate to be a better behaved superclass

0.5
    Minor tweaking

0.4
     - Added methods __cmp__, __hash__
     - Added Epoch variable, scoped to the module
     - Added setDay, setMonth, setYear methods

0.3
    Minor touch-ups

0.2
     - Fixed bug for certain B.C.E leap years
     - Added Jim Fulton's suggestions for short alias class name =ND
       and __getstate__, __setstate__ methods

Special thanks:  Roedy Green
```

def add(self, days):

```
    add days to date; use negative integers to subtract
```

def clone(self):

```
    return a cloned instance of this normalDate
```

def day(self):

```
    return the day as integer 1-31
```

def dayOfWeek(self):

```
    return integer representing day of week, Mon=0, Tue=1, etc.
```

def dayOfWeekAbbrev(self):

```
    return day of week abbreviation for current date: Mon, Tue, etc.
```

def dayOfWeekName(self):

```
    return day of week name for current date: Monday, Tuesday, etc.
```

def dayOfYear(self):

```
    day of year
```

def daysBetweenDates(self, normalDate):

```
    return value may be negative, since calculation is
    self.scalar() - arg
```

def endOfMonth(self):

```
    returns (cloned) last day of month
```

def equals(self, target):

```
    (no documentation string)
```

def firstDayOfMonth(self):

```
    returns (cloned) first day of month
```

def formatMS(self,fmt):

```
    format like MS date using the notation
    {YY}    --> 2 digit year
    {YYYY}  --> 4 digit year
    {M}     --> month as digit
    {MM}    --> 2 digit month
    {MMM}   --> abbreviated month name
    {MMMM}  --> monthname
    {MMMMM} --> first character of monthname
    {D}     --> day of month as digit
    {DD}    --> 2 digit day of month
    {DDD}   --> abrreviated weekday name
    {DDDD}  --> weekday name
```

def formatUS(self):

    return date as string in common US format: MM/DD/YY

def formatUSCentury(self):

    return date as string in 4-digit year US format: MM/DD/YYYY

def isLeapYear(self):

    determine if specified year is leap year, returning true (1) or
    false (0)

def lastDayOfMonth(self):

    returns last day of the month as integer 28-31

def localeFormat(self):

    override this method to use your preferred locale format

def month(self):

    returns month as integer 1-12

def monthAbbrev(self):

    returns month as a 3-character abbreviation, i.e. Jan, Feb, etc.

def monthName(self):

    returns month name, i.e. January, February, etc.

def normalize(self, scalar):

    convert scalar to normalDate

def range(self, days):

    Return a range of normalDates as a list. Parameter
    may be an int or normalDate.

def scalar(self):

    days since baseline date: Jan 1, 1900

def setDay(self, day):

    set the day of the month

def setMonth(self, month):

    set the month [1-12]

def setNormalDate(self, normalDate):

    accepts date as scalar string/integer (yyyymmdd) or tuple
    (year, month, day, ...)

def setYear(self, year):

    (no documentation string)

def toTuple(self):

    return date as (year, month, day) tuple

def year(self):

    return year in yyyy format, negative values indicate B.C.

Class NormalDate:

NormalDate is a specialized class to handle dates without
all the excess baggage (time zones, daylight savings, leap
seconds, etc.) of other date structures. The minimalist
strategy greatly simplifies its implementation and use.

Internally, NormalDate is stored as an integer with values
in a discontinuous range of -99990101 to 99991231. The
integer value is used principally for storage and to simplify
the user interface. Internal calculations are performed by
a scalar based on Jan 1, 1900.

Valid NormalDate ranges include (-9999,1,1) B.C.E. through
(9999,12,31) C.E./A.D.

1.0

```
    No changes, except the version number.  After 3 years of use by
    various parties I think we can consider it stable.

0.8
    Added Prof. Stephen Walton's suggestion for a range method
     - module author resisted the temptation to use lambda <0.5 wink>

0.7
    Added Dan Winkler's suggestions for __add__, __sub__ methods

0.6
    Modifications suggested by Kevin Digweed to fix:
     - dayOfWeek, dayOfWeekAbbrev, clone methods
     - Permit NormalDate to be a better behaved superclass

0.5
    Minor tweaking

0.4
     - Added methods __cmp__, __hash__
     - Added Epoch variable, scoped to the module
     - Added setDay, setMonth, setYear methods

0.3
    Minor touch-ups

0.2
     - Fixed bug for certain B.C.E leap years
     - Added Jim Fulton's suggestions for short alias class name =ND
       and __getstate__, __setstate__ methods

Special thanks:  Roedy Green
```

def add(self, days):

```
    add days to date; use negative integers to subtract
```

def clone(self):

```
    return a cloned instance of this normalDate
```

def day(self):

```
    return the day as integer 1-31
```

def dayOfWeek(self):

```
    return integer representing day of week, Mon=0, Tue=1, etc.
```

def dayOfWeekAbbrev(self):

```
    return day of week abbreviation for current date: Mon, Tue, etc.
```

def dayOfWeekName(self):

```
    return day of week name for current date: Monday, Tuesday, etc.
```

def dayOfYear(self):

```
    day of year
```

def daysBetweenDates(self, normalDate):

```
    return value may be negative, since calculation is
    self.scalar() - arg
```

def endOfMonth(self):

```
    returns (cloned) last day of month
```

def equals(self, target):

```
    (no documentation string)
```

def firstDayOfMonth(self):

```
    returns (cloned) first day of month
```

def formatMS(self,fmt):

```
    format like MS date using the notation
    {YY}    --> 2 digit year
    {YYYY}  --> 4 digit year
    {M}     --> month as digit
    {MM}    --> 2 digit month
```

```
        {MMM}   --> abbreviated month name
        {MMMM}  --> monthname
        {MMMMM} --> first character of monthname
        {D}     --> day of month as digit
        {DD}    --> 2 digit day of month
        {DDD}   --> abrreviated weekday name
        {DDDD}  --> weekday name
```

def formatUS(self):

```
    return date as string in common US format: MM/DD/YY
```

def formatUSCentury(self):

```
    return date as string in 4-digit year US format: MM/DD/YYYY
```

def isLeapYear(self):

```
    determine if specified year is leap year, returning true (1) or
    false (0)
```

def lastDayOfMonth(self):

```
    returns last day of the month as integer 28-31
```

def localeFormat(self):

```
    override this method to use your preferred locale format
```

def month(self):

```
    returns month as integer 1-12
```

def monthAbbrev(self):

```
    returns month as a 3-character abbreviation, i.e. Jan, Feb, etc.
```

def monthName(self):

```
    returns month name, i.e. January, February, etc.
```

def normalize(self, scalar):

```
    convert scalar to normalDate
```

def range(self, days):

```
    Return a range of normalDates as a list.  Parameter
    may be an int or normalDate.
```

def scalar(self):

```
    days since baseline date: Jan 1, 1900
```

def setDay(self, day):

```
    set the day of the month
```

def setMonth(self, month):

```
    set the month [1-12]
```

def setNormalDate(self, normalDate):

```
    accepts date as scalar string/integer (yyyymmdd) or tuple
    (year, month, day, ...)
```

def setYear(self, year):

```
    (no documentation string)
```

def toTuple(self):

```
    return date as (year, month, day) tuple
```

def year(self):

```
    return year in yyyy format, negative values indicate B.C.
```

reportlab.lib.pdfencrypt module

def checkU(encryptionkey, U):

```
(no documentation string)
```

def computeO(userPassword, ownerPassword, revision):

```
(no documentation string)
```

def computeU(encryptionkey, encodestring=PadString,revision=2,documentId=None):

(no documentation string)

def encodePDF(key, objectNumber, generationNumber, string, revision=2):

Encodes a string or stream

def encryptCanvas(canvas,
userPassword, ownerPassword=None,
canPrint=1, canModify=1, canCopy=1, canAnnotate=1,
strength=40):

Applies encryption to the document being generated

def encryptDocTemplate(dt,
userPassword, ownerPassword=None,
canPrint=1, canModify=1, canCopy=1, canAnnotate=1,
strength=40):

For use in Platypus. Call before build().

def encryptPdfInMemory(inputPDF,
userPassword, ownerPassword=None,
canPrint=1, canModify=1, canCopy=1, canAnnotate=1,
strength=40):

accepts a PDF file 'as a byte array in memory'; return encrypted one.

This is a high level convenience and does not touch the hard disk in any way.
If you are encrypting the same file over and over again, it's better to use
pageCatcher and cache the results.

def encryptPdfOnDisk(inputFileName, outputFileName,
userPassword, ownerPassword=None,
canPrint=1, canModify=1, canCopy=1, canAnnotate=1,
strength=40):

Creates encrypted file OUTPUTFILENAME. Returns size in bytes.

def encryptionkey(password, OwnerKey, Permissions, FileId1, revision=2):

(no documentation string)

def hexText(text):

a legitimate way to show strings in PDF

def hexchar(x):

(no documentation string)

def main():

(no documentation string)

def scriptInterp():

(no documentation string)

def test():

(no documentation string)

def unHexText(hexText):

(no documentation string)

def xorKey(num,key):

xor's each bytes of the key with the number, which is <256

Class EncryptionFlowable:

Drop this in your Platypus story and it will set up the encryption options.

If you do it multiple times, the last one before saving will win.

def draw(self):

(no documentation string)

def wrap(self, availWidth, availHeight):

(no documentation string)

Class StandardEncryption:

(no documentation string)

```
    def encode(self, t):
        encode a string, stream, text
    def info(self):
        (no documentation string)
    def permissionBits(self):
        (no documentation string)
    def prepare(self, document, overrideID=None):
        (no documentation string)
    def register(self, objnum, version):
        (no documentation string)
    def setAllPermissions(self, value):
        (no documentation string)
Class StandardEncryptionDictionary:
    (no documentation string)
    def format(self, document):
        (no documentation string)
```

reportlab.lib.PyFontify module

```
    Module to analyze Python source code; for syntax coloring tools. Interface:: tags =
    fontify(pytext, searchfrom, searchto) - The 'pytext' argument is a string containing
    Python source code. - The (optional) arguments 'searchfrom' and 'searchto' may contain
    a slice in pytext. - The returned value is a list of tuples, formatted like this::
    [('keyword', 0, 6, None), ('keyword', 11, 17, None), ('comment', 23, 53, None), etc. ]
    - The tuple contents are always like this:: (tag, startindex, endindex, sublist) - tag
    is one of 'keyword', 'string', 'comment' or 'identifier' - sublist is not used, hence
    always None.
def fontify(pytext, searchfrom = 0, searchto = None):
    (no documentation string)
def replace(src, sep, rep):
    (no documentation string)
def test(path):
    (no documentation string)
```

reportlab.lib.randomtext module

```
    Like Lorem Ipsum, but more fun and extensible. This module exposes a function
    randomText() which generates paragraphs. These can be used when testing out document
    templates and stylesheets. A number of 'themes' are provided - please contribute more!
    We need some real Greek text too. There are currently six themes provided: STARTUP
    (words suitable for a business plan - or not as the case may be), COMPUTERS (names of
    programming languages and operating systems etc), BLAH (variations on the word 'blah'),
    BUZZWORD (buzzword bingo), STARTREK (Star Trek), PRINTING (print-related terms) PYTHON
    (snippets and quotes from Monty Python) CHOMSKY (random lingusitic nonsense) EXAMPLE
    USAGE: from reportlab.lib import randomtext print
    randomtext.randomText(randomtext.PYTHON, 10) This prints a random number of random
    sentences (up to a limit of ten) using the theme 'PYTHON'.
def chomsky(times = 1):
    (no documentation string)
def format_wisdom(text,line_length=72):
    (no documentation string)
def randomText(theme=STARTUP, sentences=5):
    (no documentation string)
```

reportlab.lib.rltempfile module

Helper for the test suite - determines where to write output. When our test suite runs as source, a script "test_foo.py" will typically create "test_foo.pdf" alongside it. But if you are testing a package of compiled code inside a zip archive, this won't work. This determines where to write test suite output, creating a subdirectory of /tmp/ or whatever if needed.

def get_rl_tempdir(*subdirs):

(no documentation string)

def get_rl_tempfile(fn=None):

(no documentation string)

reportlab.lib.rparsexml module

Very simple and fast XML parser, used for intra-paragraph text. Devised by Aaron Watters in the bad old days before Python had fast parsers available. Constructs the lightest possible in-memory representation; parses most files we have seen in pure python very quickly. The output structure is the same as the one produced by pyRXP, our validating C-based parser, which was written later. It will use pyRXP if available. This is used to parse intra-paragraph markup. Example parse:: text **in** xml ("this", {"type": "xml"}, ["text ", ("b", None, ["in"], None), " xml"] None) { 0: "this" "type": "xml" 1: ["text ", {0: "b", 1:["in"]}, " xml"] } Ie, xml tag translates to a tuple: (name, dictofattributes, contentlist, miscellaneousinfo) where miscellaneousinfo can be anything, (but defaults to None) (with the intention of adding, eg, line number information) special cases: name of "" means "top level, no containing tag". Top level parse always looks like this:: ("", list, None, None) contained text of None means In order to support stuff like:: AT THE MOMENT & ETCETERA ARE IGNORED. THEY MUST BE PROCESSED IN A POST-PROCESSING STEP. PROLOGUES ARE NOT UNDERSTOOD. OTHER STUFF IS PROBABLY MISSING.

def parseFile(filename):

(no documentation string)

def parsexmlSimple(xmltext, oneOutermostTag=0,eoCB=None,entityReplacer=unEscapeContentList):

official interface: discard unused cursor info

def parsexmlSimple(xmltext, oneOutermostTag=0,eoCB=None,entityReplacer=unEscapeContentList):

official interface: discard unused cursor info

def pprettyprint(parsedxml):

pretty printer mainly for testing

def skip_prologue(text, cursor):

skip any prologue found after cursor, return index of rest of text

def test():

(no documentation string)

def testparse(s):

(no documentation string)

def unEscapeContentList(contentList):

(no documentation string)

reportlab.lib.set_ops module

From before Python had a Set class...

def intersect(seq1, seq2):

(no documentation string)

def union(seq1, seq2):

(no documentation string)

def unique(seq):

(no documentation string)

reportlab.lib.styles module

Classes for ParagraphStyle and similar things. A style is a collection of attributes, but with some extra features to allow 'inheritance' from a parent, and to ensure nobody makes changes after construction. ParagraphStyle shows all the attributes available for formatting paragraphs. getSampleStyleSheet() returns a stylesheet you can use for initial development, with a few basic heading and text styles.

def getSampleStyleSheet():

Returns a stylesheet object

def testStyles():

(no documentation string)

Class LineStyle:

(no documentation string)

def prepareCanvas(self, canvas):

You can ask a LineStyle to set up the canvas for drawing the lines.

Class ParagraphStyle:

(no documentation string)

Class PropertySet:

(no documentation string)

def listAttrs(self, indent=''):

(no documentation string)

def refresh(self):

re-fetches attributes from the parent on demand; use if you have been hacking the styles. This is used by __init__

reportlab.lib.testutils module

Provides support for the test suite. The test suite as a whole, and individual tests, need to share certain support functions. We have to put these in here so they can always be imported, and so that individual tests need to import nothing more than "reportlab.whatever..."

def getCVSEntries(folder, files=1, folders=0):

Returns a list of filenames as listed in the CVS/Entries file.

'folder' is the folder that should contain the CVS subfolder. If there is no such subfolder an empty list is returned. 'files' is a boolean; 1 and 0 means to return files or not. 'folders' is a boolean; 1 and 0 means to return folders or not.

def isWritable(D):

(no documentation string)

def makeSuiteForClasses(*classes):

Return a test suite with tests loaded from provided classes.

def outputfile(fn):

This works out where to write test output. If running code in a locked down file system, this will be a temp directory; otherwise, the output of 'test_foo.py' will normally be a file called 'test_foo.pdf', next door.

def printLocation(depth=1):

(no documentation string)

def setOutDir(name):

Is it a writable file system distro being invoked within test directory? If so, can write test output here. If not, it had better go in a temp directory. Only do this once per process

Class CVSGlobDirectoryWalker:

An directory tree iterator that checks for CVS data.

def filterFiles(self, folder, files):

Filters files not listed in CVS subfolder.

This will look in the CVS subfolder of 'folder' for
a file named 'Entries' and filter all elements from
the 'files' list that are not listed in 'Entries'.

Class ExtConfigParser:

A slightly extended version to return lists of strings.

def getstringlist(self, section, option):

Coerce option to a list of strings or return unchanged if that fails.

Class GlobDirectoryWalker:

A forward iterator that traverses files in a directory tree.

def filterFiles(self, folder, files):

Filter hook, overwrite in subclasses as needed.

Class RestrictedGlobDirectoryWalker:

An restricted directory tree iterator.

def filterFiles(self, folder, files):

Filters all items from files matching patterns to ignore.

reportlab.lib.textsplit module

Helpers for text wrapping, hyphenation, Asian text splitting and kinsoku shori. How to split a 'big word' depends on the language and the writing system. This module works on a Unicode string. It ought to grow by allowing ore algoriths to be plugged in based on possible knowledge of the language and desirable 'niceness' of the algorithm.

def cjkwrap(text, width, encoding="utf8"):

(no documentation string)

def dumbSplit(word, widths, availWidth):

This function attempts to fit as many characters as possible into the available space, cutting "like a knife" between characters. This would do for Chinese. It returns a list of (text, extraSpace) items where text is a Unicode string, and extraSpace is the points of unused space available on the line. This is a structure which is fairly easy to display, and supports 'backtracking' approaches after the fact.

Test cases assume each character is ten points wide...

```
>>> dumbSplit(u'Hello', [10]*5, 60)
[[10.0, u'Hello']]
>>> dumbSplit(u'Hello', [10]*5, 50)
[[0.0, u'Hello']]
>>> dumbSplit(u'Hello', [10]*5, 40)
[[0.0, u'Hell'], [30, u'o']]
```

def getCharWidths(word, fontName, fontSize):

Returns a list of glyph widths. Should be easy to optimize in _rl_accel

```
>>> getCharWidths('Hello', 'Courier', 10)
[6.0, 6.0, 6.0, 6.0, 6.0]
>>> from reportlab.pdfbase.cidfonts import UnicodeCIDFont
>>> from reportlab.pdfbase.pdfmetrics import registerFont
>>> registerFont(UnicodeCIDFont('HeiseiMin-W3'))
>>> getCharWidths(u'\u6771\u4EAC', 'HeiseiMin-W3', 10)   #most kanji are 100 ems
[10.0, 10.0]
```

def kinsokuShoriSplit(word, widths, availWidth):

Split according to Japanese rules according to CJKV (Lunde).

Essentially look for "nice splits" so that we don't end a line
with an open bracket, or start one with a full stop, or stuff like
that. There is no attempt to try to split compound words into
constituent kanji. It currently uses wrap-down: packs as much
on a line as possible, then backtracks if needed

```
        This returns a number of words each of which should just about fit
        on a line.  If you give it a whole paragraph at once, it will
        do all the splits.

        It's possible we might slightly step over the width limit
        if we do hanging punctuation marks in future (e.g. dangle a Japanese
        full stop in the right margin rather than using a whole character
        box.
```

def wordSplit(word, availWidth, fontName, fontSize, encoding='utf8'):

```
    Attempts to break a word which lacks spaces into two parts, the first of which
    fits in the remaining space.  It is allowed to add hyphens or whatever it wishes.

    This is intended as a wrapper for some language- and user-choice-specific splitting
    algorithms.  It should only be called after line breaking on spaces, which covers western
    languages and is highly optimised already.  It works on the 'last unsplit word'.

    Presumably with further study one could write a Unicode splitting algorithm for text
    fragments whick was much faster.

    Courier characters should be 6 points wide.
    >>> wordSplit('HelloWorld', 30, 'Courier', 10)
    [[0.0, 'Hello'], [0.0, 'World']]
    >>> wordSplit('HelloWorld', 31, 'Courier', 10)
    [[1.0, 'Hello'], [1.0, 'World']]
```

reportlab.lib.units module

```
    Defines inch, cm, mm etc as multiples of a point You can now in user-friendly units by
    doing:: from reportlab.lib.units import inch r = Rect(0, 0, 3 * inch, 6 * inch)
```

def toLength(s):

```
    convert a string to  a length
```

reportlab.lib.utils module

```
    Gazillions of miscellaneous internal utility functions
```

def annotateException(msg,enc='utf8'):

```
    add msg to the args of an existing exception
```

def commajoin(l):

```
    Inverse of commasplit, except that whitespace around items is not conserved.
    Adds more whitespace than needed for simplicity and performance.

    >>> commasplit(commajoin(['a', 'b', 'c']))
    ['a', 'b', 'c']
    >>> commasplit((commajoin(['a,', ' b ', 'c']))
    ['a,', 'b', 'c']
    >>> commasplit((commajoin(['a ', ',b', 'c']))
    ['a', ',b', 'c']
```

def commasplit(s):

```
    Splits the string s at every unescaped comma and returns the result as a list.
    To escape a comma, double it. Individual items are stripped.
    To avoid the ambiguity of 3 successive commas to denote a comma at the beginning
    or end of an item, add a space between the item seperator and the escaped comma.

    >>> commasplit('a,b,c')
    ['a', 'b', 'c']
    >>> commasplit('a,, , b , c    ')
    ['a,', 'b', 'c']
    >>> commasplit('a, ,,b, c')
    ['a', ',b', 'c']
```

def escapeOnce(data):

```
    Ensure XML output is escaped just once, irrespective of input

        >>> escapeOnce('A & B')
        'A &amp; B'
        >>> escapeOnce('C &amp; D')
        'C &amp; D'
        >>> escapeOnce('E &amp;amp; F')
```

```
        'E &amp;amp; F'
```

def escapeTextOnce(text):

Escapes once only

def findInPaths(fn,paths,isfile=True,fail=False):

search for relative files in likely places

def find_locals(func,depth=0):

apply func to the locals at each stack frame till func returns a non false value

def flatten(L):

recursively flatten the list or tuple L

def getArgvDict(kw):**

Builds a dictionary from its keyword arguments with overrides from sys.argv.
Attempts to be smart about conversions, but the value can be an instance
of ArgDictValue to allow specifying a conversion function.

def getHyphenater(hDict=None):

(no documentation string)

def getImageData(imageFileName):

Get width, height and RGB pixels from image file. Wraps Java/PIL

def getStringIO(buf=None):

unified StringIO instance interface

def import_zlib():

(no documentation string)

def isCompactDistro():

return truth if not a file system distribution

def isFileSystemDistro():

return truth if a file system distribution

def isSeqType(v,_st=(tuple,list)):

(no documentation string)

def isSourceDistro():

return truth if a source file system distribution

def markfilename(filename,creatorcode=None,filetype=None):

(no documentation string)

def open_and_read(name,mode='b'):

(no documentation string)

def open_and_readlines(name,mode='t'):

(no documentation string)

def open_for_read(name,mode='b', urlopen=urllib2.urlopen):

attempt to open a file or URL for reading

def open_for_read_by_name(name,mode='b'):

(no documentation string)

def prev_this_next(items):

Loop over a collection with look-ahead and look-back.

From Thomas Guest,
http://wordaligned.org/articles/zippy-triples-served-with-python

Seriously useful looping tool (Google "zippy triples")
lets you loop a collection and see the previous and next items,
which get set to None at the ends.

To be used in layout algorithms where one wants a peek at the
next item coming down the pipe.

```
def recursiveGetAttr(obj, name):
    Can call down into e.g. object1.object2[4].attr
def recursiveImport(modulename, baseDir=None, noCWD=0, debug=0):
    Dynamically imports possible packagized module, or raises ImportError
def recursiveSetAttr(obj, name, value):
    Can call down into e.g. object1.object2[4].attr = value
def rl_get_module(name,dir):
    (no documentation string)
def rl_getmtime(pn,os_path_isfile=os.path.isfile,os_path_normpath=os.path.normpath,os_path_getmtime=os.path.
    (no documentation string)
    def rl_glob(pattern,glob=glob.glob):
    (no documentation string)
def rl_isdir(pn,os_path_isdir=os.path.isdir,os_path_normpath=os.path.normpath):
    (no documentation string)
def rl_isfile(fn,os_path_isfile=os.path.isfile):
    (no documentation string)
def rl_listdir(pn,os_path_isdir=os.path.isdir,os_path_normpath=os.path.normpath,os_listdir=os.listdir):
    (no documentation string)
def simpleSplit(text,fontName,fontSize,maxWidth):
    (no documentation string)
Class ArgvDictValue:
    A type to allow clients of getArgvDict to specify a conversion function
Class DebugMemo:
    Intended as a simple report back encapsulator

        Typical usages:

        1. To record error data::

            dbg = DebugMemo(fn='dbgmemo.dbg',myVar=value)
            dbg.add(anotherPayload='aaaa',andagain='bbb')
            dbg.dump()

        2. To show the recorded info::

            dbg = DebugMemo(fn='dbgmemo.dbg',mode='r')
            dbg.load()
            dbg.show()

        3. To re-use recorded information::

            dbg = DebugMemo(fn='dbgmemo.dbg',mode='r')
                dbg.load()
            myTestFunc(dbg.payload('myVar'),dbg.payload('andagain'))

        In addition to the payload variables the dump records many useful bits
        of information which are also printed in the show() method.
    def add(self,**kw):
        (no documentation string)
    def dump(self):
        (no documentation string)
    def dumps(self):
        (no documentation string)
    def load(self):
        (no documentation string)
    def loads(self,s):
```

(no documentation string)

def payload(self,name):

(no documentation string)

def show(self):

(no documentation string)

Class FmtSelfDict:

mixin to provide the _fmt method

reportlab.lib.validators module

Standard verifying functions used by attrmap.

Class Auto:

(no documentation string)

def test(self,x):

(no documentation string)

Class AutoOr:

(no documentation string)

def test(self,x):

(no documentation string)

Class DerivedValue:

This is used for magic values which work themselves out.
An example would be an "inherit" property, so that one can have

```
drawing.chart.categoryAxis.labels.fontName = inherit
```

and pick up the value from the top of the drawing.
Validators will permit this provided that a value can be pulled
in which satisfies it. And the renderer will have special
knowledge of these so they can evaluate themselves.

def getValue(self, renderer, attr):

Override this. The renderers will pass the renderer,
and the attribute name. Algorithms can then backtrack up
through all the stuff the renderer provides, including
a correct stack of parent nodes.

Class EitherOr:

(no documentation string)

def test(self, x):

(no documentation string)

Class Inherit:

(no documentation string)

def getValue(self, renderer, attr):

(no documentation string)

Class NoneOr:

(no documentation string)

def test(self, x):

(no documentation string)

Class OneOf:

Make validator functions for list of choices.

Usage:
f = reportlab.lib.validators.OneOf('happy','sad')
or
f = reportlab.lib.validators.OneOf(('happy','sad'))
f('sad'),f('happy'), f('grumpy')
(1,1,0)

```
    def test(self, x):
        (no documentation string)
Class SequenceOf:
    (no documentation string)
    def test(self, x):
        (no documentation string)
Class Validator:
    base validator class
    def normalize(self,x):
        (no documentation string)
    def normalizeTest(self,x):
        (no documentation string)
Class isInstanceOf:
    (no documentation string)
    def test(self,x):
        (no documentation string)
Class isNumberInRange:
    (no documentation string)
    def test(self, x):
        (no documentation string)
Class matchesPattern:
    Matches value, or its string representation, against regex
    def test(self,x):
        (no documentation string)
```

reportlab.lib.xmllib module

```
    From before xmllib was in the Python standard library. Probably ought to be removed
def test(args = None):
    (no documentation string)
Class FastXMLParser:
    (no documentation string)
    def close(self):
        (no documentation string)
    def feed(self, data): # overridden by reset
        (no documentation string)
    def finish_endtag(self, tag):
        (no documentation string)
    def finish_starttag(self, tag, attrs):
        (no documentation string)
    def handle_cdata(self, data):
        (no documentation string)
    def handle_charref(self, name):
        (no documentation string)
    def handle_comment(self, data):
        (no documentation string)
    def handle_data(self, data):
```

```
        (no documentation string)
    def handle_endtag(self, tag, method):
        (no documentation string)
    def handle_entityref(self, name):
        (no documentation string)
    def handle_proc(self, name, data):
        (no documentation string)
    def handle_special(self, data):
        (no documentation string)
    def handle_starttag(self, tag, method, attrs):
        (no documentation string)
    def reset(self):
        (no documentation string)
    def setliteral(self, *args):
        (no documentation string)
    def setnomoretags(self):
        (no documentation string)
    def syntax_error(self, lineno, message):
        (no documentation string)
    def translate_references(self, data):
        (no documentation string)
    def unknown_charref(self, ref): pass
        (no documentation string)
    def unknown_endtag(self, tag): pass
        (no documentation string)
    def unknown_entityref(self, ref): pass
        (no documentation string)
    def unknown_starttag(self, tag, attrs): pass
        (no documentation string)
Class SlowXMLParser:
    (no documentation string)
    def close(self):
        (no documentation string)
    def feed(self, data):
        (no documentation string)
    def finish_endtag(self, tag):
        (no documentation string)
    def finish_starttag(self, tag, attrs):
        (no documentation string)
    def goahead(self, end):
        (no documentation string)
    def handle_cdata(self, data):
        (no documentation string)
    def handle_charref(self, name):
        (no documentation string)
    def handle_comment(self, data):
```

(no documentation string)

def handle_data(self, data):

(no documentation string)

def handle_endtag(self, tag, method):

(no documentation string)

def handle_entityref(self, name):

(no documentation string)

def handle_proc(self, name, data):

(no documentation string)

def handle_special(self, data):

(no documentation string)

def handle_starttag(self, tag, method, attrs):

(no documentation string)

def parse_cdata(self, i):

(no documentation string)

def parse_comment(self, i):

(no documentation string)

def parse_endtag(self, i):

(no documentation string)

def parse_proc(self, i, res):

(no documentation string)

def parse_starttag(self, i):

(no documentation string)

def reset(self):

(no documentation string)

def setliteral(self, *args):

(no documentation string)

def setnomoretags(self):

(no documentation string)

def syntax_error(self, lineno, message):

(no documentation string)

def translate_references(self, data):

(no documentation string)

def unknown_charref(self, ref): pass

(no documentation string)

def unknown_endtag(self, tag): pass

(no documentation string)

def unknown_entityref(self, ref): pass

(no documentation string)

def unknown_starttag(self, tag, attrs): pass

(no documentation string)

Class TestXMLParser:

(no documentation string)

def close(self):

(no documentation string)

def flush(self):

```
        (no documentation string)
    def handle_cdata(self, data):
        (no documentation string)
    def handle_comment(self, data):
        (no documentation string)
    def handle_data(self, data):
        (no documentation string)
    def handle_proc(self, name, data):
        (no documentation string)
    def handle_special(self, data):
        (no documentation string)
    def syntax_error(self, lineno, message):
        (no documentation string)
    def unknown_charref(self, ref):
        (no documentation string)
    def unknown_endtag(self, tag):
        (no documentation string)
    def unknown_entityref(self, ref):
        (no documentation string)
    def unknown_starttag(self, tag, attrs):
        (no documentation string)
Class XMLParser:
    (no documentation string)
    def close(self):
        (no documentation string)
    def feed(self, data): # overridden by reset
        (no documentation string)
    def finish_endtag(self, tag):
        (no documentation string)
    def finish_starttag(self, tag, attrs):
        (no documentation string)
    def handle_cdata(self, data):
        (no documentation string)
    def handle_charref(self, name):
        (no documentation string)
    def handle_comment(self, data):
        (no documentation string)
    def handle_data(self, data):
        (no documentation string)
    def handle_endtag(self, tag, method):
        (no documentation string)
    def handle_entityref(self, name):
        (no documentation string)
    def handle_proc(self, name, data):
        (no documentation string)
    def handle_special(self, data):
```

(no documentation string)

def handle_starttag(self, tag, method, attrs):

(no documentation string)

def reset(self):

(no documentation string)

def setliteral(self, *args):

(no documentation string)

def setnomoretags(self):

(no documentation string)

def syntax_error(self, lineno, message):

(no documentation string)

def translate_references(self, data):

(no documentation string)

def unknown_charref(self, ref): pass

(no documentation string)

def unknown_endtag(self, tag): pass

(no documentation string)

def unknown_entityref(self, ref): pass

(no documentation string)

def unknown_starttag(self, tag, attrs): pass

(no documentation string)

reportlab.lib.yaml module

.h1 Welcome to YAML! YAML is "Yet Another Markup Language" - a markup language which is easier to type in than XML, yet gives us a reasonable selection of formats. The general rule is that if a line begins with a '.', it requires special processing. Otherwise lines are concatenated to paragraphs, and blank lines separate paragraphs. If the line ".foo bar bletch" is encountered, it immediately ends and writes out any current paragraph. It then looks for a parser method called 'foo'; if found, it is called with arguments (bar, bletch). If this is not found, it assumes that 'foo' is a paragraph style, and the text for the first line of the paragraph is 'bar bletch'. It would be up to the formatter to decide whether on not 'foo' was a valid paragraph. Special commands understood at present are: dot image filename - adds the image to the document dot beginPre Code - begins a Preformatted object in style 'Code' dot endPre - ends a preformatted object.

def parseFile(filename):

(no documentation string)

def parseText(textBlock):

(no documentation string)

Class BaseParser:

"Simplest possible parser with only the most basic options.

This defines the line-handling abilities and basic mechanism.
The class YAMLParser includes capabilities for a fairly rich
story.

def beginPre(self, stylename):

(no documentation string)

def endPara(self):

(no documentation string)

def endPre(self):

(no documentation string)

def image(self, filename):

(no documentation string)

def parseFile(self, filename):

(no documentation string)

def parseText(self, textBlock):

Parses the a possible multi-line text block

def readLine(self, line):

(no documentation string)

def reset(self):

(no documentation string)

Class Parser:

This adds a basic set of "story" components compatible with HTML & PDF.

Images, spaces

def custom(self, moduleName, funcName):

Goes and gets the Python object and adds it to the story

def nextPageTemplate(self, templateName):

(no documentation string)

def pageBreak(self):

Inserts a frame break

def vSpace(self, points):

Inserts a vertical spacer

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
```